# *Performance Analysis of*

# *Single- and Multiprocessor Computing Systems*

*Dem Fachbereich Elektrotechnik und Informationstechnik*
*der Universität Hannover*

*zur Erlangung des Grades*

## *Dr.-Ing. habil.*

*vorgelegt von*

*Dr.-Ing. Jürgen Brehm*

*Hannover - 1999*

# Contents

## List of Figures

## List of Tables

## Abbreviations

- ALU
  Arithmetic - logic unit
- BLAS
  Basic Linear Algebra Subprograms
- CFD
  Computational Fluid Dynamics
- CG
  Conjugate Gradient
- CMi
  Communication phase i
- CPi
  Computation phase i
- CPU
  Central Processing Unit
- DEC
  Digital Equipment Corporation
- DMA
  Direct Memory Access
- d_n_procs
  PerPreT variable for number of processors
- d_p_size
  PerPreT variable for problem size
- DSPN
  Deterministic and Stochastic Petri Net
- FCFS
  First Come First Serve
- FESC
  Flow Equivalent Service Center
- FFT
  Fast Fourier Transform
- GAMESS
  General Atomic and Molecular Electronic Structure System
- GBIS
  Graphical Benchmark Information Service
- GSPN
  Generalized Stochastic Petri Net
- HP
  Hewlett Packard
- HPG
  High Performance Group
- HPF
  High Performance Fortran
- IS
  Integer Sort
- λ
  Arrival rate
- LAN
  Local Area Network
- lflop
  PerPreT variable for node performance
- LT
  Legendre Transform
- μ
  Departure rate
- mflop[][]
  PerPreT variable for node performance
- MFLOPS
  Mega Floating Point Operations per Second
- MG
  Multigrid
- MIMD
  Multiple Instruction Multiple Data
- MM
  Matrix Multiplication
- MIPS
  Millions of Instructions per Second
- MPI
  Message Passing Interface
- MVA
  Mean Value Analysis
- N
  Problem size

- NAS
  Numerical Aerodynamic Simulation
- NPB
  NAS Parallel Benchmarks
- nprocs
  Number of processors
- psize
  Problem size
- P
  Number of processors
- PARKBENCH
  PARallel Kernels and BENCHmarks
- PERFECT
  Performance Evaluation for Cost-effective Transformations
- PerPreT
  Performance Prediction Tool
- PICL
  Portable Instrumented Communication Library
- PIP
  Parallel Instrumented Program
- PSTSWM
  Parallel Spectral Transform Shallow Water Model
- PTG
  Program Task Graph
- PVM
  Parallel Virtual Machine
- SIMD
  Single Instruction Multiple Data
- SLALOM
  Scalable, Language-independent, Ames Laboratory, One-minute Measurement
- SPMD
  Single Program Multiple Data
- SPEC
  System Performance Evaluation Group
- st
  PerPreT variable for number of statements
- SUT
  System under test
- $t_{comm}$
  Communication time
- $t_{exec}$
  Execution time
- $t_{over}$
  Overhead time
- TPS
  Transactions per second

# 1. Introduction

## 1.1. Basic Classification of Performance Analysis

The growing number of different parallel computer architectures demands methods for a comparison of these systems relative to performance issues. To increase reliability and performance of computing systems is an important motivation for the development of new processors and computer architectures.

Two basic techniques to achieve higher performance can be observed, either improvements in chip technology are made and/or architectural improvements, especially parallelism, are used. The chips are getting larger, the transistors on chip are getting smaller as are the connections on chip. Higher frequencies are used. Thus, the performance upgrades of the last decades were possible. It is difficult to predict, when the chip technology will reach its physical restraints that can no longer be overcome (speed of light for frequency, connections that are less than one atom wide). If the physical limits of chip technology are reached, parallelism is the only alternative to further upgrade performance. Parallelism is already used in all computing systems on levels ranging from register width to number of concurrently running processors.

Existing parallel systems can roughly be classified in two categories, namely, SIMD- (single instruction, multiple data) and MIMD- (multiple instruction, multiple data) architectures. As the classification indicates, the performance of SIMD-architectures is mainly based on data parallelism. Many processors or ALUs (arithmetic logical units) are executing the same instruction synchronously on different data. Such systems were very successful at the beginning of the development of parallel systems. Later they were replaced by more flexible and more powerful MIMD-architectures. The methods presented in this book mainly describe approaches for the performance analysis of MIMD-systems.

Performance analysis of computing systems is carried out by:

- computer architects
- computer vendors and buyers
- system software developers
- application developers.

The results of the performance analysis are used to:

- improve the architecture
- detect and eliminate bottlenecks
- compare systems.

The performance of a system is vitally important to the vendor and buyer. Looking at figures like MIPS (mega instructions per second), or MFLOPS (mega floating point operations per second), or TPS (transactions per second) in order to describe the performance of a system in a vendor‘s catalogue does not necessarily inform the potential buyer of the true performance. First of all, these numbers are always peak performances, meaning these numbers are true (if at all) only for a very limited selection of applications. They are less useful if a user is interested in the performance of the target system for specific applications. This problem is well known from single processor architectures, however, it is more important and more complex for multiprocessor systems. The total performance of such a system is not only determined by the processor performance but also by the memory management strategy (in case of shared memory systems), network architecture, system throughput, communication, and synchronization management strategies.

The approaches for performance analysis of computer systems can be divided into three categories:

- performance measurement,
- performance modeling,
- and performance simulation.

In [Lin98] a basic classification of these performance evaluation techniques is given (compare Fig. 1.1.).

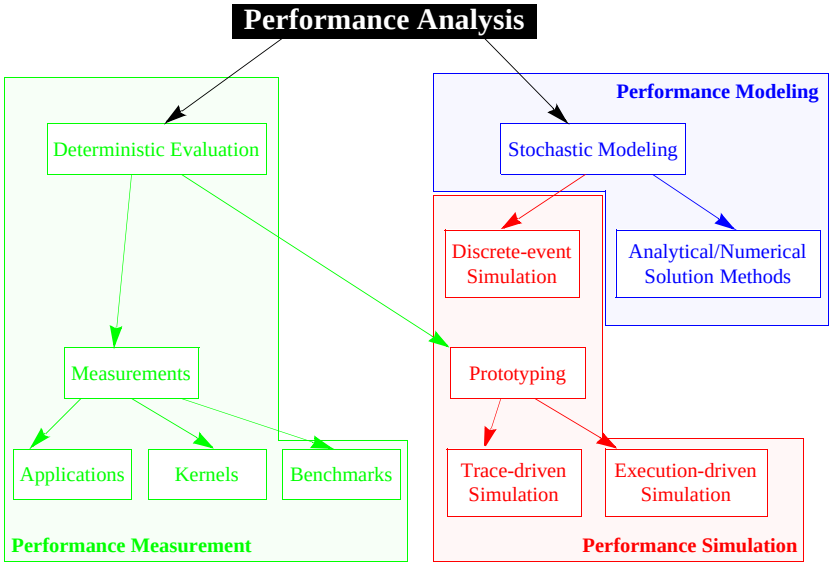



Fig. 1.1. Classification of performance analysis techniques

### Performance Measurement

The main goal for deterministic evaluation techniques based on performance measurement of real systems is to find performance bottlenecks and, if possible, to minimize them. Normally it is not possible to change the hardware of existing systems, but the results can be used to improve the design of successor systems. Another possibility is to add hardware to reduce system bottlenecks if the system is scalable. A tuning of the system software is also often possible.

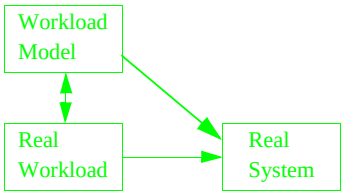



Fig. 1.2. Performance measurement

The load used for the performance measurement can be a suite of real applications or a workload model can be used (compare Fig. 1.2.). The results of performance measurement techniques compared with performance modeling techniques are more dependable and accurate, since no losses of accuracy through model abstraction can occur.

### Performance Modeling

For performance modeling techniques, the system load or the system architecture, or both, are represented through a model (compare Fig. 1.3.).

Deterministic and stochastic models are easier and more efficient to analyze the performance of future systems than performance measurement, because hardware prototypes are expensive to realize. Any representation of reality through models is an abstraction of reality and, thus, suffers from a loss of accuracy. The more detailed and complex a model is, the better the accuracy is likely to be. It is important to make reasonable assumptions regarding the structure and detail of the models. The art of modeling is to find a good trade-off between needed accuracy and degree of abstraction. Performance modeling using deterministic and stochastic models becomes very complex and difficult if the dynamic behavior of parallel applications on MIMD-systems has to be examined in detail.

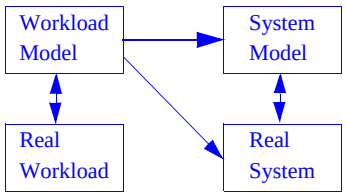



Fig. 1.3. Performance modeling

### Performance Simulation

Performance simulation techniques use a system simulator to execute a model of the workload or the real workload (compare Fig. 1.4.). Performance simulation is well suited for complex architectures which are difficult to model (especially dynamic behavior). The problems in performance simulation are extremely long runtimes which allow only small workload models or small real workloads.

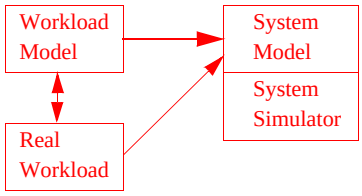



Fig. 1.4. Performance simulation

## 1.2. Refined Classification of Performance Analysis

This text includes a survey of methods from two categories (performance measurement and performance modeling) and a short summary on performance simulation. An answer to the question which method should be used for which problem will be given. Possible problems are:

- purchase decision
- architectural improvements
- optimization of applications

**Purchase Decision**

One use of models is to help determine the computer with the best price/performance ratio for a user's needs. Two scenarios are possible: the best machine for a given amount of money has to be found, or the cheapest machine to fulfill given requirements has to be found.

**Architectural Improvements**

A main goal of the system designer is to improve the architecture of the system. These improvements cannot always be realized in hardware because it is too expensive or too time consuming. Using performance analysis techniques (measurement, modeling, simulation) can help to dramatically reduce the cost and time in computer development.

**Optimization of Applications**

In addition to buyers and system designers, application developers use performance analysis techniques. The main goal is, in this case, improvement of the user's applications. Parallel applications especially offer a high potential of optimization which is often not fully realized. Performance analysis allows the comparison of different versions of parallel applications in terms of runtime and efficiency without the need to implement them on a real machine. Furthermore, analysis can save time and money compared to real implementations. This effect is especially important for multiprocessor systems, because the implementation and testing of different versions of an algorithm is much more complex than doing the same on a monoprocessor.

Additionally, since implementations are not always portable among different multiprocessor systems, a new workload program for every system is necessary. Only a few approaches for common parallel languages ([PICL90], [MPI95], [Lin85]) exist, but none of them is accepted as a standard language. The systems do not only differ in programming language but also in the programming model used which often is a reflection of the system architecture (message passing, shared memory). This is one of the reasons for the low acceptance of multiprocessor systems which often show a better performance/price ratio than comparable vector processor systems.

A survey of existing and important performance analysis techniques is given in this text. Two new approaches for the performance evaluation of multiprocessor systems are presented, a benchmark generator and a modeling technique for massively parallel systems. The latter techniques is used for a complete performance evaluation and prediction tool which is presented in the last part of this text.

The basic classification of performance evaluation techniques (compare Fig. 1.1.) is now refined to be better be able to show the reader of this text the context of the described techniques. As shown in Fig. 1.5. performance evaluation can be carried out on real systems, on prototypes, or on models of real systems.

If a real system and its workload is available benchmarking is used to determine the performance. The prototype of a real system allows profiling techniques, low level test of the hardware and elapsed time benchmarking. If the real system is not built yet or not available, model based techniques have to be applied. they are divided into stochastic methods using queueing theory, Petri Nets or analytical models. The deterministic approaches are either simulation based or the system and workload are described by analytical models.

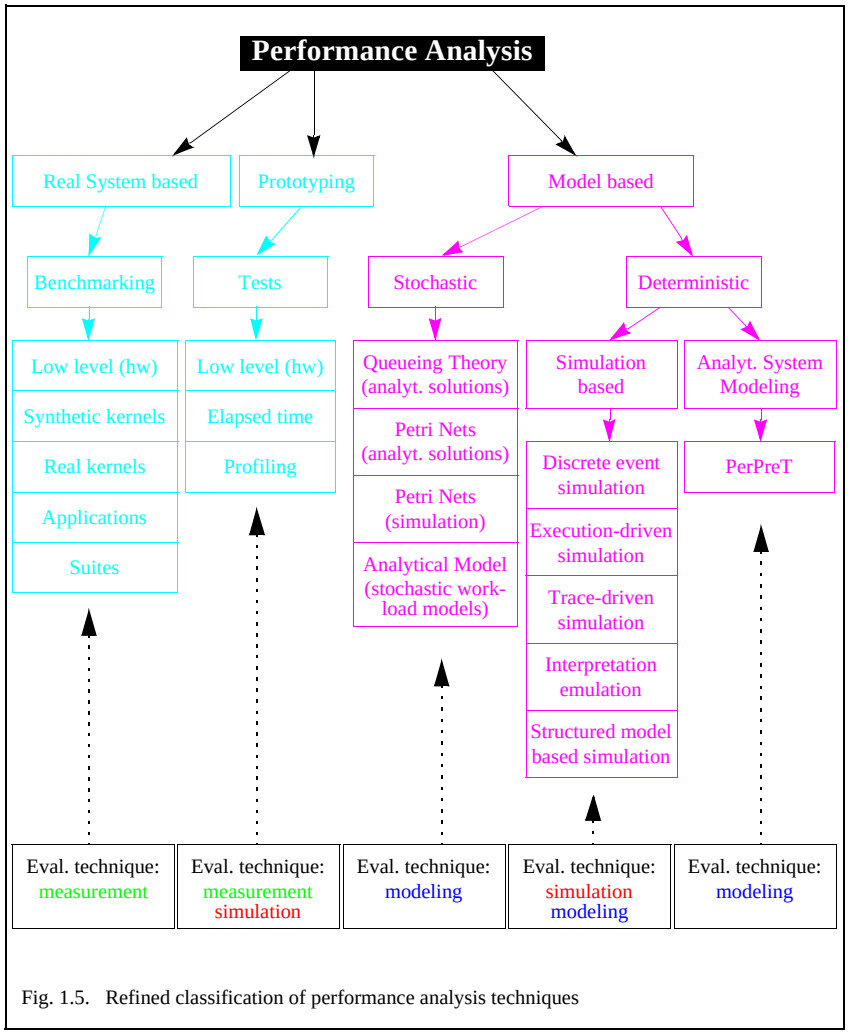



Fig. 1.5. Refined classification of performance analysis techniques

## 1.3. Organization of this Book

The first part of this book introduces and defines the basic notation in connection with performance analysis. The performance measures used and their meaning for computer architects are explained.

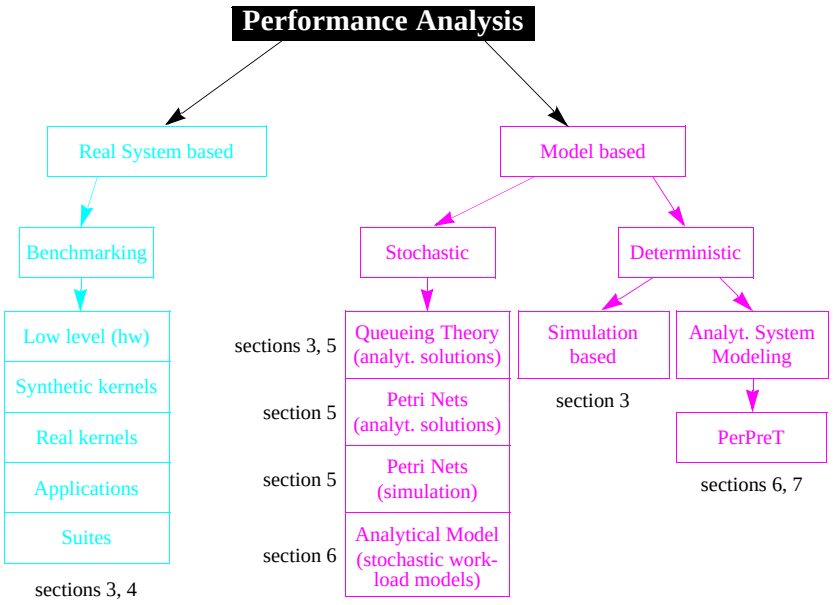
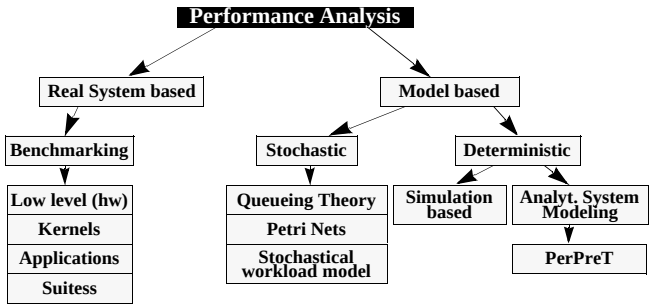



Fig. 1.6. Performance analysis techniques covered by this book

The next two sections describe the state of the art of performance measurement techniques based on deterministic evaluation for mono- and multiprocessor systems. A new approach for multiprocessor performance evaluation, the LOOP Language, is presented. Especially for multiprocessors, performance modeling techniques seem to be important. The different approaches for stochastic performance modeling are presented in the next part of the book. Stochastic modeling using Markov Chains or Petri Nets is discussed in section 5. A new approach for performance prediction using analytical parameterized models is presented in section 6. A case study shows the accuracy and usability of this approach. The last part of the book describes the resulting software tool of this performance evaluation

and prediction approach called PerPreT (Performance Prediction Tool).

The classification scheme shown in Fig. 1.6. is used throughout the rest of the book as an orientation help. It is displayed at the beginning of each chapter and the topics covered by the chapter are highlighted through bold boxes and names. Performance simulation is not subject of this book. However, some general information on this topic and some links to literature are given in section 3.3.




# 2. Basic Notation and Definitions

## 2.1. Notation

Computer scientists and researchers in performance analysis use many abbreviations and notations which are not always precise or sometimes used in different interpretations. Because of this disparity and for the better understanding, all relevant notation is defined and briefly explained in this section. If the meaning of a term is not unique it is explained how it is used in the context of this text.

### 2.1.1. General Terms in the Context of this Work

**Benchmark Program**

A benchmark program is a specific software with the goal to analyze the performance of a target system to be able to compare it with other systems. A benchmark program can be a real application or a synthetic program with the characteristics of a real application. Depending on the application, the benchmark program can stress single resources as memory access, arithmetic units, logical units, input/output devices or it can stress the system as a whole. In order to use the benchmark program to compare various target systems it should be portable to a wide variety of machines.

**Benchmark Test**

A benchmark test is the execution of a benchmark program on a target system. To keep results comparable, some rules (e.g., compiler options, no manual optimizations, system

configuration) apply, or the system and benchmark properties are written down in a protocol. The results of a benchmark test are typically performance figures like MIPS (mega instructions per second), MFLOPS (mega floating point operations per second), TPS (transactions per second), or predefined performance characteristics like SPECints and SPECrates (compare section 3.1.3. and [SPEC95]).

**Workload**

The workload of a system consists of a set of operations which are executed on the system. Using different levels of abstraction, a workload can be an algorithm, a kernel of an application, a synthetic program, or a real application. The operations can be arithmetic operations, instructions, database operations, or other operations to stress specific parts of the system.

**Kernel**

A kernel is the computationally intensive part of an application. It often consists of a small number of code lines or of some loop cascades. For simplicity, kernels are extracted from complex applications and used as benchmark programs. As in synthetic workloads, operating system issues such as I/O are often neglected in kernels.

**Application**

An application is a program to solve a problem on a computer. In contrast to kernels or synthetic programs, real problems are solved.

**Parallel Application**

A parallel application is a program to solve a problem on a multiprocessor. The program includes programming constructs which allows the problem to be solved concurrently using more than one processor. Besides constructs for the concurrent execution it normally also provides constructs for the communication and synchronization of subtasks. The goal is to reduce the application's execution time when allocated an increasing number of processors.

**Processor, Node**

In this text, processor, node, processor node, processing element, and node processor are used synonymously in connection with multiprocessors. A node consists of a CPU (central processing unit) and a link to a main memory module. The CPU can be directly connected (exclusive access) to a main memory module or it can be connected via a interconnection network with the main memory module (multiple CPUs have access).

**Monoprocessor**

A monoprocessor is a computer that consists of one processor to execute programs. In the literature a monoprocessor system is also called single processor system or uniprocessor system. These terms are used synonymously throughout this text.

**Multiprocessor**

A multiprocessor consists of several processors configured to execute programs concurrently. In contrast to a workstation cluster or computers loosely coupled via a LAN (large area network), the processors are tightly coupled, operate under the same operating system, and include a programming environment or language with programming constructs or routines for synchronization and communication. In the context of this book, the terms multiprocessor, parallel processing system, multicomputer, and MIMD system are equivalent.

**Massively Parallel System**

A multiprocessor is called massively parallel if it consists of more than 100 processors. This number is arbitrarily chosen to distinguish between parallel and massively parallel. It reflects today's systems. Systems consisting of up to several thousands of processors are commercially available.

**Performance Prediction**

The results of performance modeling or simulation allow performance predictions for a system. The predictions can be for workload models, for application kernels, for applications, or more general systems. For example, the peak performance of a computing system can be predicted. Examples for performance measures are execution time, speedup, and efficiency (for definition of performance measures, see section 2.2.2.).

**Performance Evaluation / Analysis**

The results of benchmark tests are used for the performance evaluation of a system. The performance of the system is measured in terms of operations per time unit. The operations can be basic instructions, arithmetic operations (floating point or integer), or transactions in case of data base systems. Methods for performance evaluation use measurement, modeling, and simulation. The terms performance evaluation and performance analysis are used synonymously in this book.

**Peak Performance**

The maximum performance of a system is called peak performance. There are two different ways to find the peak performance of a system:

- Perform several benchmark tests and use the results as the sustained peak performance for these tests.
- Calculate the peak performance using the architectural parameters. For example, the peak performance can be found by taking the performance of the execution unit, multiply it by the number of execution units per processor and multiply it by the number of processors in case of a multiprocessor systems. The result is the theoretical peak performance, an unrealistic number often used by computer vendors.

Since the theoretical peak performance is calculated without a workload, performance losses by memory access, system software (compiler, operating system, etc.), and possibly synchronization and communication in case of multiprocessors, the theoretical peak performance is, in general, much higher than the sustained peak performance.

**Communication Bandwidth**

Parallel applications running concurrently on multiprocessors require mechanisms for communication and synchronization. The maximum amount of data that can be transferred during a time unit between a pair of processors is defined as communication bandwidth. The communication can be realized by links connecting processors in the case of distributed memory systems or by access to common data in the case of shared memory systems. The different multiprocessor architectures are explained in section 4.1. The peak communication bandwidth of a distributed memory multiprocessor is the sum of the concurrent link bandwidths. The peak communication bandwidth of a shared memory multiprocessor is determined by the bandwidth between processors and memory. As discussed for computational peak performance, the communication peak performance can be calculated using the architectural parameters (theoretical communication peak performance) or it can be determined by communication benchmarks (sustained communication peak performance).

### 2.1.2. Program Task Graphs

Performance analysis often use Program Task Graphs (PTGs) to formally describe workloads [Kat96]. A PTG is defined as a directed acyclic graph. The nodes represent tasks and the edges represent communication relations between the tasks. The edges and the nodes can have attributes to describe the timings of the tasks and communications.

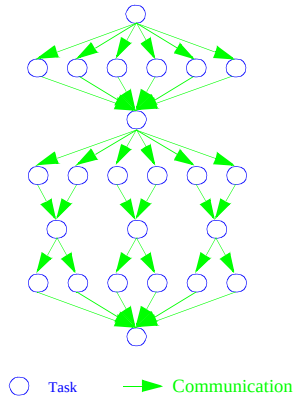



Fig. 2.1. Program Task Graph (PTG)

In Fig. 2.1. a simple example for a PTG is given. The workload is divided into small units (subtasks). Each subtask is represented by a PTG node. The nodes can carry attributes to represent the weight of the subtask. Possible attributes are: number of instructions, time units, number of arithmetic operations, number of transactions when dealing with a data base, number of memory accesses, and others. The edges (arrows) between the subtasks show the data dependencies. In the case of multiprocessor workloads, data dependencies are solved by synchronization and communication between the subtasks. Like the nodes, the edges can carry attributes with weight functions, such as timings for a message transfer, or synchronization timings, or message lengths. The description of a parallel (multiprocessor) workload as a PTG makes it easy to distribute the subtasks onto the processors.

As an example, the PTG of Fig. 2.1. is mapped onto six processors in Fig. 2.2. The data dependencies (edges) represent the communications between the processors. A limitation of the PTG representation is that communication arrows have to be deterministic.

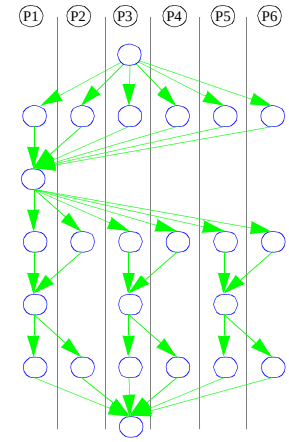



Fig. 2.2. Mapping of a parallel workload described by a PTG onto 6 processors

If the execution times of the subtasks and the timing of the communication are known, an optimization of the mapping of the PTG onto the processors can be carried out. The goal of the optimization is to minimize the total execution time [Kat96].

## 2.2. Performance Analysis Definitions

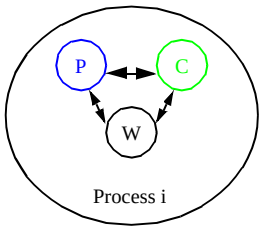
### 2.2.1. Timing

**Execution Time**

The execution time of a program $t_{exec}$ in the scope of this text is the time from the execution of the first program instruction until the last program instruction. Program loading times are not considered here. For parallel programs that are executed by one or more processors the same definition is used.

$$t_{exec} = T_{stop} - T_{start} \qquad \text{(Eq.2.1)}$$

The execution time of a parallel program normally consists of three portions, namely program execution time (instructions and operations), communication time, and waiting time.




P = Program execution
C = Communication
W = Waiting

Fig. 2.3. Possible process states during the execution of a parallel program

The portions result from the times the processes are in the different states "Program execution (P)", "Communication (C)", or "Waiting (W)" (compare Fig. 2.3. and Fig. 2.4.).

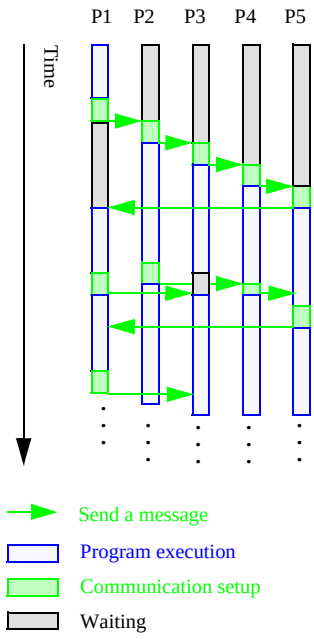



Fig. 2.4. Parallel program execution

**Communication Time**

The communication time of a parallel program $t_{comm}$ is the sum of the times for the communication between the processors:

$$t_{comm} = \sum_i t_{msg}^i \qquad \text{(Eq.2.2)}$$

with *i = 1, 2,..., number of messages*.

The time for one communication between a pair of processors in a message passing system $t_{msg}$ is divided into three portions: setup time for the sending of a message $t_{ss}$, transfer time $t_t$ and receive time $t_{rec}$.

$$t_{msg} = t_{ss} + t_t + t_{rec} \qquad \text{(Eq.2.3)}$$

**Synchronization Time**

The synchronization time $t_{sync}$ is the time spend for synchronization operations. A typical synchronization operation is a global barrier, i.e. all concurrently running processes wait at a global synchronization point specified in the parallel program until the last process arrives at this point (called barrier). This arrival releases all processes and they may continue.

**Waiting Time**

The difference between synchronization time $t_{sync}$ and waiting time $t_{wait}$ is that the time $t_{wait}$ is caused by waiting for communication, i.e. one process would like to receive a message that is not available yet. Since the operation *receive message* is normally blocking, the process has to wait until the message has arrived.

**Overhead Time**

The sum of times used for synchronization $t_{sync}$ added to the sum of waiting times $t_{wait}$ results in a time called overhead time $t_{over}$:

$$t_{over} = \sum_k t_{sync}^k + \sum_m t_{wait}^m \qquad \text{(Eq.2.4)}$$

with *k = 1,2,..., number of synchronizations* and with *m = 1,2,..., number of waits*.

**Computation Time**

The computation time $t_{comp}$ is the sum of all times during which the processes are in the program execution state.

**Total Execution Time**

The total execution time $t_{total}$ of a parallel program is defined as $t_{exec}$ in the beginning of this section. Using its portions, the total execution time can be rewritten as the sum of computation, communication, and overhead time, divided by the number of processors:

$$t_{total} = \frac{t_{comp} + t_{comm} + t_{over}}{P} \qquad \text{(Eq.2.5)}$$

with *P = number of processors*.

Since the portions $t_{comm}$, $t_{comp}$, and $t_{over}$ are always sums of times for all processors, the division by $P$ (number of processors) has to be carried out. Minimizing the total execution time of a parallel program is achieved by minimizing all portions of the sum. Communication, synchronization, and overhead times expressed by $t_{comm}$ and $t_{over}$. can be reduced by intelligent mapping of subtasks onto the processors. Computation time can be reduced by more efficient algorithms. Unfortunately, more efficient algorithms often lead to a more demanding communication structure in the parallel program, thus creating a trade-off between minimizing $t_{comm}$ and $t_{comp}$. This trade-off is also system dependent leading to different implementations for different systems. One goal of performance modeling techniques is to help the user to minimize the total execution time of a parallel program.

### 2.2.2. Performance Measures

The basic definition of performance is work divided by time which is also valid for computers.

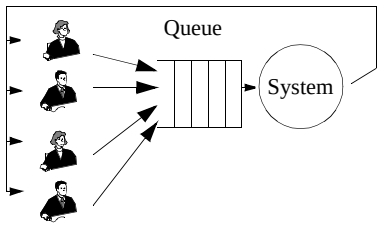



Fig. 2.5. Basic system model

#### Throughput

Throughput matches the basic definition of performance best. This performance measure describes in general the number of requests that are completed per time unit. Request is another word for workload, a request can be a transaction in terms of data base systems, it can be a job in terms of batch systems, it can simply be a user request in terms of interactive systems, or it can be the execution of an instruction or operation.The resulting performance measures are MIPS (millions of instructions per second), MFLOPS (millions of floating point operations per second), and TPS (transactions per second), or simply jobs per second. Other performance measures depending on the definition of work units are possible.

Performance measures are system and application specific. To determine the performance of an application for a system, the total execution time $t_{exec}$ has to be calculated or measured. The work (characterized by its units) divided by $t_{exec}$ (and if needed by $10^6$ for "Mega") results in the corresponding performance measure.

Unfortunately, marketing reasons have led to a widespread misuse of performance measures. A hardware provider is interested in offering a "high" performance system. Often, performance measures are calculated using the hardware description of the system without taking the compiler and operating systems into account. These theoretical "peak" performances are normally much higher than the performances for real applications.

#### Utilization

Utilization is a measure of how busy the resource system is when used by a specific workload. In terms of parallel systems, the utilization is also called efficiency (as defined later in this section).

#### Response Time

As outlined in Fig. 2.5., some users can share an interactive system, wait for the results, and, based on the results, initiate new jobs to the queue. The time a job spends in the system (from entering the queue until completion) is the job’s response time.

#### Queue Length

The queue length directly relates to the time a job must wait until it is executed by the system.The queue length is the average number of jobs in the queue or receiving service.

### 2.2.3. Relative Performance Measures

Since absolute performance measures such as MIPS, MFLOPS, or TPS for a computer architecture are theoretically calculated numbers that are not reached for real applications, it is often better to work with relative performance measures. By which factor the execution of an application can be accelerated using a multiprocessor is a very common question. System designers think about by which factor a new architecture is faster than its predecessor. These and similar questions can be answered using relative performance measures such as speedup, scaleup, efficiency, and reference numbers such as performance ratios.

#### Speedup

The goal of the application parallelization is the minimization of the total execution time $t_{exec}$ through the concurrent execution of the application on several processors. Gene Amdahl [Amd67] was one of the first who considered the problem of the maximum possible speedup. The definition of speedup is the relation of total execution time $t_{e,1}$ on one processor to total execution time $t_{e,P}$ on $P$ processors. In Amdahl's approach, an application is divided into a sequential and a parallel fraction. The execution time of the sequential fraction is constant no matter how many processors are involved. The execution time of the parallel fraction decreases proportionally to the number of allocated processors.

Amdahl's law describes the speedup behavior of parallel applications. It is derived from the following approach:

The execution time of an application $t_{e,1}$ is:

$$t_{e,1} = t_{es} + t_{ep,1} \qquad \text{(Eq.2.6)}$$

where $t_{es}$ is the execution time for the sequential fraction and $t_{ep,1}$ is the execution time of the parallel fraction on one processor. The sequential fraction *seq* and the parallel fraction *par* are defined as:

$$seq = \frac{t_{es}}{t_{ep,1} + t_{es}} \qquad \text{(Eq.2.7)}$$

$$par = \frac{t_{ep,1}}{t_{ep,1} + t_{es}} \qquad \text{(Eq.2.8)}$$

and thus, $$par = 1 - seq \qquad \text{(Eq.2.9)}$$

The total execution time $t_{e,P}$ for $P$ processors is:

$$t_{e,P} = t_{es} + \frac{t_{ep,1}}{P} \qquad \text{(Eq.2.10)}$$

Using $P$ processors results in the speedup factor $S_P$:

$$S_P = \frac{t_{e,1}}{t_{e,p}} = \frac{t_{es} + t_{ep,1}}{t_{es} + \frac{t_{ep,1}}{P}} \qquad \text{(Eq.2.11)}$$

Using (Eq.2.7), the expression from (Eq.2.11) can be transformed to:

$$S_P = \frac{1}{\frac{t_{es}}{t_{es}+t_{ep,1}} + \frac{t_{ep,1}}{t_{es}+t_{ep,1}} \cdot \frac{1}{P}}$$

which is equivalent to:

$$S_P = \frac{1}{seq + par \cdot \frac{1}{P}} \qquad \text{(Eq.2.12)}$$

This last equation is called *Amdahl's law*. It shows the heavy impact of the sequential fraction *seq* of a parallel program that is executed on *P* processors. If *seq* is only one percent (*seq* = 0.01) the maximum speedup $S_P$ is less than 100 (even for an unlimited number of processors *P*).

Since *seq* and *par* are constant, the $S_p$ function approaches 1/*seq* as the number of processors increases. Fig. 2.6. shows speedup functions for different values of *seq*. For *seq*=0 (and, thus, *par*=1), the ideal speedup function $S_p = P$ is derived from (Eq.2.11):

$$S_P = \frac{t_{e,1}}{t_{e,p}} = \frac{1}{0 + \frac{1}{P}} = P \qquad \text{(Eq.2.13)}$$

Fig. 2.6. also shows speedup functions for *seq*=0.01, *seq*=0.02 and *seq*=0.03. For example, 1000 processors result in a speedup of less than 33 if the sequential fraction of a parallel application *seq* is 3 percent.

One of the consequences of Amdahl's law is that there are few applications that offer enough parallelism (i.e., *seq* is very small) to result in close to linear speedup functions on massively parallel systems.

This is also demonstrated by the efficiency of parallel applications.

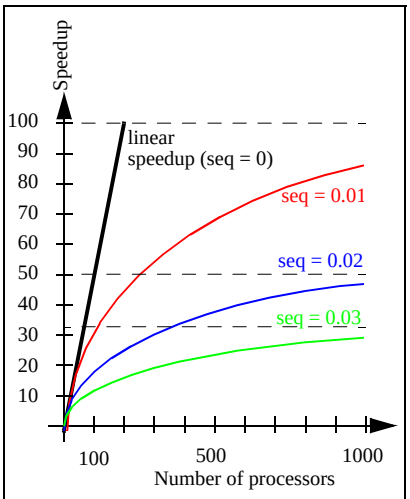



Fig. 2.6. Speedup functions for various seq using Amdahl's law.

**Efficiency**

The efficiency of a parallel application running on *P* processors is defined as:

$$\varepsilon = \frac{S_P}{P} \qquad \text{(Eq.2.14)}$$

Speedup is defined in (Eq.2.12). If *par* is replaced in this equation by (1-*seq*) using (Eq.2.9), ε can be written as:

$$\varepsilon = \frac{1}{1 + seq \cdot (P-1)}$$

Examples of efficiency functions over *seq* for various numbers of processors *P* are shown in Fig. 2.7. Considering the steep descent for a sequential fraction *seq* of 10% even for small numbers of processors, it becomes clear that

reasonable efficiency of parallel applications running on massively parallel systems is only possible for very small values of *seq*, i.e., *seq* << 0.01.

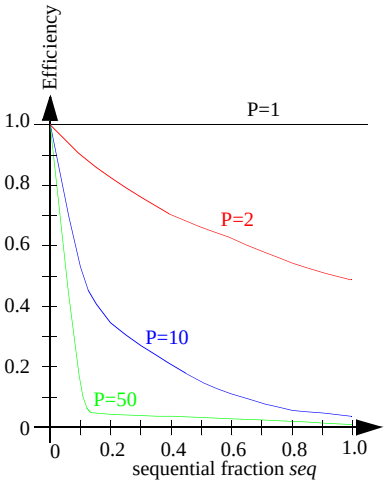



Fig. 2.7. Efficiency functions for various numbers of processors P

### Scaleup

One reaction to Amdahl's law is the assumption that only few applications exist that offer enough parallelism for efficient implementations on multiprocessors with large numbers of processors. A resulting question is whether it is worth investing much money and effort in building these machines. Despite that pessimistic outlook, several multiprocessors with up to thousands of processors have been built (nCUBE Hypercube, Connection Machine, INTEL Paragon, CRAY T3D, compare section 6.1.3.) and many applications with reasonable efficiency have been implemented on these machines.

Gustafson explains this seeming contradiction to Amdahl's law in his article "Reevaluating Amdahl's Law" [Gus88]. The derivation of Amdahl's law (Eq.2.6) to (Eq.2.13) is mathematically correct. The only problems are the pessimistic assumptions which are not necessarily true in real life. Amdahl assumes that the sequential fraction *seq* is constant and he does not examine the behavior of the sequential fraction for different problem sizes. These assumptions are different in Gustafson's approach. Gustafson points out that the sequential fraction does not necessarily grow linearly with the problem size. For many parallel applications *seq* grows slower or is even constant. This leads to the assumption observed in parallel applications that the total execution time of a parallel program remains constant for larger problem sizes if processors are added to execute it.

Fig. 2.8. shows the difference in the approaches of Amdahl and Gustafson. In the latter approach, the total execution time on a multiprocessor is the sum of the execution time for the sequential fraction *seq* and the execution time for the parallel fraction *par*. If only one processor is used to execute the program, the runtime of the parallel fraction *par* is multiplied by the number of processors *P*.

Gustafson defines the execution of a program on one processor as:

$$t_P = t_{es} + t_{ep} \cdot P \qquad \text{(Eq.2.15)}$$

Thus, the execution time on *P* processors is:

$$t_1 = t_{es} + t_{ep} \qquad \text{(Eq.2.16)}$$

and the resulting scaled speedup is:

$$SS_P = \frac{t_1}{t_p} = \frac{t_{es} + \frac{t_{ep}}{P}}{t_{es} + t_{ep}} \qquad \text{(Eq.2.17)}$$

Using (Eq.2.7):

$$SS_P = \frac{seq + par \cdot P}{seq + par} \qquad \text{(Eq.2.18)}$$

Using (Eq.2.9):

$$SS_P = P + (1 - P) \cdot seq \qquad \text{(Eq.2.19)}$$

In contradiction to Amdahl, the scaled speedup function is linear. For small sequential fractions *seq*, the function is close to the ideal speedup curve. Gustafson's intention is not to disprove Amdahl's law but to show that there are real applications that offer enough parallelism for large problem sizes to run efficiently on massively parallel systems. It also shows that it does not make good sense to use large multiprocessor systems for the solution of small problems.

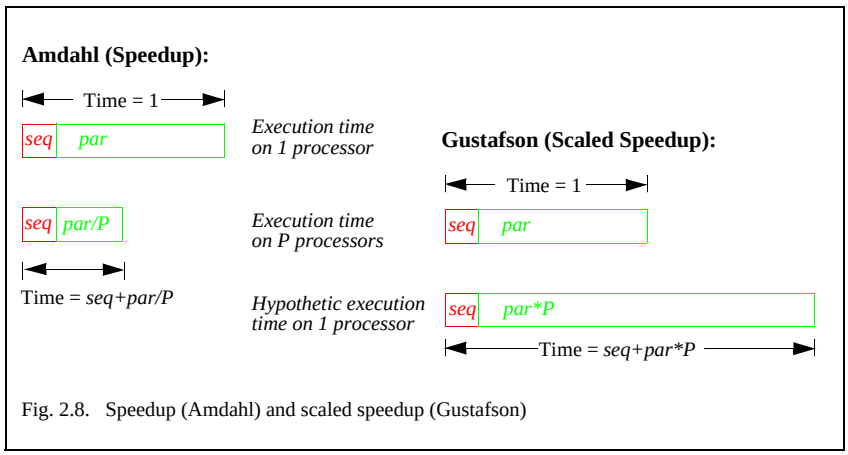



Fig. 2.8. Speedup (Amdahl) and scaled speedup (Gustafson)

**Reference Measures**

In the late 1980's several major hardware providers (IBM, SUN, HP, SIEMENS, and others) founded an institution with the goal of devising a fair performance evaluation of computer systems. To reach its goal they collected real applications and used them for the performance evaluation. The consortium was called SPEC (System Performance Evaluation Cooperative). A list with the 10 applications of the release 1 of the SPEC consortium is shown in Tab. 2.1.

These applications are execution time intensive and the influence of operating system calls during execution is negligible. The calculation of the so called SPECmark for a target system takes four steps:

- Determination of total execution time for every program *i* on target system *z* results in $t_{iz}$.
- Determination of total execution time for every program *i* on reference system *r* (VAX 11/780) results in $t_{ir}$.
- The relation of $t_{ir}$ to $t_{iz}$ is called SPECratio $R_{izr}$:

$$R_{izr} = \frac{t_{ir}}{t_{iz}}$$

- Calculation of geometric mean $\overline{R}_{g\,zr}$ for the 10 test programs. $\overline{R}_{g\,zr}$ is called SPECmark.

Speedup and scaleup are important relative measures for multiprocessor system whereas SPECmarks are mainly used to compare monoprocessor systems. A more detailed description of the SPEC Benchmarks can be found in section 3.1.

| **Abbreviation** | **Workload** | **Program** | **Data** |
|---|---|---|---|
| gcc | GNU C-Compiler | C | int |
| espresso | PLA-Simulator | C | int |
| spice2g6 | Analog circuit simulation | Fortran | float |
| doduc | Monte Carlo Simulation | Fortran | float |
| nasa7 | Collection of numerical "kernels" | Fortran | float |
| li | LISP Interpreter | C | int |
| eqntott | Minimization of boolean functions | C | int |
| matrix300 | Several matrix multiplications | Fortran | float |
| fpppp | Solution of the Maxwell Equations | Fortran | float |
| tomcatv | Network computations, strongly vectorized | Fortran | float |

Tab. 2.1. SPEC CPU benchmarks (Release 1)

### 2.2.4. System Under Test

There are several approaches to the evaluation of systems, depending on the chosen level of abstraction. Although there is a continuum of possible views, two examples of different abstraction levels are illustrated in Fig. 2.9.

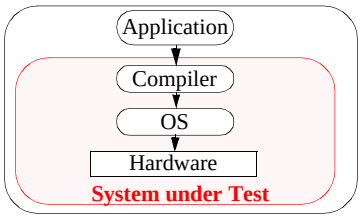
For the programmer of a high level application, the system under test (SUT) includes several components such as the compiler, the operating system, and the underlying hardware. In a multiprocessor environment, the interconnection network is also part of the SUT. The programmer might be interested in several performance features including:

- response time,
- elapsed time,
- resource utilization,
- communication patterns,
- concurrency profile, and
- space-time-diagram.

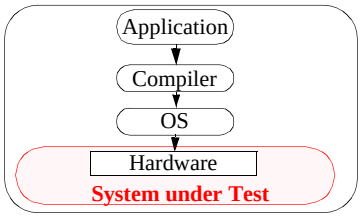
A different view of the same computer system is shown in Fig. 2.9.b. A hardware developer is normally less interested in the performance of the compiler or the operating system.Thus, the benchmarks of most use are specifically designed for the evaluation of certain hardware components of interest. The features of most interest to the hardware developer include:

- native MIPS,
- cache hit rate,
- bus utilization, and
- memory access times.

The system under test considered throughout this text is the one of the application programmer, Fig. 2.9.a. The compiler, the operating system, and the underlying hardware will be regarded as a black box. The instrumentation results in data for the single node performance, the communication behavior, and the overall performance. Looking at the amount and speed of communication at the nodes, potential and existing bottlenecks in the interconnection network are identified.




a. SUT for application programmer




b. SUT for hardware developer

Fig. 2.9. SUT depending on the level of abstraction

The methods for performance evaluation of monoprocessor systems and the methods for performance evaluation of multiprocessor systems mainly differ in the way to determine or define the workload for the performance tests and in the way the tests are realized on the target system. Three different approaches are used:

- The target system is a real system and the workload is executed on the system. This method is called benchmarking.
- The properties of the target system are captured by a simulator. The SUT is defined as in Fig. 2.9., where the hardware is replaced by a simulator in this figure. Since modern microprocessors are very complex, the simulator runs are very time consuming. Depending on the complexity of the hardware and the simulators, slowdowns with a factor from 100 to $10^6$ or more are possible. This means that to simulate one second of execution time, the simulator might need more than one week.
- This slowdown leads to a third approach where the simulator is replaced by a model for the hardware and workload. The accuracy, and thus, complexity of the hardware model depends on the desired accuracy of the results.

For all three approaches the workload can be either real applications or models with the properties of the desired applications.

### 2.2.5. Evaluation Triangle

Generally speaking, any kind of evaluation can be described by the evaluation triangle as shown in Fig. 2.10. An evaluation requires an experiment which runs a *workload* on a specific platform within a given environment *(SUT)*. The experiment is subject to an *observation* producing results which possibly lead to a redesign of the workload and/or the target system. In the next two sections of this text, the experiment is benchmarking for monoprocessors and multiprocessors. The workloads are synthetic (low level, kernels) and real (applications). The SUT is a real machine.

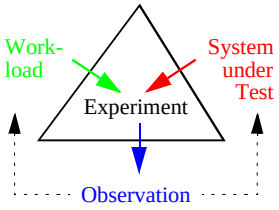
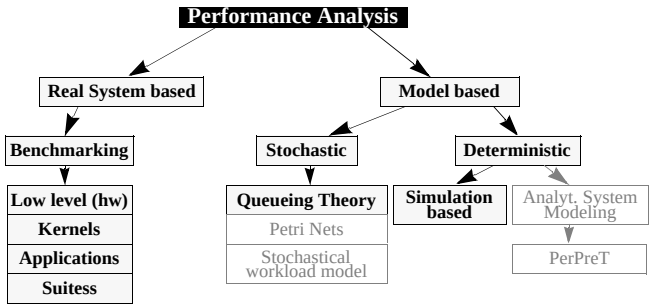



Fig. 2.10. Evaluation triangle




# 3. Performance Analysis of Monoprocessors

The first part of this section describes the generation and definition of workloads for performance tests and some examples for benchmarking with synthetic workloads and applications. The second part of this section presents an important approach for performance modeling of monoprocessor systems, namely, queueing networks. This general stochastic modeling technique, its graphical system and workload representation, the derivation of the steady-state diagram, and the solution of the linear system of equations for birth-death models is explained.

Benchmarking consists of two steps. First a workload is selected. The user has the choice of predefined standard workloads as presented in the next section or the workload is specifically designed to meet the user's needs or stress the system's bottlenecks. The definition of the workload also includes its parameters (e.g., problem size, compiler options, operating system, etc.). The second step of benchmarking is the implementation and execution of the workload on the target system. The execution time is used to calculate the results of the benchmark. These results are either performance measures like MFLOPS (mega floating point operations per second), MIPS (mega instructions per second), TPS (transactions per second) or relative performance measures compared to a reference system.

## 3.1. Benchmarking

### 3.1.1. Synthetic Workloads

The goals of benchmarking using synthetic workloads are to simulate the system load caused by real applications. One possible approach to derive an appropriate synthetic workload is to determine the program behavior in terms of memory access, memory traf-

fic, memory volume, number of instructions, input/output requirements, and similar parameters. The result is a test program which reflects a profile for the workload. This test program is used as the desired synthetic workload.

**Dhrystone:**

Possibly the best known synthetic workload is the Dhrystone program written by R. Weicker [Wei88]. This program was developed using the programming language ADA and later implemented in C by R. Richardson. The main focus of Dhrystone is an evaluation of the CPU performance. The quality of the compiler, and in particular, the code generation and optimization also plays an important role.

The program tries to be balanced with respect to the three aspects:

- statement type,
- operand type, and
- operand locality.

The average number of parameters in procedure or function calls is 1.82 (not counting function values as implicit parameters).

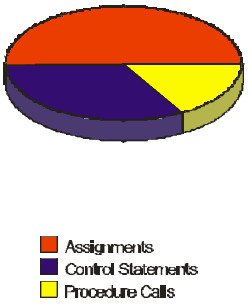



Fig. 3.1. Dhrystone (version 2.1) distribution of statements

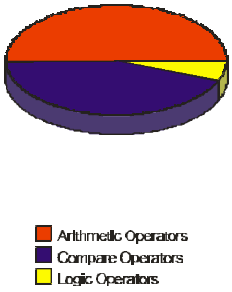



Fig. 3.2. Dhrystone (version 2.1) distribution of operators

The Dhrystone benchmark program contains 103 dynamically executed statements of a high level program in a distribution considered representative (compare Fig. 3.1.):

- assignments 52 (51.0%)
- control statements 33 (32.4%)
- procedures, function calls 17 (16.7%)

The distribution of the 63 executed operations per dhrystone loop is (compare Fig. 3.3.):

- **Arithmetic operators**, total 32 (50.8%):
  - \+ 21 (33.3%)
  - \- 7 (11.1%)
  - * 3 (4.8%)
  - / 1 (1.6%)

- **Compare operators**, total 27 (42.8%):

| | | |
|---|---|---|
| == | 9 | (14.3%) |
| != | 4 | (6.3%) |
| > | 1 | (1.6%) |
| < | 3 | (4.8%) |
| >= | 1 | (1.6%) |
| <= | 9 | (14.3%) |

- **Logic operators**, total 4 (6.3%):

| | | |
|---|---|---|
| && | 1 | (1.6%) |
| \|\| | 1 | (1.6%) |
| ! | 2 | (3.2%) |

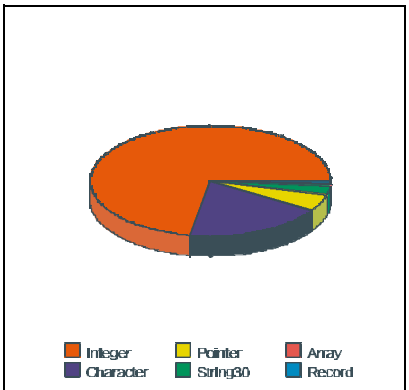



Fig. 3.3. Dhrystone (version 2.1) distribution of operands with respect to types

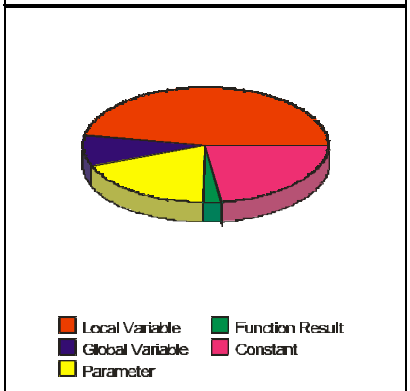



Fig. 3.4. Dhrystone (version 2.1) distribution of operands with respect to locality

The distribution of the 242 operands (counted once per operand reference) with respect to their type is (compare Fig. 3.3.):

| | | |
|---|---|---|
| Integer | 175 | (72.3%) |
| Character | 45 | (18.6%) |
| Pointer | 12 | (5.0%) |
| String30 | 6 | (2.5%) |
| Array | 2 | (0.8%) |
| Record | 2 | (0.8%) |

The distribution of the 242 operands (counted once per operand reference) with respect to their locality is (compare Fig. 3.4.):

| | | |
|---|---|---|
| Local Variable | 114 | (47.1%) |
| Global Variable | 22 | (9.1%) |
| Parameter | 45 | (18.6%) |
| Function Result | 6 | (2.5%) |
| Constant | 55 | (22.7%) |

The resulting synthetic workload intends to mimic typical program behavior of applications running on monoprocessor systems. The result of the benchmark test is the number of dhrystones per second. Since most of the operands are integer and 60% to 80% (depending on system and compiler) of the execution time is spend on integer operations, the dhrystones per second can also be interpreted as measure for the integer performance of a CPU. One problem of Dhrystone is the small size of the code which fits into caches of modern computers. Another problem appeared with the popularity and acceptance of Dhrystone. The hardware and compiler vendors installed special options in their compilers to "optimize" the performance results for a system executing Dhrystone.

The program does not compute anything meaningful, but it is syntactically and semantically correct. Semantically correct means that all variables have a value assigned to them before they are used as a source operand. As a result, the Dhrystone benchmark program outputs the microseconds for one run through Dhrystone and the corresponding Dhrystones per second.

**Dhrystone Results:**

The Dhrystone C Programs (dhry.shar), and the latest table of results (dhry.tbl) are available via anonymous *ftp* from *ftp.nosc.mil* in directory *pub/aburto*. Using the *GNU C*-Compiler (gcc, g++ - GNU project C and C++ Compiler v2.7) with optimization turned off, the following results for SUN systems (running SOLARIS 2.5) were obtained for the Sun Workstations of the Institut für Rechnerstrukturen und Betriebssysteme of the University of Hannover (compare Fig. 3.5.):

***SUN SPARC II, 40 MHz***
One run through Dhrystone: 78.6 µs
Dhrystones per Second: 12717.8

***SUN SPARC 10, 33 MHz***
One run through Dhrystone: 37.6 µs
Dhrystones per Second: 26574.5

***SUN ULTRA 1, 143 MHz***
One run through Dhrystone: 14.6 µs
Dhrystones per Second: 68642.0

***SUN ULTRA 2, 168 MHz***
One run through Dhrystone: 12.4 µs
Dhrystones per Second: 80829.9

.

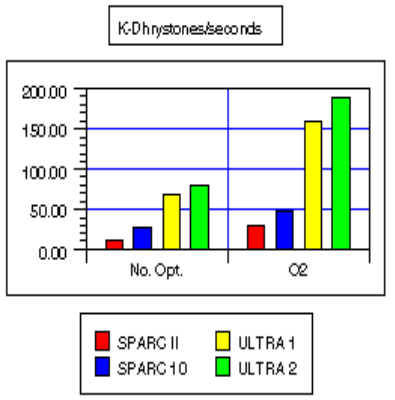



Fig. 3.5. Dhrystone (version 2.1) results for SUN microprocessors

To show the significant impact of the quality of the generated code, the compiler optimization (O2) was used (compare Fig. 3.5.):

***SUN SPARC II, 40 MHz***
One run through Dhrystone: 34.6 µs
Dhrystones per Second: 28861.4
Factor to no optimization 2.27

***SUN SPARC 10, 33 MHz***
One run through Dhrystone: 21.3 µs
Dhrystones per Second: 46915.3
Factor to no optimization 1.77

***SUN ULTRA 1, 143 MHz***
One run through Dhrystone: 6.2 µs
Dhrystones per Second: 160599.6
Factor to no optimization 2.34

***SUN ULTRA 2, 168 MHz***
One run through Dhrystone: 5.3 µs
Dhrystones per Second: 188798.0
Factor to no optimization 2.34

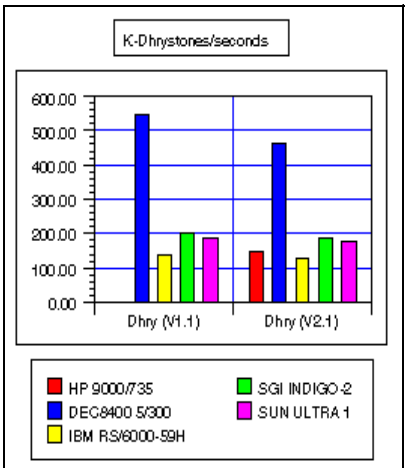



Fig. 3.6. Dhrystone results for workstations

**Dhrystone Results for different architectures:**

Fig. 3.6. and Tab. 3.1. show results of the two Dhrystone workloads (version 1.1 and ver-

sion 2.1) for some well known workstation architectures. The initial Dhrystone was programmed using ADA. The C and PASCAL versions helped to make it portable to all systems. The initial version 1.1 allowed compilers with good optimization to do some "illegal" optimizations. Since the code does not compute anything useful and some branches in the program are never executed, good optimizers simply deleted these branches. In version 2.1 of Dhrystone some code is added to these branches and prevents them from being eliminated. Furthermore, the overhead necessary for the measurement loop was hard to determine. To improve accuracy the loop check of the measurement loop became part of the benchmark itself. In summary, version 2.1 of Dhrystone is more realistic which results in slightly worse performance numbers compared with version 1.1.

The selection of the systems is not meant to show which company offers the best system. The selection criterion is simply the availability of results for different workloads. Throughout the rest of the section, the benchmarking results for the five systems of Fig. 3.6. are used. The results are taken from the Performance Data Base of the University of Tennessee at Knoxville[1].

1. The results can be found on the Internet at the URL: *http://performance.netlib.org/performance /html/PDStop.htm*l

| **Computer** | **Operating System** | **CPU** | **CPU MHz** | **Dhry V1.1** | **Dhry V2.1** |
|---|---|---|---|---|---|
| HP 9000/735 | HP-UX 9.01 | PA-RISC7150 | 99 | --- | 147 |
| DEC8400 5/300 | UNIX V4.0 | DEC21164 | 300 | 550 | 465 |
| IBM RS/6000-59H | AIX 3.2.5 | POWER-RISC | 66 | 140 | 128 |
| SGI INDIGO-2 | IRIX 6.2 | MIPS R10000 | 195 | 200 | 188 |
| SUN ULTRA 1 | Solaris 2.5 | Ultra SPARC | 167 | 190 | 179 |

Tab. 3.1. Dhrystone results for workstations

### Pros and Cons

The example above indicates the advantages and disadvantages of using a method like Dhrystone.

- ***Advantages:***
  - It offers a fast, portable evaluation program,
  - it is easy to use, and
  - the results are easy to compare.
- ***Disadvantages:***
  - Results can only be used as reference, illegal optimizations are possible,
  - a static workload without user specific parameters is used, and
  - it is unsuitable for multiprocessor systems.

**Whetstone:**

The Dhrystone workload was first published in 1984. It was designed to evaluate the integer performance of a system. Especially for scientific applications the floating point performance is of equal or greater value. The predecessor of Dhrystone, the Whetstone, was published in 1976 evaluates the floating point units besides the integer units. A run of the Whetstone workload is a set of loops requiring integer, boolean, and floating point arithmetic. Iterative calls to subroutines and inline transcendental functions are also made. With the benchmark coded in C, multiple Whetstone runs were performed on the same SUN workstations with SPARC microprocessors as in the previous section. The GNU C++ compiler (version 2.7) was used to compile the Whetstone program. Because the code does not support cooperative multitasking and requires less than 200 KB RAM (random access memory), even on RISC processors, benchmark results tend to be insensitive to system bus bottlenecks. The first series of experiments was run without optimization settings offered by the compiler (compare Fig. 3.5.):

***SUN SPARC II, 40 MHz***
One run through Whetstone: 108350 µs
(1 Million Whetstone instructions)
MWhetstones per Second: 92.3

***SUN SPARC 10, 33 MHz***
One run through Whetstone: 69580 µs
MWhetstones per Second: 143.7

***SUN ULTRA 1, 143 MHz***
One run through Whetstone: 36110 µs
MWhetstones per Second: 276.9

***SUN ULTRA 2, 168 MHz***
One run through Whetstone: 17740 µs
MWhetstones per Second: 563.7

The second series of experiments was run with the optimization parameter O2 turned on (compare Fig. 3.5.):

***SUN SPARC II, 40 MHz***
One run through Whetstone: 72620 µs
(1 Million Whetstone instructions)
MWhetstones per Second: 137.7
Factor to no optimization 1.49

***SUN SPARC 10, 33 MHz***
One run through Whetstone: 49230 µs
MWhetstones per Second: 203.1
Factor to no optimization 1.41

***SUN ULTRA 1, 143 MHz***
One run through Whetstone: 25240 µs
MWhetstones per Second: 396.2
Factor to no optimization 1.43

***SUN ULTRA 2, 168 MHz***
One run through Whetstone: 12430 µs
MWhetstones per Second: 804.5
Factor to no optimization 1.43

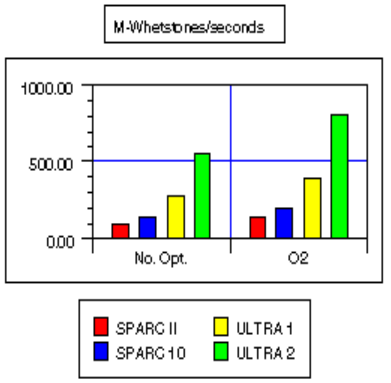



Fig. 3.7. Whetstone results for SUN microprocessors

The Pros and Cons for the Whetstone workload are the same as those for the Dhrystone workload.

### 3.1.2. Kernels

In contrast to synthetic workloads which do not compute anything useful but stress the hardware, kernels are the time consuming portions of real applications. In general, an application consists of four steps:

- input data,
- initialize,
- calculate, and
- output results.

The second (initialization) and the third (calculation) steps are used to extract kernel codes. Depending on the goals of performance evaluation, these kernels can be used directly. If needed, smaller fractions (e.g., some time consuming loops) can also be extracted.

**LINPACK:**

The collection of LINPACK application kernels collected by Jack Dongarra [Don79] is probably the best known benchmark workload. LINPACK consists of numerical subroutines for the solution of problems from linear algebra. In contrast to other more general workloads, these subroutines specifically show how well a system performs the solution of dense matrix problems. A linear system of equations $Ax=b$ is called dense if only few of the matrix coefficients of $A$ are zero.

| Computer | Operating System, Compiler | CPU MHz | LINP. 100 | LINP. 1000 | Theoretical |
|---|---|---|---|---|---|
| HP 900/0735 | +OP3 -Wl -aarchive -WP -nv -w ConvexMlib1.2 | 99 | 41 | 120 | 198 |
| DEC8400 5/300 | -inline=daxpy -ur=3 -fast -O5 -tune ev5 | 300 | 140 | 411 | 600 |
| IBM RS/6000-59H | v3.1.1 xlf -Pv -Wü -me -ew -O3 -qarch=pwrx -qtune=pwrx- qhot -qhsflt -qnosave | 66 | 132 | 230 | 264 |
| SGI ORIGIN 2000 | -n32 -mips4 -Ofast=ip27 -TENV:X=4 -LNO:blocking=off:ou_max=6:pf2=0 | 195 | 114 | 344 | 390 |
| SUN ULTRA 1 | -V -fast -native -dalign -libmil -xO4 -Qoption cg=20 -Qms_pipe+float_loop_ld=3D16 -onetrip -crossfile -xsafe=3Dmem | 167 | 70 | 237 | 333 |

Tab. 3.2. Linpack results for workstations

The LINPACK subroutines cause a high load for the floating point units compared to other operations (integer operations, memory access). Most of the floating point operations are executed by the so-called Basic Linear Algebra Subprograms (BLAS). The data structures for the BLAS routines are mainly one dimensional arrays. This results in an excellent test for vectorization units which typically speedup the computation.

The LINPACK workload is scalable with two standard workloads defined for matrix size (100 by 100) and matrix size (1000 by 1000). In principle, every other matrix size is possible (matrix size is a parameter), but mainly two different matrix sizes are used to make the results easier comparable.

Tab. 3.2. shows an example table of how the LINPACK results are listed in the Performance Data Base of the University of Tennessee at Knoxville[1]. Each row of the table contains the results for one system. The first column contains the name of the system, the second column the parameters that were used to execute the benchmark, the third column the frequency used for the microprocessor, the fourth column the results for LINPACK 100, the fifth column the results for LINPACK 1000, and the last column the theoretically possible performance.

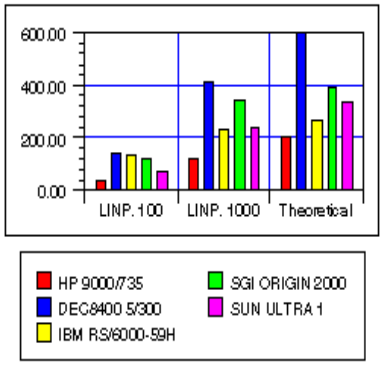



Fig. 3.8. Linpack results

Fig. 3.8. shows LINPACK results for the workstations. The execution time of the LINPACK workload is used to calculate the MFLOPS performance of the target system. The first part of the figure shows the results for matrix size (100 by 100), the second part shows the results for matrix size (1000 by 1000) and the third part shows the theoretically possible MFLOPS performance of the system. Since LINPACK is especially designed to stress the floating point units, the results for LINPACK 1000 should be close to the theoretically possible maximum performance of the system.

Besides the LINPACK kernels, there are a several other kernels used as benchmark workloads. Examples of kernels from the Performance Data Base of the University of Tennessee are MM (Matrix Multiplication), FFT (Fast Fourier Transforms with single and double precision), Queens (find all ways that 14 queens can be placed on a 14 by 14 chessboard), FIB (Fibonacci test), and others.

**Pros and Cons**

The example above indicate the advantages and disadvantages of using a kernel based workload like LINPACK for benchmarking:

- ***Advantages:***
  - It offers a fast, portable evaluation program,
  - it is easy to use,
  - it is scalable, and
  - the results are easy to compare.
- ***Disadvantages:***
  - Results can only be used as reference, illegal optimizations are possible,
  - a static workload without user specific parameters is used, and
  - operating system and compiler code generation have significant impact on the results.

1. The results can be found on the Internet at the URL: *http://performance.netlib.org/performance /html/PDStop.html*

### 3.1.3. Application Suites (SPEC)

The System Performance Evaluation Group (SPEC) was founded in 1988 by a small group of workstation vendors to overcome the lack of realistic standard workloads for benchmarking workstations.

Many of the major computer producers are represented in the SPEC organization. The main goal of the SPEC consortium is to collect programs based on real world applications in order to use them as benchmark workloads. Thus, the resulting SPEC Benchmark suite is a mixture of applications with floating point intensive (compare Tab. 3.4.) and integer intensive computations (compare Tab. 3.3.). SPEC used the SUN SPARC station 10/40 (40MHz SuperSPARC with no L2 cache) as a reference machine to normalize the performance metrics used in the SPEC95 suites. Each benchmark is run and measured on this machine to establish a reference time for that benchmark. These times are then used in the SPEC calculations.

| Benchmark | Ref. time (s) | Application area | Task |
|---|---|---|---|
| 099.go | 4600 | Games, AI | An internationally ranked go-playing program |
| 124.m88ksim | 1900 | Simulation | Simulates the Motorola 88100 processor running Dhrystone and a memory test |
| 126.gcc | 1700 | Compiler | Compiles source code to an optimized SPARC assembler code |
| 129.compress | 1800 | Data compression | Compresses large text file (approx. 16 MB) using adaptive Level-Ziv coding |
| 130.li | 1900 | Language interpreter | Xlisp Interpreter |
| 32.ijpeg | 12400 | Image processing | jpeg image compression with different parameters |
| 134.perl | 1900 | Shell interpreter | Perl Interpreter |
| 147.vortex | 2700 | Database | Object oriented data base |

Tab. 3.3. SPEC CINT95 (integer) benchmarks

#### Criteria for SPEC Benchmarks

In the process of selecting applications to be used as benchmarks, SPEC considered the following criteria:

- Portability to all SPEC hardware architectures (32- and 64-bit including Alpha,

Intel Architecture, PA-RISC, Rxx00, Sparc, etc.)

- Portability to various operating systems, particularly UNIX, NT and VMS.
- Benchmarks should not include measurable I/O.
- Benchmarks should not include networking or graphics.
- Benchmarks should run in 64MB RAM without swapping. (SPEC is assuming this will be a minimal memory requirement for the life of SPEC95; and the emphasis is on compute-intensive performance and not disk activity).
- Benchmarks should run at least five minutes on a DEC 200MHz Alpha system. Benchmarks should not spend more than five percent of time in non-SPEC provided code.

The user of the SPEC suite receives the source code to compile it for the target system. The user is allowed to tune the target systems to obtain the best possible results. The workload generated by the code is useful in evaluating the processor, the memory hierarchy, and the compiler of the target system. Unfortunately, all components are evaluated as a whole and the result is a single number. Thus, it is impossible to evaluate the individual components using the SPEC benchmark suite. Using source code leads to comparable results for the systems. One problem known from previous benchmarks is the danger of special compiler options in order to improve results for SPEC benchmark tests. The more important a benchmark gets, the bigger the danger for manipulations is. Since all results have to be published with details of the compiler and operating system used, the test results should be reproducible and the danger of such manipulations is minimized.

| **Benchmark** | **Ref. time (s)** | **Application area** | **Task** |
|---|---|---|---|
| 101.tomcatv | 3700 | Fluid dynamics | Vectorized mesh generation |
| 102.swim | 8600 | Weather prediction | Shallow water equations |
| 103.su2cor | 1400 | Quantum physics | Monte-Carlo method |
| 104.hydro2d | 2400 | Fluid dynamics | Navier Stokes equations |
| 107.mgrid | 2500 | Electromagnetism | 3D potential field |
| 110.applu | 2200 | Fluid dynamics | Partial differential equations |
| 125.turb3d | 4100 | Simulation | Turbulence modeling |
| 141.apsi | 2100 | Weather prediction | Weather and climate modeling |
| 145.fpppp | 9600 | Chemistry | Gaussian series of quantum chemistry benchmarks |
| 146.wav | 3000 | Electromagnetism | Maxwell equations |

Tab. 3.4. SPEC CFP95 (floating point) benchmarks

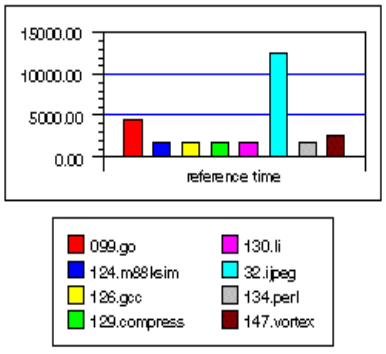



Fig. 3.9. SPEC CINT95 (integer) benchmarks (reference times in seconds)

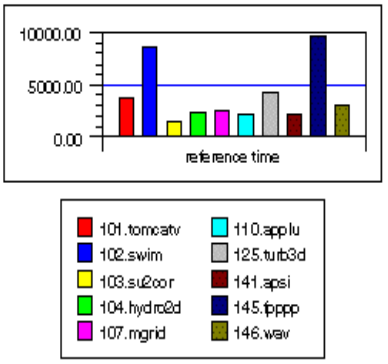



Fig. 3.10. SPEC CFP95 (float. point) benchmarks (reference times in seconds)

The SPEC organization provides workloads in form of C programs to evaluate the floating point performance and the integer performance of computer systems. Using these workloads results for the speed and throughput of several target systems are produced. The metric for speed is formed SPECint95 (integer tests) or SPECfp95 (floating point tests). The CINT95 and CFP95 suites can be used to measure and calculate the following metrics:

**CINT95:**

- SPECint95: The geometric mean of eight normalized ratios (one for each integer benchmark) when compiled with aggressive optimization for each benchmark.
- SPECint_base95: The geometric mean of eight normalized ratios (one for each integer benchmark) when compiled with conservative optimization for each benchmark.
- SPECint_rate95: The geometric mean of eight normalized throughput ratios (one for each integer benchmark) when compiled with aggressive optimization for each benchmark.
- SPECint_rate_base95: The geometric mean of eight normalized throughput ratios (one for each integer benchmark) when compiled with conservative optimization for each benchmark.

**CFP95:**

- SPECfp95: The geometric mean of 10 normalized ratios (one for each floating point benchmark) when compiled with aggressive optimization for each benchmark.
- SPECfp_base95: The geometric mean of 10 normalized ratios (one for each floating point benchmark) when compiled with conservative optimization for each benchmark.
- SPECfp_rate95: The geometric mean of 10 normalized throughput ratios (one for each floating point benchmark) when compiled with aggressive optimization for each benchmark.
- SPECfp_rate_base95: The geometric mean of 10 normalized throughput ratios (one for each floating point benchmark) when compiled with conservative optimization for each benchmark.

The difference between a "base" metric and a "non-base" metric is caused by compiler options. In order to provide comparisons across different computer hardware, SPEC had to provide the benchmarks as source code. Thus, in order to run the benchmarks, they must be compiled. There was agreement that the benchmarks should be compiled the way users compile programs. But how do users compile programs? On one side, people may experiment with many different compilers and compiler flags to achieve the best performance. On the other side, people may just compile with the basic options suggested by the compiler vendor. SPEC recognizes that they can not exactly match how everyone uses compilers, but two reference points are possible. The base metrics (i.e., "SPECint_base95") are required for all reported results and have set guidelines for compilation (i.e., the same flags must be used in the same order for all benchmarks). The non-base metrics (i.e., "SPECint95") are optional and have less strict requirement (i.e., different compiler options may be used on each benchmark.

A full description of the distinctions can be found in the SPEC95 Run and Reporting rules available with SPEC95[1].

1. The results can be viewed using the internet URL: *http://www.specbench.org*

| Computer | Operating System | CPU | CPU MHz | SPEC fp95 | SPEC int95 |
|---|---|---|---|---|---|
| HP 9000/735 | HP-UX 9.01 | PA-RISC7150 | 99 | 3.98 | 3.27 |
| DEC8400 5/300 | UNIX V4.0 | DEC21164 | 300 | 11.70 | 7.43 |
| IBM RS/6000-59H | AIX 3.2.5 | POWER-RISC | 66 | 8.75 | 3.10 |
| SGI INDIGO-2 | IRIX 6.2 | MIPS R10000 | 195 | 10.20 | 8.50 |
| SUN ULTRA 1 | Solaris 2.5 | Ultra SPARC | 167 | 8.45 | 5.58 |

Tab. 3.5. SPEC 95 results published by SPEC

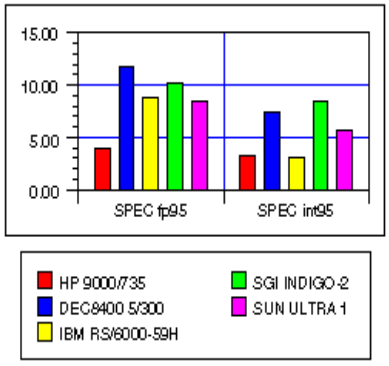



Fig. 3.11. SPECfp95 and SPECint95 results

The ratio for each of the benchmarks are calculated using a SPEC-determined reference

time and the run time of the benchmark. A metric to determine the throughput of a system is the SPECint_rate95 or the SPECfp_rate95.

The following steps are necessary to produce the SPECint95 and SPECfp95 numbers:

1. Implementation of the benchmarks from Tab. 3.3. or Tab. 3.4. on the target system and measurement of the execution times.
2. Calculation of the SPECratio for the single workloads:

*SPECratio for workload x = x.reference.time/x.execution.time,*

i.e., the metric SPECratio is a relative number for the execution time of the workload compared to a reference time for the workload.

3. After the SPECratio is determined for all workloads, the geometric means is calculated for the SPECratios of the integer workloads (=SPECint95) and the geometric means is calculated for the SPECratios of the floating point workloads (=SPECfp95).

Tab. 3.5. and Fig. 3.11. show SPECfp95 and SPECint95 results for the workstations published by SPEC[1].

The following steps are necessary to produce the SPECint95 and SPECfp95 rates:

1. For each workload, a so called SPECrate is calculated. The SPECrate is a function
   - of the number of copies of the program that were executed,
   - of the time that was needed to execute all copies, and
   - of a reference factor[2].
2. After the SPECrate is determined for all workloads, the geometric means is calculated for the integer benchmarks (=SPECint_rate95), and the geometric means is calculated for the floating point benchmarks (=SPECfp_rate95).

Fig. 3.12. and Tab. 3.6. show SPECfp_rate95 and SPECint_rate95 results for the workstations published by SPEC[3].

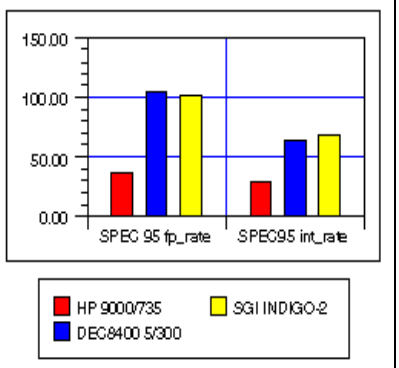



Fig. 3.12. SPEC95 rates

1. The results can be viewed using the internet URL: *http://www.specbench.org*
2. The formula for the exact determination of the SPECrates can be found using the internet URL: *http://www.specbench.org.*
3. The results can be viewed using the internet URL: *http://www.specbench.org*

| **Computer** | **Operating System** | **CPU** | **CPU MHz** | **SPEC95 fp_rate** | **SPEC95 int_rate** |
|---|---|---|---|---|---|
| HP 9000/735 | HP-UX 9.01 | PA-RISC7150 | 99 | 35,8 | 29,4 |
| DEC8400 5/300 | UNIX V4.0 | DEC21164 | 300 | 104 | 64,2 |
| SGI INDIGO-2 | IRIX 6.2 | MIPS R10000 | 195 | 102 | 68 |

Tab. 3.6. SPEC 95 rates published by SPEC

**Pros and Cons**

The examples above indicate the advantages and disadvantages of using an application based workload for benchmarking:

- ***Advantages:***
  - It offers fast, portable evaluation programs,
  - it is easy to use,
  - the results are easy to compare.
- ***Disadvantages:***
  - Results can only be used as reference, illegal optimizations are possible,
  - a static workload without user specific parameters is used,
  - it is not scalable, and
  - operating system and compiler code generation have significant impact on the results.
  - There is no obvious relation between SPEC results and architectural parameters.

### 3.1.4. Comparison of Benchmark Results

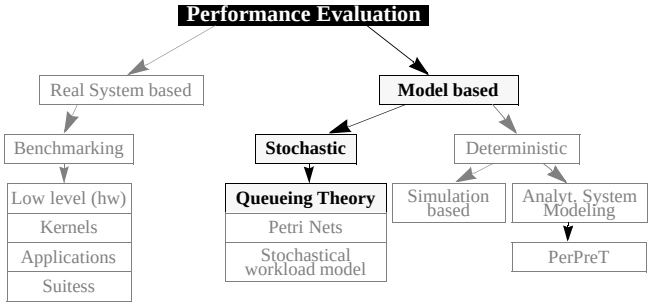
Tab. 3.7. illustrates that the benchmark results roughly describe the performance potential of the target systems. In general, a single number is not enough for a detailed performance description of a target system. The table contains the ranks of the workstations for the different benchmarks. Since none of the systems is the best for all benchmarks, the question which is the best system for a particular need may be asked. Even if all systems were equally priced, a user would not automatically buy the DEC system, if the application is run with a more integer oriented profile.

Most benchmark tests are useful tools to compare different computing systems. Since the result often is a single number, it is only useful for comparison. Question like "Do I really need this system?" or "Does my application run sufficiently fast on this system?" are generally not answered using "standard" benchmarks.

In summary, benchmarks are of minor importance for system designers and application developers with respect to the optimization of systems or applications. They evaluate existing systems, but they do not provide specific information to find, for example, system bottlenecks. The evaluation of a system under test as defined in Fig. 2.9.a (SUT includes compiler, operating system and hardware) makes it impossible to trace any performance problems. Nevertheless, benchmarking is important for purchasing decisions and for comparing and ranking computer systems.

| | | | | Ranking | | | |
|---|---|---|---|---|---|---|---|
| **Computer** | **Operating System** | **CPU** | **CPU MHz** | **N-MIPS V2.1** | **SPEC int95** | **SPEC fp95** | **LINP. 1000** |
| HP 9000/735 | HP-UX 9.01 | PA-RISC7150 | 99 | 4 | 4 | 5 | 5 |
| DEC8400 5/300 | UNIX V4.0 | DEC21164 | 300 | 1 | 2 | 1 | 1 |
| IBM RS/6000-59H | AIX 3.2.5 | POWER-RISC | 66 | 5 | 5 | 3 | 4 |
| SGI INDIGO-2 | IRIX 6.2 | MIPS R10000 | 195 | 2 | 1 | 2 | 2 |
| SUN ULTRA 1 | Solaris 2.5 | Ultra SPARC | 167 | 3 | 3 | 4 | 3 |

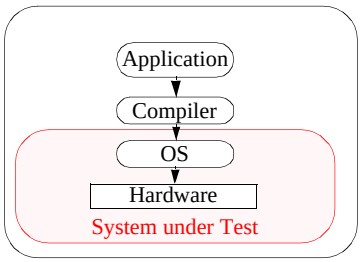
Tab. 3.7. Comparison of benchmark results for workstations




## 3.2. Modeling

In the previous section, real workloads are implemented on real machines to determine the performance of the target system in terms of executed instructions or operations per time unit. These benchmarks are necessary because the complexity of the hardware and the quality of the compiler generated codes make it impossible to predict the performance of a system by simply looking at the hardware. It is possible to predict peak performances (numbers selected by the vendors), but as the tests in the previous section show, the peak performance is generally quite different from that measured by real applications.

For system designers the task of performance evaluation is crucial, but in general the target system is still under design and measurements are not possible. In this case, the target system and the workload which is intended to run on that system have to be modeled. The system under test to be optimized is outlined in Fig. 3.13., it consists of hardware and some functionality of the operating system (such as scheduling, I/O).




SUT for hardware developer

Fig. 3.13. SUT depending on the level of abstraction

In [Men94] a model is defined as a representation of a system. Applied to computer systems, modeling is used for at least two purposes:

- functional modeling to evaluate the operation of the system, and
- performance modeling to predict performance measures of a system.

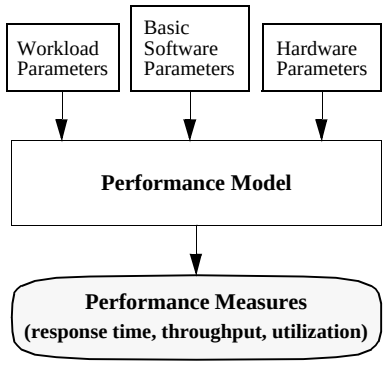



Fig. 3.14. Performance modeling

**Functional models to represent the operation of the system:**

As an example of functional modeling, Petri Nets can be used to prove the absence of deadlocks or starvation in concurrent systems [Pet81]. Thus, the focus of functional models is on the study of certain properties of the behavior of a system. A brief overview on modeling multiprocessors using Petri Net models is given in section 5.2. of this book.

**Performance models to predict performance measures of a system:**

As shown in Fig. 3.14., the workload, basic software, and hardware are represented through parameters. The performance model uses the parameters to predict performance measures such as response time, throughput, and utilization of the system. Workload parameters describe the load imposed on the target systems. They include the arrival rates of jobs and their demands on the target system resources such as execution time, memory requirements, and I/O behavior. The basic software parameters such as maximum degree of multiprogramming and CPU dispatching priority reflect the effect of the operating system on the system performance. Hardware parameters describe the expected performance of the hardware and include the for example processor speed, memory access time, disk latency and transfer rates, and local area network latency and transfer rates.

### 3.2.1. Queueing Networks

In computer systems the cooperation between hardware and operating system consists of jobs sharing system resources such as CPU, disks and other devices. Since generally one job can use the resource at any time, all other jobs wanting to use that resource wait in queues. As described in [Jai91] queueing theory helps in determining the time that jobs spend in various queues in the system.

To demonstrate the application of performance modeling techniques based on queueing networks an example is given. Suppose the manager of a computer center has the choice of two systems. The operating costs for system A are five cents per second. System A completes the average job in two seconds. For every completed job the computer center gets 25 cents. System B is twice as fast as system A completing the average job in one second, but it also costs more money to operate system B, which is eight cents per second. In average every five seconds a new job arrives at the queue, the job arrival time is assumed to be exponentially distributed, the job arrival rate $\lambda$ is 0.2 (one job every 5 seconds) for both systems. The queue can hold up to 4 jobs. Fig. 3.15. shows the two system alternatives. The service rate of system A $\mu_A$ is 0.5 (A needs 2 seconds to finish a job), the service rate $\mu_B$ is 1 (B needs one second to finish a job). This figure shows the basic building blocks of queueing networks, the so called *station*. A *service center* consists of a *queue*, a *service station*, and *a queueing discipline*. The service center is visited by customers. Modeling a computer system, a customer can represent different entities, such as memory requests, computer programs, individual processes, communication messages, jobs in an interactive or batch system. The service stations have associated the parameters of a service time distribution. The queueing discipline describes the sequence to process customers in the queue. Throughout the examples considered in this book, the queues have the FCFS (first come first serve) queueing discipline. Other examples for queueing disciplines would be round robin, processor sharing and disciplines based on priority schemes. The graphical representation of a service station in a queueing network is a circle, the graphical representation of a queue is a rectangle with on side missing and some vertical lines (compare Fig. 3.15.). The customers follow the directed arcs connecting the service stations and queues. If a customer has the choice of two arcs to follow next, routing probabilities have to be associated with each feasible branch. All examples in this text only use queueing networks in which the number of customers is fixed and all customers have the same routing probabilities and service requirements.

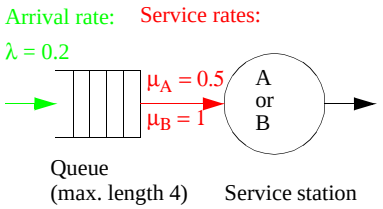



Fig. 3.15. Service Center

### 3.2.2. Basic Birth-Death Models

To answer the question whether the manager should buy system A or system B, the throughput, utilization and costs per job are now calculated using a queueing network. From the queueing network a finite state diagram can be derived (compare Fig. 3.16.). The steady state probabilities can be calculated using markovian analysis.

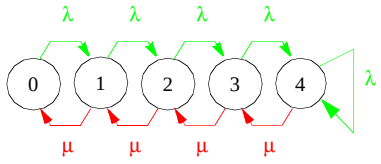



Fig. 3.16. State diagram

The circles of the steady state diagram contain the number of jobs in the queue, λ and μ are arrival and service rates, respectively. To model the behavior of the system *stochastic processes* [Jai91] are used. Such processes are useful in representing the state of queueing systems. The commonly used types of stochastic processes are:

- *Discrete-State, Continuous-State Processes*
  A process is called a *discrete-state* process if the number of values its state can take is countable. For example, the number of jobs a system can only take discrete values, therefore the function number of jobs over time is a discrete-state process which is also called a stochastic chain.
  A process is called a *continuous-state* process if the state can take any real values. For example, the function of the waiting time of a job is a continuous state process.

- *Markov Process*
  If the future states of a process are independent of the past and depend only on the present, the process is called a *Markov Process*. The Markov property makes a process easier to analyze since we do not have to remember the complete past trajectory. Knowing the present state of the process is sufficient. The processes are named after A. A. Markov, who defined and analyzed the in 1907. A discrete-state Markov process is called a *Markov chain*. The diagram in Fig. 3.16. is more formally known as a *Markov diagram*. Its special properties arise from the fact that it uses the exponential distribution to model service and arrival times. The exponential distribution has a so-called "memoryless" property which means that for any state the system can enter, the next state depends solely on the current state of the system. States visited previously to the current state and the amount of time spent in the current state or previous state have no bearing on the next transition. This allows time to be factored out of the analysis.

- *Birth-Death Processes*
  The discrete-space Markov processes in which the transitions are restricted to neighboring states only are *called birth-death processes*. For these processes, it is possible to represent states by integers such that a process in state *n* can change only to state *n+1* or *n-1* (compare Fig. 3.16.).

The average number of times an arc of Fig. 3.16. is traversed per standard time unit is thought of as the amount of flow along that arc. It is obvious that the amount of flow along an arc depends upon being in the state

from which that arc departs. For a steady state system, the flow along an arc is the product of the probability of being in the state from which the arc departs and the rate associated with the arc. For steady state diagrams it is clear that the amount of flow into a state must be equal to the amount of flow out of that state. If the throughput of the system has to be calculated the sum of the throughputs of the individual states has to be calculated. The throughput of the individual states is the flow along the arcs labelled with μ.

The problem now is to calculate the steady state probabilities $P_0, P_1,...,P_4$, with $P_i$ = *probability to be in state i*. Using the following facts, it is possible to form a system of linear equations:

- Flow along an arc is the product of the probability of being in the state which the arc departs and the rate associated with the arc.
- The sum of all flows into a state is equal to the sum of all the flows out of that state.
- The sum over all $P_i$ $(i = 0, 1,...,4)$ is 1, since at any time the system must be in one of the four states.
- λ and μ are known values.

The resulting linear system of equations is:

$$\mu P_1 = \lambda P_0 \qquad \text{(Eq.3.1)}$$

$$\lambda P_0 + \mu P_2 = \lambda P_1 + \mu P_1 \qquad \text{(Eq.3.2)}$$

$$\lambda P_1 + \mu P_3 = \lambda P_2 + \mu P_2 \qquad \text{(Eq.3.3)}$$

$$\lambda P_2 + \mu P_4 = \lambda P_3 + \mu P_3 \qquad \text{(Eq.3.4)}$$

$$\lambda P_3 + \lambda P_4 = \lambda P_4 + \mu P_4 \qquad \text{(Eq.3.5)}$$

These are the global balance equations which yield the steady state solution. Since the sum of all probabilities is one, an additional equation is available:

$$P_0 + P_1 + P_2 + P_3 + P_4 = 1 \qquad \text{(Eq.3.6)}$$

For five unknowns $(P_0,...,P_4)$, six equations are available ((Eq.3.1),..., (Eq.3.6)), but one of the equations is redundant. By using substitution of variables, the system can be solved in terms of $P_0$:

(Eq.3.1) =>

$$P_1 = \frac{\lambda}{\mu} P_0 \qquad \text{(Eq.3.7)}$$

(Eq.3.2), (Eq.3.7)=>

$$P_2 = \frac{\lambda^2}{\mu^2} P_0 \qquad \text{(Eq.3.8)}$$

(Eq.3.3), (Eq.3.7), (Eq.3.8)=>

$$P_3 = \frac{\lambda^3}{\mu^3} P_0 \qquad \text{(Eq.3.9)}$$

(Eq.3.4), (Eq.3.8), (Eq.3.9)=>

$$P_4 = \frac{\lambda^4}{\mu^4} P_0 \qquad \text{(Eq.3.10)}$$

Using (Eq.3.7),...,(Eq.3.10) the equation (Eq.3.6) can be rewritten as:

$$P_o + \frac{\lambda}{\mu} P_0 + \frac{\lambda^2}{\mu^2} P_0 + \frac{\lambda^3}{\mu^3} P_0 + \frac{\lambda^4}{\mu^4} P_0 = 1 \qquad \text{(Eq.3.11)}$$

Now, the solution to $P_0$ is given:

$$P_0 = \frac{1}{1 + \frac{\lambda}{\mu} + \frac{\lambda^2}{\mu^2} + \frac{\lambda^3}{\mu^3} + \frac{\lambda^4}{\mu^4}} \quad \text{(Eq.3.12)}$$

With the input values $\lambda$ and $\mu$ all answers to the performance questions can be given. Under the given circumstances (arrival rate, service rate, system costs per second), the manager should not buy any of the two systems, because they both result in negative operational costs. Given the results of Tab. 3.8. the operational costs of system A should be less than 4.9212 cents per second and the operational costs for system B should be less than 4.9936 cents per second to make money for the given workload.

| **Input** | **System A** | **System B** |
|---|---|---|
| $\lambda$ (arrival rate) | 1/5 | 1/5 |
| $\mu$ (service rate) | 1/2 | 1 |
| **Results** | | |
| Steady State Prob: | | |
| $P_o$ | 625/1031 | 625/781 |
| $P_1$ | 250/1031 | 125/781 |
| $P_2$ | 100/1031 | 25/781 |
| $P_3$ | 40/1031 | 5/781 |
| $P_4$ | 16/1031 | 1/781 |
| Utilization U | $1-P_0$ = 406/1031 | $1-P_0$ = 156/781 |
| Throughput X | U*$\mu$ = 203/1031 | U*$\mu$ = 156/781 |
| Revenue per sec. | X*0.25 = 0.049212 | X*0.25 = 0.049936 |
| Cost per sec. | 0.05 | 0.08 |
| **Profit per sec.** | **-0.000778** | **-0.030064** |

Tab. 3.8. Comparison of systems A and B

What happens if a higher load is expected for both systems. Assume the arrival rate goes up from 1/5 (one job every five seconds) to 1 (one job every second). Since system B is faster than system A, one would intuitively expect that B would outperform A in terms of profit. Tab. 3.9. shows the results for the new workload, which makes both systems more profitable. Now, System B clearly outperforms A in terms of profit per second.

But this is only half the truth, the additional problem with system A is that the average number of jobs in the queue is much higher than the average number of jobs in the queue for system B, thus the waiting time for each job and the response time of system A is much higher than the corresponding times for system B.

| **Input** | **System A** | **System B** |
|---|---|---|
| $\lambda$ (arrival rate) | 1 | 1 |
| $\mu$ (service rate) | 1/2 | 1 |
| **Results** | | |
| Steady State Prob: | | |
| $P_o$ | 1/31 | 1/5 |
| $P_1$ | 2/31 | 1/5 |
| $P_2$ | 4/31 | 1/5 |
| $P_3$ | 8/31 | 1/5 |
| $P_4$ | 16/31 | 1/5 |
| Utilization U | $1-P_0$ = 30/31 | $1-P_0$ = 4/5 |
| Throughput X | U*$\mu$ = 15/31 | U*$\mu$ = 4/5 |
| Revenue per sec. | X*0.25 = 0.120968 | X*0.25 = 0.2 |
| Cost per sec. | 0.05 | 0.08 |
| **Profit per sec.** | **0.070968** | **0.12** |

Tab. 3.9. Comparison of systems A and B with new arrival rates

The average number of jobs in the system can be calculated as:

$$\varnothing Jobs = P_1 \cdot 1 + 2 \cdot P_2 + 3 \cdot P_3 + 4 \cdot P_4$$

(Eq.3.13)

For system A ($\lambda = 1$, $\mu = 1/2$) the average number of jobs in the system is:
2/31 + 8/31 + 24/31 + 48/31 = 2.645

for system B ($\lambda = 1/3$, $\mu = 1$) the average number of jobs in the system is:
1/5 + 2/5 + 3/5 + 4/5 = 2

Assume, the systems are interactive systems, then the time for system A is definitely too high. This will lead to unsatisfied customers, thus, decrease the arrival rate and profit.

The intentions of this example were:

- To show the basic technique of getting a markovian model for a system.
- To show how the linear system of equations can be built using the steady state diagram.
- To show which kind of performance questions can be answered by modeling techniques.
- To show that the model is very sensitive with respect to its parameters. Thus, the determination of the parameters should be carefully done.

In [Men94] an approach to build a validated system model is described (compare Fig. 3.17.). The initial step is to construct a suitable performance model. Then, the necessary parameters must be obtained and the necessary assumptions stated. Using the parameters, the model can be constructed and solved. By solving it is meant to obtain performance measures through "manipulation" of the parameters. For this so called model solution step, many techniques have been suggested. Exact solutions of approximate models or approximate solutions of approximate models are possible. Since the approximations always involve assumptions, the results should be validated by experiments. The result of this validation step is either that the model is acceptable or unacceptable. If found unacceptable, the critical task is now to locate the model errors and build a new so called calibrated model. The calibrated model has to be validated, and if found unacceptable, the search for model errors begins again. Once a successfully validated baseline model is found, it can be used to build the prediction model. The prediction model is designed to find answers to the "what if" questions, for example if the baseline model describes a device, it can be used to build the prediction model of the upgraded device. The question "what if the device is upgraded?" can now be answered. Since the baseline model is changed in order to build the prediction model, the validation procedure has to be repeated if possible.

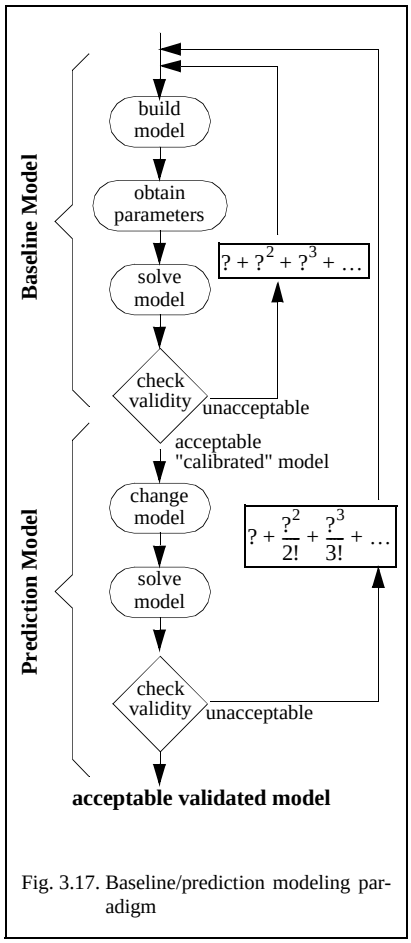



Fig. 3.17. Baseline/prediction modeling paradigm

### 3.2.3. Birth-Death Models with Infinite Number of States

The queue of example system A is now supposed to be unlimited (compare Fig. 3.18.). Using this assumption, is it possible to generate formulae to be able to calculate the systems performance characteristics such as processors utilization, response time, throughput, average queue length?

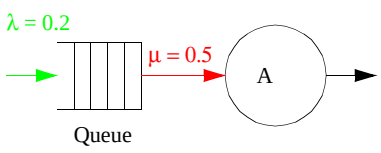



Fig. 3.18. System model (unlimited queue length)

Fig. 3.19. shows the resulting state space diagram:

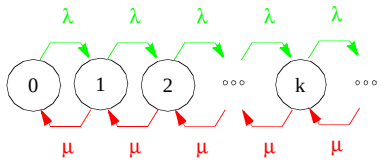



Fig. 3.19. State diagram

In this diagram the system "flows" from state to state depending on the system parameters λ and μ. Suppose, the queue holds *k* jobs (i.e. the system is in state *k*), there is one job in execution at rate μ and there are *(k-1)* jobs waiting in the queue. The state *k* is only left, when a job is finished (with rate μ), or when a new job arrives at the queue (with rate λ). At any point in time at maximum one of these events can happen. Thus, from state *k* it is only possible to go to state *(k-1)* or *(k+1)*. Such a system is also called birth-death system. As before, the flow balance equations can now be formulated (using the basic assumption flow in = flow out):

$$\mu P_1 = \lambda P_0 \qquad \text{(Eq.3.14)}$$

$$\lambda P_0 + \mu P_2 = \lambda P_1 + \mu P_1 \qquad \text{(Eq.3.15)}$$

$$\vdots$$

$$\lambda P_{k-1} + \mu P_{k+1} = \lambda P_k + \mu P_k \qquad \text{(Eq.3.16)}$$

$$\vdots$$

Using these linear equations, the probabilities $P_i$ *(i=0,1,2,...)* can be determined. $P_i$ is the probability that *(i-1)* jobs are in the queue, one job is executed. $P_0$ is the probability that the system is idle (i.e. there are no jobs to be executed). As before, all $P_i$s can be represented in terms of $P_0$:

(Eq.3.14) =>

$$P_1 = \frac{\lambda}{\mu} P_0 \qquad \text{(Eq.3.17)}$$

(Eq.3.14),(Eq.3.17)=>

$$P_2 = \frac{\lambda}{\mu}P_1 = \frac{\lambda^2}{\mu^2}P_0 \qquad \text{(Eq.3.18)}$$

This scheme can be applied to all $P_i$s:

$$P_k = \frac{\lambda}{\mu}P_{k-1} = \frac{\lambda^2}{\mu^2}P_{k-2} = \ldots = \frac{\lambda^k}{\mu^k}P_0 \qquad \text{(Eq.3.19)}$$

After all $P_i$s are expressed in terms of $P_0$, the fact that the sum of all probabilities must be one is used to calculate $P_0$:

$$P_o + \frac{\lambda}{\mu}P_0 + \frac{\lambda^2}{\mu^2}P_0 + \ldots = 1 \qquad \text{(Eq.3.20)}$$

$$P_o \cdot \left(1 + \frac{\lambda}{\mu} + \frac{\lambda^2}{\mu^2} + \ldots\right) = 1$$

$$P_o = \left[1 + \frac{\lambda}{\mu} + \frac{\lambda^2}{\mu^2} + \ldots\right]^{-1}$$

$$P_o = \left[\sum_{i=0}^{\infty}\left(\frac{\lambda}{\mu}\right)^i\right]^{-1} = \left[\frac{1}{1-\frac{\lambda}{\mu}}\right]^{-1} = 1 - \frac{\lambda}{\mu} \qquad \text{(Eq.3.21)}$$

Using (Eq.3.19) and (Eq.3.21) the expression for the steady state probability to be in state *k* is:

$$P_k = \left(\frac{\lambda}{\mu}\right)^k \cdot \left(1 - \frac{\lambda}{\mu}\right) \qquad \text{(Eq.3.22)}$$

Now the probability of each $P_i$ *(i=0,1,2,...)* can easily be calculated, but more important, these probabilities can be used for more useful performance measures. The system is busy (utilized), when at least one job is present. Thus, the utilization of the system is:

$$utilization = 1 - P_o = \frac{\lambda}{\mu} \qquad \text{(Eq.3.23)}$$

For system A ($\lambda$=1/5, $\mu$=1/2) with unlimited queue from the previous example, the utilization would be 0.4, for system B ($\lambda$=1/5, $\mu$=1) with unlimited queue the utilization would be 0.2, which is not surprising since system B is twice as fast as system A.

The throughput is defined as the rate at which jobs leave the system after being executed. Since the flow in = flow out assumption is also valid for the total system, it is not surprising that the throughput is identical with the arrival rate $\lambda$.

$$throughput = \lambda \qquad \text{(Eq.3.24)}$$

Thus, the throughput of both systems A and B is 0.2.

The mean queue length (average number of jobs in the system) is also an important performance measure. Using a state-by-state

enumeration, the mean queue length (*mql*) can be calculated as:

$$mql = 0 \cdot P_o + 1 \cdot P_1 + 2 \cdot P_2 + \ldots \quad \text{(Eq.3.25)}$$

$$mql = \sum_{k=1}^{\infty} kP_k = \sum_{k=1}^{\infty} k\left(\frac{\lambda}{\mu}\right)^k \cdot \left(1-\frac{\lambda}{\mu}\right)$$

$$= \left(1-\frac{\lambda}{\mu}\right) \cdot \frac{\frac{\lambda}{\mu}}{\left(1-\frac{\lambda}{\mu}\right)^2} = \frac{\frac{\lambda}{\mu}}{1-\frac{\lambda}{\mu}} = \frac{\lambda}{\mu-\lambda}$$

(Eq.3.26)

Mean queue length:

$$mql = \frac{\lambda}{\mu-\lambda} \quad \text{(Eq.3.27)}$$

Applied to the systems A and B the $mql_A = 0.\overline{6}$ and the $mql_B = 0.25$.

Especially in case of interactive systems, the average response time is a useful performance measure. It is defined as the average total time (from arrival upon completion) a job spends in the system. The average response time is thus the sum of the waiting time for a job and the service time. The waiting time for a job is the product of the *mql* with the time to execute one job ($1/\mu$). The service time of the job itself is also $1/\mu$. Thus, the response time (*rt*) is:

$$rt = \frac{1}{\mu} \cdot \frac{\lambda}{\mu-\lambda} + \frac{1}{\mu} = \frac{1}{\mu-\lambda} \quad \text{(Eq.3.28)}$$

Using Little's Law [Lit61] is another way to calculate the response time *rt*. Little proofed that the average number of jobs in the system is equal to the arrival rate of jobs times the average time each job stays in the system:

$$mql = throughput \cdot rt \quad \text{(Eq.3.29)}$$

Using Little's Law, (Eq.3.24) and (Eq.3.26) the response time *rt* can be calculated as:

$$rt = \frac{mql}{throughput} = \frac{\frac{\lambda}{\mu-\lambda}}{\lambda} = \frac{1}{\mu-\lambda}$$

(Eq.3.30)

The response time for system A is then $3.\overline{3}$ seconds, for system B the response time is 1.25 seconds.

The examples in the previous section assumed that the arrival rate and the service rate is independent of the system state. Looking at an interactive system with four users submitting a job in average every 2 seconds to the queue ($\lambda = 0.5$), it is clear that the arrival rate depends on the state the system currently has. After submitting a job the user waits until the job is finished before he submits a new job at the same rate. If no job is in the queue, the arrival rate $\lambda_0$ will be $\lambda_0 = 4*\lambda = 2.0$, if three jobs are already submitted to the queue, the arrival rate will be $\lambda_3 = \lambda = 0.5$. Assume that the service request are similar and that the system can shorten service time by pipeline and cache effects, if more than one job is in the queue. If one job is in the system, the time for the request is 0.5 seconds ($\mu_1 = 2$), if two jobs are in the system, the time per request is 0.4 seconds ($\mu_2 = 2.5$), if three jobs are in the

system, the rime per request is $0.\overline{3}$ seconds ($\mu_3 = 3$) and if 4 jobs are in the system, the time per job is 0.25 seconds ($\mu_4 = 4$). Thus, the service rate also depends on the system state. Fig. 3.20. shows the resulting system model

.

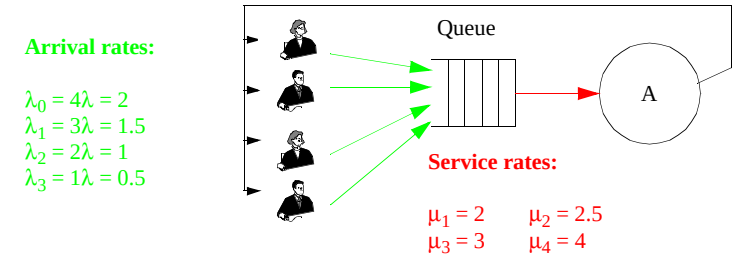



Fig. 3.20. System model for a generalized birth-death model

The state diagram for the interactive system is described in Fig. 3.21. Using the flow balance equations (flow-in = flow-out) the following system of equations is generated:

$$
\begin{aligned}
2P_1 &= 2P_0 \\
2P_0 + 2.5P_2 &= 2P_1 + 1.5P_1 \\
1.5P_1 + 3P_3 &= 1P_2 + 2.5P_2 \\
1P_2 + 4P_4 &= 0.5P_3 + 3P_3 \\
0.5P_3 &= 4P_4
\end{aligned}
$$

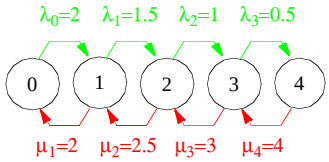



Fig. 3.21. State diagram

Since one of these five equations for five unknowns is redundant and can be deleted, the sum of the probabilities is used to replace one of the equations:

$$P_0 + P_1 + P_2 + P_3 + P_4 = 1$$

Thus the solution of the system of equations is:

$$
\begin{aligned}
P_0 &= 40/113 = 35.4\% \\
P_1 &= 40/113 = 35.4\% \\
P_2 &= 24/113 = 21.2\% \\
P_3 &= 8/113 = 7.1\% \\
P_4 &= 1/113 = 0.9\%
\end{aligned}
$$

Now the performance measures utilization, throughput, queue length and response time can be recalculated:

$$
\begin{aligned}
\textit{utilization} &= 1 - P_0 \\
&= 64.6\%
\end{aligned}
$$

$$
\begin{aligned}
\textit{throughput} &= 2P_1 + 2.5P_2 + 3P_3 + 4P_4 \\
&= 1.49 \textit{ jobs per seconds}
\end{aligned}
$$

$$\begin{aligned} \textit{queue length} &= P_1+2P_2+3P_3+4P_4 \\ &= 1.03\ \textit{jobs} \end{aligned}$$

$$\begin{aligned} \textit{response time} &= \textit{queue length / throughput} \\ &= 0.69 \end{aligned}$$

### 3.2.4. Generalized Birth-Death Models

If a system is in a given state *k*, indicating that *k* jobs are in the system, in a birth death system only two events can cause the system to leave state *k*. Either one job is added to the system (system proceeds from state *k* to state *(k+1)*), or a job is completed and leaves the system (system proceeds from state *k* to state *(k-1)*).

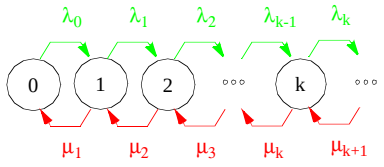



Fig. 3.22. Generalized birth death state-space diagram

The state space diagram of Fig. 3.19. shows that the arrival rate λ and the service rate μ are independent of the current state. To generalize this model, state dependent birth and death rates are introduced. Being in a state *k*, the arrival rate of new jobs is $\lambda_k$, and the completion rate of a job is $\mu_k$ (compare Fig. 3.22.).

This state-space diagram describes a load dependent service center. At a load dependent service center the service rate varies with the number of customers present. A practical example for a load dependent service center is a disk device where accesses are served in an order that attempts to minimize head movement. The greater the number of requests queued at such a device, the smaller the average time required to satisfy each request, since the effectiveness of the scheduling policy increases with queue length.

As shown in the previous example, the flow balance equations can be generated:

$$\mu_1 P_1 = \lambda_1 P_0$$

$$\lambda_0 P_0 + \mu_2 P_2 = \lambda_1 P_1 + \mu_1 P_1$$

$$\vdots$$

$$\lambda_{k-1} P_{k-1} + \mu_{k-1} P_{k+1} = \lambda_k P_k + \mu_k P_k$$

$$\vdots$$

This can be rewritten as:

$$\begin{aligned} P_k &= \frac{\lambda_{k-1}}{\mu_k} \cdot P_{k-1} = \frac{\lambda_{k-1}}{\mu_k} \cdot \frac{\lambda_{k-2}}{\mu_{k-1}} \cdot P_{k-1} \\ &= \ldots = \frac{\lambda_{k-1}}{\mu_k} \cdot \frac{\lambda_{k-2}}{\mu_{k-1}} \cdot \ldots \cdot \frac{\lambda_0}{\mu_1} \cdot P_{k-1} \\ &= P_0 \cdot \prod_{k=0}^{k-1} \frac{\lambda_k}{\mu_{k+1}} \end{aligned}$$

(Eq.3.31)

Using the conservation of total probability $P_0 + P_1 + P_2 + \ldots = 1$, substitution and simplifications, $P_0$ can be calculated as:

$$P_0 = \left[ \sum_{i=0}^{\infty} \prod_{k=0}^{k-1} \frac{\lambda_k}{\mu_{k+1}} \right]^{-1}$$

(Eq.3.32)

The first term in this summation is defined to be 1. Using (Eq.3.31) and (Eq.3.32), any state $P_k$ can be expressed in terms of $\lambda$ and $\mu$:

$$P_k = \left[\sum_{i=0}^{\infty} \prod_{k=0}^{k-1} \frac{\lambda_k}{\mu_{k+1}}\right]^{-1} \cdot \prod_{k=0}^{k-1} \frac{\lambda_k}{\mu_{k+1}}$$

(Eq.3.33)

Now the performance measures utilization, throughput, queue length and response time can be recalculated:

**Utilization:**

$$utilization = 1 - P_0$$

(Eq.3.34)

**Throughput:**

$$throughput = \sum_{k=1}^{\infty} \mu_k \cdot P_k$$

(Eq.3.35)

**Queue length (*mql*):**

$$mql = \sum_{k=1}^{\infty} k \cdot P_k$$

(Eq.3.36)

**Response time (*rt*):**

$$rt = \frac{mql}{throughput} = \frac{\sum_{k=1}^{\infty} k \cdot P_k}{\sum_{k=1}^{\infty} \mu_k \cdot P_k}$$

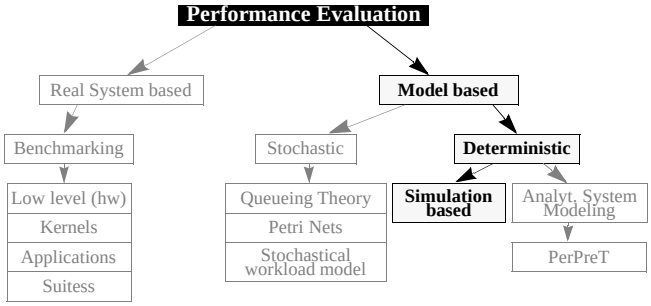
(Eq.3.37)

The focus of this text is not on markovian modeling techniques, this section serves as an overview of the basic modeling technique with respect to use steady state diagrams, generate flow balance equations and solve the equations in order to answer performance questions (e.g utilization, throughput, average queue length, or response time of the target system). More information on markov modeling techniques can be found in [Men94], [Sev81], [Den78], [Laz84] and many others.

**Pros and Cons**

The examples above indicate the advantages and disadvantages of using modeling for performance evaluation:

- ***Advantages:***
  - applicable without real system or prototype of real system
  - changes in workloads are easily modeled through changes in the corresponding parameters
  - graphical representation of model helps to understand the modeled system
  - accurate results for many systems
  - very useful for system designers to model system changes without the need to build a prototype
- ***Disadvantages:***
  - models become very complex if the system under test is complex
  - the resulting diagrams, flow equations and systems of linear equations tend to become very complex
  - very sensitive to accurate parameter values




## 3.3. Simulation

As mentioned in the introduction, this text will not describe techniques for performance simulation. Yet, it is important to discuss the basic differences of simulation techniques compared with other performance evaluation techniques. Fig. 3.23. shows the basic approach for performance evaluation using a system simulator. Instead of using a performance model as in Fig. 3.14., a simulator for the hardware is used. This simulator can also include some of the basic software parameters (such as scheduling or I/O behavior). The principal strength of simulation is its flexibility. There are few restrictions on the behavior that can be simulated, so a computer system can be represented at an arbitrary level of detail. The principal weakness of simulation is its relative expense in developing a simulator and executing workloads on the simulator. The simulator can also be expensive to parametrize, because a highly detailed model implies a large number of parameters.

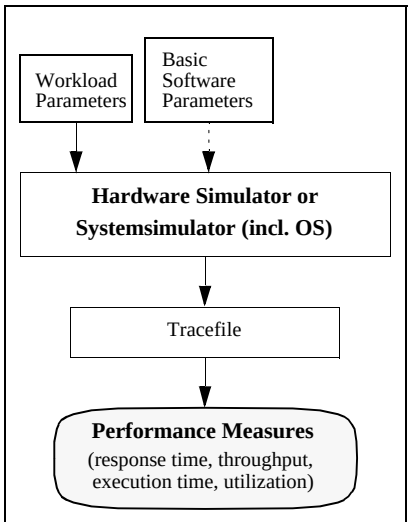



Fig. 3.23. Performance simulation

Today's high performance microprocessors are very complex, the simulation of complete real applications is mostly too time consuming to be executed on a simulator. A slowdown between $10^5$ and $10^7$ makes it almost impossible to run complex real applications using a simulator. Kernels or synthetic workloads are used instead of real applications. The answers to the performance questions are often not direct results of the hardware simulator, they have to be found analyzing the tracefile created by the simulator.

The tracefile contains timestamped information of the workload behavior on the system. This information can be memory references (for trace-driven simulation), references for the use of arithmetic or logic units (for execution driven simulation), or references for the use of communication units in the case of multiprocessors (for execution driven simulation). Once the tracefile is available, the information can be visualized using tools like ParaGraph [Para92]. It can also be used to calculate performance measures like throughput, idle time, execution time, and others. With the detail of the observation (trace information), the size of the tracefiles grows very fast and thus, restricts the simulation time.

The development of the hardware simulator requires considerable programming effort, even if software packages to support this effort are available. The benefit of simulation techniques, if applied properly, are the derivation of the performance measures with very high accuracy. Detailed information and examples for the application of simulation techniques for the performance evaluation of computing systems can be found in [SPL91] and in [DNS95]. The first reference describes the SPLASH (Stanford Parallel Applications for Shared Memory) approach. Parallel applications are executed on a simulator for a shared memory multiprocessor. The SPLASH package is also used as a benchmark for multiprocessors as described in the next section. The second reference describes the simulation of the Power PC (PC620) microarchitecture.

**The System Simulator ClearSim**

Large networks of embedded control systems are of growing importance. More demanding applications, greater functionality of hardware and system software and the usage of distributed subsystems, require an analysis in each stage of development. In [BMS97] the system simulator ClearSim to simulate the behavior of embedded control systems is presented.

ClearSim has been developed within the ESPRIT project OMI/CLEAR. This project aimed at providing a configurable RTOS to support the designer of embedded control systems to build application software of high complexity in less time. To support the development of real time applications, a system simulator was included, because simulation offers advantages to a real experiment.

ClearSim helps to evaluate the system to be developed in the early design stages. The simulation environment includes models of the microcontroller (ARM610 in the CLEAR project), the RTOS (EOS [Cas95]) and additional hardware like e.g. timers, motors etc. With ClearSim it is possible to simulate and evaluate systems, even when some parts, e.g special hardware or the real target microcontroller, are non-existent. For the description of the models normal C language is used, so no special simulation language has to be learned. The simulation not only covers functional modeling but also correct timing with a typical accuracy of ±10%. Nevertheless, simulation speed is very fast, typical slowdowns range from 5 to 50.

The simulator is based on the event-driven method which is extended by an execution-driven algorithm for software simulation [Gol90], [Spi93], [CMS93]. The *execution-driven simulation* has a high execution speed on relatively high abstraction levels. Compared to the instruction-driven simulation where assembler instructions are read and emulated as an interpreted language, the execution-driven method is based on the execution of the target instructions on the host machine, i.e. the workstation. The observation is

managed by subroutine calls to control routines being automatically inserted by an instrumenter, before linking the executable. The simulation speed of the execution driven algorithm is much higher than that of the instruction driven method since it maps most of the target components on the host system, such as processor registers, processor status, memory and the complete instruction set.

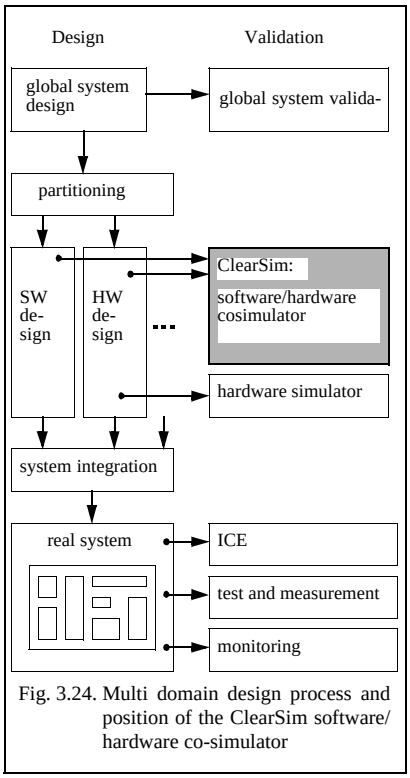



Fig. 3.24. Multi domain design process and position of the ClearSim software/ hardware co-simulator

A typical ClearSim simulation deals with one or more processes and their interaction over a communication medium. A process can be an execution-driven or a physical process. The actual software is implemented as one or more execution-driven *application processes*, with special modeling support for the OS and the real target processor. The rest of the simulated system, the peripherals and the environment consisting of timers, devices, sensors etc., is built with *physical processes*.

The description of software is the source code itself as it can be used in the final product. Even the makefile is identical. No simulator instructions need to be inserted manually although it is possible to influence the observation behavior if necessary. The instrumenter also automatically inserts information of the correct timing of the application software.

The required hardware to run the application on is simulated or mapped on the host architecture. Every system component of the SUC which interacts with the system in any kind of communication (shared memory, signals, interrupts, hardware registers) is defined as a *physical process*. The definition of a physical process comprises the timing and the interactional behavior with the software or even other physical processes. It is in the responsibility of the developer to define how detailed the specification of a process should be.

Every physical process is given through a C coded program. These programs have to be written by the user or taken from a library because they are simulation specific in the sense that they are not needed in "reality“ on the target system. Contrary to the application processes, where the source code includes both, function and timing, the timing for the physical processes has to be explicitly inserted by the user. The simulation environment offers library functions to the user handling the timing and communication of the process. These functions are similar to known simulation languages, especially SIM++ [Jad89] and ModSim [Her90]. Examples for such library functions are event handling routines, random number generators and statistical evaluation functions.

The communication between the processes is performed by the event list manager ELM. This central manager starts and stops certain activities according to the central event list and is responsible for the bookkeeping of the simulation time. In case of a simulation of distributed systems, the ELM has also to syn-

chronize and coordinate the different processes involved in the simulation [Mis86].

The observation of program behavior is based on tracing. Unlike programs running on the real hardware, tracing of a simulation run does not distort the monitored results, since the trace functions can be made invisible for the time bookkeeping mechanism.

ClearSim offers a wide range of observation options. There is information about the corresponding source code, the memory usage, interrupts or timing. In addition the user can insert commands for generating self defined trace stamps. With several analysis tools relevant data can be extracted from trace files and visualized in different sights. Moreover, a source code debugger is available to find logical errors as well as algorithmic and timing errors.

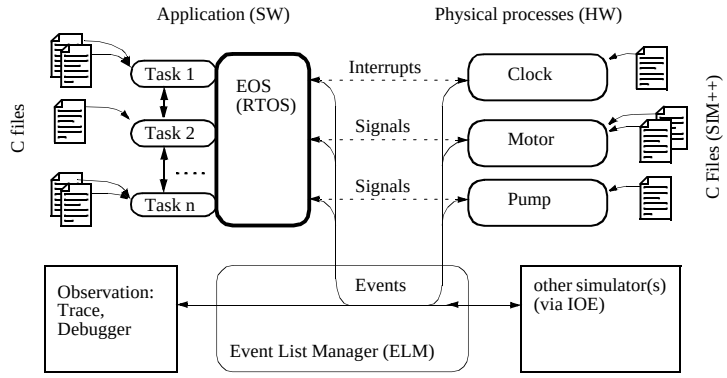
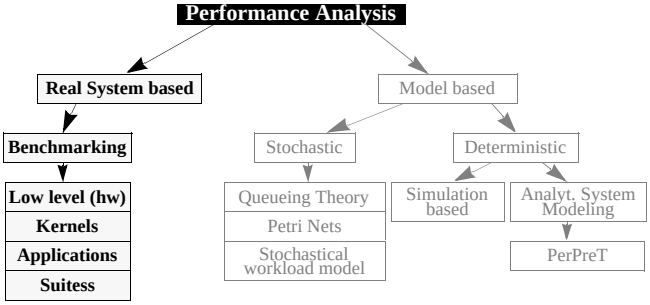



Fig. 3.25. Combination of application and physical processes




# 4. Benchmarking of Multiprocessors

The demand for increased performance forced system designers to exploit parallelism on all computer architectural levels. Starting with on chip level parallelism (register width, memory bus width, instruction bus width, several execution units) and going up to system level parallelism (multiprocessor systems). Except the better performance, the exploitation of parallelism was transparent to the user if done under the system level. Smart compilers handled the often difficult task to produce machine code which efficiently used the parallel resources.

Up to today there is neither software that automatically generates efficient parallel code for all available multiprocessors, nor exists a common programming language. This is why it is difficult for a user to write multiprocessor programs and why the task of performance evaluation for multiprocessors is much different compared to performance evaluation of monoprocessors. Looking at the benchmark techniques in the previous chapter where basically some piece of software was implemented and monitored on a target system, this approach has to be refined to be applicable to multiprocessors.

This section starts with a short description of multiprocessor architectures and some examples for massively parallel systems. After describing existing benchmarking approaches for the performance evaluation of multiprocessors a new approach, the LOOP method is described.

## 4.1. Architecture of Multiprocessors

### 4.1.1. Shared Memory versus Message Passing

The state of the art commercially available MIMD-systems are divided into two architecture classes, namely, systems with shared memory and message passing systems.

### Shared Memory Systems

Shared memory systems (compare Fig. 4.1.) work with a global address space which can be used by all processors.The memory modules are connected to the processors via a network or a bus system. Each processor can optionally be equipped with a private cache. Communication and synchronization of the concurrently executed tasks is realized by reading and writing shared data.

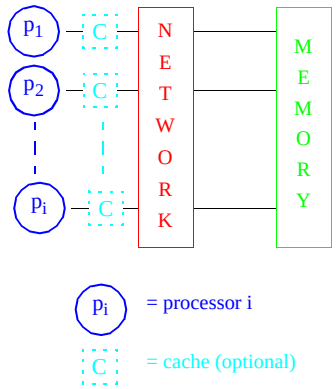



Fig. 4.1. MIMD-systems with global shared memory

The main problem of these systems is the global resource memory which is used by all processors at the same time and thus the most probable performance bottleneck.This is also the reason why commercially successful shared memory systems consist of a relatively small number of processors (typically between 2 and 64 processors).

### Message Passing Systems

All processors in a message passing system (compare Fig. 4.2.) have access to their own local memory modules. Communication and synchronization between the processors is realized through messages sent through a connection network.

Programming message passing multiprocessors showed to be a lot more difficult than programming monoprocessors. In the beginning these machines did not have a common programming language or even worse a common programming paradigm. Most machines were delivered with a sequential programming language (such as Fortran or C) with extensions for message passing (send and receive routines) and program loading. After realizing that reprogramming for new or different machines becomes too expensive, portable communication libraries such as PICL (Portable Instrumented Communication Library) [PICL90] or MPI (Message Passing Interface) [MPI94] were developed. Now the programs became portable among a variety of message passing machines, but the development of new programs is still complicate and lacks tools to help the user exploit the potential performance of the system. Since most of the applications which run on these machines have performance demands that can only be satisfied by these machines, the programmer has no choice but to spend the effort of writing message passing programs.

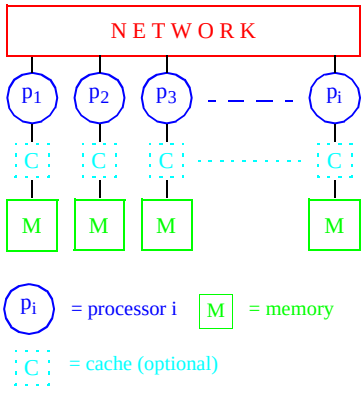



Fig. 4.2. Message passing MIMD-system

**Massively Parallel Systems**

Massively or highly parallel systems consist of a large number of processors (usually more than 1000). They offer very high performance which, unfortunately, is not always easy to use. The nature of highly parallel systems asks for highly parallel applications. In scientific computing, these systems are mainly used to solve special problems (e.g. grand challenges) or they are used as parallel data base servers. Massively parallel systems are typically message passing architectures. Since some people assume that it is easier to produce software for shared memory systems rather than for message passing systems, it is possible to work with a virtual shared memory which is automatically mapped onto a message passing scheme [PVM94].

### 4.1.2. System Examples

Massively parallel message passing systems are of special interest for performance evaluation and performance prediction techniques described in the next sections of this text. In order to get a better understanding of the architectural features of these systems and a better understanding of the problems which can arise with the performance evaluation three example systems are now presented.

#### 4.1.2.1 nCUBE/2

**Logical Configuration of nCUBE/2**

The nCUBE/2 is a message passing multiprocessor. The nodes are connected via a network with a multidimensional cube topology which is called hypercube:

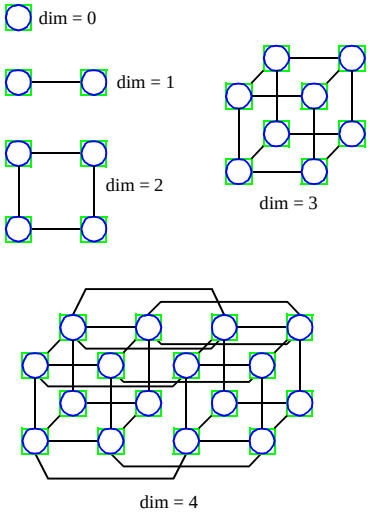



Fig. 4.3. Hypercube topology

The hypercube network is organized so that connections among processors form cubes. As more processors are added, the cube grows to larger dimensions. Two hypercubes of the same dimension, joined together, form a hypercube of the next dimension. Fig. 4.3. starts with a single computing node which is defined as hypercube with dimension zero. Adding a computing node and connecting the two nodes results in the hypercube of dimension one. Copying the new hypercube of dimension one and connecting the nodes results in the hypercube of the next dimension two. This construction scheme can be used to create hypercubes of any dimension. Hypercube networks have some interesting features for communication:

- The hypercube is built of $2^N$ nodes, with $N$ = *dimension of the hypercube*.
- The longest path between any two nodes is equal to the dimension of the hypercube, N.
- The number of link interfaces per node is equal to the dimension of the hypercube.
- Other logical configurations like ring, array, torus, 3D-array can easily be mapped onto hypercube topologies.

The nodes of the nCUBE/2 hypercube are used for computing only, the program development can be done on a workstation which is connected to the hypercube. Input and output operations can be done via the workstations connection or via I/O channels which directly connect the hypercube with harddrives. This leads to the logical configuration for the nCUBE/2 system as outlined in Fig. 4.4.

### Physical Configuration of nCUBE/2

The nCUBE/2 system was delivered in the late eighties. The workstation used as frontend was a SUN SPARC workstation with a VME board for the connection to the hypercube. The operating system of the frontend was UNIX, the programming language for the hypercube was an extension of C with parallel constructs for loading programs onto the hypercube, constructs for input and output from the hypercube on the frontend, and constructs for synchronization and communication of the concurrently running node programs. The hypercube nodes were proprietary microprocessors running at several MFLOPS. The direct connection of nodes to disks was realized through communication with I/O nodes which were connected to harddrives through I/O channels. In summary, the physical configuration of the nCUBE/2 hypercube is described in Fig. 4.5.

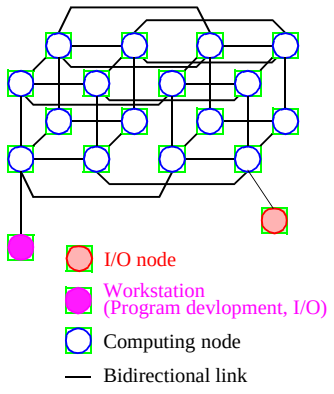
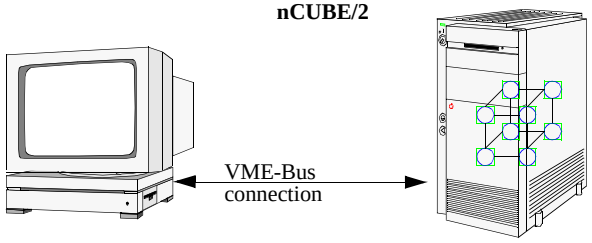



Fig. 4.4. Logical configuration of the nCUBE/2




Hostcomputer (frontend):

-SUN SPARC
-VME Board for the connection to the nCUBE/2
-Operating system UNIX
-C-Compiler (for Hostprograms)
-C-Crosscompiler (for Nodeprograms)

nCUBE/2 Nodes:

-Compute Nodes
-I/O Nodes for the connection to local disks
-Operating system Vertex

Fig. 4.5. Physical configuration of the nCUBE/2

#### 4.1.2.2 INTEL Paragon

**Logical Configuration of INTEL MP/150**

Except the network topology, the logical configuration of the INTEL MP/150 multiprocessor system looks very similar to the logical configuration of nCUBE/2. A workstation frontend is used for program development, input and output operations, and as user interface to the node processors. Special I/O nodes provide fast access to local disk drives.

The computing nodes are arranged as a two dimensional array. Each row of the resulting matrix is connected to an I/O node for fast disk I/O operations and to a service node as an interface to other I/O services.

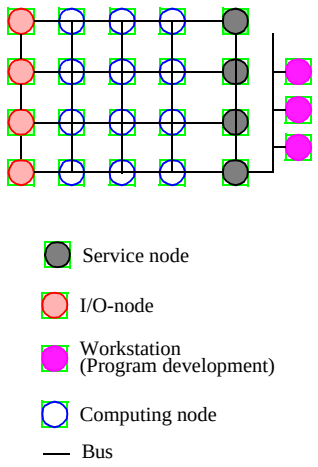



Fig. 4.6. Logical configuration of INTEL Paragon MP/150

**Physical Configuration of INTEL MP/150**

The INTEL Paragon MP/150 multiprocessor was delivered in the mid nineties and is about 10 years younger than the nCUBE/2 system. As a comparison of the logical configurations of the two systems shows, the way to develop and run programs on both machines is very similar. Also the idea to use I/O nodes to provide the computing nodes with fast connections for disk operations remained the same. Using PICL it is possible to run old programs on the new machine. There are only two real differences to be observed:

- The hypercube network is replaced by a two dimensional array. One reason to do this is the complexity of communication hardware for the computing nodes. In a mesh topology each node only needs link interfaces for four neighbors (north, east, south, west). A constant number of link interfaces is easier to handle for hardware designers than a number that varies with the size of the multiprocessor. Since the links are busses for INTEL MP/150 compared to point to point connections of the nCUBE/2 there are no drawbacks as to the length of the longest path between any two processors. Since several pairs of processors can communicate concurrently sharing the same bus, the bandwidth of the bus has to be larger than the bandwidth of links for point to point connections [Smi95].
- The second major difference of INTEL MP/150 compared to nCUBE/2 is the complexity of the computing node. Two CPUs for computation and one CPU for communication issues are provided. All CPUs are

based on the INTEL i860xp microprocessor.

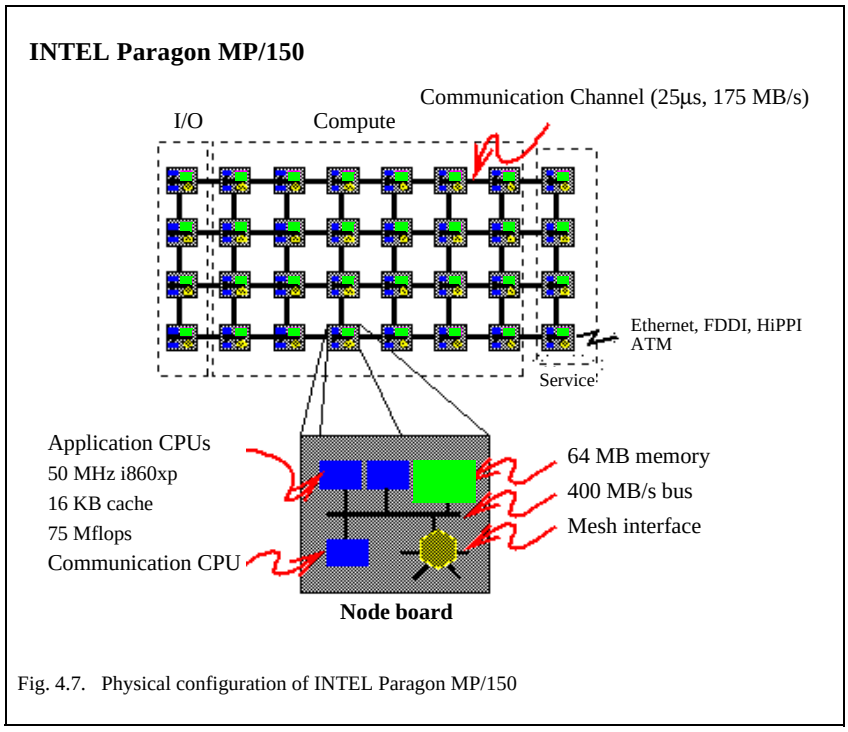



Fig. 4.7. Physical configuration of INTEL Paragon MP/150

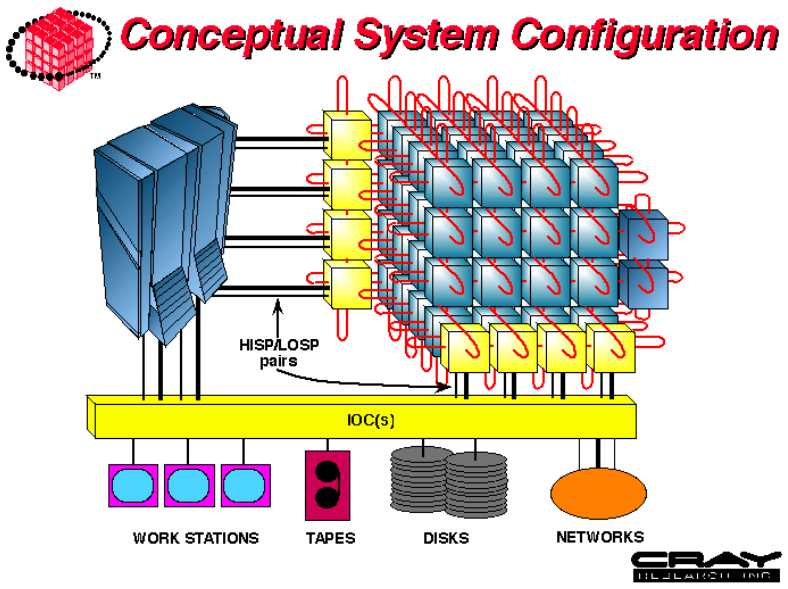
#### 4.1.2.3 Cray T3D

**Logical Configuration of Cray T3D**

Looking at the logical configuration of a state of the art multiprocessor system as the CRAY T3D in Fig. 4.9. does not bring up major differences compared to the logical configurations of the systems described before. In contrast to the INTEL MP/150 a three dimensional torus network is used instead of a two dimensional array. As in the two previously described systems, the nodes are still purely used for computation, program development is done in a workstation environment.

**Physical Configuration of Cray T3D**

The main difference of the CRAY T3D system from the physical aspect of view is the performance of its nodes and the performance of the network (compare Fig. 4.8.).




Technology:

- 128 to 256 processing elements (PEs)
- Up to 16 GBytes of globally adressable memory
- High bandwidth, parallel I/O
- 19.2-38.4 GFLOP/s peak performance
- High-performance interconnect 3-D Torus

Fig. 4.8. Physical configuration of CRAY T3D

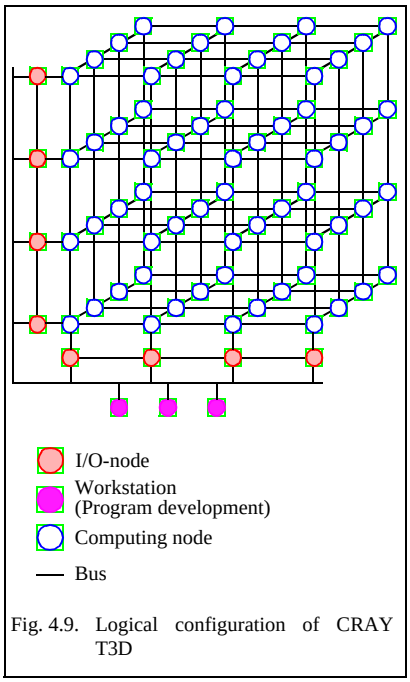



Fig. 4.9. Logical configuration of CRAY T3D

## 4.2. Workloads for Multiprocessors

The need for high performance systems triggered the development of multiprocessors of different architectures in the early eighties. In the scientific and academic community they were immediately accepted and very successful. They offered chances to run complex applications in reasonable amounts of time. Often the hardware development outperformed the software development in terms of parallel languages and compilers, tools to monitor and debug parallel applications, and tools to evaluate the system performance of the new architectures. This section describes the development of benchmarks for multiprocessors (shared memory and message passing) which up today are collections of parallel kernels and applications. At the end of the section a new approach, the LOOP benchmark, is presented. The main difference to existing techniques is:

- Scalability in terms of problem size and number of processors.
- User defined workloads possible.
- Automatic instrumentation and measurement environment included.
- Graphical result analysis included.

### 4.2.1. Kernel and Application Benchmarks

#### 4.2.1.1 LINPACK

The collection of application kernels LINPACK programmed by Jack Dongarra [Don79] and its use as a monoprocessor benchmark is presented in section 3.1.2. Since its first appearance in the late seventies it was used to test almost any high performance system on the market. The problem size was initially set to (100x100) matrix sizes. After increasing performance of the systems and advances in memory technology (larger caches), the problem size was increased to (300x300) matrix sizes. As the race for performance and large main memory was and is still going on, the problem size was recently set to (1000x1000) matrices. LINPACK is acknowledged as a standard benchmark test and it is also used for multiprocessor target systems. In [Don97] Jack Dongarra published new rules for using the LINPACK benchmark for massively parallel systems. He realized that these systems are sometimes too large to be efficiently used with a LINPACK problem size of (1000x1000). The new ground rules are set as follows:

- Solve systems of linear equations by some method, allow the size of the problem to vary, and measure the execution time for each size problem. In computing the floating point execution rate (MFLOPS), use

  $$\frac{2n^3}{3} + 2n^2$$

  operations independent of the actual method used. If Gaussian Elimination is used, partial pivoting must be used. Compute and

report a residual for the accuracy of solution as:

$$\frac{||Ax - b||}{(||A|| ||x||)}$$

Tab. 4.10. shows some LINPACK results[1] for massively parallel systems (more than 1000 processors are used for the execution of the benchmark). The columns in Tab. 4.10. are defined as follows:

- *Procs* is the number of processors used to execute the LINPACK benchmark.
- $R_{max}$ is the performance in GFLOP/s (Giga floating point operations per second) for the largest problem run on a machine.
- $N_{max}$ is the size of the largest problem run on a machine.
- $N_{1/2}$ is the size where half the $R_{max}$ execution rate is achieved.
- $R_{peak}$ is the theoretical peak performance in GFLOPS for the machine.

1. Results are taken from PDS (Performance Data Base Server) at URL: h*ttp://performance.netlib.org/performance/html/linpack-parallel.data.col0.html*

**Summary for LINPACK**

In summary, the pros and cons of the LINPACK benchmark are:

**Pro:**

LINPACK is scalable, by adjusting the matrix size with the performance increase of the target system, it is still possible to make reasonable measurements.

**Con:**

The workload created by LINPACK was never meant to evaluate all aspects of the system performance. Even the author of LINPACK Jack Dongarra pointed out that LINPACK is a good test for the arithmetic behavior in case of matrix operations but that LINPACK should not be used to evaluate the system as a whole. By simply increasing the problem size, the workload created by LINPACK gets even more "exotic", since there are not too many real life situations where the multiplication of two dense (1000x1000) matrices is needed. Usually, real life applications of that size work with sparse matrices which involves a totally different storage and operation scheme.

| **Computer** | **Procs.** | **$R_{max}$ (GFLOPS)** | **$N_{max}$ (order)** | **$N_{1/2}$ (order)** | **$R_{peak}$ (GFLOPS)** |
|---|---|---|---|---|---|
| Intel ASCI Option Red (200 MHz Pentium Pro) | 9152 | 1338.0 | 235000 | 63000 | 1830 |
| CRAY T3E-900 (450 MHz) | 1320 | 670.0 | 128832 | 23184 | 1188 |
| CP-PACS* (150 MHz PA-RISC based CPU) | 2048 | 368.2 | 103680 | 30720 | 614 |
| Cray T3D 1024 (150 MHz) | 1024 | 100.5 | 81920 | 10224 | 152 |
| Thinking Machines CM-5 | 1024 | 59.7 | 52224 | 24064 | 131 |
| Thinking Machines CM-200 (10 MHz) | 2048 | 9.8 | 29696 | 11264 | 20 |
| MasPar MP-1216 (80ns) | 16384 | 0.473 | 11264 | 1280 | 0.55 |

Tab. 4.10. Linpack results for massively parallel machines

#### 4.2.1.2 NAS Parallel Benchmarks

The NAS (Numerical Aerodynamic Simulation) program is a group at NASA Ames Research Center focused on high performance computing. Parallel numerical applications from computational fluid dynamics and related aerosciences disciplines are implemented on high performance systems. To measure objectively the performance of highly parallel computers and to compare their performances with that of conventional supercomputers, NAS developed the NAS Parallel Benchmarks (NPB) [Bai91].

Since NAS used several massively parallel systems with different programming models and parallel languages, the idea of creating a "pencil and paper" benchmark suite was realized. This technique tells the user which algorithm he should use to solve a problem, but he has the choice of programming model and language. Within the framework of Fortran 77, Fortran 90, C and HPF (High Performance Fortran), implementors are free to use language constructs, data structures, data partitioning and algorithmic details that maximize performance on a target system. Some rules for the implementation of the NAS Parallel Benchmarks must however be observed:

- All floating point operations must be performed using 64-bit floating point arithmetic.
- All benchmarks must be coded in either Fortran or C, with certain approved extensions for parallel processing.
- Implementations of the benchmarks may not include a mix of Fortran and C code, one or the other must be used.
- HPF extensions are allowed.
- Any language extension or library routine that is employed in any of the benchmarks must be supported by the target system vendor and available to all users.
- All rules apply equally to subroutine calls. language extensions and compiler directives (i.e., special comments).
- Subprograms and library routines not written in Fortran or C may only perform the following functions:

  • Indicate code that can be executed in parallel or loops that can be distributed among different computational nodes.
  • Specify the allocation and organization of data among or within computational nodes.
  • Communicate data between processing nodes.
  • Communicate data between the computational nodes and service nodes.
  • Rearrange data stored in multiple computational nodes, including constructs to perform indirect addressing and array transpositions.
  • Synchronize the action of different computational nodes.
  • Initialize for a data communication or synchronization operation that will be performed or completed later.
  • Perform high speed input or output operations between main memory and the mass storage system.
  • Perform any of the following array reduction operations on an array either residing within a single computational node or distributed among multiple nodes: +,*,MAX,MIN, AND, OR, XOR.
  • Combine communication between nodes with one of the operations listed in the previous item.

- Perform any of the following computational operations on arrays either residing within a single computational node or distributed among multiple nodes: dense or sparse matrix multiplication, dense or sparse matrix-vector multiplication, 1D, 2D, or 3D FFTs, sorting, block tri-diagonal system solution and block penta-diagonal system solution. Such routines must be callable with general array dimensions.

The NPBs consist of three kernels and five applications. The kernels are EP (embarrassingly parallel), MG (Multigrid), and CG (Conjugate Gradient). The applications are FT (3 dimensional FFT for solving partial differential equations), IS (Integer sort), LU (LU solver), SP (pentadiagonal solver), and BT (tridiagonal solver). Since the NPBs are intended to be implemented on a wide variety of systems with substantially different performances, three classes of NPBs with respect to the problem sizes are defined.

| Benchmark code | Class A | Class B | Class C |
|---|---|---|---|
| Kernels: | | | |
| Embarrassingly parallel (EP) | $2^{28}$ | $2^{30}$ | $2^{32}$ |
| Multigrid (MG) | $256^3$ | $256^3$ | $512^3$ |
| Conjugate Gradient (CG) | 14000 | 75000 | 150000 |
| Applications: | | | |
| 3-D FFT PDE (FT) | $256^2$x128 | $256^2$x512 | $512^3$ |
| Integer sort (IS) | $2^{23}$ | $2^{25}$ | $2^{27}$ |
| LU solver (LU) | $64^3$ | $102^3$ | $162^3$ |
| Pentadiagonal solver (SP) | $64^3$ | $102^3$ | $162^3$ |
| Block tridiagonal solver (BT) | $64^3$ | $102^3$ | $162^3$ |

Tab. 4.11. NAS Parallel Benchmarks problem sizes

Tab. 4.12. gives an example for a result table. It uses NAS sample codes implemented on one node of a CRAY-MP. Problem sizes, memory requirements (measured in Mw (Mega word)), run times (in seconds) and Performance rates (measured in MFLOP/s) for each of the eight benchmarks are given.

| | Class A | Mem Mw | Time secs | Rate MFLOP/s | Class B | Mem Mw | Time secs | Rate MFLOP/s |
|---|---|---|---|---|---|---|---|---|
| EP | $2^{28}$ | 1 | 151 | 147 | $2^{30}$ | 18 | 512 | 197 |
| MG | $256^3$ | 57 | 54 | 154 | $256^3$ | 59 | 114 | 165 |
| CG | 14000 | 10 | 22 | 70 | 75000 | 97 | 998 | 55 |
| FT | $256^2$x128 | 59 | 39 | 192 | $256^2$x512 | 162 | 366 | 195 |
| IS | $2^{23}$ | 26 | 21 | 37 | $2^{25}$ | 114 | 126 | 25 |
| LU | $64^3$ | 30 | 344 | 189 | $102^3$ | 122 | 1973 | 162 |
| SP | $64^3$ | 6 | 806 | 175 | $102^3$ | 22 | 2160 | 207 |
| BT | $64^3$ | 24 | 923 | 192 | $102^3$ | 96 | 3554 | 203 |

Tab. 4.12. NAS Parallel Benchmarks results for one node of CRAY Y-MP

After the message passing library MPI [MPI94] was published and implemented on most existing message passing multiprocessors, NAS offered source codes using Fortran 77 and MPI for their eight benchmarks [Bai95]. More information on the NAS Parallel Benchmarks can be found in [Bai94] and [Bai95] or on the WWW under:
*http:///science.nas.nasa.gov/Software/NPB/*

The result page of that URL is kept updated and currently contains results for multiprocessors such as IBM SP2, IBM SP2-160tn, Cray T3D, SGI Power Challenge Array, INTEL Paragon, Cray J90, Cray C90, Cray T3D. Cray T3E-900. Cray T3E-1200. UC Berkeley-NOW. The results were obtained by NAS (not submitted by vendors). In addition, vendor submitted results are published for the following machines: IBM SP (P2SC 120MHz nodes), SGI Origin 2000, HP/Convex Exemplar SPP2000.

**Summary for NAS Benchmarks**

In summary, the pros and cons of the NAS idea of "paper and pencil" benchmarks are:

**Pros:**

It is very useful when no common parallel programming model exists. Looking at different architectures (shared memory versus message passing) it is still an interesting approach. The NPB is scalable with respect to different problem sizes and thus, it is applicable to a wide range of computing systems.

**Cons:**

The problem of the NAS Parallel Benchmarks is its focus on only eight workloads that are probably not important for every user. In that case the NPBs can still be used to simply compare different machines.

#### 4.2.1.3 PERFECT Club Benchmarks

Similar to the NAS parallel benchmarks the PERFECT (Performance Evaluation for Cost-effective Transformations) Club benchmarks [PCB94] consist of a suite of 13 parallel applications which run on high-performance computers has been collected in a joint effort of several US research and development organizations. The codes are being used for benchmarking computer systems as well as for the purpose of evaluating new concepts and developments in high-performance computing. The applications are computationally extensive at a fixed problem size. Thus, they are not scalable. The I/O problem is left out, i.e. no time spent for I/O operations is measured. In characterizing the applications, particular emphasis is placed on understanding the underlying algorithms in the hope that there is greater generality at the algorithmic level. The PERFECT Club benchmark suite consists of the following applications:

- ADM
  A complete three-dimensional system of hydrodynamic equations is solved which simulates pollutant concentration and deposition patterns in lakeshore environments [Chr86], [Chr87]. The advection-diffusion equation for the transport, diffusion, and deposition of pollutants is also included in the model. The advection term is treated by a Fourier pseudospectral method in the x and y directions. Convection-diffusion processes in the vertical direction are treated using a semi-implicit Crank-Nicolson method.

- ARC3D
  The Euler and Navier Stokes equations for analyzing three-dimensional fluid flow problems are solved using an implicit finite difference code [Pul85]. The equations are discretized onto a curve linear mesh, and a Beam-Warming [Bea76] method is used to cast the equations in implicit form. Three-point central difference approximations are used, leading to a pentadiagonal system of equations.

- BDNA
  The BDNA code performs molecular dynamics simulations of biomolecules in water [Saw87]. BDNA is a simulation of the hydration structure of potassium counter ions and water in B-DNA. There are 1500 water molecules and 20 counter ions placed in a parallelepiped box of dimension 33.8 Å x 44.0 Å x 44.0 Å. One complete helical turn of B-DNA consisting of ten pairs is considered.

- DYFESM
  A two-dimensional, dynamic, finite element code for the analysis of symmetric anisotrophic structures [Noo85]. An explicit leap-frog temporal method with substructuring is used to solve for the displacement and stresses, along with velocities and accelerations at each time step. The accelerations are computed via a preconditioned Conjugate Gradient method at each time step.

- FLO52Q
  The unsteady Euler equations are solved to analyze the transonic inviscid flow past an airfoil [Jam83]. The two-dimensional domain is discretized into quadrilateral cells. In the case considered there were 128 intervals around the profile, and 32 radially. Application of the Euler equations to these cells yields a set of coupled ordinary differential equations, to each of which a dissipative term is added to suppress non-physical oscillations near regions of steep gradients, such as shock waves. The set of differential equations is solved by incorporating a simple saw tooth multigrid strategy into the multistage time stepping scheme. This scheme is good at modeling shock waves, and the algorithm can be vectorized with a vector length equal to a grid dimension.

- MDG
  A molecular dynamics calculation of 343 water molecules in the liquid state at room temperature and pressure is performed by MDG [Lie86]. The Newtonian equations of motion are solved for 343 water molecules in a cubical box. The total potential is the sum of the intra- and intermolecular potentials.

- MG3D
  Seismic migration code [Res89] to investigate the geological structure of the Earth. Signals of different frequencies are emitted at the Earth's surface, and after interacting with the geological strata, are received at another point on the surface. The data collected at the surface by this technique can then be used to extrapolate backwards in time to get a three-dimensional image of the structure below the surface. This data is Fourier transformed in time, and the depth extrapolation in the z-direction can proceed independently (in parallel) for each frequency. The sample code uses an X-Y grid of 125x120 points, respectively, and on step (downward in the earth) in z-direction.

- OCEAN
  The dynamical equations of a two-dimensional Boussinesq fluid layer are solved using a spectral method [Cur84]. The nonlinear terms are evaluated by fast transform methods with aliasing terms removed. Timestepping is done by a leapfrog scheme for the non linear terms, and an implicit scheme for the viscous terms. The pressure term is computed in Fourier representation by local algebraic manipulation of the incompressibility constraint. The flow is assumed to occur in a box region with

periodic and free-slip (no-stress) impermeable boundary conditions applied.

- QCD
  Quantum Chromodynamics (QCD) is the gauge theory of the strong interaction which binds quarks and gluons into hadrons, which in turn make up the constituents of nuclear matter. Computer simulations are necessary to study long term effects in QCD theory [Ott87]. In these lattice gauge simulations the quantum field is discretized onto a periodic, four dimensional, space-time lattice. Quarks are located at the lattice sites, and the gluons that bind them are associated with the lattice links. The gluons are represented by SU(3) matrices, which are a particular type of 3x3 complex matrix. A major component of the QCD code involves updating these matrices. The PERFECT sample code uses a Monte Carlo technique to perform the matrix update. The lattice size for the sample code is $8^4$.

- SPEC77
  The atmospheric flow is simulated using a global spectral method [Sel80], [Sel82]. The formulation of the model is based on the unknown functions in terms of their spherical harmonic expansions in the horizontal direction. In the vertical direction, a quadratic conserving finite difference formula is used. A semi-implicit backward time integration scheme is applied, and initial conditions are obtain from a spectral Hough operational analysis. The ocean interacts with the atmosphere by means of evaporation and sensible heating. The moisture cycle consists of large-scale precipitation and convection.

- SPICE
  This code is a general-purpose circuit simulation program for non-linear DC, non-linear transient and linear AC analysis[Nag75]. Problems are formulated as stiff differential-algebraic equations. The numerical methods of solution include stiffy stable numerical integration algorithms, the Newton-Raphson method for solving the non-linear algebraic equations, and sparse linear system solvers [New83].

- TRACK
  This code [Got88] determines the course of an unknown number of targets, such as rocket boosters, from observations of the target s taken by sensors at regular time intervals. The targets may be launched from a number of different sites. If the target's acceleration is assumed to be known, then the path of an individual object is described fully by a 4-component launch vector made up of the latitude and longitude of the launch site, the time of the launch, and the initial launch azimuth relative to due north. At each time step a kinematical model with a stochastic acceleration component is used to estimate the position, velocity, and acceleration of the targets. The output from this phase is then passed to the precision parameter estimation module which uses Newton-Raphson iteration to solve an equation giving a more precise estimate of the launch parameter vector.

- TRFD
  This code is a kernel simulating the computational aspects of a two-electron integral transformation. It is part of the HONDO quantum mechanical package [Dup88]. The evaluation of these types of integral transformations is a necessary first step in computing correlated wavefunctions [Dup87], and is used in determinations of molecular electronic structure. The evaluation of the integral transformations is formulated as a series of matrix multiplications.

For the benchmark test the codes are implemented on the target system and the runtime of the programs are measured. From this measure the MFLOP/s rate is calculated as usual:

$$\text{MFLOP/S} = \frac{\text{\# of floating point operations}}{\text{CPU time in seconds} \cdot 10^6}$$

The MFLOP/s rate is compressed for each machine using the harmonic mean, which is defined as:

$$MEAN = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

where each $x_i$ is the performance measure of interest and $n$ is the number of such measures.

| Program | Hitachi S-820/80 | Cray Y-MP/832 | Cray 2S/4128 | NEC SX-2 | Cray X-MP/416 | ETA 10G |
|---|---|---|---|---|---|---|
| **Fluid Flow** | | | | | | |
| ADM | 22.3 | 18.7 | 13.1 | 16.1 | 14.8 | 7.3 |
| ARC2D | 499.2 | 448.2 | 100.3 | - | 183.8 | - |
| ARC3D | - | 233.1 | 58.5 | 71.4 | 130.7 | 33.3 |
| FLO52 | 229.7 | 328.7 | 61.7 | 177.1 | 194.0 | 62.2 |
| OCEAN | - | 35.5 | 22.7 | 16.9 | 25.1 | 9.6 |
| SEPC77 | 37.9 | 51.5 | 17.5 | 41.2 | 30.1 | 9.8 |
| **MEAN** | **(51.6)** | **54.2** | **27.2** | **(30.3)** | **37.7** | **12.9** |
| **Chemical & Physical** | | | | | | |
| BDNA | 123.5 | 121.5 | 81.3 | 170.2 | 83.0 | 58.4 |
| MDG | 15.2 | 16.6 | 14.0 | 14.1 | 13.4 | 6.0 |
| QCD | 9.3 | 12.6 | 7.9 | 9.6 | 10.1 | 5.4 |
| TRFD | - | 56.4 | 25.8 | 57.9 | 44.8 | 24.7 |
| **MEAN** | **(16.5)** | **24.1** | **16.1** | **20.2** | **19.2** | **9.7** |
| **Engineering Design** | | | | | | |
| DYFESM | 69.8 | 59.4 | 32.0 | 66.6 | 41.1 | 36.0 |
| SPICE | 5.7 | 5.7 | 3.9 | 4.7 | 3.9 | 2.1 |
| **MEAN** | **10.5** | **10.5** | **6.9** | **8.7** | **7.2** | **4.0** |
| **Signal Processing** | | | | | | |
| MG3D | - | 27.1 | 19.4 | 35.0 | 21.2 | 6.5 |
| SPICE | 7.4 | 7.9 | 5.1 | 6.2 | 6.5 | 3.7 |
| **MEAN** | **(7.4)** | **12.3** | **8.1** | **10.5** | **9.9** | **4.7** |
| **All Codes** | | | | | | |
| **MEAN** | **(17.1)** | **22.2** | **13.9** | **(16.6)** | **16.5** | **(7.5)** |

Tab. 4.13. PERFECT Benchmark baseline results (in MFLOP/s)

Tab. 4.13. is published in [Poi93]. It shows PERFECT Benchmark results for some computing systems. In this table the codes are simply compiled and executed. The bold numbers in Tab. 4.14. represent measured performances for **hand** optimized codes. Especially for the Cray systems, the baseline results and the optimized results are significantly different.

The Perfect Benchmarks codes are available free of charge for non-commercial use. They can be ordered in hardcopy from: *librarian@csrd.uiuc.edu.*
Further information is available via *ftp* from *ftp.csrd.uiuc.edu:/pub/CSRD_Software/Perfect/version1*.

| Program | Hitachi S-820/80 | Cray Y-MP/832 | Cray 2S/4128 | NEC SX-2 | Cray X-MP/416 | ETA 10G |
|---|---|---|---|---|---|---|
| **Fluid Flow** | | | | | | |
| ADM | 22.3 | **90.6** | **19.3** | 16.1 | **61.1** | 7.3 |
| ARC2D | 499.2 | **682.3** | **118.5** | - | **261.7** | - |
| ARC3D | - | **792.6** | 58.5 | **129.5** | **310.9** | **50.0** |
| FLO52 | 229.7 | **347.4** | **75.6** | 177.1 | **218.7** | **63.2** |
| OCEAN | - | **275.4** | **78.8** | 16.9 | **136.7** | 9.6 |
| SEPC77 | 37.9 | **543.3** | **27.8** | **52.7** | **220.0** | **10.8** |
| **MEAN** | **(51.6)** | **271.3** | **43.1** | **(32.5)** | **150.7** | **13.6** |
| **Chemical & Physical** | | | | | | |
| BDNA | 123.5 | **288.4** | **83.5** | 170.2 | **156.4** | 58.4 |
| MDG | 15.2 | **594.9** | **28.5** | **138.6** | **195.9** | 6.0 |
| QCD | 9.3 | **249.6** | **15.8** | 9.6 | **81.4** | 5.4 |
| TRFD | - | **444.2** | **52.2** | 57.9 | **206.2** | 24.7 |
| **MEAN** | **(16.5)** | **350.7** | **30.9** | **29.7** | **139.7** | **9.7** |
| **Engineering Design** | | | | | | |
| DYFESM | 69.8 | **295.2** | **74.3** | **69.3** | **191.7** | **64.6** |
| SPICE | 5.7 | **18.9** | **6.7** | 4.7 | **14.7** | 2.1 |
| **MEAN** | **10.5** | **35.6** | **12.3** | **8.7** | **27.3** | **4.1** |
| **Signal Processing** | | | | | | |
| MG3D | - | **1146.2** | **149.5** | 35.0 | **453.3** | 6.5 |
| SPICE | 7.4 | **38.7** | **5.4** | 6.2 | **24.7** | 3.7 |
| **MEAN** | **(7.4)** | **74.8** | **10.5** | **10.5** | 46.8 | **4.7** |
| **All Codes** | | | | | | |
| **MEAN** | **(17.1)** | **120.2** | **22.5** | **(18.4)** | **75.9** | **(7.7)** |

Tab. 4.14. PERFECT Benchmark optimized results (in MFLOP/s)

**Summary for PERFECT Benchmarks**

In summary, the pros and cons of the PERFECT benchmarks are:

**Pros:**

A wide area of scientific computation is covered by the 13 applications which can be implemented on parallel systems. They reflect real workloads for high performance systems.

**Cons:**

The problem of the PERFECT Benchmarks is that they are not scalable in terms of problem size. Tab. 4.13. and Tab. 4.14. show that the codes are very sensitive for optimizations which have to be done manually.

#### 4.2.1.4 PARKBENCH

The PARKBENCH (PARallel Kernels and BENCHmarks) committee, originally called the Parallel Benchmark Working Group (PBWG) was founded at Supercomputing '92 in Minneapolis, when a group of about 50 people interested in computer benchmarking met. The objectives of the PARKBENCH group are:

- To establish a comprehensive set of parallel benchmarks that is generally accepted by both users and vendors of parallel systems.
- To provide a focus for parallel benchmark activities and avoid unnecessary duplication of effort and proliferation of benchmarks.
- To set standards for benchmarking methodology and result-reporting together with a control database/repository for both the benchmarks and the results.
- To make the benchmarks and results freely available in the public domain.

The initial focus of the parallel benchmarks is on the new generation of scalable distributed memory message passing architectures for which there is a notable lack of existing benchmarks. For this reason the initial benchmark release concentrates on Fortran77 message passing codes using the widely available PVM message-passing interface [PVM94] for portability.

The releases 2.0 of the benchmark suite adopted the MPI [MPI94] interface, Future releases will include Fortran90 and High Performance Fortran (HPF) versions of the benchmark codes. The committee is currently working on benchmarks for shared memory architectures as well. Currently the PARKBENCH committee offers four different types of benchmarks:

- Low level benchmarks
- Kernel benchmarks.
- Compact application benchmarks.
- HPF compiler benchmarks.

**Low Level Benchmarks**

As the name indicates, these benchmarks are specifically designed to stress certain hardware capabilities of the target system. Two types of low level benchmarks are offered:

- Single processor benchmarks:
  Besides reference to existing monoprocessor benchmarks, PARKBENCH included some low level benchmarks to evaluate the performance of a computing node of the (multiprocessor) target system. These benchmarks help to find the timer resolution and the correct timer value which is important for measurements. Secondly, they provide a test for the basic arithmetic operations performance including a test of pipeline units. The last two low level benchmarks try to explore memory bottlenecks of the system.
- Multiprocessor benchmarks:
  Since the computational performance is evaluated by the PARKBENCH low level single processor benchmarks, the multiprocessor benchmarks provide routines to stress the communication system. The first two routines use so called ping-pong communication with varying message length (messages are sent forth and back between pairs of processors) to characterize the performance for short and long messages. Varying the message length is necessary since some target systems use different message passing protocols for short and long messages. A bandwidth saturation benchmark tries to find the limits of the communication system. Using a "worst case" scenario for communication load, all processors broadcast a message to all other processors at the same time. Executing this test with a varying number of processors points out potential communication bottlenecks and bandwidth limitations of the system. Besides communication the synchronization speed in massively parallel systems is always an important issue. This is addressed by the last low level communication benchmark which tests the time to carry out a barrier synchronization on the target system.

**Kernel Benchmarks**

The low-level benchmark codes are designed to measure the basic architectural features of parallel machines. Full application codes obviously measure the performance of a parallel system on the full problem and this is ultimately what the user wants. However, in many instances, the full application codes are complex, contain many hundreds of thousands of lines of Fortran, and are not available in a suitable parallel version. In order to obtain a guide to the performance of any given parallel system on a particular application something less complex than the full application is useful. A profile of the sequential version of the application enables the compute intensive portions of the program to be identified. It is these compute-intensive sections of an application that are modeled with the parallel kernel benchmarks.

The kernel codes are typically up to a few thousand lines of Fortran and are sufficiently simple that the performance of a given parallel machine on this program may be related to the underlying architectural parameters. It must be acknowledged, however, that the performance on kernels alone is insufficient to completely assess the performance potential of a parallel machine on full scientific applications. The main difficulty is that a certain data structure may be very efficient on a certain system for one of the isolated kernels, and yet this data structure would be inappropriate if incorporated into a larger application. For example, the performance of a real computational fluid dynamics (CFD) application on a parallel system is critically dependent on data motion between different computational kernels. In addition, full applica-

tions typically have initialization phases, I/O and so on, so complete reproduction of these features can be of critical importance for a realistic guide to performance.

To cover a large spectrum of applications run on highly parallel systems the PARKBENCH suite includes:

- Matrix kernel benchmarks such as:
  dense matrix multiply, transpose matrix, dense LU factorization with partial pivoting, QR decomposition, matrix tridiagonalization.
- FFT kernel benchmarks (3-D FFT PDE from the NAS Parallel Benchmark suite).
- PDE (Partial Differential Equations) kernel benchmarks (MG from the NAS Parallel Benchmark suite).
- Other kernel benchmarks (EP, CG, IS) taken from the NAS Parallel Benchmarks.

**Compact Application Benchmarks**

The codes in the compact applications suite should be representative of the fields in which parallel computers are actually used. The codes should exercise a number of different algorithms and possess different communication and I/O characteristics. The compact applications suite is open and expected to grow in the future. Currently two compact application benchmarks are available:

- NPB-CFD codes (NAS Parallel benchmarks computational fluid dynamics codes).
- PSTSWM (Parallel Spectral Transform Shallow Water Model code, compare section 6.3.3.).

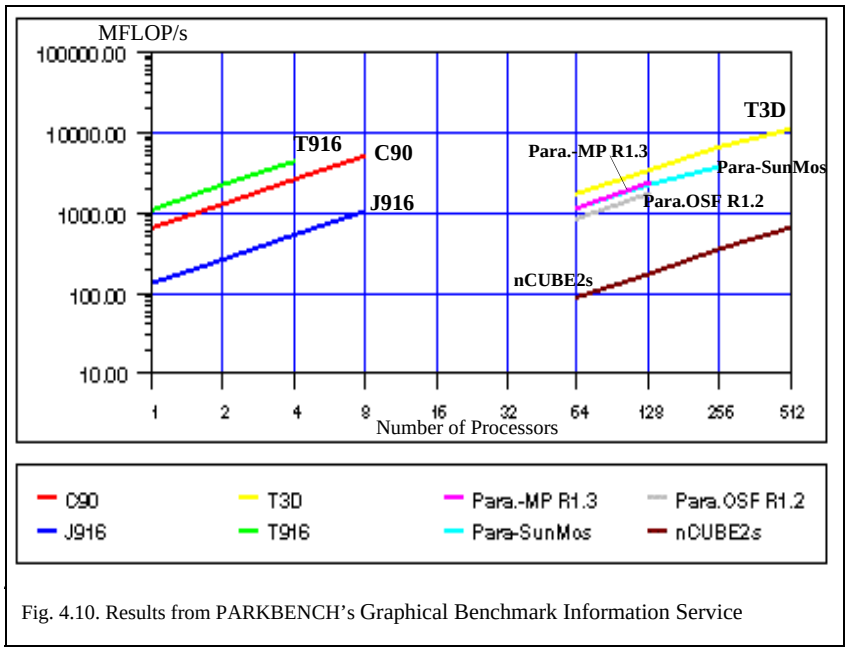



Fig. 4.10. Results from PARKBENCH's Graphical Benchmark Information Service

**HPF Compiler Benchmarks**

The benchmark suite comprises several simple, synthetic applications which test several aspects of HPF compilation. The current version of the suite addresses the basic features of HPF, and it is designed to measure performance of early implementations of the compiler. They concentrate on testing parallel implementation of explicitly parallel statements, i.e., array assignments, *FORALL* statements, *INDEPENDENT DO* loops, and intrinsic functions with different mapping directives. In addition, the low level compiler benchmarks address the problem of passing distributed arrays as arguments to subprograms. The language features not included in the HPF subset are not addressed in the current release of the suite. The codes included in the current suite are either adopted from existing benchmark suites, NAS suite, Livermore Loops, and the Purdue Set, or are developed at Syracuse University.

**Results**

Results from PARKBENCH are available via the Graphical Benchmark Information Service (GBIS) at two sites:

The University of Tennessee in the United States under the URL:
*http://www.netlib.org/parkbench/gbis/html/*

The University of Southampton in the United Kingdom under the URL:
*http://www.ccg.ecs.soton.ac.uk/gbis/papiani-new-gbis-top.html*

These sites contain results for a wide variety of multiprocessors. Being equipped with a graphical user interface it is simple to view result data. Fig. 4.10. shows an example for a result graph. The benchmark used to produce this figure is *3-D FFT PDE* from the PARKBENCH kernel benchmarks (originally taken from the NAS Parallel Benchmarks suite). The target systems are CRAY-T3D, INTEL Paragon, and nCUBE/2 (compare section 6.1.3.).

**Summary for PARKBENCH Benchmarks**

In summary, the pros and cons of the PARKBENCH benchmarks are:

**Pros:**

Different workload types from low level to application programs are considered to offer tests for different users and purposes. The PARKBENCH's Graphical Benchmark Information Service makes it easy to access performance measures for a wide variety of machines. Also, the objectives of the PARKBENCH group which are described in the introduction of this section are all realized.

**Cons:**

The problem of the PARKBENCH Benchmarks is that they are not scalable in terms of problem size.

#### 4.2.1.5 GENESIS Benchmarks

The GENESIS Benchmark [GEN91] suite originated within ESPRIT project P2702 (GENESIS) with the codes being written by a number of the project partners. It is a distributed-memory benchmark suite which tries to evaluate the performance of distributed memory MIMD systems on scientific and engineering applications. A Graphical Bench-

mark Information Service (GBIS)
*http://www.ccg.ecs.soton.ac.uk/gbis/papiani-new-gbis-top.html*
was set up to act as a repository for the benchmark suite, a source of papers and information concerning the benchmarks and a database of benchmark results.

Most of the benchmarks are message passing codes written in Fortran 77, using mainly double precision arithmetic. There are two different versions of each message passing code, one using the PARMACS 5.1 message passing macros and the other using PVM 3.x. There are also data parallel versions of some of the codes, written in High Performance Fortran (HPF), and sequential versions where applicable. The benchmarks are divided into three categories:

- low level codes
- application kernels
- compact applications

As the three categories already indicate, the GENESIS Benchmark suite is very similar to the PARKBENCH benchmark from the previous section. The only difference is its focus on distributed memory architectures. The GENESIS Benchmark suite is available at anonymous *ftp.par.soton.ac.uk*. The GENESIS files are located in the directory: *pub/genesis*.

**Summary for GENSIS Benchmarks**

In summary, due to the similarity of the two approaches, the pros and cons of the GENESIS do not differ from the pros and cons of the PARKBENCH benchmarks.

#### 4.2.1.6 SPLASH-2

The Stanford Parallel Applications for Shared Memory (SPLASH) [SPL91] suite is a set of parallel applications designed to facilitate the performance study of shared memory multiprocessors. The properties being investigated are:

- *Concurrency and load balancing:*
  The concurrency and load balancing characteristics of a program indicate how many processors can be effectively utilized by that program, assuming a perfect memory system and communication architecture. This indicates whether a program with a certain data set is appropriate for evaluating the communication architecture of a machine for a given scale. For example, if a program does not speed up well, it may not be appropriate for evaluating a large scale machine.
- *Communication to computation ratio:*
  The communication to computation ratio indicates the potential impact of communication latency on performance, as well as the potential bandwidth needs of the application. Additionally, the total communication traffic and local traffic is characterized for a set of architectural parameters.
- *Working set:*
  The working sets of a program [Den68], [RSG93] indicates its temporal locality. They can be identified as the knees in the curve of cache missrate versus cache size. Knowledge of the working sets can help to prune the cache size dimension of the parameter space.
- *Spatial locality:*
  The spatial locality in a program has tremendous impact on its memory and communication behavior. In addition to the single processor trade-offs in using long cache lines (prefetching, fragmentation and transfer time), cache coherent multiprocessors have the potential drawback of false sharing, which causes communication and can be very expensive.

Similar to the NAS and PARKBENCH benchmarks, the SPLASH-2 [SPL95] suite consist of a mixture of complete applications and computational kernels. It has eight complete applications and 4 kernels, which represent a variety of computations in scientific, engineering, and graphics computing.

**Applications:**

- Barnes
  The Barnes application simulates the interaction of a system of bodies in a three-dimensional space over a number of time steps, using the Barnes-Hut hierarchical N-body method.
- FMM
  Like Barnes, the FMM application also simulates a system of bodies over a number of time steps. However, it simulates interactions in two dimensions using a different hierarchical N-body method called the adaptive Fast Multipole Method [Gre87].
- OCEAN
  A red-black multigrid Gauss-Seidel solver [Bra77] is used to study large-scale ocean movements based on eddy and boundary currents.
- Radiosity
  The equilibrium distribution of light in a scene is computed. Radiosity uses the iterative hierarchical diffuse radiosity method [HSA91].
- Raytrace
  A three dimensional scene is rendered using ray tracing [SGL94].

- Volrend
  A three dimensional scene is rendered using a ray casting technique [NiL92].
- Water-Nsquared
  This application evaluates forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed using an $O(n^2)$ algorithm, and a predictor-corrector method is used to integrate the motion of the water molecules over time.
- Water-Spatial
  The same problem as in Water-Nsquared is solved using a more efficient O(n) algorithm for large number of molecules.

**Kernels:**

- Cholesky
  Factorization of a sparse matrix into the product of a lower triangular matrix and its transpose.
- FFT
  Complex 1-D version of the radix-$\sqrt{n}$ six step FFT algorithm [Bai90].
- LU
  Factorization of a dense matrix into the product of a lower triangular matrix and an upper triangular matrix.
- Radix
  Iterative integer radix sort kernel [Ble91].

All kernels and applications are scalable with respect to problem size and numbers of processors. Tab. 4.15. contains a breakdown of instructions executed for default problem sizes on a 32 processor machine. Instructions executed are broken down into total floating point operations across all processors for applications with significant floating point computation, reads and writes. The number of synchronization operations is broken down into number of barriers (column B in Tab. 4.15.) encountered per processor, and total number of locks (column L in Tab. 4.15.) and pauses (flag based synchronizations, column P in Tab. 4.15.) encountered on all processors.

More details on SPLASH-2 and results can be found in [SPL95].

| | Problem Size | Total Inst. (M) | Total FLOPS (M) | Total Reads (M) | Total Writes (M) | Shared Reads (M) | Shared Writes (M) | B | L | P. |
|---|---|---|---|---|---|---|---|---|---|---|
| **Application Code** | | | | | | | | | | |
| Barnes | 16 K part. | 2002.79 | 239.24 | 406.85 | 313.29 | 225.05 | 93.23 | 8 | 34648 | 0 |
| FMM | 16 K part. | 1250.02 | 423.88 | 226.23 | 38.58 | 217.84 | 30.10 | 20 | 28088 | 0 |
| Ocean | 258x258 ocean | 379.93 | 101.54 | 81.89 | 18.93 | 80.26 | 17.27 | 364 | 2592 | 0 |
| Radiosity | room,-ae 5000 -en 0.05 -bf 0.1 | 2832.47 | --- | 499.72 | 284.61 | 261.08 | 21.99 | 10 | 231190 | 0 |
| Raytrace | car | 829.32 | --- | 208.90 | 79.95 | 159.97 | 22.22 | 0 | 94471 | 0 |
| Volrend | head | 754.77 | --- | 152.19 | 59.57 | 81.93 | 3.07 | 15 | 28934 | 0 |
| Water-N. | 512 molec. | 460.52 | 98.15 | 81.27 | 35.25 | 69.07 | 26.60 | 10 | 17728 | 0 |
| Water-S. | 512 molec | 435.42 | 91.50 | 72.31 | 32.73 | 60.54 | 22.64 | 10 | 353 | 0 |
| **Kernel Code** | | | | | | | | | | |
| Cholesky | tk 15.0 | 539.17 | 172.00 | 111.86 | 28.03 | 75.87 | 23.31 | 3 | 54054 | 4203 |
| FFT | 64K points | 34.79 | 6.36 | 4.07 | 2.88 | 4.05 | 2.87 | 6 | 0 | 0 |
| LU | 512x512 matrix | 494.05 | 92.20 | 104.00 | 48.00 | 93.20 | 44.74 | 66 | 0 | 0 |
| Radix | 1 M integers | 50.99 | --- | 12.06 | 7.03 | 12.06 | 7.03 | 10 | 0 | 124 |

Tab. 4.15. Characteristics of SPLASH-2 applications and kernels

**Summary for SPLASH Benchmarks**

In summary, the pros and cons of the SPLASH benchmarks are:

**Pros:**

Different workload types consisting of applications and kernels for shared memory multiprocessors are considered to offer tests for different users and purposes. All workloads are scalable with respect to problem size and number of processors. The number of synchronizations and the number of accesses to shared data for all workloads are given as a measure of the communication load produced by the parallel programs.

**Cons:**

SPLASH benchmarks as the name already indicates are designed for shared memory multiprocessors and thus, they cannot be used for distributed memory architectures.

#### 4.2.1.7 SLALOM

All previous benchmark approaches defined some (scalable or parametrized) workload to implement it on the target system and monitor its execution. The results are execution times or performance measures such as MFLOPS or MIPS. The SLALOM (Scalable, Language-independent, Ames Laboratory, One-minute Measurement) approach tries to answer a different question, namely "How big a SLALOM problem will my computer solve?". The advantage of this approach is that it makes it possible to compare a very wide range of computers from the low end of performance to powerful multiprocessors.

Fortran, C, and Pascal definitions of the revised benchmark are available, with variants for SIMD, shared-memory MIMD, distributed-memory MIMD, and vector computers, through *anonymous ftp* to a workstation at Ames, *tantalus.al.iastate.edu*.

The application used to generate the workload is a radiosity program as described in the previous chapter. The program is available in different languages (C, Fortran, PASCAL) and changes due to code optimization are allowed. The reason for the choice of radiosity were scalability, radiosity involves solving a nearly-dense system of equations, has I/O and set up costs, and appears to lack hidden "shortcuts" that might be unevenly exploited.

In contrast to the previously discussed benchmarks, SLALOM also takes I/O and the setup of the problem into account. Constructing an N by N dense matrix takes an order of $O(N^2)$ steps, whereas solving that matrix requires $O(N^3)$ steps. For large N, the solving will dominate the benchmark. Yet, the operations needed to set up each element is in the hundreds, so the operation counts are roughly in balance when N = 200. The slower machines are spending much of the time setting up the problem instead of solving it. For the I/O operations, the input is in the form of reading a geometry file from mass storage. The output consists of writing the answer (position, size, and color of every patch) to mass storage.

The performance is fed back to the user for each problem size, in a form that summarizes and profiles the run. Tab. 4.16.shows the result for a typical SLALOM iteration. The residual is shows the accuracy of solution. it is defined as usual:

$$residual = \frac{||Ax - b||}{(||A||||x||)}$$

The results of the SLALOM benchmark are verified. The setup of the matrix is cross-checked by seeing that the rows sum to unity. The residual of the matrix solution is found after the job is done. Lastly, answer files can be compared for small and large problem sizes, against examples maintained with the program versions. Also, displaying the result graphically can quickly show errors.

| Task | Seconds | Operations | MFLOPS | % of Time |
|---|---|---|---|---|
| Reader | 0.001 | 258. | 0.025800 | 0.0% |
| Region | 0.001 | 1148. | 1.148000 | 0.0% |
| SetUp1 | 10.520 | 20532123. | 1.951723 | 17.8% |
| SetUp2 | 23.130 | 39372520. | 1.702227 | 39.1% |
| SetUp3 | 0.130 | 236856. | 1.821969 | 0.2% |
| Solver | 24.890 | 135282624. | 5.435220 | 42.1% |
| Storer | 0.480 | 25440. | 0.053000 | 0.8% |
| **TOTAL** | **59.160** | **195425529.** | **3.303339** | **100.0%** |

residual is 2.2566160051696E-12
Approximate data memory use: 2311600 bytes

Tab. 4.16. SLALOM benchmark results for single processor SGI 4D/380S

In the SLALOM benchmark the problem execution time is fixed at 60 seconds. The benchmark has logic to time itself and adjust automatically to find that problem size, or the user can do the trial-and-error manually.

The smallest SLALOM run possible is one with only 6 patches: one for each face of the box. That job has 8812 floating-point operations, so a computer must be capable of at least 148 FLOPS to run SLALOM in under a minute.

The performance of the system under test is measured in number of patches that were solved. Tab. 4.17. contains SLALOM results for some single and multiprocessor machines. MFLOPS are also quoted in this table for continuity with earlier benchmarks, but the number of patches determine the rank. See the MasPar versus the NCUBE 6400 ranking, for example. More information on SLALOM can be found in [Gus91] and [Gus92].

**Summary for SLALOM Benchmarks**

In summary, the pros and cons of the SLALOM benchmarks are:

**Pros:**

The idea to benchmark a system through the amount of work which can be done in a given time interval sounds very reasonable. Compared to many other benchmarks, the time to set up a problem and to output the result is also included. The workload is scalable and well suited for a wide range of computing systems.

**Cons:**

The problem of the SLALOM Benchmarks is that only one workload is considered. The conclusions from this workload are not

enough for a performance evaluation of a computing system.

| Machine, environment | Procs. | Patches | Ops. | Secs. | MFLOPS |
|---|---|---|---|---|---|
| **Cray Y-MP/8, 167 MHz** | | | | | |
| Fortran+tuned LAPACK solver | 8 | 5120 | 126 G | 59.03 | 2130. |
| **Cray 2S/8-128, 244 MHz** | | | | | |
| Fortran+directives, FPP 3.00Z25 | 8 | 2443 | 14.4 G | 59.83 | 240. |
| **NCUBE 6400, 20 MHz** | | | | | |
| Fortran+assembler | 64 | 1438 | 2.83 G | 59.95 | 47.2 |
| **MasPar MP-1, 12.5 MHz** | | | | | |
| C with plural variables (mpl) | 8192 | 1407 | 2.93 G | 57.65 | 50.8 |
| **Silicon Graphics 4D/380S, 33 MHz** | | | | | |
| Fortran (-O2 -mp -lparalin) | 8 | 1010 | 1.15 G | 59.85 | 19.2 |
| **IBM RS/6000 POWERstation 320** | | | | | |
| Fortran (xlf -O -Q) | 1 | 642 | 328. M | 59.+ | 5.6 |
| **iPSC/860, 40 MHz** | | | | | |
| Fortran (-OLM -i860) | 1 | 419 | 105. M | 59.91 | 1.75 |
| **Myrias SPS2 (mc68020, 16.7 MHz)** | | | | | |
| Fortran (mpfc -Ofr) | 64 | 399 | 92.2. M | 59.26 | 1.56 |
| **SUN 4/370, 25 MHz,** | | | | | |
| C (ucc -O4 -dalign etc.) | 1 | 380 | 81.1 M | 59.85 | 1.35 |
| **NCUBE 2, 20 MHz** | | | | | |
| Fortran + assembler subroutines (-O2) | 1 | 354 | 67.5 M | 59.73 | 1.13 |
| **DECStation 2100, 12.5 MHz,** | | | | | |
| Fortran (f77 -O2) | 1 | 285 | 38.8 M | 59.72 | 0.649 |
| **Cogent XTM (T800 Transputer)** | | | | | |
| Fortran 77 (-O -u) | 1 | 149 | 7.89 M | 59.37 | 0.133 |

Tab. 4.17. SLALOM benchmarks results

#### 4.2.1.8 The SPEC High-Performance Group

In 1994, the Standard Performance Evaluation Corp. (SPEC) established the High Performance Group (HPG), which establishes, maintains and endorses a suite of benchmarks that represent high-performance computing applications for standardized, cross-platform performance evaluation. The benchmarks target high performance system architectures, such as symmetric multiprocessor systems, workstation clusters, distributed memory parallel systems, and traditional vector and vector parallel supercomputers.

The SPEChpc96 suite includes two application areas:

- seismic processing (SPECseis96), an industrial application that performs time and depth migrations used to locate gas and oil deposits
- and computation chemistry (SPECchem96) which contains GAMESS (General Atomic and Molecular Electronic Structure System). Many of the functions found in GAMESS are duplicated in commercial packages used in the pharmaceutical and chemical industries for drug design and bonding analysis.

The applications are scalable in terms of problem size, input data sets for a small, medium, large, and extra large problem are provided. For SPECseis96, problem sizes relate directly to the number and the type of analysis the application performs. For SPECchem96, problem sizes relate to the complexity of the molecule under analysis. Additional applications from CFD (computational fluid dynamics), molecular dynamics, and weather predictions are under investigation for future releases of SPEChpc suites.

An overview on the structure and members of the SPEC HPG is given in Fig. 4.11., additional information and results are available at:
*http://www.specbench.org/hpg/*

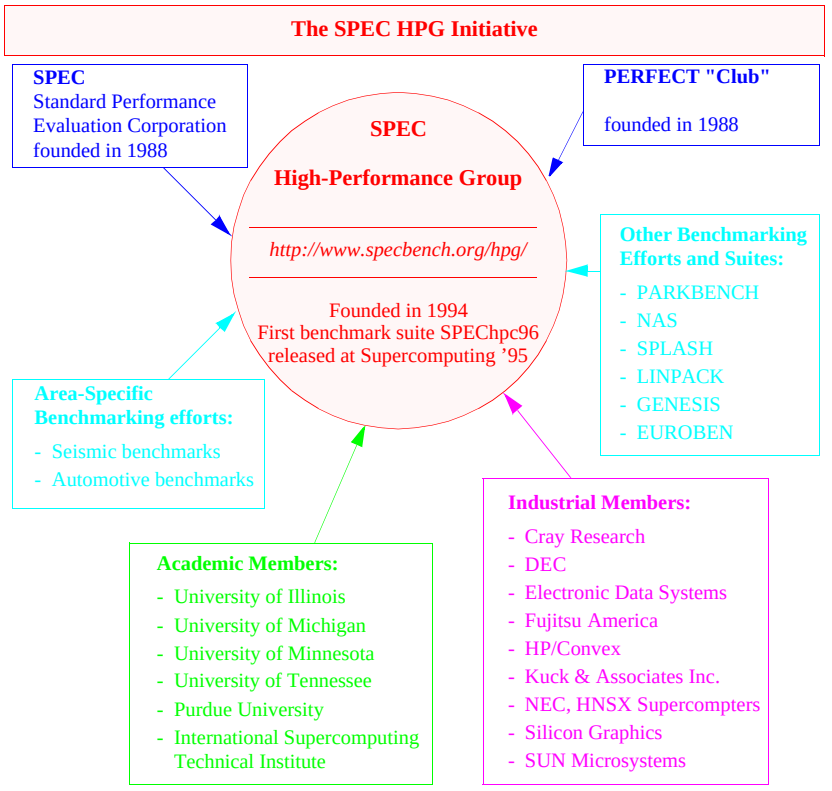



Fig. 4.11. The SPEC High-Performance Group

### 4.2.2. LOOP Programs

The number of multiprocessor architectures has substantially increased in the last years, so did the efforts to evaluate these machines. The popular kernel benchmarks that have been used for traditional vector supercomputers, such as the Livermore Loops, the LINPACK benchmark and the original NAS kernels, are clearly inappropriate for the performance evaluation of highly parallel machines. First of all, the tuning restrictions of these benchmarks rule out many widely used parallel extensions. More importantly, the computation and memory requirements of these programs do not do justice to the vastly increased capabilities of the new parallel machines. Another problem is that none of the previously discussed approaches is feasible for a wide range of existing multiprocessors or is adaptable to user defined workloads.

The LOOP approach describes a portable high level workload description language (LOOP language) for parallel systems. To automatically produce program code for the different systems, a program generator was developed that translates the LOOP computation and the LOOP communication instructions into instrumented parallel C code. The instrumentation results in trace data that are visualized by appropriate tools. Thus, a user can evaluate a system for the specific workloads of his applications. Additionally, LOOP descriptions of a set of parameterized workloads are part of the LOOP benchmark package. These workloads are used to compare different systems.

In the field of parallel computing a broad black box approach with fixed workloads (e.g., as used by SPEC) is no longer adequate. Some knowledge of the machine architecture always influences the design of the application program. The *LOOP* method introduced in the next section allows certain machine dependent optimizations.

The term *parallel systems* used here refers to *massively parallel computer systems* and not to architectures such as multiprocessor workstations. Benchmarking the latter is similar to the evaluation of single processor architectures. These environments normally consider a number of processes per processor with little communication between them. They are programmed in a code-parallel manner. Besides evaluating pure processor performance, benchmark tests must determine processor capabilities, e.g., how many processes can be handled at the same time and how long are the context switches. Under this scenario workload mixes consisting of conventional benchmarks can be used, as long as it is guaranteed that all available processors have some computational work to do. Workstation networks running applications in an SPMD (single program multiple data) mode can also be considered as massively parallel systems and can thus be evaluated using the LOOP approach.

The situation in evaluating massively parallel computers is more difficult than for single processor architectures. Standard benchmark tests cannot be used for such systems since they have not been specially designed for these architectures. Special algorithms are required since applications normally are tuned to certain processor or cache topologies. Contrary to single processor architectures, massively parallel machines are typically not stressed under normal workload conditions. This creates the need for a new kind of benchmark. The communication features of the system should be evaluated, and special workload characteristics should be described. The remainder of this section summarizes the

approach of a new benchmark that is able to evaluate the overall system performance of massively parallel computer systems. The advantages of this *parameterized benchmark* over previous benchmarks are given, and the methods by which the new approach can be applied are explained.

#### 4.2.2.1. Using Parameters

Most of the existing benchmark tests described previously do not allow the use of parameters by which the user load can be calibrated. This restricts the influence that a user has on the execution of a benchmark program. This restriction is useful to make sure that results are uniform and comparable. On the other hand, the user is bound to a program which probably does not represent the same workload as the specific application of interest. Giving the user the ability to adapt the behavior of a certain application enables the benchmark to mimic the behavior of the described program. Two approaches are possible.

A first and rather static approach is to create a single program that is able to change its behavior according to input parameters from the user. Such a program can change its behavior in a restricted manner.

Another, more flexible approach is to develop a program that not only makes use of these parameters, but also *generates* different programs. These synthetic programs are then to comprise the benchmark workload. This implies the creation of a Benchmark Generator rather than developing a benchmark program in isolation.

In both cases, the use of parameters has the important advantage that only a single program has to be ported to different machines in order to obtain a wide variety of synthetic workloads. This implies that a user does not have to port an application program to the new architecture to investigate its behavior. By incorporating several scaling parameters it is possible to simulate the application's (communication and computation) behavior under different conditions.

A second important advantage of this approach is the fact that special features of a system's performance can be tested individually. For example, different kinds of message patterns can be generated by manipulating message size and frequency parameters.

The benchmark generator also has another advantage over simply porting special applications and using them as benchmark programs. Evaluation facilities and tracing capabilities can automatically be included. Using a set of parameters to describe an application implies a trade-off between the conflicting goals of easy usability and model representativeness. A large number of parameters makes it easier to create a workload with behavior close to the application from which these parameters are derived. However, the extra parameters add to the complexity of the benchmark. Ideally, the benchmark should be characterized by a small set of parameters while not sacrificing representativeness.

#### 4.2.2.2. The *LOOP* Method

In the LOOP approach [Bre94], [Sch93] workloads are not defined using one specific workload described in detail. Instead, an environment for a user specified evaluation of parallel computer systems is provided. The *LOOP method* has been developed assuming that the user has structural knowledge of the intended workload. The benchmark generator then constructs a workload with the same structural characteristics.

It is often useful to obtain a first impression of a new algorithm's behavior on a known machine. The exact amount of code related to communication handling does not have to be specified. Instead, one can concentrate on the algorithm itself. Some predefined standard workloads described in later sections can be used to get a first impression of the system without the need to fully implement a user specific application.

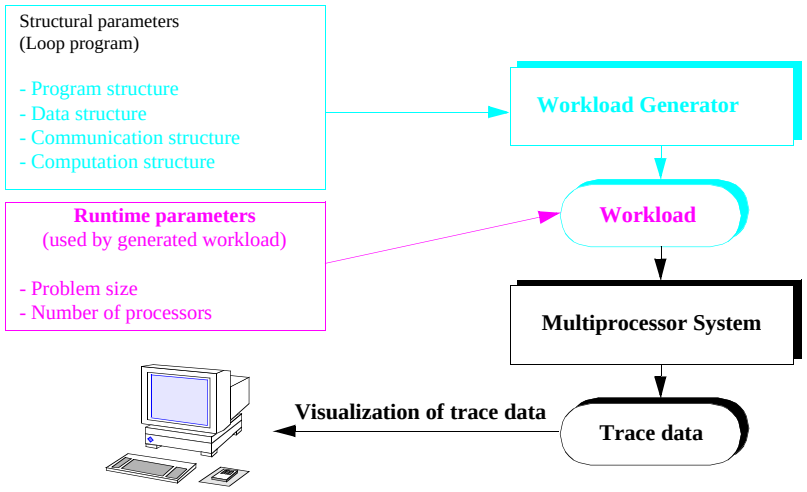



Fig. 4.12. Workload generation with the LOOP approach

In Fig. 4.12., the LOOP approach is illustrated. A structural load description (LOOP program) is fed into the generator. The generator produces the corresponding parallel instrumented program. The program can be run on the target architecture with different input parameters and the behavior can be analyzed using collected trace information. The central part of the *LOOP* method is the Workload Generator. This generator is the only program that has to be ported to a new machine to test the new machine with a wide variety of workloads.

Although the problems evaluated using the *LOOP method* can be defined at a high level of abstraction, the use of a powerful visual-

ization tool allows the user to examine such things as the communication structure in detail. Communication bottlenecks in the hardware or in the chosen algorithm can be detected. It can be determined if a certain network topology is suitable for a problem with specific characteristics. Examples are described in later sections.

Given a description of the workload and the architecture to be tested, the workload generator constructs a program in standard C which is executed on the target system. To have a widely accepted communication model, the generator uses the PICL library [PICL90] (Portable Instrumented Communication Library). Besides the ability to write portable code for massively parallel systems, this library is capable of tracing basic communication instructions. PICL trace information can be analyzed with a visualization tool, ParaGraph [Para92].

The main goal in the design of the LOOP model is to ensure ease of use. This is accomplished by making the description of the workload significantly more concise than the user's actual application. In situations where a new architecture is to be evaluated, this is especially important. Another important goal in the LOOP design is to make the programming of parallel program communication for message passing systems as easy as possible.

In the LOOP method, the user can describe programs in a pseudo-code like manner similar to that often found in literature, for example [Gol83], [Pre88]. To accomplish this, the *LOOP* language is an extension to standard C and PICL [Sch93]. *LOOP* constructs and data structures can easily be manipulated. The implementation of the abstract constructs for the description of communication workloads guarantees deadlock-free workloads, because sender and corresponding receiver are automatically addressed as pairs.

### How to write *LOOP* programs

The first step in the evaluation of a parallel system with the *LOOP* method is to give structural information of the desired workload to the generator. Based on this information, the generator produces executable code for the target system. By using the *LOOP* language, all structural information is given.

In a typical experiment, it is often useful to execute the same workloads with different problem sizes and with varying number of processors. Such sensitivity analysis finds limitations in the hardware regarding memory or cache behavior. Therefore, at runtime certain parameters can be specified to the generated workload. This implies that even without rebuilding the program, problem size limitations of algorithms or hardware can be tested.

### Structural Parameters

The structural parameters are specified via *LOOP* language constructs. In this section, the most important constructs are explained and their usage is demonstrated via examples.

### Programs and Data Structures

Considering problems normally solved on massively parallel systems, numerical applications are arguably the most important. Numerical programming problems mainly deal with operations on matrices and vectors. Programs for these kinds of problems, therefore, consist of iterations over matrices and vectors. For this reason, the design of the *LOOP* language focuses on offering convenient ways to describe such operations.

Along with the \LOOP construct which determines loop nesting, several instructions handling high-level data structures are available. In Fig. 4.13., a LOOP program abstraction of a simple parallel matrix multiplication is shown. The use of the construct \LOOP and the declaration of high-level data structures can be seen. The communication is specified via the \COMMUNICATE statement which is explained later. The declared matrices are allocated dynamically by the \MAT construct and initialized as specified by the \INIT_ARRAY construct. This Initialization

can be omitted if specific array values are not required.

All *LOOP* language instructions are prefixed by the sign \. The generator recognizes these tagged backslash instructions and converts them to normal C. This implies that additional C code can be incorporated directly into the LOOP program.

The construct

\LOOP [iterator]

in Fig. 4.13. is used to describe iterations over the complete problem size. The use of iteration variables (i1 - i3 in our example) is optional. The variables can be omitted if they are not needed. The default number of LOOP iterations is the problem size. No definition or initialization for the iterators and the matrices is necessary. Iteration variables are automatically declared by their use in the LOOP construct. High-level data structures like matrices are declared and initialized by using special instructions which, in our example, are the constructs

\MAT and \INIT_ARRAY.

The construct

\LOOP [iterator]

in Fig. 4.13. is used to describe iterations over the complete problem size. The use of iteration variables (i1 - i3 in our example) is optional. The variables can be omitted if they are not needed. The default number of LOOP iterations is the problem size. No definition or initialization for the iterators and the matrices is necessary. Iteration variables are automatically declared by their use in the LOOP construct.

The code given in Fig. 4.13. is the complete input for the generator to produce a parallel, parameterized program for a matrix multiplication. The result of the generator is a parallel instrumented program (PIP). The benchmark package includes a user friendly interface that aids in all phases of the machine evaluation. After the PIP is generated, the code is compiled and executed. A tracefile is collected and written to disk.

```
int main(void){
  int nprocs, me, host;
  int psize, amount;
  \OPEN0
    (&nprocs,&me,&host, &psize,
      TRACE_BUF_SIZE);
  /* Declarations: */
  \Mat(double) Mat1;
  \Mat(double) Mat2, Mat3;
  /*random init */
  \INIT_ARRAY(Mat1);
  \INIT_ARRAY(Mat2);
  \INIT_ARRAY(Mat3, 0);
  amount= sizeof(double)*
      psize*psize/nodes;
  \LOOP i3 {
    \LOOP i2 {
      \LOOP i1 {
      Mat3[i3][i2] +=
       Mat1[i3][i1]*
       Mat2[i1][i2];
      }
    \COMMUNICATE(amount,1,1);
  }}
  \CLOSE0();
}
```

Fig. 4.13. Complete LOOP program for a parallel matrix multiplication

### Computational Load

In the above example the computational load results from the statement

```
Mat3[i3][i2] +=
    Mat1[i3][i1]*Mat2[i1][i2]
```

in the inner loop. Often the user may not be able to give the exact statements generating the desired computational load. In such cases it is important to have a set of statements with which the computational behavior can be described. Since the resulting computational load depends upon the position in the loop hierarchy, these statements have to be determined for each loop level of the program structure. Three examples of such statements illustrate various options.

\MATPROD

indicates the calculation of a matrix multiplication. The type of operations can be determined by the declaration of the matrices which are to be multiplied. The sizes of the matrices (submatrices) can be given as arguments.

\MATVECPROD

is similar to the matrix multiplication instruction and generates code for a matrix-vector product.

\SCALPROD

calculates a scalar product. Two vectors and the name of the resulting scalar are given as arguments.

These and other operations often used in massively parallel programming are provided to make the description of the computational load easy. Since various data structures can be specified it is easy to generate different classes of workloads. Using these constructs, it is possible, for example, to describe a diverse set of abstract tests for integer and floating point performance.

### Modeling Communication

The design of a workload for multiprocessor machines should include a description of workload placed on the interconnection network. Several goals are discussed in the following.

First, it is important to have a simple, easy to use description of several communication related parameters. Such parameters include:

- the frequency and type of messages,
- the size and pattern of messages, and
- the locality and communication distance, including the sending and receiving nodes.

Another goal, related to the programming of message-passing architectures, is to make the communication code deadlock-free. The development of such code is problematic since statements for receiving messages are normally blocking. In order to make the code deadlock-free, a mechanism is needed which assures that an adequate number of messages are sent to nodes which are waiting to receive.

Describing communication in an abstract way implies that neither setup routines nor special point-to-point programming should be necessary. The *LOOP* language provides high level instructions from which the generator is able to produce correct communication code for each node. These are motivated through the following example.

### The COMMUNICATE Statement

One common communication function is the transfer of messages between several processor nodes of a given distance. In parallel computing, situations can be found in which several processors of a certain distance communicate regularly. Nodes typically send results to another node and receive new data from a third node.

As an example, the simple parallel matrix multiplication LOOP program of Fig. 4.13. is considered. The underlying algorithm is described in Fig. 4.14.

Each node performs the following step number of nodes times:

- gets part of A and B
- computes a submatrix of C
- sends own part of B to another node
- receives new part of B from a third node

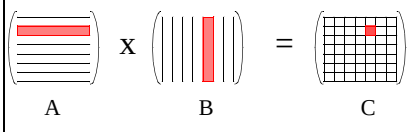



Fig. 4.14. Structure of a parallel matrix multiplication

In this figure, it is shown that both matrices A and B are distributed in blocks of rows and columns, respectively. Each node is able to compute a certain sub-matrix of the resulting

matrix C. Having calculated the sub-matrix, it is necessary that each node sends its column block of matrix B to another node and receives a new block of columns from a third node. Typically, the blocks of columns are exchanged cyclically.

Reconsidering the LOOP program shown in Fig. 4.13., the communication can be modeled with the

\COMMUNICATE (size, distance, partners)

instruction. The arguments specify the amount of data which has to be communicated, the distance between the communicating processors, and the number of processors to which the data shall be transferred, respectively. Thus, the

\COMMUNICATE(amount,1,1)

statement in Fig. 4.13. indicates that a certain number of matrix entries (amount) has to be transferred to one partner of distance one. Optionally, the \COMMUNICATE statement can be given a compound statement. Such compound statements indicate cases in which the low level send and receive operations generated by the high level \COMMUNICATE instruction are positioned at different places in the parallel code. The sending operations are done before the execution of the compound statement and the receiving of the messages is done afterwards. Inside the compound statement a certain amount of computation may be performed. Because of the non blocking send operation, overlapping between communication and computation can be achieved. Generally speaking, with the \COMMUNICATE instruction it is possible to check the performance of the interconnection network with respect to

- the communication distance and
- the message length.

**Information Exchange**

Similar to the \COMMUNICATE construct, the \EXCHANGE statement generates communication between two processors of a certain distance. Although it might initially seem possible, this construct cannot be replaced by the use of two \COMMUNICATE instructions which generate only very few bidirectional communications by coincidence. The parameters *size*, *distance* and *partners* of the \COMMUNICATE statement do not allow to specifically allocate pairs of processors for \COMMUNICATE statements executed on two processors. Therefore, a construct which generates only bidirectional communication between pairs of processors is provided. The construct

\EXCHANGE (size, distance, partners)

generates bidirectional communication by sending and receiving messages of length ‘size’ between ‘partners’ (i.e., processors) of a certain communication ‘distance’.

In parallel linear algebra and image processing there are several algorithms which make use of data exchange between pairs of processors. As an example, the principles of a parallel red-black relaxation algorithm are considered.

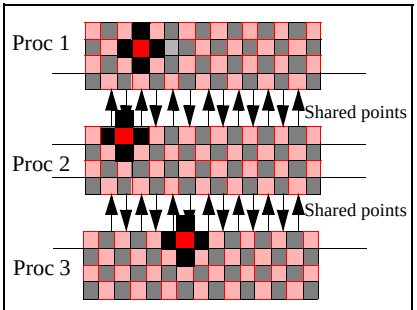



Fig. 4.15. Parallel red-black relaxation

All elements of the matrix are marked in a chess-board like manner using the colors red and black. Each processor gets a part of the matrix as shown in Fig. 4.15. In each iteration step, all elements of a processor’s submatrix are recalculated. New values are calculated by using a function which takes a certain neighborhood of the point into account. This means for each point, a statement

```
P'[i,j] = f(P[i][j],
          P[i-1][j], P[i+1][j],
          P[i][j-1],P[i][j+1] )
```

has to be calculated.

Red elements are recalculated first in which elements shared between two processors have to be exchanged. After that, recalculation and exchange of black elements is done. This process continues until changes in the matrix values are below a certain error threshold where the algorithm is assumed to have found the solution to the problem.

```
int main(void)
{
int amount, host, me;
int psize, nprocs;
int psize_node =
        (int) ( psize / nodes );
\MAT (TYPE: MY_TYPE,
        S1: psize_node) Mat;
\OPEN0(&nprocs,&me,&host,
        &psize, T_BUF);
\INIT_ARRAY(Mat);
amount=psize*sizeof(MY_TYPE);
\LOOP (ITERATIONS) {
   \EXCHANGE ( amount, 1, 2 );
   relax (0, psize, psize_node);
   \EXCHANGE ( amount, 1, 2 );
   relax (1, psize, psize_node);
} /* END LOOP */
\CLOSE0();
return EXIT_SUCCESS;
}
```

Fig. 4.16. LOOP program for parallel red-black relaxation

A *LOOP* program which is capable to model this algorithm is shown in Fig. 4.16. The subroutine relax performs the iteration of one color (red or black) on the matrix. The function for the computation of new elements is the arithmetic mean of the four neighboring elements. In contrast to a real algorithm which would terminate if the error bound between two iteration steps is smaller than a given limit, this example iterates a distinct number of times. At runtime the user can specify the amount of iterations by defining the value of *iterations*.

Besides placing computational load on each processor, the program as it is described in Fig. 4.16. stresses the interconnection network with a large amount of bidirectional messages expressed by the \EXCHANGE construct. Here it can be seen how some normal C constructs are integrated in the LOOP program. The distribution of the matrix itself and other setup overhead is not regarded in this example. To handle the distribution of data some high-level communication routines as shown below are provided.

### High-level Communication Routines

Two major types of high-level communication routines are provided: multi-broadcast and vector communication kernels. The first one has been implemented to place a massive load on the interconnection network. All of these massively communicating routines are implemented by sending a certain amount of data from all nodes to all others. This functionality is called a multi-broadcast operation. The *LOOP* package has three slightly different implementations of this multi-broadcast statement.

\MULTIBCAST0(amount)

All nodes start by performing all *send* operations first; after that all *receives* are executed. The amount of data sent by each node is given as an argument. All nodes begin by sending to node 0 and proceed by sending to the remaining nodes in numerical order. (Nodes do not send messages to themselves). Message receiving is done in the same order starting at node 0.

\MULTIBCAST_ME(amount)

This procedure is similar to the above described operation. The only difference is that nodes do not start sending to node 0. Instead each starts with the node which is numbered one greater than itself (modulo the highest

processor number). This is to make sure that node 0 and the communication paths near node 0 are not overwhelmingly loaded.

\MULTIBCAST_ALTER(amount)

In contrast to the two above mentioned functions this one does not separate all *send* and *receive* operations. Instead, as the name indicates, it places a receive operation after each send. The order of these operations is such that *send* operations are done with increasing and *receive* operations are done with decreasing processor numbers. This strategy avoids hot spots and puts an evenly distributed communication load on the interconnection network.

The other class of high-level communication routines are vector communication kernels. Such routines are often used in parallel computing. Routines for distributing, collecting, and broadcasting vectors are provided. The communication pattern used by these procedures is based on a virtual (binary) tree topology. Although a tree topology might not map well on the target architecture, it does offer the advantage that collection and distribution can be done in logarithmic time. Three different routines are provided by the LOOP language.

\TREE_BCAST(start_data,amount)

Node 0 sends (broadcasts) ‘amount’ bytes to all other nodes. The data sent is located at the position indicated by ‘start_data’.

\TREE_COLLECT_VEC(start_vec)

Processor 0 collects the parts of a distributed vector from all other processors and rebuilds the original vector at ‘start_vec’.

\TREE_DISTRIB_VEC(start_vec)

Processor 0 starts distributing a vector at ‘start_vec’ to all other processors. Each node gets its own part of the vector. These tree communications are carried out in $\log_2$(processors) steps. In each step the data to be collected/distributed is passed to the next upper/lower level of the assumed virtual tree topology. A default logical tree topology is implemented with the LOOP package. The mapping of the nodes on the target architecture can be tuned by the user.

**Runtime Parameters**

In the preceding, the structural parameters needed to generate a certain type of workload have been described. For different executions, this generation step need not be repeated, only different runtime parameters are needed.

One important runtime parameter is the problem size. The overall execution time and communication behavior depend directly on this parameter. For example, in the SLALOM benchmark, by varying the problem size it is possible to analyze the cache influence. Workloads with a large problem size do not fit in small data caches. A second important runtime parameter is the number of processors allocated to the generated workload. Both, the problem size and the number of allocated processors, are important in determining the granularity at which the problem is solved most efficiently on the tested machine. To be able to model iterative algorithms, a third runtime parameter, the number of iterations is provided.

**Problem size:**
The size of the data structures over which the program iterates.

**Processors:**
The number of processors allocated to the workload.

**Iterations:**
In case of iterative algorithms, the number of passes the algorithm makes over the specified data structures.

#### 4.2.2.3 PICL and ParaGraph

An important feature of any successful benchmark is to design it to be portable across as many machines as possible. This is a difficult task in the case of multiprocessor architectures, because there is no standard programming language. There are also various programming models (e.g., host node model, node model, synchronous communication, asynchronous communication). A group of researchers at Oak Ridge National Laboratory (ORNL) addressed this task by constructing a communication library. The idea is simple:

1. Identify the communication needs of a message passing program (e.g., send, receive, barrier, broadcast, etc.).
2. Provide the user with routines for those needs.
3. Put the routines in a software library that is easy to install for a wide variety of multiprocessors.
4. Make it publicly available.

The result is PICL (Portable Instrumented Communication Library) which has been implemented on several multiprocessor systems. PICL programs are portable between machines where PICL is implemented. PICL includes all communication routines that are needed for parallel message passing programs. The generator of the benchmarking package transforms the LOOP description of a parallel workload into a parallel program with C and PICL statements. A detailed description of PICL can be found in [PICL90], which is also part of the LOOP benchmark documentation package[1].

PICL automatically instruments the code for tracing purposes. The resulting traces can be interpreted with ParaGraph, a graphical display system for visualizing the behavior and performance of parallel programs on message passing multicomputer architectures. Visual animation is provided based on execution trace information monitored during an actual run of a parallel program. The resulting trace data is replayed pictorially and provides a dynamic depiction of the behavior of the parallel program. Graphical summaries of overall performance behavior is also provided. Different visual perspectives provide different insights of the same performance data. A description of ParaGraph can be found in [Para92], which is also part of the benchmark documentation package.

The output of the generator was chosen to be PICL programs for three reasons: instrumentation, availability, and portability. PICL provides instrumentation, it is public domain software and it is implemented on several systems. The generator output is not inherently restricted to PICL. Whenever a new message passing paradigm becomes available that meets the three requirements above, the generator output can easily be changed. This implies that the basic LOOP approach is independent of the underlying message passing hardware. In this book it is not possible to describe all LOOP statements, a complete description can be found in [Bre94].

#### 4.2.2.4 Predefined Benchmarks

Once the basic LOOP structure has been specified it is possible to write generic LOOP programs to analyze a wide range of system features. The LOOP package includes some programs that can be used as predefined benchmarks. The predefined benchmarks consist of LOOP programs for typical parallel workloads (e.g., matrix multiplication,

1. available via *ftp (ftp.irb.uni-hannover.de)*

conjugate gradient, relaxation, fast fourier transformation) and of one special synthetic test program that provides an overall impression of the computation and communication performance of the machine. This special test program is termed the *Fingerprint* LOOP program. The predefined benchmarks are designed for users who want to evaluate and compare different machines.

**The Fingerprint**

```
#include "LOOP.h"
#define TRACE_BUF_SIZE 250000
#define MY_TYPE double
int main(void)
{
  /* declarations */
  int myself,allnodes;
  int host,problemsize ;
  \VEC(TYPE:MY_TYPE) sc, vec1;
  \VEC(TYPE:MY_TYPE) vec2;
  /* main program */
  \OPEN0(&allnodes,
       &myself, &host,
       &problemsize,
       TRACE_BUF_SIZE) ;
  \VISIBLE_SYNC();
  \LOOP {
    \SCALPROD(sc, vec1, vec2);}
  \VISIBLE_SYNC();
  /* multi-broadcast, 1 bytes */
  \MULTIBCAST0(1);
  \VISIBLE_SYNC();
  /* multi-bcast,500 bytes */
  \MULTIBCAST0(500);
  \VISIBLE_SYNC();
  /* multi-bcast, 1000 bytes */
  \MULTIBCAST0(1000);
  \VISIBLE_SYNC();
  \CLOSE0();
  return EXIT_SUCCESS;
}
```

Fig. 4.17. LOOP source code for Fingerprint

To assess the communication capabilities of a machine in comparison to the computational power, the *Fingerprint* benchmark has been developed. The goal is to provide quick, visual reference information for a first glance comparison between different machines. As shown in the annotated version of the space time diagram in Fig. 4.18., the *Fingerprint* was designed to illustrate

(1) the time needed for a certain computation intensive phase,

(2a-c) the time needed for communications of different message length, and

(3) the effect of heavy communication loads which partially saturate the communication network.

This latter effect is provoked by concentrating on node 0, then on node 1, and so on. Thus, communication delays tend to be compounded for higher processor numbers. Therefore, a severe "V-type" profile indicates a high number of conflicts in the communication network. A more rectangular (i.e., vertical) profile is typical for a non-saturated network as evidenced by the ends of phases 2a and 2b. In Fig. 4.17. the LOOP source code for the *Fingerprint* workload is given.

The execution of the *Fingerprint* workload falls into two parts. In the first part, a scalar product of two vectors of size *problemsize* is calculated on each node. The parameter *problemsize* is specified at runtime making several different executions possible. The second part consists of three multibroadcast instructions with different message lengths. In each multibroadcast the selected amount of information is transmitted from each node to all other nodes. This produces a heavy load for the interconnection network. The different amount of data sent increases the network load and produces space time diagrams which can be compared across various machines.

To provide a boundary between the computation and communication phases easier a

*\VISIBLE_SYNC()* construct is used. Since the PICL *sync0* instruction does not produce any trace data to be visualized by ParaGraph, the *LOOP* system provides this special form of synchronization. All nodes execute a *sync0* operation. Next, every node sends a short message to its right hand neighbor[1]. This produces a vertical line in the space-time diagram. To synchronize the execution of all nodes after sending a message to and receiving a message from the neighbors, a second *sync0* is performed by all nodes. The second *sync0* minimizes the time difference for the processors to start the next phase of the program.

1. Using a virtual ring topology

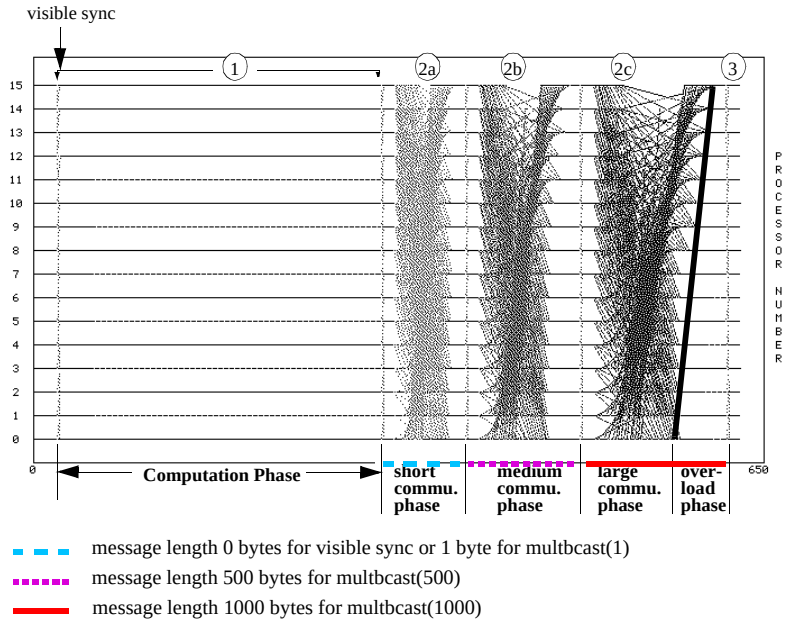



Fig. 4.18. Space time diagram for a typical 16 processor Fingerprint execution

### Comparison of nCUBE/2 and MEIKO using Fingerprint

As an example, two distributed multiprocessors of similar technology are now compared using the LOOP Fingerprint operation. One test machine is the nCUBE/2 which is described in section 4.1.2.1 of this book. The other machine is a MEIKO multiprocessor

which is based on T800 Transputers running at 20 MHz.

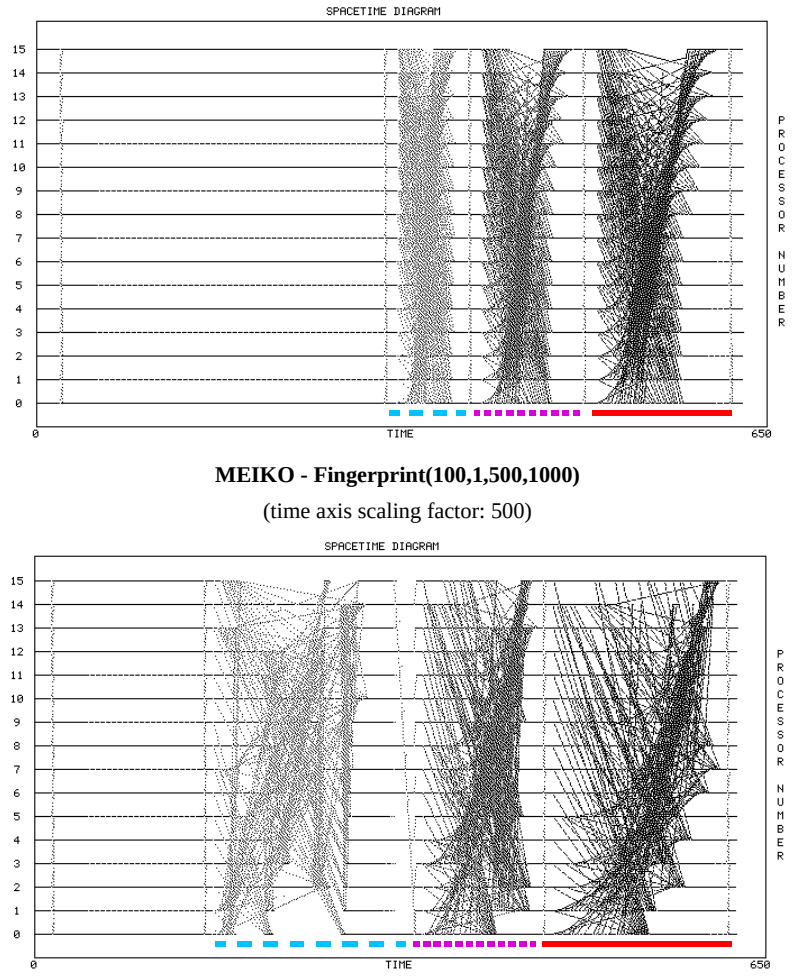



Fig. 4.19. Space-time diagram of 16 processor Fingerprint on MEIKO and nCUBE

Fig. 4.19. shows space-time diagrams for the first experiment. A Fingerprint with the parameter quadruple (100,1,500,1000) is executed on 16 processors on each machine. The first parameter 100 is the problem size of the scalar product of the computation phase. The second parameter 1 is the number of bytes used for the multibroadcast of the first com-

munication phase. The third and fourth parameters are the number of bytes being used for the second and third multibroadcasts.

The two space-time diagrams indicate that communication costs are higher on the MEIKO. For a better optical comparison, different time axis scaling is used in Fig. 4.19. for the two space-time diagrams. If the scaling of the MEIKO diagram is used for the nCUBE, the resulting diagram is shown in Fig. 4.20.

**nCUBE/2 - Fingerprint(100,1,500,1000)**

(time axis scaling factor: 500, same scale as MEIKO)

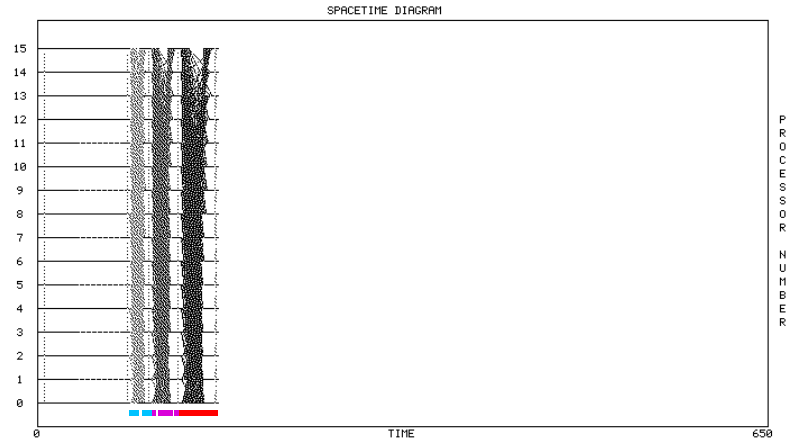



Fig. 4.20. Space-time diagram of 16 processor Fingerprint on nCUBE

Tab. 4.18. summarizes the execution times for the experiment. The total execution time is better for the nCUBE system. Of greater importance is the computation/communication ratio. A value of one for this ration means that the time spent for computation equals the time spent for communication. A value of smaller than one of this ration means that more time is spent for communication, a value greater than one means that more time is spent for computation. A value close to one

indicates a balanced system in terms of computation versus communication.

| **nCUBE/2** | | **MEIKO** | |
|---|---|---|---|
| number of nodes: | 16 | number of nodes: | 16 |
| total execution time: | 0.0765 sec | toal execution time: | 0.282 sec |
| computation time | 0.0367 sec | computation time | 0.060 sec |
| communication time | 0.0398 sec | communication time | 0.222 sec |
| compu./commu ratio | 0.922 | compu./commu ratio | 0.270 |
| avg time send (usec): | 4353 | avg time send (usec): | 18276 |

Tab. 4.18. Comparison of MEIKO and nCUBE using a 16 processor Fingerprint

In the second experiment, the same Fingerprint (parameters 100, 1, 500, 1000) is executed on 32 processors on each machine. Since the computational load is unchanged and the communication load is increased the computation/communication ration is expected to become smaller. Tab. 4.19. confirms the expected results. If the relative speed of nCUBE versus MEIKO is compared for Tab. 4.18. (nCUBE is 3.68 times faster than MEIKO) and Tab. 4.19. (nCUBE is 4.98 times faster than MEIKO) it shows that the higher communication costs of the MEIKO system slows down the system for larger numbers of processors. Also, the average time for the send operation has increased by a higher factor for the MEIKO system.

| **nCUBE/2** | | **MEIKO** | |
|---|---|---|---|
| number of nodes: | 32 | number of nodes: | 32 |
| total execution time: | 0.1122 sec | total execution time: | 0.559 sec |
| computation time | 0.0367 sec | computation time | 0.060 sec |
| communication time | 0.0755 sec | communication time | 0.499 sec |
| compu./commu ratio | 0.486 | compu./commu ratio | 0.120 |
| avg time send (usec): | 9836 | avg time send (usec): | 49669 |

Tab. 4.19. Comparison of MEIKO and nCUBE using a 32 processor Fingerprint

**Parameterized Applications**

For the comparison of different computer systems the benchmark package provides five different LOOP workload programs:

- fingerprint (fp),
- conjugate gradient method (cg),
- matrix multiplication sync (mmm_s),

- matrix multiplication async (mmm_a), and
- red-black relaxation (red_black).

The fingerprint workload is described in the previous section. The second workload (cg) is a LOOP workload for a parallel conjugate gradient method. The two different matrix multiplication versions are with asynchronous communication (mmm_a) and with synchronous communication (mmm_s). In the first case, sending of messages is overlapped with computation. This can be important for architectures being capable of doing computation and communication in parallel. The synchronous version is favorable for architectures with a synchronous message passing hardware. The last workload simulates a parallel red-black relaxation algorithms. The LOOP workload program is described in Fig. 4.15.

#### 4.2.2.5 Results

For the comparison of a MEIKO (a 64 node T800 based multiprocessor), an nCUBE/2 (a 128 node hypercube connected multiprocessor, compare section 4.1.2.), and an INTEL Paragon (a 512 node i860 based mesh connected multiprocessor, compare section 4.1.2.), five different LOOP workloads that are described are executed on each system. The workloads are executed with 16 and 32 processors on all three machines [Bre94]. The timings are given in Tab. 4.20. On the nCUBE/2 and the Paragon the workloads are also executed with 64 and 128 nodes. The results are shown in Tab. 4.20.The result tables are organized as follows. First, the name for the LOOP workload program is given. The first parameter is the number of allocated processors, the second parameter is the problem size, and the third parameter is the number of iterations (if applicable).

**Execution Times**

Tab. 4.20.shows the results of the LOOP benchmarks executed on the different systems. An interesting result is that none of the three target architectures profits from the overlapping communication in the second matrix multiplication algorithm. On the Paragon and the nCUBE, asynchronous communication results in a lower bandwidth. The asynchronous communication can also slow down computation because the processor and the communication unit try to access main memory simultaneously. On the MEIKO, asynchronous communications are converted to synchronous communications at the hardware level resulting in additional overhead. [Note: In a separate experiment, the message passing paradigm “send as soon as possible, receive as late as possible“ does not necessarily improve performance.]

| | Runtime in seconds | | |
|---|---|---|---|
| | MEIKO | nCUBE | Parag. |
| fp 16/100 | 0.282 | 0.073 | 0.024 |
| fp 32/100 | 0.559 | 0.112 | 0.036 |
| cg 16/256/8 | 0.601 | 0.299 | 0.062 |
| cg 32/256/8 | 0.624 | 0.237 | 0.055 |
| mmm_s 16/256 | 13.461 | 8.586 | 1.692 |
| mmm_s 32/256 | 7.788 | 4.643 | 0.855 |
| mmm_a 16/256 | 13.680 | 8.437 | 1.692 |
| mmm_a 32/256 | 7.922 | 4.470 | 0.886 |
| r_b 16/1024/5 | 6.885 | 4.439 | 0.567 |
| r_b 32/1024/5 | 3.621 | 2.210 | 0.284 |

Tab. 4.20. Execution times for the LOOP benchmarks

To see the results from a relative viewpoint, the times are converted to “paragon seconds“ (seeTab. 4.21.). The workloads in the tables are ordered from communication bound loads to computation bound loads. That is, the fingerprint workload has the highest communication/computation ratio, while the relaxation workload has the lowest communication/ computation ratio. From the published single node peak performances, one could expect that the performance of the nCUBE/2 and the MEIKO are similar and that the Paragon is an order of magnitude faster.

The first experiment (Fingerprint) shows that this expectation is not necessarily true. For a communication bound synthetic workload (i.e., fp16/100) a slowdown of only 3 is observed for the nCUBE/2 and a slowdown of 11 is observed for the MEIKO. However, the lower the communication/computation ratio is, the more the MEIKO and the nCUBE/2 are outperformed by the Paragon. For the most computation bound workload, red_black32/1024/5, the closer the MEIKO and nCUBE/2 are to each other and are approximately an order of magnitude slower than the Paragon. The Fingerprint results (execution time, space time diagram) show that the nCUBE/2 scores better with respect to communication bound workloads.

| | Slowdown | | |
|---|---|---|---|
| | MEIKO | nCUBE | Para. |
| fp 16/100 | 11.729 | 3.033 | 1.0 |
| fp 32/100 | 15.517 | 3.111 | 1.0 |
| cg 16/256/8 | 9.931 | 4.823 | 1.0 |
| cg 32/256/8 | 11.345 | 4.309 | 1.0 |
| mm_s 16/256 | 7.956 | 5.074 | 1.0 |
| mm_s 32/256 | 8.800 | 5.246 | 1.0 |
| mm_a 16/256 | 8.085 | 4.986 | 1.0 |
| mm_a 32/256 | 8.941 | 5.045 | 1.0 |
| r_b 16/1024/5 | 12.121 | 7.743 | 1.0 |
| r_b 32/1024/5 | 12.750 | 7.782 | 1.0 |

Tab. 4.21. Slowdown against Paragon

Some of the results are expected (e.g., overall performance). However, some tests provide interesting insight into machine behavior (through trace visualization tools). These results can be used by both parallel programmers and system developers to improve performance. Examples include balancing the computation and communication performance (e.g., Fingerprint) and the improving of asynchronous communication (e.g., matrix multiplication). It is noted that the parallelization for message passing systems is still rather coarse (i.e., a certain amount of computation between communication is needed) otherwise slowdowns can easily result from adding processors (e.g., the result for the conjugate gradient workload on the MEIKO for 16 and 32 processors).

**Standard Result Sheets**

For a more complete overview on the test results, three standard result sheets for each experiment are developed. The first page contains information on the workload, including the structural and runtime parameters, information on the system hardware (e.g., number of processors, type of interconnection network), the measured performance metrics (e.g., execution time, percentages for busy, idle and overhead times), a profile of the parallel workload, and a utilization summary for each processor. The second page gives an overview of various statistical information of an experiment. It contains information such as the number of messages sent and received, the average, maximum and minimum times for the send and receive operations, and message queue lengths[1].

The PICL tracefiles contain all the information on communication and computation events. Paragraph offers a wide variety of dis-

1. A report explaining in more detail the standard evaluation sheets for all tests is available via *ftp* from *ftp.irb.uni-hannover.de* (directory */pub/bench*)

plays to visualize these events. Thus, the user can go into as much detail as desired.

**Summary for LOOP Benchmarks**

In summary, the pros and cons of the LOOP benchmarks are:

**Pros:**

The LOOP language makes it easy to define workloads for target systems. The benchmark generator automatically transforms the workload descriptions into benchmark programs. Furthermore, the benchmark programs are instrumented and a complete measurement environment is available. Additionally, the program behavior can be visualized using the ParaGraph tool.

The benchmarks are scalable, problem size and number of processors are runtime parameters.

Besides the advantage of user defined workloads, a set of standard workloads is delivered with the benchmark in order to be able to compare different target systems running the same workload.

**Cons:**

The LOOP benchmarks are designed to evaluate message passing multiprocessor architectures and thus, they are not portable to shared memory machines.

## 4.3. Summary Benchmarking

The main goal for deterministic evaluation techniques based on performance measurement of real systems is to find performance bottlenecks and, if possible, to eliminate them. Normally it is not possible to change the hardware of existing ready to market systems, but the results can be used to improve the design of successors.

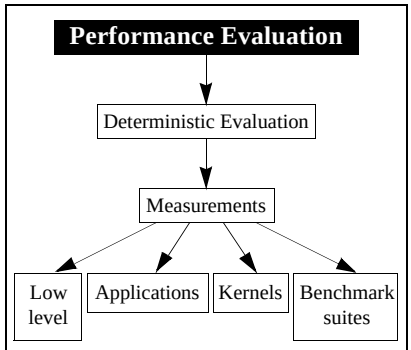



Fig. 4.21. Deterministic performance evaluation

The previous two sections of this text, namely performance evaluation of monoprocessors and performance evaluation of multiprocessors, gave an overview of existing benchmark workloads and benchmark technologies. For all benchmark efforts, the starting point is to run a real workload or a model of a real workload on an existing system.

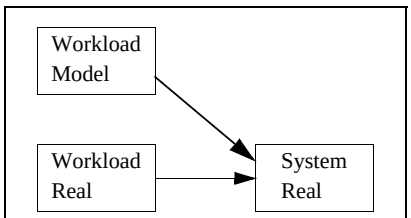
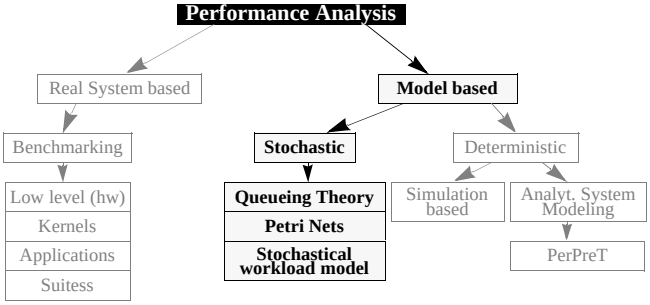



Fig. 4.22. Performance Measurement

The results of benchmarking for *system designers* can be used to improve future systems. For *application developers* benchmarking can help to identify architectural weaknesses which might be avoided. But the most important use of benchmarking is to compare systems on the market, to provide information for *purchase decisions* and to classify systems in terms of performance.

Since a real system is not always available for performance evaluation and since the implementation of benchmark workloads on parallel systems is a complex and time consuming task, the next sections present modeling techniques for parallel systems.




# 5. Performance Modeling for Multiprocessors

Performance models of multiprocessors (including their communication systems) with the goal to optimize and guide new designs have been studied in many publications and books (compare [Pat92], [Men94], [Laz84], and others). With the growing complexity of the hardware, these methods are of growing importance and their use in practise will increase. The approach so far has often been tuning a new system using educated guesses and experimentation results with existing similar systems. With the growing complexity of the systems, this intuitively based approach is harder to realize, since the understanding of the system behavior is harder to acquire. Additionally, the concurrent behavior of multiprocessor systems is a degree more complex than the behavior of monoprocessor systems

As indicated in the header, this section covers model based stochastic performance evaluation techniques. In contrast to modeling monoprocessor systems, to model a multiprocessor system requires the ability to model the concurrent execution of tasks on a system. Additionally the interconnection networks between processors and memory are more complex for multiprocessor systems. In the case of message passing systems, the interconnection network between processors is also part of the system model. In summary, the modeling of multiprocessors is a demanding task, but it is also necessary for the development of new systems and for the tuning and understanding of existing systems.

This section will demonstrate that the queueing network systems presented in section 3.2. of this book are also well suited to model multiprocessor systems. Additionally, the Petri Net approach to model concurrent systems is also presented in this section. Both approaches allow graphical descriptions of the problems to be solved. This feature helps to describe and understand the model of the target systems. In the last part of this section a quantitative approach to model multiprocessor behavior is presented. This approach uses workload properties to estimate the possible speedup and efficiency of an application.

### Typical Monoprocessor Model

Fig. 5.1. is a scheme for the typical interaction between the user and a monoprocessing system. The user submits jobs to the (time shared) system, which are processed by the system and the results are sent back to the us-

er. The workload for this type of interaction [Kos95] can be modeled using:

- transaction workload
  (characterized by arrival rates of requests)
- batch workload
  (constant number of requests)
- terminal workload
  (characterized by number of users and their think times)

The main difference between performance modeling for monoprocessors and multiprocessors is the complexity of the workload and the systems. No model is able to capture every detail of a contemporary computing system. Instead, the modeler must abstract the essential features which account for the cause and effect relationships that result in the observed behavior of the system. The intention of this section is to show how basic techniques can be applied to multiprocessor systems.

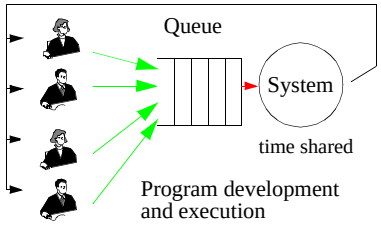



Fig. 5.1. Basic monoprocessor system model

### Typical Multiprocessor Model

The typical model for different multiprocessor classes (loosely coupled and tightly coupled) is described now.

### Message Passing Multiprocessors

Fig. 5.2. shows the basic architecture of distributed memory (message passing) multiprocessors (in the modeling literature often referenced as loosely coupled systems). The program development is realized using a "conventional" computing system, the so called host system. From there the programs are downloaded onto the parallel processing system (multiprocessor) to be executed. For many systems (nCUBE, INTEL Paragon, MEIKO), the multiprocessor operates in a space sharing mode. In contrast to time sharing, the user-allocated resources are not time slots, but the number of processors the user needs to execute the parallel program. These processors are arranged as a subsystem (subcubes for hypercubes, subarrays for meshes) for the time the program is running. Subsystems can be gathered to build a larger subsystem or distributed to build some smaller subsystems. In contrast to sequential workloads, most parallel workloads already contain parameters for the target system (size of subsystem to be used). The model for the whole multiprocessor including user interaction would be very complex with a difficult choice of adequate parameters. This is why most approaches for the modeling of multiprocessors focus on the modeling of the behavior of the subsystems when executing a single parallel program. The workload for such an experiment can be characterized as a batch workload. The workload model will consist of a quantitative description of the single program. User commands can be neglected. If different single programs creating different workloads are considered, a multiclass workload model [Pat92] has to be chosen.

### Shared Memory Multiprocessors

The modeling approach for shared memory multiprocessors (in the modeling literature often referenced as tightly coupled systems) is similar to the modeling of monoprocessor systems. From the modeling point of view, a tightly coupled system consists of several processors that operate under the coordination of a single operating system. The communication and synchronization is realized through shared data in the main memory. This design allows a high degree of interac-

tion between processors without a significant degradation of performance. Thus, the set of processors can be seen as a single entity. The tightly coupled multiprocessor can then be modeled as a single server with a single queue for job scheduling [Men94]. Whenever a processor becomes free, the next job is taken from the queue for execution. The problem with this approach is the correct model assumption for the processor speed. Software contention, operating system overhead and workload properties can lead to performance degradation that reduce the effective processor speed. A solution to this problem is a queueing network model with a load-dependent server representation (compare [Men94], chapter 7). The solution of queueing networks with load-dependent devices requires the definition of a service rate multiplier function. Shared memory multiprocessors are state of the art technology in today's computers. Besides compute and file servers, even personal computers (PCs) or workstations are realized as shared memory multiprocessors. But the focus of this text is more oriented towards massively parallel systems. These systems are realized as loosely coupled systems.

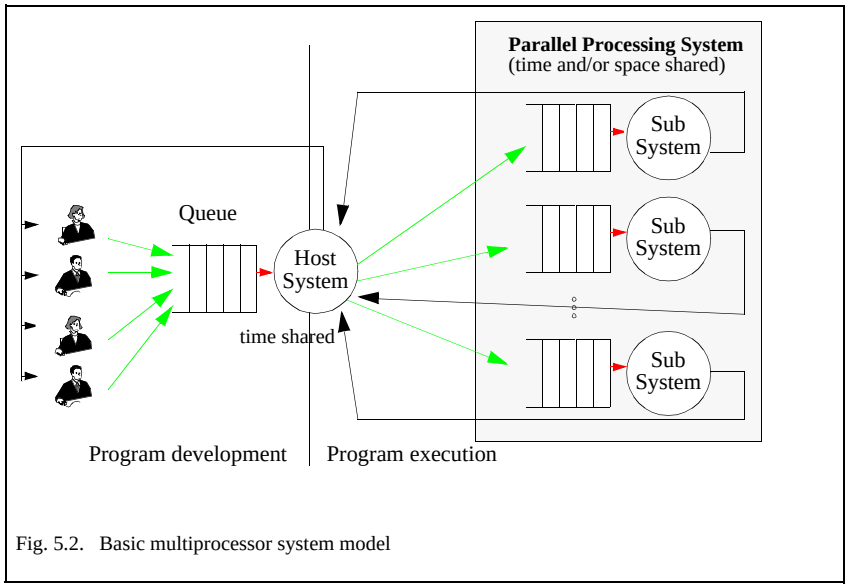



Fig. 5.2. Basic multiprocessor system model

**Interconnection Networks**

Both, message passing and shared memory multiprocessors require interconnection networks. The shared memory system needs the interconnection network to connect the processor modules to the main memory (compare (Fig. 4.1.), the message passing system needs the interconnection network to connect the processors (compare Fig. 4.2.). Thus, the interconnection network is an integral part of every multiprocessor system and needs to be represented in each model for the whole system. In [Pat92] some networks (switching and multistage) and approaches to model them using queueing networks are presented.

# 5.1. Queueing Networks

## 5.1.1. Queueing Network Example

Based on the example in section 3.2. a small multiprocessor system is considered (compare Fig. 5.3.) on a very high level. It consists of two processors (A1 and A2, each with a service rate of $\mu = 1$, i.e. one job is executed per time unit) and of a queueing system which can hold at most 4 jobs (the arrival rate $\lambda$ is 1, i.e. in average one job arrives per time unit, if there are less than 4 jobs waiting).

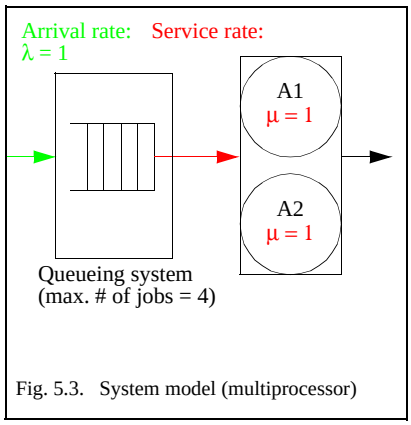



Fig. 5.3. System model (multiprocessor)

Before the system designer realizes the hardware for the queueing system, he is interested in comparing different strategies in terms of throughput and response time. The first strategy is to implement two separate queues (each can hold up to four jobs, but the total number of jobs in the system is still limited to four jobs). When a job enters the queueing system, one of the queues is randomly assigned to it. This strategy is called random queue.

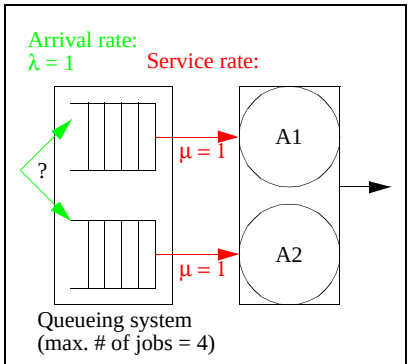



Fig. 5.4. Random queue strategy

The second strategy is to realize only one queue which can hold up to four jobs. The jobs wait in the queue until one of the processors is available and are then sent to that processor. This strategy is called common queue.

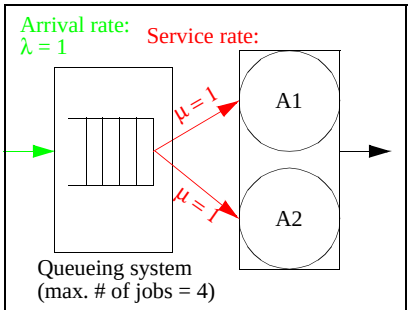



Fig. 5.5. Common queue strategy

The third strategy is to realize two queues (each can hold up to two jobs), when a job enters the queueing system, the job is sent to the currently shorter queue. This strategy is called shortest queue.

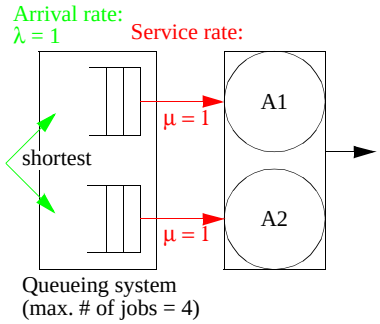



Fig. 5.6. Shortest queue strategy

The last strategy is similar to the first one. There are two separate queues (each can hold up to four jobs, but the total number of jobs in the system is still limited to four jobs). The strategy is to alternate between the two queues: When a job enters the queueing system, it is sent to the next queue from point of view of its predecessor. This strategy is called next queue.

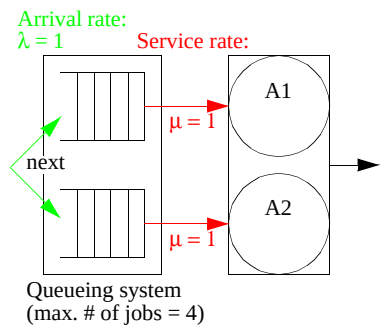



Fig. 5.7. Next queue strategy

The question now is which of the strategies results in the best performance. The performance measures to be determined are utilization, throughput, average waiting time, and response time. As in section 3.2. a markovian based approach is used to solve this problem.

**Random Queue Strategy**

Every job which enters the queueing system is randomly assigned to one of the two queues. Randomly assumes that the probability for the job to enter either one of the two queues is the same, namely 1/2. Therefore, jobs entering the queueing system arrive at each queue with rate λ/2.

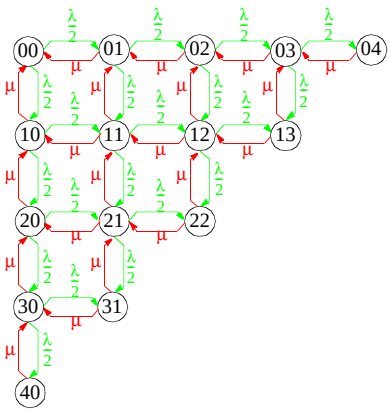



Fig. 5.8. State diagram of random queue

The steady state diagram for this strategy is outlined in Fig. 5.8. A state is described by the pair *(ij)*, with *i=0,1,...,4* and *j=0,1,...,4*. The indices *i* and *j* are the number of jobs in the corresponding queue (*i* for the jobs in the first queue, *j* for the jobs in the second queue). The probability $P_{12}$ is, for example, the probability for one job in the first queue and two jobs in the second queue. The jobs are only removed from the queue when the processing of the job has finished. At every point in time at most one state transition using one of the arcs can happen. The arcs show the rates for the state transitions with λ = arrival rate and μ = service rate. The resulting system of flow balance equations (flow-in = flow-out) is the following:

$$\lambda P_{00} = \mu P_{01} + \mu P_{10}$$

$$(\lambda + \mu) P_{01} = \frac{\lambda}{2} P_{00} + \mu P_{11} + \mu P_{02}$$

$$(\lambda + \mu) P_{02} = \frac{\lambda}{2} P_{01} + \mu P_{03} + \mu P_{12}$$

$$(\lambda + \mu) P_{03} = \frac{\lambda}{2} P_{02} + \mu P_{04} + \mu P_{13}$$

$$\mu P_{04} = \frac{\lambda}{2} P_{03}$$

$$(\lambda + \mu) P_{10} = \frac{\lambda}{2} P_{00} + \mu P_{11} + \mu P_{20}$$

$$(\lambda + 2\mu) P_{11} = \frac{\lambda}{2} P_{01} + \frac{\lambda}{2} P_{10} + \mu P_{12} + \mu P_{21}$$

$$(\lambda + 2\mu) P_{12} = \frac{\lambda}{2} P_{02} + \frac{\lambda}{2} P_{11} + \mu P_{13} + \mu P_{22}$$

$$2\mu P_{13} = \frac{\lambda}{2} P_{03} + \frac{\lambda}{2} P_{12}$$

$$(\lambda + \mu) P_{20} = \frac{\lambda}{2} P_{10} + \mu P_{21} + \mu P_{30}$$

$$(\lambda + 2\mu) P_{21} = \frac{\lambda}{2} P_{11} + \frac{\lambda}{2} P_{20} + \mu P_{22} + \mu P_{31}$$

$$2\mu P_{22} = \frac{\lambda}{2} P_{12} + \frac{\lambda}{2} P_{21}$$

$$(\lambda + \mu) P_{30} = \frac{\lambda}{2} P_{20} + \mu P_{31} + \mu P_{40}$$

$$2\mu P_{31} = \frac{\lambda}{2} P_{21} + \frac{\lambda}{2} P_{30}$$

$$\mu P_{40} = \frac{\lambda}{2} P_{30}$$

The sum of all probabilities equals one:

$$\sum P_{ij} = 1$$

This is a linear system with 15 unknowns and 16 equations. One of the equations is redundant. This system can be solved for given rates λ and μ. However, it is more elegant to

solve this system for all $P_{ij}$ as functions of the parameters $\lambda$ and $\mu$. Changes in the workload (parameter $\lambda$) or the system (parameter $\mu$) can then be done without the need to solve the linear system again. The resulting state probabilities are listed in Tab. 5.1.

$$P_{00} = \frac{16\mu^4}{5\lambda^4 + 8\lambda^3\mu + 12\lambda^2\mu^2 + 16\lambda\mu^3 + 16\mu^4}$$

$$P_{01} = P_{10} = \frac{8\lambda\mu^3}{5\lambda^4 + 8\lambda^3\mu + 12\lambda^2\mu^2 + 16\lambda\mu^3 + 16\mu^4}$$

$$P_{02} = P_{11} = P_{20} = \frac{4\lambda^2\mu^2}{5\lambda^4 + 8\lambda^3\mu + 12\lambda^2\mu^2 + 16\lambda\mu^3 + 16\mu^4}$$

$$P_{03} = P_{12} = P_{21} = P_{30} = \frac{2\lambda^3\mu}{5\lambda^4 + 8\lambda^3\mu + 12\lambda^2\mu^2 + 16\lambda\mu^3 + 16\mu^4}$$

$$P_{04} = P_{13} = P_{22} = P_{31} = P_{40} = \frac{\lambda^4}{5\lambda^4 + 8\lambda^3\mu + 12\lambda^2\mu^2 + 16\lambda\mu^3 + 16\mu^4}$$

Tab. 5.1. State probabilities for random queue

Now the performance measures utilization, throughput, average waiting time, and response time can be calculated.

**State Probabilities for $\lambda$=1 and $\mu$=1:**

Using the formulae from Tab. 5.1., the steady state probabilities for $\lambda$=1 and $\mu$=1 of the random queue strategy are:

$P_{00}$ = 16/57
$P_{01} = P_{10}$ = 8/57
$P_{02} = P_{11} = P_{20}$ = 4/57
$P_{03} = P_{12} = P_{21} = P_{30}$ = 2/57
$P_{04} = P_{13} = P_{22} = P_{31} = P_{40}$ = 1/57

**Utilization:**

With probability $P_{00}$ the system is idle, with probability $P_{0*}$ or $P_{*0}$ with (*=*1,2,3,4*) the system is utilized at 50%. Thus, the utilization can be calculated as:

$$U_{rand} = 1 - \left(P_{00} + \frac{1}{2}P_{01} + \frac{1}{2}P_{10} + \frac{1}{2}P_{02} + \frac{1}{2}P_{20} + \frac{1}{2}P_{03} + \frac{1}{2}P_{30} + \frac{1}{2}P_{40} + \frac{1}{2}P_{04}\right)$$

The state diagram outlined in Fig. 5.8. shows symmetry along the diagonal $P_{ii}$ with (*i=0,1,2*). This results in $P_{ij} = P_{ji}$. Using this symmetry, the utilization $U_{rand}$ can be calculated as:

$$U_{rand} = 1 - (P_{00} + P_{01} + P_{02} + P_{03} + P_{04})$$

$$U_{rand} = \frac{26}{57} = 0.4561$$

**Throughput:**

The service demand $D$ for each job is one time unit, thus the throughput $X_{rand}$ is:

$$X_{rand} = \frac{U_{rand}}{D} = U_{rand} = \frac{26}{57} = 0.4561$$

The average number of jobs (*njob*) in the system is needed to compute the response time *rt*. Knowing steady state probabilities makes it easy to calculate the average number of jobs in the system. For each state the number of jobs in the system is determined as number of jobs in that state weighted by the steady state probability of being in that state. Summing these values for all states will yield the

average number of jobs in the system. Thus, the formula for *njob* is:

$$njob_{rand} = \sum n(ij) P_{ij} \qquad \text{(Eq.5.1)}$$

with *n(ij) = the number of jobs in state (ij)*.

Using symmetry, it can be calculated for the random queue system as:

$$njob_{rand} = 2P_{01} + 6P_{02} + 12P_{03} + 20P_{04}$$

$$njob_{rand} = \frac{84}{57} = 1.4737$$

After *njob* is calculated, it is possible to use **Little's Law** [Lit61]:

$$njob = X \cdot rt \qquad \text{(Eq.5.2)}$$

to calculate the response time *rt* of the system.

$$rt = \frac{njob}{X} \qquad \text{(Eq.5.3)}$$

The response time $rt_{rand}$ for the random queue system is.

$$rt_{rand} = \frac{\frac{84}{57}}{\frac{26}{57}} = \frac{84}{26} = 3.2308$$

The waiting time for a job is defined as the time a job spends in the system without service. It can be calculated by subtracting the average service time from the average response time. The service rate is μ. The average service time *D* is the reciprocal value of μ. Thus, the waiting time *wt* is calculated as:

$$wt = rt - D \qquad \text{(Eq.5.4)}$$

For the example system *D=1/μ=1*. The waiting time $wt_{rand}$ for the random queue strategy is:

$$wt_{rand} = rt_{rand} - 1 = 2.2308$$

The four strategies are compared under different load conditions. Doubling the workload parameter λ yields the following results.

**State Probabilities for λ=2 and μ=1:**

Using the formulae from Tab. 5.1., the steady state probabilities for λ=2 and μ=1 of the random queue strategy are:

$P_{00}$ = 2/28
$P_{01} = P_{10}$ = 1/28
$P_{02} = P_{11} = P_{20}$ = 2/28
$P_{03} = P_{12} = P_{21} = P_{30}$ = 2/28
$P_{04} = P_{13} = P_{22} = P_{31} = P_{40}$ = 2/28

Utilization:

$$U_{rand} = 1 - \left(P_{00} + \frac{1}{2}P_{01} + \frac{1}{2}P_{10} + \frac{1}{2}P_{02} + \frac{1}{2}P_{20} + \frac{1}{2}P_{03} + \frac{1}{2}P_{30} + \frac{1}{2}P_{40} + \frac{1}{2}P_{04}\right)$$

Using symmetry:

$$U_{rand} = 1 - (P_{00} + P_{01} + P_{02} + P_{03} + P_{04})$$

$$U_{rand} = \frac{19}{28} = 0.6786$$

The service demand D for each job is one time unit, thus the throughput is:

$$X_{rand} = \frac{U_{rand}}{D} = U_{rand} = \frac{19}{28} = 0.6786$$

Using symmetry the mean queue length ($njob_{rand}$) can be calculated as:

$$njob_{rand} = 2P_{01} + 6P_{02} + 12P_{03} + 20P_{04}$$

$$njob_{rand} = \frac{78}{28} = 2.7857$$

The corresponding response time $rt_{rand}$ is then:

$$rt_{rand} = \frac{\frac{78}{28}}{\frac{19}{28}} = 4.1053$$

The waiting time $wt_{rand}$ for the random queue strategy is:

$$wt_{rand} = rt_{rand} - 1 = 3.1053$$

**Common Queue Strategy**

Every job enters the system through the common queue, from there it proceeds to the first available processor. In Fig. 5.9. the steady state diagram of the common queue strategy is outlined. A state is defined as triple *(ijk)*, with *(i=0,1,2; j=0,1; k=0,1)*. The first index *i* is the number of jobs waiting for execution in the common queue, the index *j* is the number of jobs being currently executed by the first processor and the index *k* is the number of jobs being currently executed by the second processor.

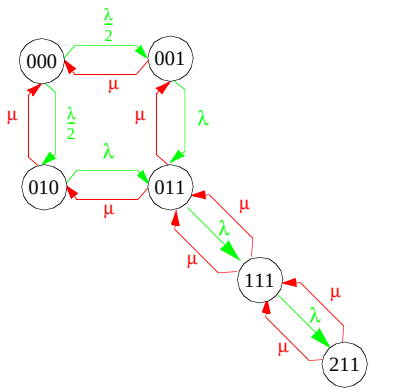



Fig. 5.9. State diagram of common queue

The resulting system of flow balance equations (flow-in = flow-out) is the following:

$$\lambda P_{000} = \mu P_{001} + \mu P_{010}$$

$$(\lambda + \mu) P_{001} = \frac{\lambda}{2} P_{000} + \mu P_{011}$$

$$(\lambda + \mu) P_{010} = \frac{\lambda}{2} P_{000} + \mu P_{011}$$

$$(\lambda + 2\mu) P_{011} = \lambda P_{001} + \lambda P_{010} + 2\mu P_{111}$$

$$(\lambda + 2\mu) P_{111} = \lambda P_{011} + 2\mu P_{211}$$

$$2\mu P_{211} = \lambda P_{111}$$

The sum of all probabilities equals one:

$$\sum P_{ijk} = 1$$

Tab. 5.2. lists the steady state probabilities for the common queue strategy.

| |
|---|
| $P_{000} = \frac{8\mu^4}{\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 8\lambda\mu^3 + 8\mu^4}$ |
| $P_{010} = P_{001} = \frac{4\lambda\mu^3}{\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 8\lambda\mu^3 + 8\mu^4}$ |
| $P_{011} = \frac{4\lambda^2\mu^2}{\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 8\lambda\mu^3 + 8\mu^4}$ |
| $P_{111} = \frac{2\lambda^3\mu}{\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 8\lambda\mu^3 + 8\mu^4}$ |
| $P_{112} = \frac{\lambda^4}{\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 8\lambda\mu^3 + 8\mu^4}$ |

Tab. 5.2. Results for common queue

**State Probabilities for $\lambda$=1 and $\mu$=1:**

Using the formulae from Tab. 5.2., the steady state probabilities $P_{ijk}$ for $\lambda$=1 and $\mu$=1 of the common queue strategy are can be calculated as before.

Utilization:

$$U_{comm} = 1 - \left(P_{000} + \frac{1}{2}P_{010} + \frac{1}{2}P_{001}\right)$$

Using the calculated steady state probabilities $P_{ijk}$ for $\lambda$=1 and $\mu$=1, the resulting value for the utilization $U_{comm}$ is:

$$U_{comm} = \frac{11}{23} = 0.4783$$

Using the calculated utilization $U_{comm}$, the resulting value for the throughput $X_{comm}$ is:

$$X_{comm} = \frac{11}{23} = 0.4783$$

Using symmetry, the mean queue length $njob_{comm}$ can be calculated as:

$$njob_{comm} = 2P_{010} + 2P_{011} + 3P_{111} + 4P_{112}$$

Using the calculated steady state probabilities $P_{ijk}$ for $\lambda$=1 and $\mu$=1, the resulting value for the mean queue length $njob_{comm}$ is:

$$njob_{comm} = \frac{26}{23} = 1.1304$$

Using the calculated the mean queue length $njob_{comm}$ and the calculated throughput $X_{comm}$, the resulting value for the response time $rt_{comm}$ is:

$$rt_{comm} = \frac{\frac{26}{23}}{\frac{11}{23}} = 2.3636$$

The waiting time $wt_{comm}$ for the common queue strategy is:

$$wt_{comm} = rt_{comm} - 1 = 1.3636$$

As before, the workload parameter $\lambda$ is doubled:

**State Probabilities for $\lambda$=2 and $\mu$=1:**

Using the formulae from Tab. 5.2., the steady state probabilities $P_{ijk}$ for $\lambda$=2 and $\mu$=1 of the common queue strategy are can be calculated as before.

Utilization:

$$U_{comm} = 1 - \left(P_{000} + \frac{1}{2}P_{010} + \frac{1}{2}P_{001}\right)$$

Using the calculated steady state probabilities $P_{ijk}$ for λ=2 and μ=1, the resulting value for the utilization $U_{comm}$ is:

$$U_{comm} = \frac{13}{16} = 0.8125$$

Using the calculated utilization $U_{comm}$, the resulting value for the throughput $X_{comm}$ is:

$$X_{comm} = \frac{13}{16} = 0.8125$$

Using symmetry, the mean queue length $njob_{comm}$ can be calculated as:

$$njob_{comm} = 2P_{010} + 2P_{011} + 3P_{111} + 4P_{112}$$

Using the calculated steady state probabilities $P_{ijk}$ for λ=2 and μ=1, the resulting value for the mean queue length $njob_{comm}$ is:

$$njob_{comm} = \frac{38}{16} = 2.375$$

Using the calculated the mean queue length $njob_{comm}$ and the calculated throughput $X_{comm}$, the resulting value for the response time $rt_{comm}$ is:

$$rt_{comm} = \frac{\frac{38}{16}}{\frac{13}{16}} = 2.923$$

The waiting time $wt_{comm}$ for the common queue strategy is:

$$wt_{comm} = rt_{comm} - 1 = 1.923$$

**Shortest Queue Strategy**

Each job which enters the queueing system is assigned to one of the two queues by alternating between them. The probability for the job to enter either one of the two queues is the same, namely 1/2. Therefore, jobs entering the queueing system arrive at each queue with rate λ/2.

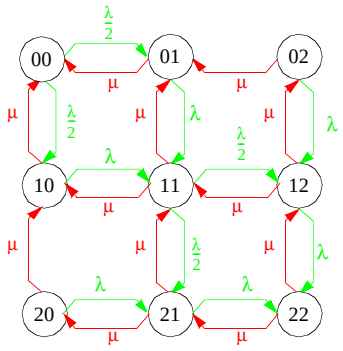



Fig. 5.10. State diagram of shortest queue

The resulting steady state diagram is shown by Fig. 5.10. A state is described by the pair *(ij)*, with *i=0,1,...,4* and *j=0,1,...,4*. The indices *i* and *j* are the number of jobs in the corresponding queue (*i* for the jobs in the first queue, *j* for the jobs in the second queue).

The results for the steady state probabilities in Tab. 5.3. are derived by solving the linear system of equations from Fig. 5.10.:

$$\lambda P_{00} = \mu P_{01} + \mu P_{10}$$

$$(\lambda + \mu) P_{01} = \frac{\lambda}{2} P_{00} + \mu P_{11} + \mu P_{02}$$

$$(\lambda + \mu) P_{02} = \mu P_{12}$$

$$(\lambda + \mu) P_{10} = \frac{\lambda}{2} P_{00} + \mu P_{11} + \mu P_{20}$$

$$(\lambda + 2\mu) P_{11} = \lambda P_{01} + \lambda P_{10} + \mu P_{12} + \mu P_{21}$$

$$(\lambda + 2\mu)P_{12} = \lambda P_{02} + \frac{\lambda}{2}P_{11} + \mu P_{22}$$

$$(\lambda + \mu)P_{20} = \mu P_{21}$$

$$(\lambda + 2\mu)P_{21} = \frac{\lambda}{2}P_{11} + \lambda P_{20} + \mu P_{22}$$

$$2\mu P_{22} = \lambda P_{12} + \lambda P_{21}$$

The sum of all probabilities equals one:

$$\sum P_{ijk} = 1$$

$$P_{00} = \frac{2\mu(3\lambda\mu^3 + 4\mu^4)}{(\lambda + \mu)(\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 6\lambda\mu^3 + 8\mu^4)}$$

$$P_{01} = P_{10} = \frac{\lambda(3\lambda\mu^2 + 4\mu^4)}{(\lambda + \mu)(\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 6\lambda\mu^3 + 8\mu^4)}$$

$$P_{02} = P_{20} = \frac{\lambda^3\mu^2}{(\lambda + \mu)(\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 6\lambda\mu^3 + 8\mu^4)}$$

$$P_{11} = \frac{2\lambda^2\mu^2(\lambda + 2\mu)}{(\lambda + \mu)(\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 6\lambda\mu^3 + 8\mu^4)}$$

$$P_{12} = P_{21} = \frac{\lambda^3\mu}{\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 6\lambda\mu^3 + 8\mu^4}$$

$$P_{22} = \frac{\lambda^4}{\lambda^4 + 2\lambda^3\mu + 4\lambda^2\mu^2 + 6\lambda\mu^3 + 8\mu^4}$$

Tab. 5.3. Results for shortest queue

**State Probabilities for λ=1 and μ=1:**

Using the formulae from Tab. 5.3., the steady state probabilities $P_{ij}$ for λ=1 and μ=1 of the common queue strategy are can be calculated as before.

Utilization:

$$U_{short} = 1 - \left(P_{00} + \frac{1}{2}P_{01} + \frac{1}{2}P_{10} + \frac{1}{2}P_{02} + \frac{1}{2}P_{20}\right)$$

Using symmetry the formula for the utilization $U_{short}$ can be simplified to:

$$U_{short} = 1 - (P_{00} + P_{01} + P_{02})$$

Using the calculated steady state probabilities $P_{ij}$ for λ=1 and μ=1, the resulting value for the utilization $U_{short}$ is:

$$U_{short} = \frac{20}{42} = 0.4762$$

Using the calculated utilization $U_{short}$, the resulting value for the throughput $X_{short}$ is:

$$X_{short} = \frac{20}{42} = 0.4762$$

Using symmetry, the mean queue length $njob_{short}$ can be calculated as:

$$njob_{short} = 2P_{01} + 4P_{02} + 2P_{11} + 6P_{12} + 4P_{22}$$

Using the calculated steady state probabilities $P_{ij}$ for λ=1 and μ=1, the resulting value for the mean queue length $njob_{short}$ is:

$$njob_{short} = \frac{50}{42} = 1.1905$$

Using the calculated the mean queue length $njob_{short}$ and the calculated throughput $X_{short}$,

the resulting value for the response time $rt_{short}$ is:

$$rt_{short} = \frac{\frac{50}{42}}{\frac{20}{42}} = 2.5000$$

The waiting time $wt_{short}$ for the common queue strategy is:

$$wt_{short} = rt_{short} - 1 = 1.5000$$

As before, the workload parameter $\lambda$ is doubled:

**State Probabilities for $\lambda$=2 and $\mu$=1:**

Using the formulae from Tab. 5.3., the steady state probabilities $P_{ij}$ for $\lambda$=2 and $\mu$=1 of the common queue strategy are can be calculated as before.

Utilization:

$$U_{short} = 1 - (P_{00} + P_{01} + P_{02})$$

Using the calculated steady state probabilities $P_{ij}$ for $\lambda$=2 and $\mu$=1, the resulting value for the utilization $U_{short}$ is:

$$U_{short} = \frac{39}{51} = 0.7647$$

Using the calculated utilization $U_{short}$, the resulting value for the throughput $X_{short}$ is:

$$X_{short} = \frac{39}{51} = 0.7647$$

Using symmetry, the mean queue length $njob_{short}$ can be calculated as:

$$njob_{short} = 2P_{01} + 4P_{02} + 2P_{11} + 6P_{12} + 4P_{22}$$

Using the calculated steady state probabilities $P_{ij}$ for $\lambda$=2 and $\mu$=1, the resulting value for the mean queue length $njob_{short}$ is:

$$njob_{short} = \frac{118}{51} = 2.3137$$

Using the calculated the mean queue length $njob_{short}$ and the calculated throughput $X_{short}$, the resulting value for the response time $rt_{short}$ is:

$$rt_{short} = \frac{\frac{118}{51}}{\frac{39}{51}} = 3.0256$$

The waiting time $wt_{short}$ for the common queue strategy is:

$$wt_{short} = rt_{short} - 1 = 2.0256$$

**Next Queue Strategy**

The next queue strategy is to alternate between the two queues. When a job enters the queueing system, it is sent to the next queue from point of view of its predecessor. The steady state diagram for this strategy is outlined in Fig. 5.11. A state is described by the pair *(ij)*, with *i=0,1,...,4* and *j=0,1,...,4*. The indices *i* and *j* are the number of jobs in the corresponding queue (*i* for the jobs in the first queue, *j* for the jobs in the second queue. One of the indices is always underlined. This indicates the queue that has been assigned at the previous arrival. For each arrival the queue changes, i.e. the underlined index is increased by one and the underline changes to the other index. The steady state diagram for this strategy is outlined in Fig. 5.11.

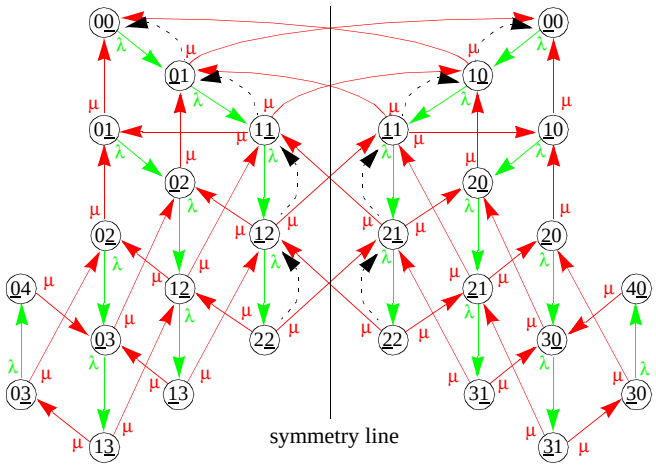



Fig. 5.11. State diagram of next queue

The system of equations resulting from the steady state diagram for this strategy is more complex compared with the previous ones. 28 states result in 28 flow balance equations that have to be solved. Since the system will be solved symbolically in terms of λ and μ, a smaller number of equations would make this task much easier. Looking at the state diagram in Fig. 5.11., symmetry along the drawn symmetry line is obvious. Four arcs cross the symmetry line from left to right and vice versa. These arcs are replaced by the corresponding dotted arcs. After this replacement, only one of the two systems (left or right from the symmetry line) has to be solved. The resulting flow balance equations for the left side are:

$$\lambda P_{\underline{00}} = \mu P_{\underline{01}} + \mu P_{\underline{01}}$$

$$(\lambda + \mu) P_{\underline{01}} = \lambda P_{\underline{00}} + \mu P_{\underline{11}} + \mu P_{\underline{02}}$$

$$(\lambda + \mu) P_{\underline{01}} = \mu P_{\underline{11}} + \mu P_{\underline{02}}$$

$$(\lambda + 2\mu) P_{\underline{11}} = \lambda P_{\underline{01}} + \mu P_{\underline{12}} + \mu P_{\underline{21}}$$

$$(\lambda + \mu) P_{\underline{02}} = \lambda P_{\underline{01}} + \mu P_{\underline{12}} + \mu P_{\underline{03}}$$

$$(\lambda + \mu) P_{\underline{02}} = \mu P_{\underline{12}} + \mu P_{\underline{03}}$$

$$(\lambda + 2\mu)P_{12} = \lambda P_{02} + \mu P_{22} + \mu P_{13}$$

$$(\lambda + 2\mu)P_{12} = \lambda P_{11} + \mu P_{22} + \mu P_{13}$$

$$(\lambda + \mu)P_{03} = \lambda P_{02} + \mu P_{13} + \mu P_{04}$$

$$(\lambda + \mu)P_{03} = \mu P_{13}$$

$$2\mu P_{22} = \lambda P_{12}$$

$$2\mu P_{13} = \lambda P_{12}$$

$$2\mu P_{13} = \lambda P_{03}$$

$$\mu P_{04} = \lambda P_{03}$$

The sum of all probabilities equals one:

$$\sum P_{ijk} = 0$$

The solution of this linear system is given in Tab. 5.4.

**State Probabilities for $\lambda$=1 and $\mu$=1:**

Using the formulae from Tab. 5.4., the steady state probabilities $P_{ij}$ for $\lambda$=1 and $\mu$=1 of the next queue strategy can be calculated as before.

Utilization using symmetry:

$$U_{next} = 1 - (2P_{00} + P_{10} + P_{01} + P_{20} + P_{02} + P_{30} + P_{03} + P_{40})$$

Using the calculated steady state probabilities $P_{ij}$ for $\lambda$=1 and $\mu$=1, the resulting value for the utilization $U_{next}$ is:

$$U_{next} = 1 - \frac{10014}{19086} = \frac{8972}{19086} = 0.4701$$

Using the calculated utilization $U_{next}$, the resulting value for the throughput $X_{next}$ is:

$$X_{short} = \frac{8972}{19086} = 0.4701$$

Using symmetry, the mean queue length $njob_{next}$ can be calculated as:

$$njob_{\text{next}} = 2P_{01}+2P_{10}+4P_{11}+4P_{02}+4P_{20}+6P_{21}+6P_{12}+6P_{03}+6P_{30}+8P_{22}+8P_{13}+8P_{31}+8P_{04}$$

Using the calculated steady state probabilities $P_{ij}$ for $\lambda$=1 and $\mu$=1, the resulting value for the mean queue length $njob_{next}$ is:

$$njob_{next} = \frac{24750}{19086} = 1.2968$$

Using the calculated the mean queue length $njob_{next}$ and the calculated throughput $X_{next}$, the resulting value for the response time $rt_{next}$ is:

$$rt_{next} = \frac{\frac{24750}{19086}}{\frac{8972}{19086}} = 2.7586$$

The waiting time $wt_{short}$ for the common queue strategy is:

$$wt_{next} = rt_{next} - 1 = 1.7586$$

As before, the workload parameter $\lambda$ is doubled:

**State Probabilities for $\lambda$=2 and $\mu$=1:**

Using the formulae from Tab. 5.4., the steady state probabilities $P_{ij}$ for $\lambda$=2 and $\mu$=1 of the next queue strategy can be calculated as before.

Utilization using symmetry:

$$U_{next} = 1 - (2P_{\underline{00}} + P_{\underline{10}} + P_{\underline{01}} + P_{\underline{20}} + P_{\underline{02}} + P_{\underline{30}} + P_{\underline{03}} + P_{\underline{40}})$$

Using the calculated steady state probabilities $P_{ij}$ for $\lambda$=2 and $\mu$=1, the resulting value for the utilization $U_{next}$ is:

$$U_{next} = 1 - \frac{926}{3284} = \frac{2358}{3284} = 0.7180$$

Using the calculated utilization $U_{next}$, the resulting value for the throughput $X_{next}$ is:

$$X_{short} = \frac{2358}{3284} = 0.7180$$

Using symmetry, the mean queue length $njob_{next}$ can be calculated as:

$$njob_{\text{next}} = 2P_{\underline{01}}+2P_{\underline{10}}+4P_{\underline{11}}+4P_{\underline{02}}+4P_{\underline{20}}+6P_{\underline{21}}+6P_{\underline{12}}+6P_{\underline{03}}+6P_{\underline{30}}+8P_{\underline{22}}+8P_{\underline{13}}+8P_{\underline{31}}+8P_{\underline{04}}$$

Using the calculated steady state probabilities $P_{ij}$ for $\lambda$=2 and $\mu$=1, the resulting value for the mean queue length $njob_{next}$ is:

$$njob_{next} = \frac{8228}{3284} = 2.5055$$

Using the calculated the mean queue length $njob_{next}$ and the calculated throughput $X_{next}$, the resulting value for the response time $rt_{next}$ is:

$$rt_{next} = \frac{\frac{8228}{3284}}{\frac{2358}{3284}} = 3.4894$$

The waiting time $wt_{short}$ for the common queue strategy is:

$$wt_{next} = rt_{next} - 1 = 2.4894$$

| | |
|---|---|
| $P_{\underline{0}0}=P_{0\underline{0}}=$ | $\frac{\mu^4(15\lambda^6 + 134\lambda^5\mu + 488\lambda^4\mu^2 + 936\lambda^3\mu^3 + 928\lambda^2\mu^4 + 416\lambda\mu^5 + 64\mu^6)}{Den}$ |
| $P_{\underline{0}1}=P_{1\underline{0}}=$ | $\frac{\lambda\mu^3(8\lambda^6 + 82\lambda^5\mu + 333\lambda^4\mu^2 + 706\lambda^3\mu^3 + 776\lambda^2\mu^4 + 384\lambda\mu^5 + 64\mu^6)}{Den}$ |
| $P_{0\underline{1}}=P_{\underline{1}0}=$ | $\frac{\lambda^2\mu^3(\lambda+\mu)(7\lambda^4 + 45\lambda^3\mu + 110\lambda^2\mu^2 + 120\lambda\mu^3 + 32\mu^4)}{Den}$ |
| $P_{1\underline{1}}=P_{\underline{1}1}=$ | $\frac{\lambda^2\mu^2(4\lambda^6 + 36\lambda^5\mu + 142\lambda^4\mu^2 + 309\lambda^3\mu^3 + 354\lambda^2\mu^4 + 184\lambda\mu^5 + 32\mu^6)}{Den}$ |
| $P_{\underline{0}2}=P_{2\underline{0}}=$ | $\frac{\lambda^3\mu^2(4\lambda^5 + 39\lambda^4\mu + 139\lambda^3\mu^2 + 242\lambda^2\mu^3 + 192\lambda\mu^4 + 48\mu^5)}{Den}$ |
| $P_{0\underline{2}}=P_{\underline{2}0}=$ | $\frac{\lambda^4\mu^2(3\lambda^4 + 23\lambda^3\mu + 65\lambda^2\mu^2 + 76\lambda\mu^3 + 28\mu^4)}{Den}$ |
| $P_{1\underline{2}}=P_{\underline{2}1}=$ | $\frac{2\lambda^4\mu(\lambda^2 + 5\lambda\mu + 7\mu^2)(\lambda^3 + 4\lambda^2\mu + 6\lambda\mu^2 + 2\mu^3)}{Den}$ |
| $P_{\underline{1}2}=P_{2\underline{1}}=$ | $\frac{2\lambda^3\mu(\lambda+\mu)(\lambda^5 + 8\lambda^4\mu + 25\lambda^3\mu^2 + 45\lambda^2\mu^3 + 36\lambda\mu^4 + 8\mu^5)}{Den}$ |
| $P_{\underline{0}3}=P_{3\underline{0}}=$ | $\frac{2\lambda^5\mu(\lambda+\mu)(\lambda^3 + 8\lambda^2\mu + 22\lambda\mu^2 + 21\mu^3)}{Den}$ |
| $P_{0\underline{3}}=P_{\underline{3}0}=$ | $\frac{\lambda^6\mu(\lambda^3 + 8\lambda^2\mu + 22\lambda\mu^2 + 21\mu^3)}{Den}$ |
| $P_{2\underline{2}}=P_{\underline{2}2}=$ | $\frac{\lambda^4\mu(\lambda+\mu)(\lambda^5 + 8\lambda^4\mu + 25\lambda^3\mu^2 + 45\lambda^2\mu^3 + 36\lambda\mu^4 + 8\mu^5)}{Den}$ |
| $P_{\underline{1}3}=P_{3\underline{1}}=$ | $\frac{\lambda^5(\lambda^2 + 5\lambda\mu + 7\mu^2)(\lambda^3 + 4\lambda^2\mu + 6\lambda\mu^2 + 2\mu^3)}{Den}$ |
| $P_{1\underline{3}}=P_{\underline{3}1}=$ | $\frac{\lambda^6(\lambda+\mu)(\lambda^3 + 8\lambda^2\mu + 22\lambda\mu^2 + 21\mu^3)}{Den}$ |
| $P_{\underline{0}4}=P_{4\underline{0}}=$ | $\frac{\lambda^7(\lambda^3 + 8\lambda^2\mu + 22\lambda\mu^2 + 21\mu^3)}{Den}$ |

Denominator for all $P_{ij}$:

$$Den = 2(4\lambda^{10} + 42\lambda^9\mu + 191\lambda^8\mu^2 + 521\lambda^7\mu^3 + 1016\lambda^6\mu^4 + 1615\lambda^5\mu^5 + 2122\lambda^4\mu^6 + 2112\lambda^3\mu^7 + 1376\lambda^2\mu^8 + 480\lambda\mu^9 + 64\mu^{10})$$

Tab. 5.4. Results for next queue

**Comparison of Results**

The performance measures throughput, number of jobs in the system, response time and waiting time of the four strategies (random queue, shortest queue, next queue and common queue) are ranked in Tab. 5.5. (absolute values) and Tab. 5.6. (relative values). The common queue strategy is the best for all performance measures. In the second table, the performance measures of the best ranked strategy are set to 100%. The other values are given as variations in per cent. If the workload parameter $\lambda$ increases, the relative difference of the strategies also increase for the performance measure throughput. It does not necessarily increase for the other performance measures. This is an effect of the closed system, the maximum number of jobs in the system is limited to four. The higher the workload is, the closer the average number of jobs in the system will be to the maximum number of jobs in the system for any of the four strategies. Since response time and waiting time are direct derivatives of the average number of jobs in the system, it is clear that they show the same behavior.

| $\lambda = 1, \mu = 1$ | **Throughput** | **njob** | **Resp. time** | **Wait. Time** | **Rank** |
|---|---|---|---|---|---|
| common queue | 0.4783 | 1.1304 | 2.3636 | 1.3636 | 1 |
| shortest queue | 0.4762 | 1.1905 | 2.5000 | 1.5000 | 2 |
| next queue | 0.4701 | 1.2968 | 2.7586 | 1.7586 | 3 |
| random queue | 0.4561 | 1.4737 | 3.2308 | 2.2308 | 4 |
| $\lambda = 2, \mu = 1$ | | | | | |
| common queue | 0.8125 | 2.3750 | 2.9231 | 1.9231 | 1 |
| shortest queue | 0.7647 | 2.3137 | 3.0256 | 2.0256 | 2 |
| next queue | 0.7180 | 2.5055 | 3.4894 | 2.4894 | 3 |
| random queue | 0.6786 | 2.7857 | 4.1053 | 3.1053 | 4 |

Tab. 5.5. Comparison of the four queueing strategies (absolute values)

| $\lambda = 1, \mu = 1$ | **Throughput** | **njob** | **Resp. time** | **Wait. Time** | **Rank** |
|---|---|---|---|---|---|
| common queue | 100.00 | 100.00 | 100.00 | 100.00 | 1 |
| shortest queue | 99.56 | 105.32 | 105.77 | 110.03 | 2 |
| next queue | 98.29 | 114.72 | 116.71 | 128.97 | 3 |
| random queue | 95.36 | 130.36 | 136.69 | 163.60 | 4 |
| $\lambda = 2, \mu = 1$ | | | | | |
| common queue | 100.00 | 100.00 | 100.00 | 100.00 | 1 |
| shortest queue | 94.12 | 97.42 | 103.51 | 105.32 | 2 |
| next queue | 88.37 | 105.49 | 119.37 | 129.44 | 3 |
| random queue | 83.52 | 117.29 | 140.44 | 161.47 | 4 |

Tab. 5.6. Comparison of the four queueing strategies (relative values)

### 5.1.2. Extended Queueing Networks

As shown by the example in the previous section, the queueing network modeling technique can make it easy to compute performance measures. Additionally, the graphical representation helps to intuitively generate models for systems. The conventional queueing networks as described in the previous section, however, cannot represent synchronization and simultaneous resource possession which is important especially for modeling multiprocessors. To overcome this problem, extended queueing networks have been introduced in [Kel76]. Besides *service stations* and *customers* as building blocks, extended queueing networks may also contain *passive centers* (e.g. Links available in Fig. 5.12.) to model passive system resources with a limited number of elements such as disk controllers, memory partitions, and communication links.

In [Lin98] an example to motivate the use of extended queueing networks is given.

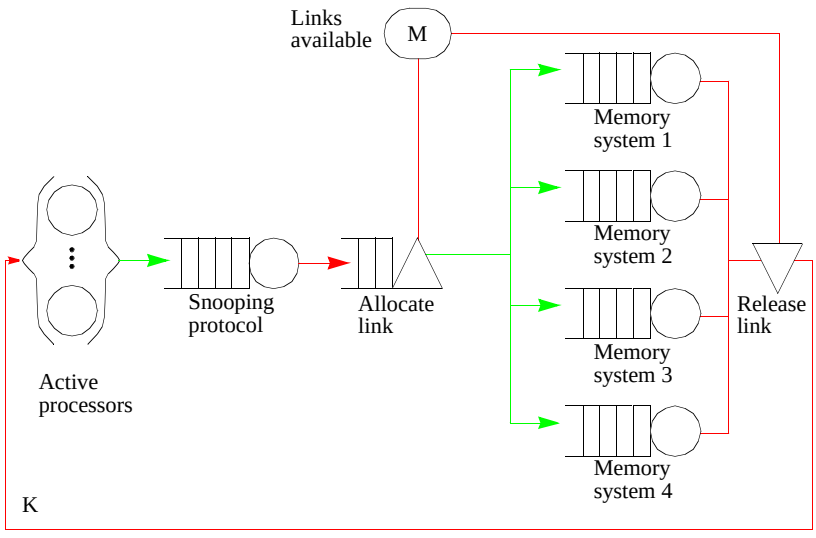



Fig. 5.12. Extended queueing network of multiprocessor system

The multiprocessor in Fig. 5.12. is a shared memory system with $K$ processors and 4 memory modules. To access one of the memory systems, the processor must get both access to a communication link and to the corresponding memory system. After issuing a

memory request, the snooping protocol of the issuing processor determines a free link of the interconnection network. The snooping delay is assumed to be exponentially distributed. The model in Fig. 5.12. takes into account contention for communication links by the passive resource *Links available*. Passive centers contain a finite number of elements, called tokens, which represent the finite number of elements of a passive system resource. Tokens of a passive center like *Links available* are allocated to customers. The customers keep the token until they are serviced at one or more service centers. At a passive center, no active service is provided to visiting customers. By means of visiting *allocate tokens* and *release tokens* which are represented by triangles and diamonds, respectively, the customers request and return a token. An *allocate token* has associated a queue (compare Fig. 5.12.), since the customer might have to wait until a requested token becomes available.

Extended queueing networks also contain building blocks for creating and destroying tokens of a passive resource which allows modeling further synchronization constructs. The introduction of passive centers to queueing networks provides a powerful extension to conventional queueing networks because they enable the representation of customers holding several system resources simultaneously. More information on extended queueing networks can be found in [MNS86].

### 5.1.3. Hierarchical Approach for Complex Systems

The basic technique to handle complex systems, namely hierarchical decomposition, can also be applied to queueing network models. In [Laz84] the concept of flow equivalent service centers (FESC) is introduced. The idea is to split one model into a number of smaller submodels, each of which can then be analyzed in isolation. The solution of the original model is formed by combining the solutions of the submodels. This combination of the submodels is derived by using flow equivalent service centers. A FESC mimics the behavior of the submodel in terms of throughput and customer departure. It is represented as a load dependent service center in the model. At a load dependent service center the service rate varies with the number of customers present. An example of a load dependent service center is pointed out in section 3.2.4. of this book.

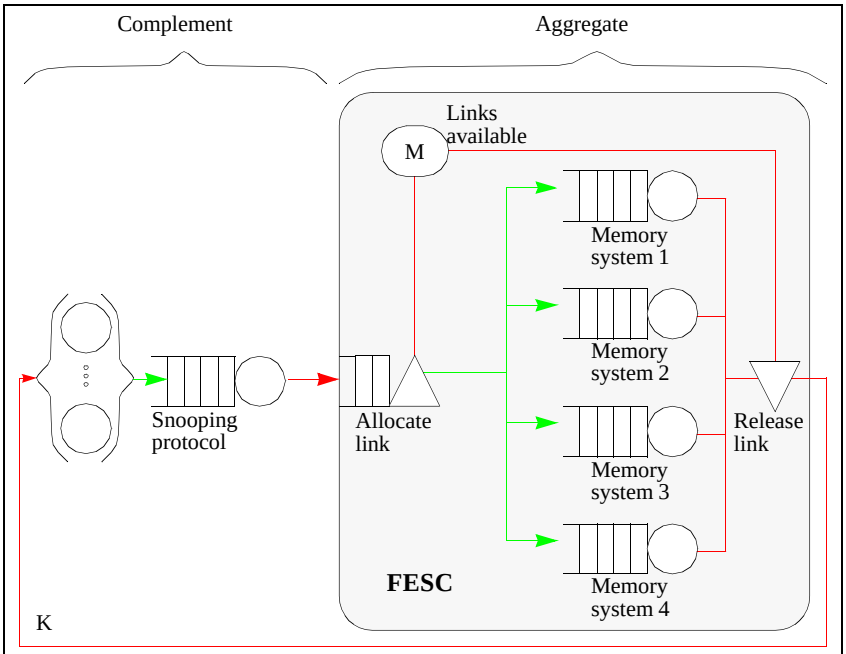



Fig. 5.13. Hierarchical modeling using a flow equivalent service center

The extended queueing network of the previous section is realized with the help of a flow equivalent service center in Fig. 5.13. In the general case, there is an arbitrarily defined subsystem, called the *aggregate*, which interacts with the other service centers in the net-

work, called collectively the *complement* or *complementary network*. The aggregate itself may or may not be representable as a network of service centers. In this example, the complement represents the processing elements, while the aggregate represents the complex memory system. The basic idea of a the hierarchical approach is to replace the entire aggregate by a single service center that mimics its behavior, thus reducing the size of the network to be solved.

For the complement, the aggregate is nothing but a black box with an arrival rate that is the same as the departure rate at the complement. The customers stay in the aggregate for a certain time interval (residence time at the black box) and they leave the black box at a rate or pattern to return to the complement. As long as the black box (FESC) mimics the residence time and the departure process correctly, it does not affect the complement. With respect to the service centers in the complement, any representation of the aggregate that results in accurate inter-departure times is sufficient to solve the network.Thus, the performance measures obtained for the complement are the same regardless of whether the aggregate is represented as a large number of service centers or as a single (mostly load dependent) service center.

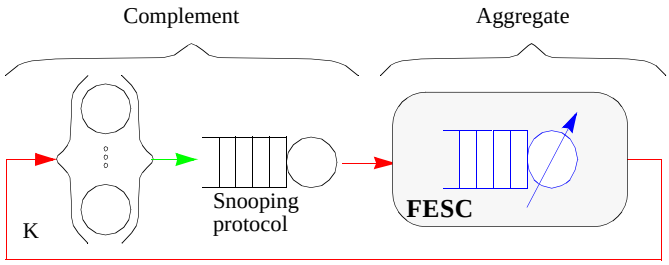



Fig. 5.14. Complement and aggregate using a FESC

**Model decomposition**

Since FESCs can be nested, the hierarchical concept is obvious. Fig. 5.15. shows the model decomposition. At the highest level (*level 0*) the computer system is modeled by FESCs, each of which represents some portion of the system. On the next level, the description for each component is more detailed, if it is a FESC on the previous level. Since FESCs may be nested, each of the models on *level 1* can contain other FESCs. Starting with the solution on the lowest level (*level 2* in Fig. 5.15.) the models are solved. The solution of each *level (k+1)* is used to solve the model on *level k* until finally the highest level (*level 0*) is reached. The definition of the model is top down, the solution of the models is bottom up. More details on hierarchical modeling using FESCs and on the solution of models including FESCs can be found in [Laz84].

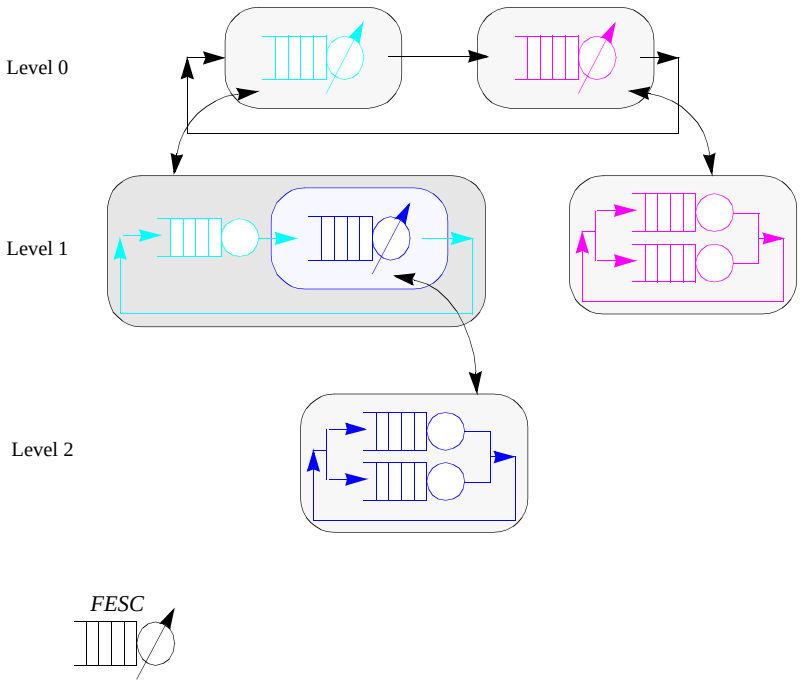



Fig. 5.15. Model decomposition in a hierarchical approach

### 5.1.4. Application Areas for Queueing Networks

Queueing networks have become important tools in the design and analysis of computer systems. For many applications this modeling technique achieves a favorable balance between accuracy and efficiency. The models can be defined, parameterized, and evaluated at relatively low cost. They can be used throughout the design cycle from specification to hardware realization. Systems can be modeled from high level to detailed models of system components.

Several problems arise with the modeling of complex systems with several service stations and several customer classes with different service needs. The graphical representation of such systems is complex, so are the steady-state diagrams and the resulting systems of equations to be solved. Special solution techniques were developed to solve these equation systems. The solution methods vary from direct methods with intelligent pivoting strategies to iterative methods such as Conjugate Gradient methods with special preconditioning.

Especially for multiprocessors with a large number of processing elements (e. g. more than 1000 processors) it is not possible to model the system as a whole. A hierarchical approach like the flow equivalent service center approach as described in the previous section can make it possible to handle these systems.

Another problem is the modeling of the workload involved. The workload is abstracted by parameters of some probability distribution. If it is not possible to describe the actual workload with such a model, it is also not possible to use the queueing networks as modeling technique.

In summary, if a system and its workload can be represented as a queueing network, this modeling technique should be used to predict its performance measures. By changing the parameters of the model, changes in both, workload and system, can easily be evaluated without the need to build expensive hardware prototypes.

## 5.2. Petri Net Models

In 1962 Carl Adam Petri [Pet62] introduced a solution for the problem of representing concurrent or competing processors by a graphical modeling formalism which is called Petri Net.

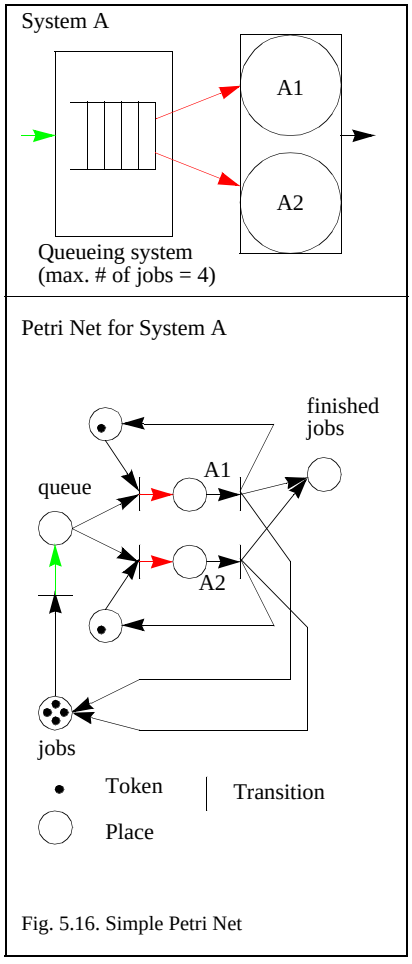



Fig. 5.16. Simple Petri Net

In Fig. 5.16. a simple example for a Petri Net representing a system is outlined. A Petri Net is a directed bipartite Graph. The first set of vertices (circles in Fig. 5.16.) are *places* and the second set of vertices (bars in Fig. 5.16.) are *transitions*. Each place of a Petri Net can hold any number of *tokens* (little blackened circles in Fig. 5.16.). A token is moved from one place to the next place following the arrows, if the transition between the two places fires. A transition can only fire if all places before the transition hold at least one token. If the transition fires, all places before the transition loose one token and all places after the transition gain one token. The place *queue* in Fig. 5.16. represents the queue, the places *A1* and *A2* represent the *service stations* (processors). Up to four jobs can circle in Fig. 5.16., but each service station can only hold up to one job. This is derived through a *"service available token"*, which is transferred to the service station when a job is executed. The absence of the *"service available token"* blocks the transition to the *service* place. After a job is finished, the "*service available token"* is released to its place. The transition before the place named *queue* mimics job arrivals, the transitions after the places *A1* and *A2* are the service transitions.

A queue implies a first come first serve (FCFS) strategy which is obviously not realized by the single place for the token in Fig. 5.16. Replacing the *queue* place of this figure by the Petri Net in Fig. 5.17. would implement a FCFS strategy for the queue. As long as places are available in the queue, the arriving jobs wander from bottom to top. The "service available token" from Fig. 5.16. becomes now a *"queue place available"* token in Fig. 5.17. When a job departs the system, the *"queue place available"* token enables the next job in the queue to forward one place.

Whenever a job forwards a step, its successors in the queue can also forward one step.

Petri Net for queue with FCFS

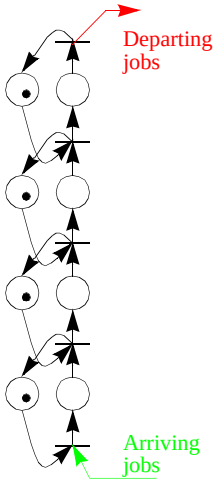



Fig. 5.17. Petri Net for FCFS strategy

Originally, Petri Nets were designed to study qualitative or logical properties of systems exhibiting concurrent and asynchronous behavior. With the lack of a time concept, Petri Nets cannot directly be used for quantitative performance evaluation. The natural approach for this problem is to implement the association of time with the transitions of a Petri Net. The first research proposals in this matter [Sym78], [Mol82] focused on the association of exponentially distributed firing delays to all transitions of a Petri Net. The next step was the introduction of generalized stochastic Petri Nets (GSPN) in 1984 [Mar84]. In a GSPN a transition has either an exponentially distributed firing delay or the transition fires without delay. Since the stochastic process underlying a GSPN can always be represented as a time-continuous Markov Chain, the GSPNs belong to the class of Markovian modeling techniques. This also implies that the stationary analysis of a GSPN requires the solution of a linear system of equations.

## 5.2.1. GSPN Example

The example of the beginning of this section, where steady state diagrams are created from Markov Chains to describe four queueing strategies for jobs arriving at a multiprocessor, is now reconsidered. The same strategies are expressed as GSPNs.
Some definitions are needed to be able to understand the process of transforming a GSPN into a steady state diagram.

### 5.2.1.1 Definitions

**Definition: Marking**

If $P$ is a finite set of places and each place may contain a finite number of tokens, a ***marking*** $M_k$ is a row vector of nonnegative integers and defines the number of tokens in each place $p_i \in P$. For a marking $M_k$, the number of tokens in a particular place $p_i$ is given by the i-th component of $M_k$.

**Definition: Reachability Set**

The set of all markings reachable from the initial marking $M_0$ by firing sequences of transitions and the initial marking itself is denoted as the ***reachability set*** of a GSPN.

**Definition: Reachability Graph**

The ***reachability graph*** of a GSPN is a directed graph *(V,E)*. The set of vertices *V* is given by the reachability set. The set of directed arcs *E* is given by the feasible marking changes in the GSPN due to transition firings in all reachable markings.

**Definition: Vanishing Marking, Tangible Marking**

The markings of the reachability set of a GSPN can be partitioned into two disjoint subsets. Markings, in which at least one immediate transition (transition without firing delay) is enabled are called ***vanishing markings***. Markings in which only timed transitions are enabled are called ***tangible markings***.

**Definition: Tangible Reachability Graph**

The ***tangible reachability graph*** is a directed graph *(V,E)*. The set of vertices *V* is given by the tangible markings of the reachability set. The set of directed arcs *E* is given by the feasible marking changes in the GSPN due to transition firings in all reachable markings.

### 5.2.1.2 Example: Queueing Strategies

**Random Queue Strategy**

As in the example in the beginning of this section, the system consists of two processors (*A1* and *A2*, each with a service rate of $\mu = 1$, i.e. one job is executed per time unit) and of a queueing system which can hold at most 4 jobs (the arrival rate $\lambda$ is 1, i.e. in average one job arrives per time unit, if there are less than 4 jobs waiting).
The random queue strategy (each queue can hold up to four jobs, but the total number of jobs in the system is limited to four jobs) is outlined in Fig. 5.18. When a job enters the queueing system, one of the queues is randomly assigned to it. The Petri Net places for the queues are *q1* and *q2*, the places for the processors are *A1* and *A2*. The places $D_i$ (with $i = 1,2,3,4$) are dummy places to ensure correct sequencing. $D_1$ and $D_2$ make sure that only one job can be processed at a time. $D_3$ is the job pool. $D_4$ gets all the finished jobs. The GSPN contains transitions with exponentially distributed firing delays and transitions

without firing delay. When a delayed transition fires, all following transitions without delay have to fire before the next delayed transition is enabled to fire. Thus, the firing of non delayed transitions has always priority to the firing of delayed transitions. Since the jobs are randomly assigned to one of the queues and the jobs arrive with an exponentially distributed rate of λ, the jobs arrive at each queue with an exponentially distributed rate of λ/2.

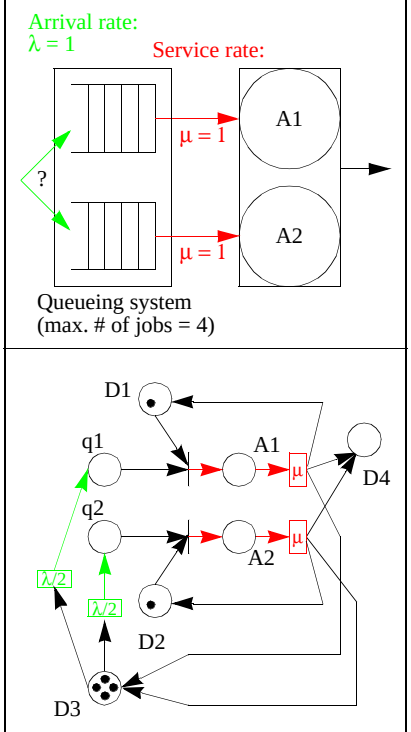



Fig. 5.18. GSPN for random queue strategy

To find a Markov Chain representing the GSPN requires to find the tangible reachability graph for the GSPN. First, the reachability set has to be found. Fig. 5.19. shows how to generate the reachability set. Beginning with a marking $M_0$, all possible markings have to be found. If the change from one marking to the other involves only immediate transitions, the marking is a vanishing marking and is not included in the tangible reachability graph. The first column in Fig. 5.19. identifies vanishing and tangible markings. The last column *S* is used to describe the system state. It consists of a pair *(ij)*, with *i* being the number of jobs in *subsystem I* (consisting of *q1* and *A1*) and *j* being the number of jobs in *subsystem II* (consisting of *q2* and *A2*). For the performance questions, it is only interesting to determine when a job enters one of the subsystems and when a job leaves the subsystems. This is described by all tangible transitions from Fig. 5.19.

| | q1 | A1 | q2 | A2 | D1 | D2 | D3 | S |
|---|---|---|---|---|---|---|---|---|
| $M_0$ | 0 | 0 | 0 | 0 | 1 | 1 | 4 | 00 |
| t | 0 | 0 | 1 | 0 | 1 | 1 | 3 | 01 |
| ***v*** | ***0*** | ***0*** | ***0*** | ***1*** | ***1*** | ***0*** | ***3*** | ***01*** |
| t | 0 | 0 | 1 | 1 | 1 | 0 | 2 | 02 |
| t | 0 | 0 | 2 | 1 | 1 | 0 | 1 | 03 |
| t | 0 | 0 | 3 | 1 | 1 | 0 | 0 | 04 |
| t | 1 | 0 | 0 | 0 | 1 | 1 | 3 | 10 |
| ***v*** | ***0*** | ***1*** | ***0*** | ***0*** | ***0*** | ***1*** | ***3*** | ***10*** |
| t | 1 | 1 | 0 | 0 | 0 | 1 | 2 | 20 |
| t | 2 | 1 | 0 | 0 | 0 | 1 | 1 | 30 |
| t | 3 | 1 | 0 | 0 | 0 | 1 | 0 | 40 |
| t | 0 | 1 | 1 | 0 | 0 | 1 | 2 | 11 |
| t | 1 | 0 | 0 | 1 | 1 | 0 | 2 | 11 |
| ***v*** | ***0*** | ***1*** | ***0*** | ***1*** | ***0*** | ***0*** | ***2*** | ***11*** |
| t | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 21 |
| t | 2 | 1 | 0 | 1 | 0 | 0 | 0 | 31 |
| t | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 22 |
| t | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 12 |
| t | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 13 |

Fig. 5.19. Generation of reachability set

From the reachability set, the tangible reachability graph in Fig. 5.20. can directly be constructed:

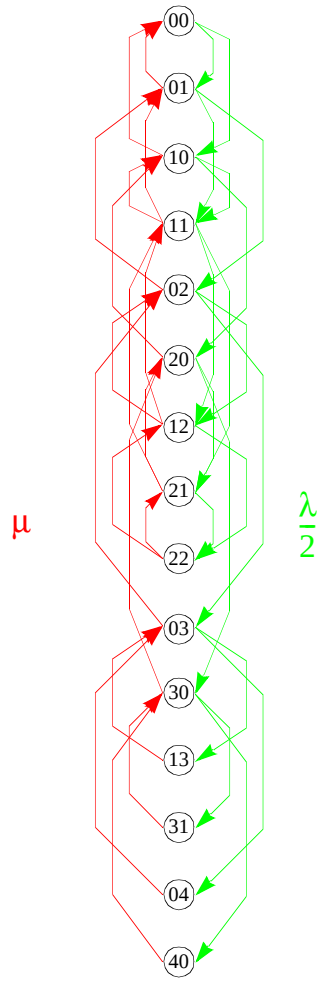



Fig. 5.20. Tangible reachability graph from Petri Net

All arrows pointing from top to bottom symbolize the arrival of a job in one of the subsystems I or II and carry the transition rate of λ/2 for this event. All arrows pointing from bottom to top symbolize the completion of a job in one of the subsystems and carry the transition rate of μ for this event. It can be shown that the graphs in Fig. 5.20. and Fig. 5.8. are identical. As demonstrated in the previous section, the resulting system of flow balance equations can be generated and solved as functions of λ and μ. Since each GSPN can be transformed to a Markov Chain by algorithmic means, the potential of both methods for performance evaluation is similar.

**Common Queue Strategy**

The second strategy is to realize only one queue which can hold up to four jobs. The jobs wait in the queue until one of the processors is available and are then sent to that processor. This strategy is called common queue.

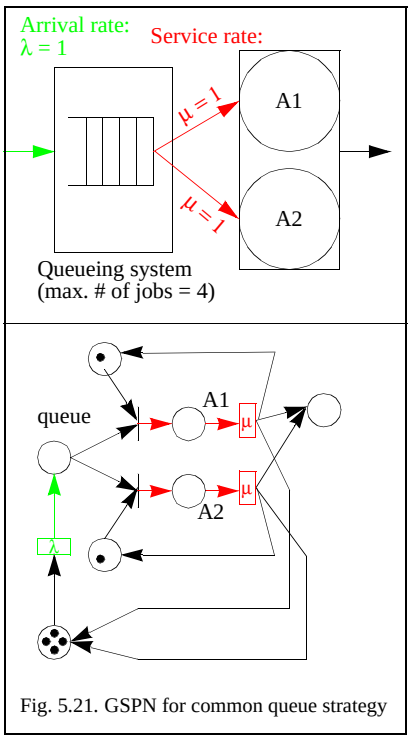


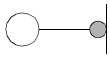

Fig. 5.21. GSPN for common queue strategy

Fig. 5.21. shows the associated GSPN for this strategy. Since the jobs are assigned to one queue at arrival rate λ and from there to the next available processor, the system in Fig. 5.21. looks the same as the basic system in Fig. 5.16., except the transitions with exponential firing delay.

After the generation of the GSPN, the tangible reachability graph and thus, the steady state diagram can be constructed.

**Shortest Queue Strategy**

The third strategy is to realize two queues (each can hold up to two jobs). When a job enters the queueing system, the job is sent to the currently shorter queue. This strategy is called shortest queue. The GSPN for this strategy is not as straightforward as for the other strategies. A new kind of arc which only makes a transition fire, if the associated place holds no token, makes it easier to realize the mechanism for finding the shorter queue. These arcs are called inhibitor arcs. Fig. 5.22. shows the graphical representation of this mechanism.



Transition fires without delay, if place holds no token

Fig. 5.22. No token firing transition

Fig. 5.23. shows the GSPN for the shortest queue strategy. Jobs enter with arrival rate λ. If both queues hold no job (token), one of them is randomly assigned the job. If only one queue holds no job, it gets the job (the transition to assign a job to the queue is then enabled). If both queues hold jobs, the shorter one has to be found. As long as the queue entry transitions are disabled (because the queue holds a token), the large transition *t1* can fire and the queues become empty. The tokens from the jobs residing in the queues are stored in place *s*. When both queues become empty, they held the same number of jobs and the new job is assigned randomly. After the new job is assigned, the transition *t2* is enabled and the queues get their previous job tokens back from place *s*.

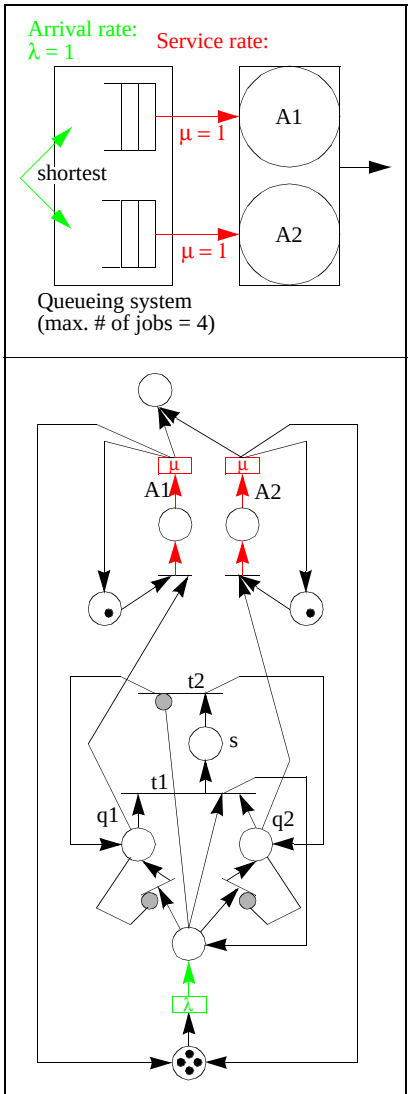



Fig. 5.23. GSPN for shortest queue strategy

**Next Queue Strategy**

The last strategy is similar to the first one. There are two separate queues (each can hold up to four jobs, but the total number of jobs in the system is still limited to four jobs). The strategy is to alternate between the two queues, when a job enters the queueing system, it is sent to the next queue from point of view of its predecessor. This strategy is called next queue, the GSPN for this strategy is outlined in Fig. 5.24.

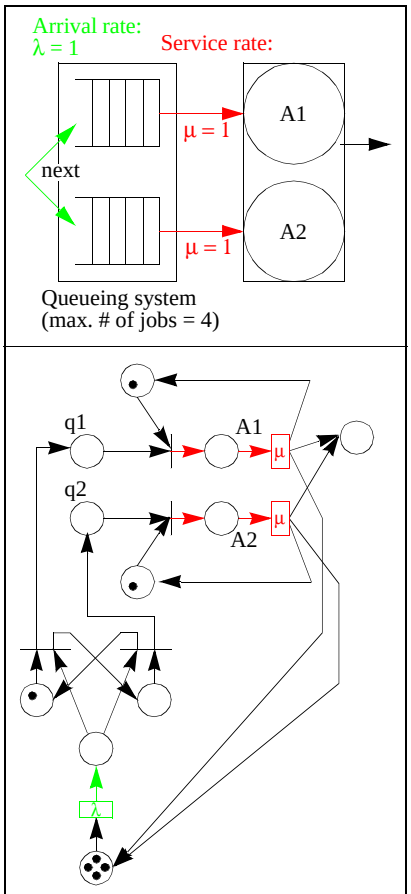



Fig. 5.24. GSPN for next queue strategy

### 5.2.1.3 Queueing Networks versus GSPNs

From the previous sections and examples it might look as if queueing networks and stochastic Petri Nets have the same modeling power. Both techniques use a graphical system representation which makes the modeling formalism attractive. Looking at the GSPN examples in the previous section, the graphical representation of the GSPN building blocks seems to be less compact than the graphical representation of queueing network building blocks. However, each queueing network model can be represented as a GSPN. This consequence is not valid vice versa. There are GSPNs for which no equivalent representation as an extended queueing network exists. In [Ver87] the authors compare the modeling power of both techniques (Queueing networks and GSPNs). They point out that probably the main difference in modeling power is the fact that synchronization can be represented in a GSPN in a more general form than in a queueing network. Using an example for a time-out representation they show a scenario which can be represented as a GSPN but which cannot represented as an extended queueing network because this formalism does not provide means for pre-emptying timed activities.

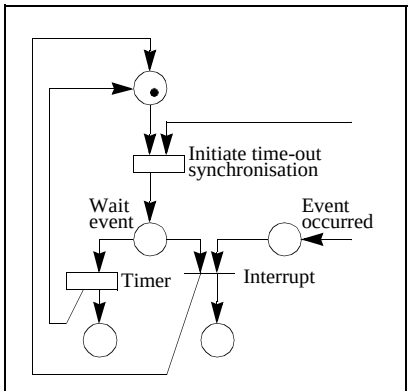



Fig. 5.25. GSPN for a time-out synchronization

Another aspect are the means for the verification of structural properties of the underlying system which are not provided by queueing networks. Since GSPNs are extensions of Petri Nets, these means are inherently existent for GSPNs. More information on modeling using GSPNs can be found in [Mar95].

The introductionary example of section 5.1.2. can also be realized as a GSPN:

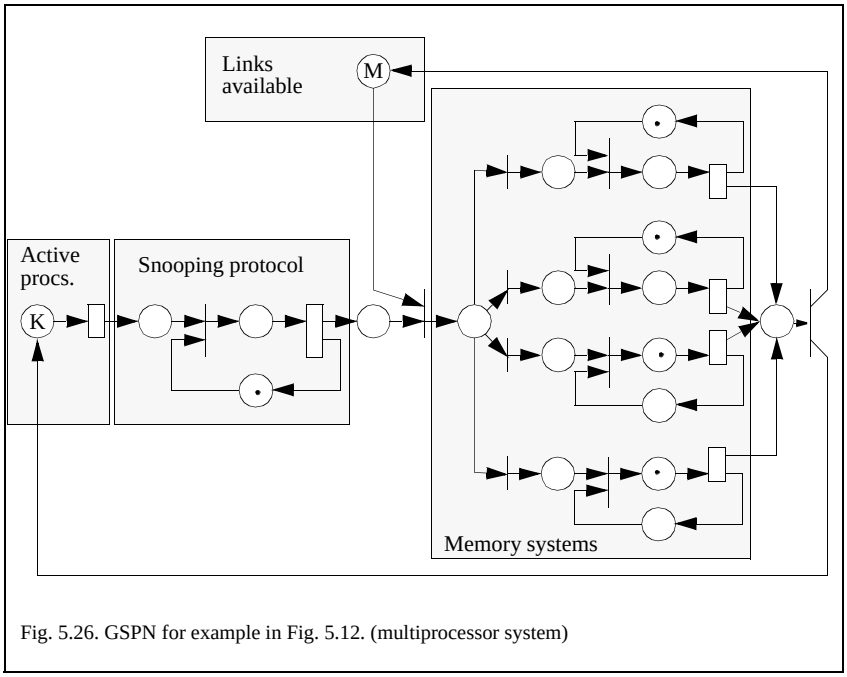



Fig. 5.26. GSPN for example in Fig. 5.12. (multiprocessor system)

### 5.2.2. DSPN Example

Generalized stochastic Petri Nets (GSPN) contain two kinds of transitions, differed by their firing time. Immediate transitions fire without delay after the input conditions are satisfied, exponential transitions fire after an exponentially distributed random delay. The lack of transitions which fire after a constant time delay led to the introduction of deterministic and stochastic Petri Nets (DSPN) as described in [Mar87]. In computer systems, examples for constant time delays are timeouts, setup times for devices (e.g. communication units, vector units), time slices in time shared systems. Especially for concurrent systems synchronization on different levels, DSPNs are well suited. In [Lin98] a multiple reader, single writer system is used as an example to introduce DSPNs.

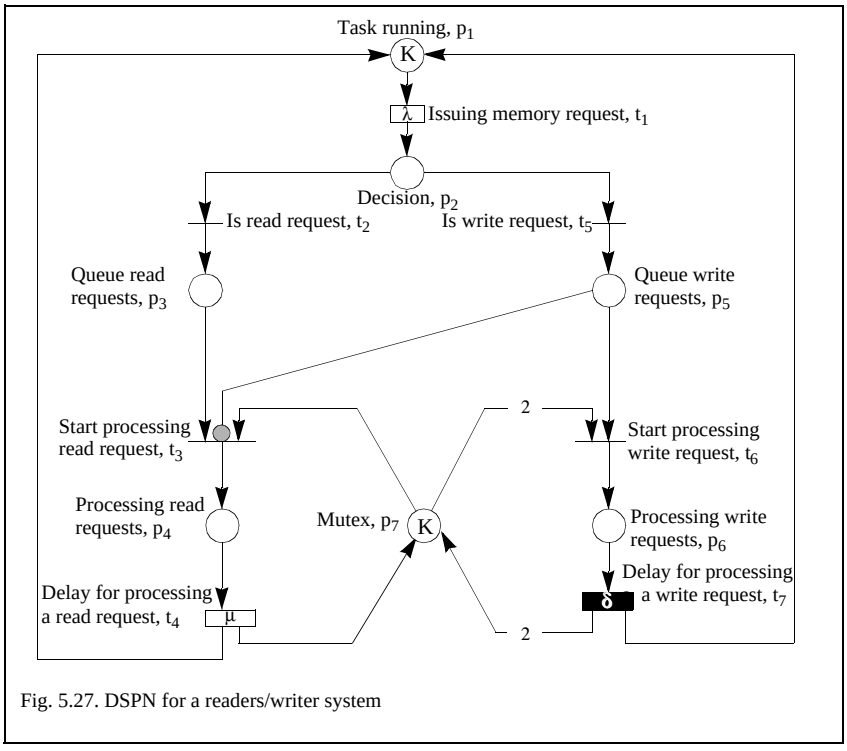



Fig. 5.27. DSPN for a readers/writer system

The elements of the DSPN in Fig. 5.27. are defined as in the GSPNs of the previous section. Two new elements are displayed in this figure. A black bar with white text stands for a transition with constant delay, the text is the delay parameter. An arc with a number associated to it has the multiplicity of that number, if no number is present, the multiplicity

of that arc is one. The text in a place stands for the number of tokens associated with that place.

In the readers system described in this figure, *K* tasks share the resource memory. As long as no write process is present, up to *K* processes are allowed to read from memory simultaneously. As soon as a write process is queued, the read queue gets blocked by the inhibitor arc, thus giving priority to the write process. When all *K* tokens are available in the *Mutex* place, the write request can be processed. After processing and a constant delay, the *K* tokens are given back to the *Mutex* place.

This example demonstrates the modeling power of the DSPN technique. It includes sharing resources, synchronization, and priorities. For a small example *(K=2)* it is now demonstrated how the tangible reachability graph of the DSPN can be constructed. As before, the first step is to generate the reachability set of the DSPN.

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ | $p_6$ | $p_7$ |
|---|---|---|---|---|---|---|---|
| $M_0$ | 2 | 0 | 0 | 0 | 0 | 0 | 2 |
| $M_1$ | 1 | 1 | 0 | 0 | 0 | 0 | 2 |
| ***$M_2$*** | ***1*** | ***0*** | ***1*** | ***0*** | ***0*** | ***0*** | ***2*** |
| ***$M_3$*** | ***1*** | ***0*** | ***0*** | ***1*** | ***0*** | ***0*** | ***1*** |
| $M_4$ | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| ***$M_5$*** | ***0*** | ***0*** | ***1*** | ***1*** | ***0*** | ***0*** | ***1*** |
| ***$M_6$*** | ***0*** | ***0*** | ***0*** | ***2*** | ***0*** | ***0*** | ***0*** |
| $M_7$ | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| $M_8$ | 1 | 0 | 0 | 0 | 1 | 0 | 2 |
| ***$M_9$*** | ***1*** | ***0*** | ***0*** | ***0*** | ***0*** | ***1*** | ***0*** |
| $M_{10}$ | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| ***$M_{11}$*** | ***0*** | ***0*** | ***1*** | ***0*** | ***0*** | ***1*** | ***0*** |
| $M_{12}$ | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

Fig. 5.28. Generation of reachability set (K=2)

The tangible markings for *K=2* are *{$M_0$, $M_1$, $M_4$, $M_7$, $M_8$, $M_{10}$, $M_{12}$}*, the vanishing markings are *{$M_2$, $M_3$, $M_5$, $M_6$, $M_9$, $M_{11}$}*. The reachability graph resulting from the reachability set in Fig. 5.28. is:

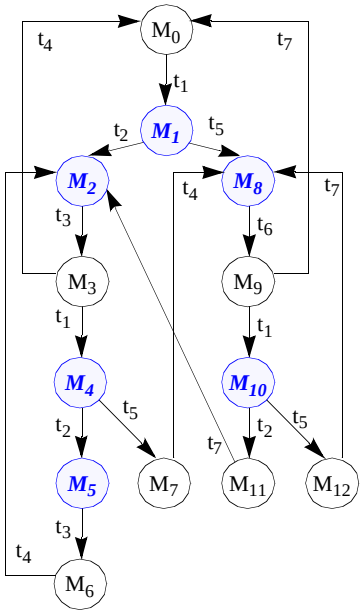



Fig. 5.29. Reachability graph (K=2)

The vanishing markings are highlighted in the reachability graph. By removing the vanishing markings and adjusting the arcs, the tangible reachability graph (as shown in Fig. 5.30.) can be generated. One arc in a tangible reachability graph can represent the firing of a transition sequence, but only one of the transitions in this sequence can be a timed transition. In Fig. 5.30. the first transition of an arc is the timed transition, the transitions after the comma are the immediate transitions. In [Lin98], an algorithm to automatically generate the tangible reachability graph of a deterministic stochastic Petri-Net is outlined. [Lin98] also contains a software package (DSPNexpress) with graphical interface to create and solve DSPNs.

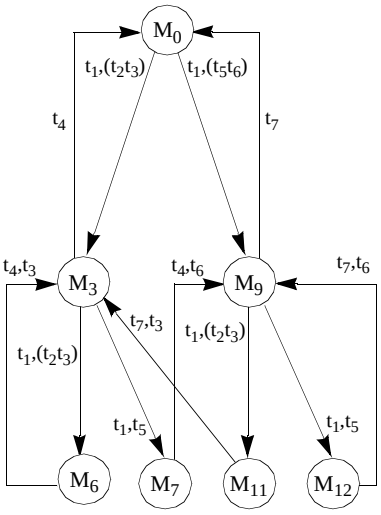



Fig. 5.30. Tangible reachability graph (K=2)

After the tangible reachability graph is generated, numerical techniques are available to solve the system and find the performance measures. More details on these methods can be found in [Lin98].

### 5.2.3. Application Areas for Petri Nets

Like queueing networks, the modeling techniques involving Petri Nets (stochastic and deterministic) have become important tools in the design and analysis of computer systems. These models can be defined, parameterized, and most of the times evaluated at relatively low cost. For complex models a considerable numerical effort might be necessary to solve them. Since great progress in numerical algorithms has led to reducing the numerical complexity of the solvers by orders of magnitude and since modern computers contain large main memories and fast processors, it is now possible to solve rather complex Petri Net models. An example for this development is the DSPNexpress software distributed with [Lin98].

The Petri Net models can be used throughout the design cycle from specification to hardware realization. Systems can be modeled from high level to detailed models of system components.

Especially in the design and development of multiprocessors, Petri Net modeling techniques are well suited. They inherently support and address issues like synchronization and concurrency. Besides modeling the timing behavior it is also possible to use them for testing formal properties (like deadlocks, aliveness), if all timed transitions are substituted by immediate transitions.

A practical problem is to model massively parallel multiprocessors with up to thousands of processors. In theory, it is no problem for this technique to model such systems, but the resulting models are too complex to be handled (in terms of memory and computation time).

As in the case of queueing networks, another problem is the modeling of the workload involved. There are basically two choices to be made, abstract the workload on a very high level (job level) or model program behavior. The first choice might not be detailed enough for the performance measures to be observed, whereas the second choice will lead to complex models. The graphical representation of system behavior is straightforward and easy to understand for a system engineer. The representation of a C-program as a Petri Net might be more complicate as the C-program itself.

In summary, if a system and its workload can be represented as a Petri Net (GSPN or DSPN), this modeling technique should be used to evaluate its performance measures. Similar to queueing networks, by changing the parameters of the model, changes in workload and system behavior, can easily be evaluated without the need to build expensive hardware prototypes.

## 5.3. Quantitative Performance Evaluation

The previous two sections describe stochastic performance evaluation techniques. The workloads and systems are realized using parameterized stochastic models. Workload and target system are modeled using the same formalism (Queuing Networks or Petri Nets). The emphasis is on both, finding the correct model for the *workload* and *system*. This section presents a different approach [CMS97], which uses *parallel profiles* on different abstraction levels to characterize a parallel workload and its execution on a target system. Together with assumptions for the underlying hardware (number of processors, communication latency), it is possible to predict performance behavior measures like *execution time* (in timesteps), *speedup* and *efficiency*.

Starting point for this so called *quantitative performance evaluation* is a formalism to describe and find an average degree of parallelism for a workload. Each problem can be considered on different levels of abstraction. These levels form a hierarchy from a *natural problem description* to a running *parallel program* (compare Tab. 5.7.). Going from one level to the next causes losses in the average degree of parallelism. The first four levels in Tab. 5.7. are independent of the target system architecture. Thus, the parallel profiles of these levels can be determined without looking at specific systems.

| **Level of abstraction** | **Loss of parallelism caused by** |
|---|---|
| Problem (in nature) | |
| natural problem description | level of description, granularity |
| algorithmic problem description | problem decomposition, granularity |
| high level programming language program | communication primitives,<br>choice of data structures |
| machine language program | number of processors,<br>mapping of task graph onto machine |
| running program | ressource conflicts,<br>communication bottlenecks,<br>data dependencies |

Tab. 5.7. Loss of parallelism caused by problem formulation

Using the different levels of abstraction from Tab. 5.7., two different parallel profiles, the *algorithmic profile* and the *execution profile*, can be defined (compare Fig. 5.31.). To determine the *algorithmic profile* an algorithm is subdivided into elementary operations. Depending on the problem nature, such elementary operations can be arithmetic operations,

logical operations, data base accesses, etc. The algorithmic profile *P(i)* is defined as the number of elementary operations which can be executed concurrently at time *i*. The interval for each timestep $i = 1, 2,..., i_m$ can be chosen arbitrarily by the user. For example, a timestep can be defined as a multiple of the clock frequency of the target system. It can also be defined as the time needed for an arithmetic operation, for a logical operation or for a data base access. To determine the algorithmic profile, an *ideal target system* is assumed, i.e. an indefinite number of processors is available which have access to a global address space without communication and synchronization delays.

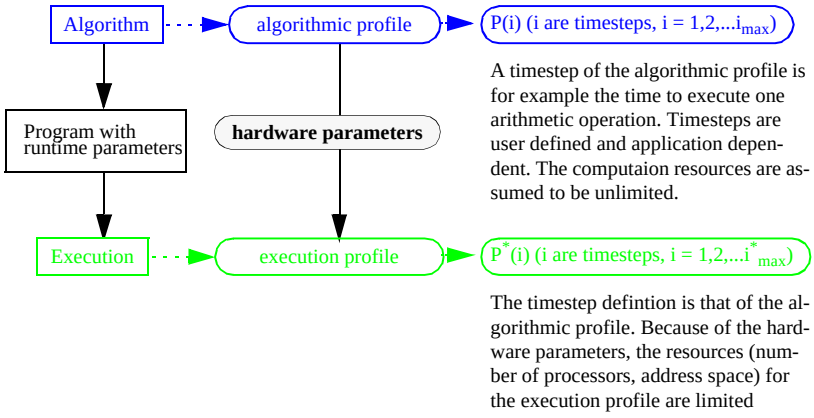



Fig. 5.31. Parallel profiles from algorithm to program execution

### Algorithmic Profile

A parallel matrix multiplication is now used as an example to demonstrate how to construct an *algorithmic profile* and to show how to transform this profile into an *execution profile*. Two matrices *A(N,N)* and *B(N,N)* (with *N = the dimension of the quadratic matrices*) are to be multiplied. The result matrix is also a quadratic matrix of dimension *N*.

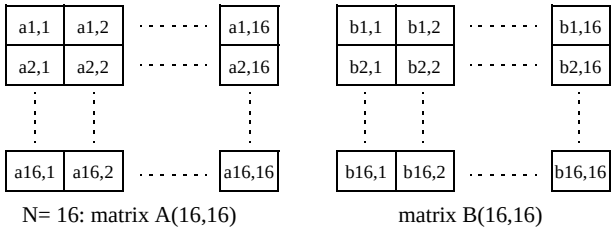



Fig. 5.32. Matrix multiplication of A and B

To compute one element of the result matrix, a dotproduct of *one row of matrix A* and *one column of matrix B* has to be carried out. This dotproduct requires *N* multiplications (compare Fig. 5.34.) and some additions (compare Fig. 5.35.). If the assumption of unlimited resources (unlimited number of processors, unlimited global address space) is made, all multiplications for one dotproduct can be performed concurrently. Furthermore, the multiplications for all dotproducts of the result matrix can be calculated concurrently in one timestep (the length of one timestep is defined as the time to execute one multiplication or one addition). In total, $N^2$ dotproducts with *N* multiplications each (=$N^3$ multiplications) can be performed in parallel at the first timestep of the algorithmic profile (compare Fig. 5.34. and Fig. 5.33.). Thus, the first bar of the algorithmic profile with $N^3$ elementary operations of Fig. 5.33. is found. The next timesteps are needed for the concurrent calculation of the dotproducts for each element of the result matrix (compare Fig. 5.35.)

.

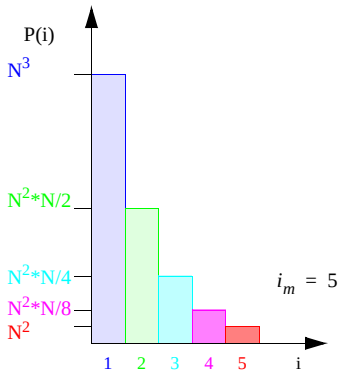
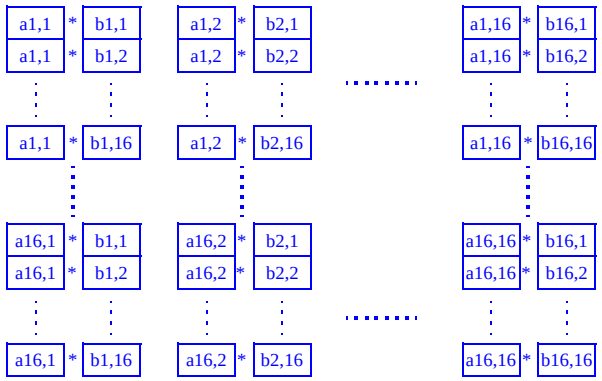



Fig. 5.33. Algorithmic profile for parallel matrix multiply with N = 16




timestep i = 1: execute all necessary multiplications (16*16*16) = ($N^3$) operations

Fig. 5.34. Parallel algorithm for matrix multiplication (part 1)

To compute the dotproducts, an algorithm with a treelike topology is used. The first step (*timestep i = 2 in Fig. 5.35.*) computes *(N/2)* sums for *N* pairs of previously calculated products. These sums are calculated concurrently for all $N^2$ elements of the result matrix. Thus, ($N^2$*N/2*) elementary operations are performed in this timestep (compare Fig. 5.33.). In the next step (*timestep i = 3 in Fig. 5.35.*), the sums of the previous step are con-

currently added by pairs to produce the next results. The number of elementary operations needed for this timestep are ($N^2$*$N/4$). This method is repeated for the next timesteps until the calculation of the dotproducts is finished in the last step. The total number of operations for the parallel matrix multiplication is:

$$16^3 + 16^2 \cdot 8 + 16^2 \cdot 4 + 16^2 \cdot 2 + 16^2 = 7936$$

Five timesteps ($i_m = 5$) are needed to perform these operations which results in an average degree of parallelism $p$ of *7936/5 = 1587*.

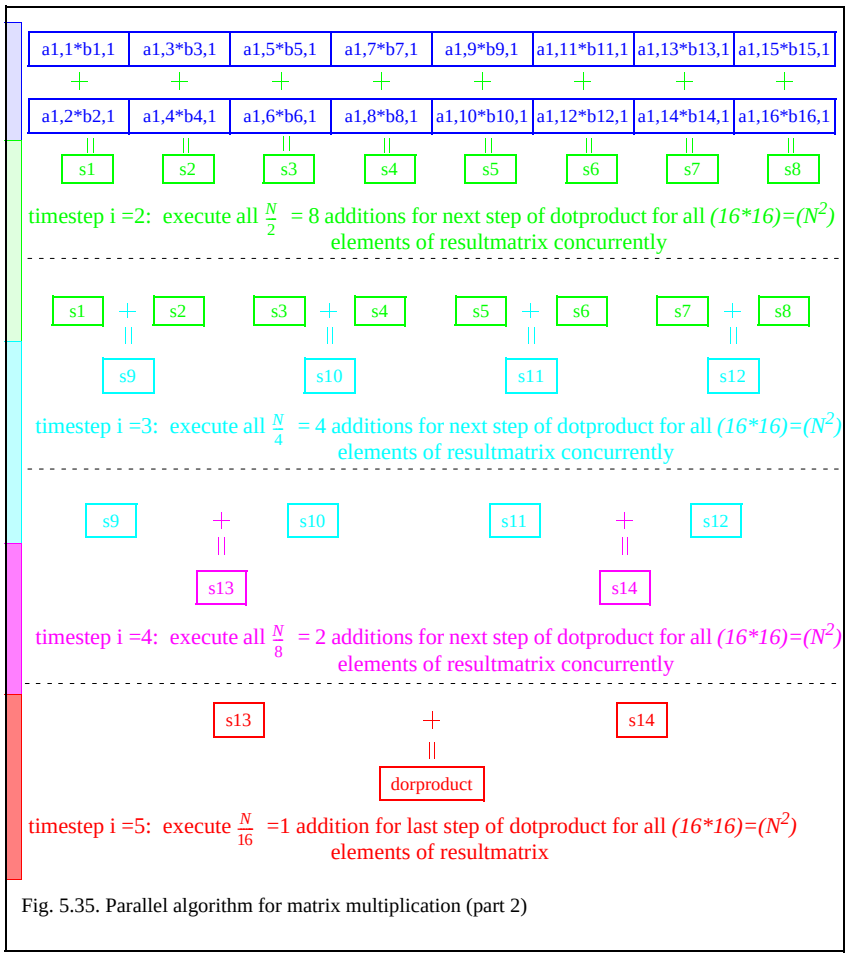



Fig. 5.35. Parallel algorithm for matrix multiplication (part 2)

More general, the total number of all elementary operations is called work $A$ with:

$$A = \sum_{i=1}^{i_m} P(i) \qquad \text{(Eq.5.5)}$$

This formula does not include additional operations for communication and synchronization) that might be necessary for the execution of the algorithm on a multiprocessor system. The underlying algorithmic profile also assumes that unlimited resources are available.

The average degree of parallelism $p$ for the algorithmic profile is defined as:

$$p = \frac{1}{i_m} \cdot \sum_{i=1}^{i_m} P(i) = \frac{A}{i_m} \qquad \text{(Eq.5.6)}$$

If all elementary operations could be evenly distributed over time and the previously made assumption of unlimited resources is used, $p$ is identical with the expected speedup $S$ of the algorithm.

**Execution Profile**

Going from the *algorithmic profile* to the *execution profile* requires to consider the available resources. Applied to the implementation of the matrix multiplication on a real multiprocessor this means that no more than *nprocs* (with *nprocs* = number of available processors) elementary operations can be computed concurrently at any timestep. In Fig. 5.36. the execution profile of the parallel matrix multiplication is visualized for *nprocs=8* processors. One computation *ci,j* *(*with *i=1,2,..,N* and *j=1,2,...,N)* is the calculation of one dotproduct of the result matrix. It can be done in 6 elementary timesteps. In the first two timesteps of each step *ci,j*, all the multiplications of the dotproduct are computed concurrently. Since only 8 processors are available, two timesteps are needed for these 16 elementary operations. In the next four timesteps of each step *ci,j* the sum of the dotproduct is calculated as described in Fig. 5.35.

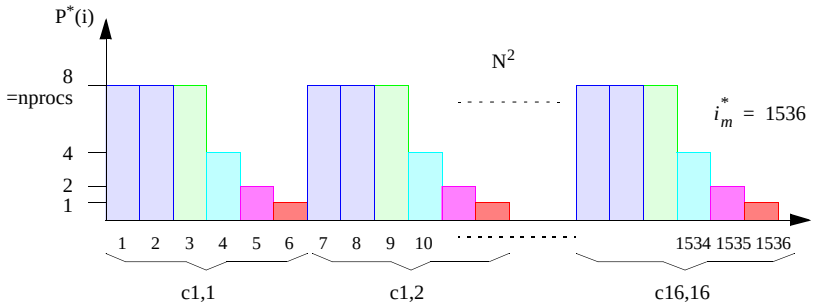



Fig. 5.36. Execution profile for parallel matrix multiply with N = 16 and nprocs = 8

The total number of timesteps for the *execution profile* $i_m^*$ increases to 1536 compared with $i_m = 5$ timesteps of the corresponding *algorithmic* profile. The definition of work which is the total number of elementary operations can also be applied to the *execution profile*:

$$A^* = \sum_{i=1}^{i^*_m} P^*(i) \qquad \text{(Eq.5.7)}$$

The total number of operations remains the same for both profiles ($A = A^*$).

The average degree of parallelism as defined in (Eq.5.3) decreases with increasing $i_m^*$:

$$p^* = \frac{A^*}{i_m^*} = \frac{A}{i_m^*} \qquad \text{(Eq.5.8)}$$

*B(i)* is defined as the number of elementary operations which can be performed concurrently at one timestep *i* for a given architecture. In the example above, *B(i) = const = nprocs*. The maximum work a parallel machine can execute for a given number of timesteps $i_m^*$ is:

$$A_{max} = \sum_{i=1}^{i_m^*} B(i) = i_m^* \cdot nprocs \qquad \text{(Eq.5.9)}$$

$A_{max}$ is also called the *capacity* of the machine. The capacity definition can be used to calculate the *efficiency* of an execution because efficiency is defined as:

$$\varepsilon = \frac{\text{performed work}}{\text{capacity}} = \frac{A}{A_{max}} \qquad \text{(Eq.5.10)}$$

Using (Eq.5.8) and (Eq.5.9), efficiency can also be expressed as:

$$\varepsilon = \frac{A}{A_{max}} = \frac{A}{i_m^* \cdot nprocs} = \frac{p^*}{N}$$

For the example of the parallel matrix multiplication, the performed work (total number of operations) is 7936. Using the execution profile of Fig. 5.36. results in a capacity of:

$$i_m^* \cdot nprocs = 1536 \cdot 8 = 12288$$

Thus, the efficiency can be calculated as:

$$\varepsilon = \frac{7936}{12288} = 0.645$$

This is an optimistic estimation since no losses in efficiency caused by communication and synchronization are considered. The efficiency loss is only caused by unbalanced load (compare Fig. 5.36.).

Using the assumption that one timestep is the time to perform one elementary operation, the total number of operations and the resulting number of timesteps in the execution profile determine the *speedup S*:

$$S_{nprocs} = \frac{\text{performed work}}{\text{\# of tmesteps for nprocs processors}}$$

For one processor, $S_1 = 1$ because the total number of operations (=performed work) equals the total number of timesteps of the execution profile (compare Fig. 5.37.).

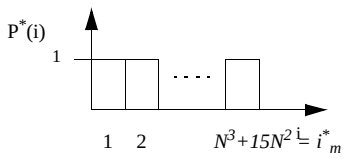



Fig. 5.37. Execution profile for matrix multiply with nprocs = 1

For *nprocs* processors, the total number of timesteps will decrease because some operations will be executed in parallel.

$$S_{nprocs} = \frac{A}{i_m^*}$$

For the example of the parallel matrix multiplication, the performed work (total number of operations) is 7936. Using the execution profile of Fig. 5.36. results in a speedup of:

$$S_8 = \frac{7936}{1536} = 5.17 \quad (\equiv p^*)$$

In section 2.2.3. of this book, the relation between efficiency and speedup is defined as:

$$\boxed{\varepsilon = \frac{S_P}{P}} \qquad \text{with } P = nprocs.$$

Thus, the efficiency can be used to calculate the speedup:

$$S_{nprocs} = nprocs \cdot \varepsilon$$

### Distribution of Parallelism

Execution or algorithmic profiles can be used to derive the *distribution of parallelism*. The behavior of a parallel program is characterized by a distribution of parallelism *V(P)*. Each value *V(P)* (with $P=0,1,2,...,P_{max}$ and $P_{max}$ is the maximum parallel degree) is the probability of a profile to have the parallel degree *P(i)* at an arbitrarily chosen timestep *i*. *V(P)* can also be defined as the percentage of timesteps that a program has the parallel degree *P*. With $P_{max}$ is the maximum parallel degree for a program, *V(P)* can be calculated as:

$$V(P) = \frac{\sum_{i=1}^{i_m} z(i, P')}{i_m}, \quad \text{with } 0 \le P \le P_{max}$$

and $$z = \begin{cases} 1 & P = P' \\ 0 & P \neq P' \end{cases}.$$

The distribution of parallelism resulting of the execution profile in Fig. 5.36. is described in Fig. 5.38.

*V(P)* for *P = 0,1,2,...,8* has the values:
*V(0) = V(3) = V(5) = V(6) = V(7) = 0*
$V(1) = V(2) = V(4) = \frac{1}{6}$

$V(8) = \frac{1}{2}$

The sum of all probabilities *V(P)* must be zero:

$$\sum V(P) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{2} = 1$$

Using *V(P)* the performed work can be calculated as:

$$A = i_m \cdot \sum_{P=0}^{P_{max}} V(P) \cdot P$$

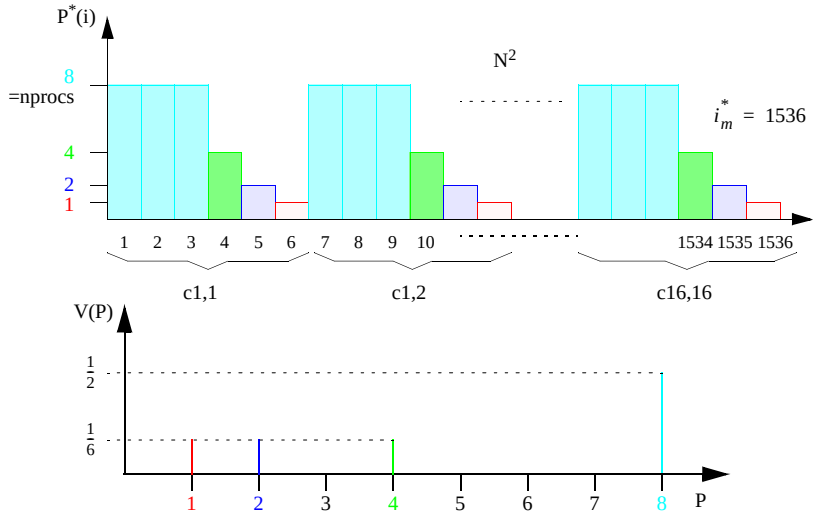



Fig. 5.38. Distribution of parallelism

### Speedup with Latency

In the previous section, communication costs are not considered. If communication is carried out as an operation which is overlapped with computation the derivation of the formulae for speedup and efficiency is correct. In general, communication cannot always be overlapped by computation. A new variable $L$ (=latency) is now introduced. The latency $L$ is a measure for the time needed for communication. It is given in multiples of the system frequency. The execution profile with latency $L$ can be used to predict upper and lower boundaries for speedup and efficiency of a parallel program.

### Lower Boundary for Speedup (worst case)

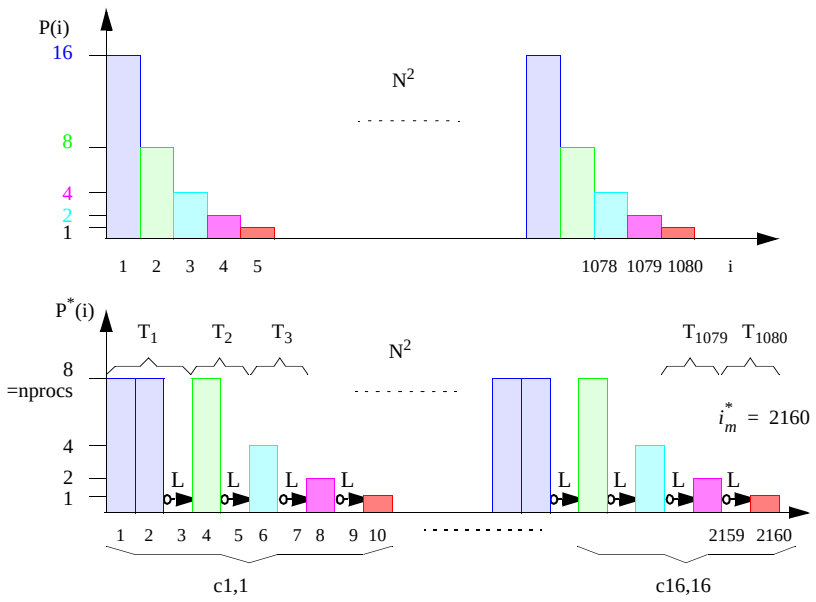



Fig. 5.39. From algorithmic profile to execution profile with latency (L=1)

The upper part of Fig. 5.39. shows the algorithmic profile for the execution profile of Fig. 5.36. The columns represent the steps of the algorithm for the parallel matrix multiplication. The first column is step one (all multiplications for one element of the result matrix). The next four columns are the steps for the calculation of the dotproduct. These five steps for the computation of one element of the result matrix have to be repeated for all $N^2$ elements of the result matrix. The lower part of Fig. 5.39. shows the resulting execution profile for eight processors including latency. In addition to the processor constraint,

the introduction of latency for the communications which occur after each computation step increases the total number of time steps (compare lower part of Fig. 5.39. and Fig. 5.36.). As an example, the assumption is made that each communication needs one time unit and that the communication cannot be overlapped by computation. The new elementary timestep $T_i$ including communication time is defined as:

$$T_i \leq \left\lceil \frac{P(i)}{N} \right\rceil + L \qquad \text{(Eq.5.11)}$$

The total time can be calculated as:

$$i^*_m = \sum_{i=1}^{i_m} T_i \qquad \text{(Eq.5.12)}$$

With (Eq.5.11) and (Eq.5.12) $i^*_m$ can be estimated as:

$$i^*_m \leq i_m \cdot L + \sum_{i=1}^{i_m} \left\lceil \frac{P(i)}{N} \right\rceil \qquad \text{(Eq.5.13)}$$

Using $\left(\frac{a}{b} + 1 > \left\lceil \frac{a}{b} \right\rceil\right)$ the total number of timesteps can be estimated as:

$$i^*_m < i_m \cdot (L+1) + \frac{\sum_{i=1}^{i_m} P(i)}{N} \qquad \text{(Eq.5.14)}$$

Using (Eq.5.5) results in:

$$i^*_m < i_m \cdot (L+1) + \frac{A}{N} \qquad \text{(Eq.5.15)}$$

Using (Eq.5.15) and

$$S = \frac{A}{i^*_m} \; ; \; p = \frac{A}{i_m} \; ; \; ly = L + 1$$

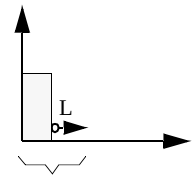



*Fig. 5.40. ly = L+1*

the lower bound for speedup can be estimated as:

$$S > \frac{N}{1 + \frac{ly}{p} \cdot N}, \; \textit{with } ly \geq 1 \qquad \text{(Eq.5.16)}$$

**Upper Boundary for Speedup (best case)**

To estimate an upper boundary for the speed-up, the assumption is made that the communication for each phase can be initiated after the first parallel computation step and that the communication can be (at least partially) overlapped by computation.

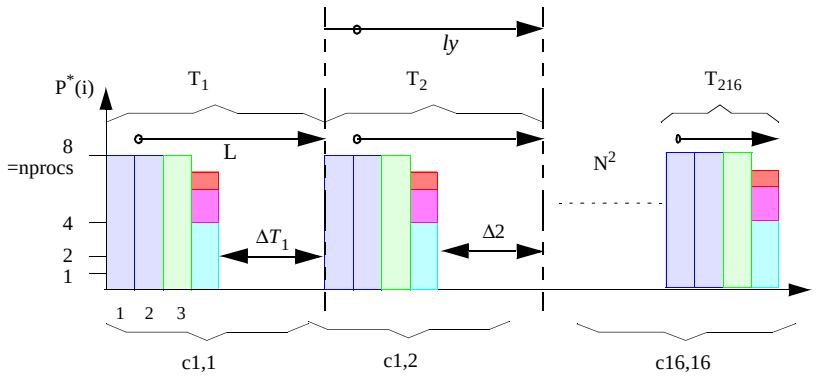



Fig. 5.41. Execution profile with latency hiding

Using these assumptions, the new elementary timestep $T_i$ including communication time is defined as:

$$T_i = max\left(ly, \left\lceil \frac{P(i)}{N} \right\rceil\right) \qquad \text{(Eq.5.17)}$$

As before, *ly* is defined as $L + 1$.

$$T_i = \left\lceil \frac{P(i)}{N} \right\rceil + max(0, \Delta T_i) \qquad \text{(Eq.5.18)}$$

Using $\Delta T_i = ly - \left\lceil \frac{P(i)}{N} \right\rceil$

the total time can be calculated as:

$$i^*_m = \sum_{i=1}^{i_m} T_i =$$

$$\sum_{i=1}^{i_m} \left\lceil \frac{P(i)}{N} \right\rceil + \sum_{i=1}^{i_m} (max(0, \Delta T_i))$$

Only positive values of $\Delta T_i$ have to be considered: $\Delta T_i > 0$: $P(i) < ly \bullet N$.

The distribution of parallelism can now be used to estimate an upper boundary for speedup. The frequency *D(P)* of a parallel degree *P(i)* is defined by the probability *V(P)* multiplied with the number of steps $i_m$:

$D(P) = i_m \bullet V(P)$

The sum of $D(P) \bullet \Delta T_i$ from $P=1$ to $P=lyN$ results in

$$i_m^* = \sum_{i=1}^{i_m} \left\lceil \frac{P(i)}{N} \right\rceil + \sum_{P=1}^{lyN} \left[ D(P) \cdot \left( ly - \left\lceil \frac{P(i)}{N} \right\rceil \right) \right]$$

With $x \le \lceil x \rceil$ :

$$i_m^* \ge \sum_{i=1}^{i_m} \frac{P(i)}{N} + ly \sum_{P=1}^{lyN} D(P) - \frac{1}{N} \sum_{P=1}^{lyN} D(P) \cdot P$$

With (Eq.5.5)

$$i_m^* \ge \frac{A}{N} + ly \sum_{P=1}^{lyN} D(P) - \frac{1}{N} \sum_{P=1}^{lyN} D(P) \cdot P \qquad \text{(Eq.5.19)}$$

**Start comment: Comparison of different distribution functions $D_i$**

Using a distribution function $D$ the total number of steps $i_m$ and the parallel work $A$ can be calculated as:

$$\sum_{P=1}^{P_{max}} D(P) = i_m \qquad \text{(Eq.5.20)}$$

$$\sum_{P=1}^{P_{max}} P \cdot D(P) = A \qquad \text{(Eq.5.21)}$$

**Distribution function $D_1$:**

Each parallel degree $P = 1, 2, \ldots, P_{max}$ appears $\frac{i_m}{P_{max}}$ times: $D_1(P) = \frac{i_m}{P_{max}}$

$$A_1 = \sum_{P=1}^{P_{max}} P \cdot \frac{i_m}{P_{max}} = (P_{max} + 1) \cdot \frac{i_m}{2}$$

$$p_1 = \frac{P_{max} + 1}{2}$$

$$p_1 \approx \frac{P_{max}}{2} \qquad (P_{max} \gg 1)$$

**Distribution function $D_2$:**

All $i_m$ steps have parallel degree $P = P_{max}$.

$A_2 = P_{max} \bullet i_m$; $\quad p_2 \approx P_{max}$

**Comparison:**

$$A_2 - A_1 = \frac{i_m}{2} \cdot (P_{max} - 1) \ge 0$$

In general, for fixed $i_m$ und $P_{max} > 1$ the relation $A_2 > A_i$ holds for any $D_i(P)$

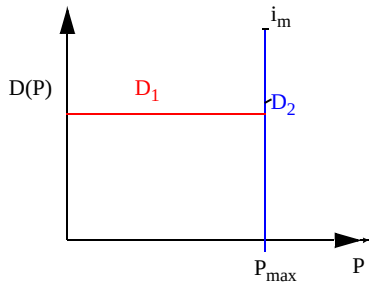



Fig. 5.42. Distribution functions $D_1$ and $D_2$

**End of comment**

**Three cases for the relation between $P_{max}$ and $lyN$ are considered:**

- **Case A: $P_{max} \le lyN$**

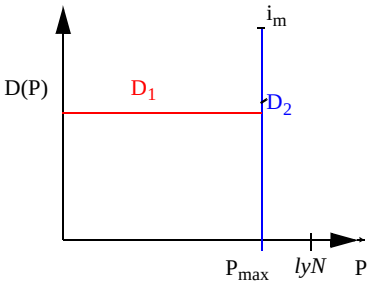



Fig. 5.43. $D_1$ and $D_2$ with $P_{max} \le lyN$

With (Eq.5.19) the total number of steps can be estimated as:

$$i^*_m \ge \frac{A}{N} + ly \cdot i_m - \frac{A}{N} \qquad \text{(Eq.5.20)}$$

(Eq.5.20) can be simplified:

$$i^*_m \ge (ly \cdot i_m) \qquad \text{(Eq.5.21)}$$

An upper boundary for speedup using the distribution functions $D_1$ and $D_2$ can be made:

$$S_1 \le \frac{p}{ly} \quad \text{for} \quad p \le \frac{lyN+1}{2}, (D_1) \qquad \text{(Eq.5.22)}$$

$$S_2 \le \frac{p}{ly} \quad \text{for} \quad p \le lyN, (D_2) \qquad \text{(Eq.5.23)}$$

- **Case B: $P_{max} > lyN$ and distribution $D_1$**

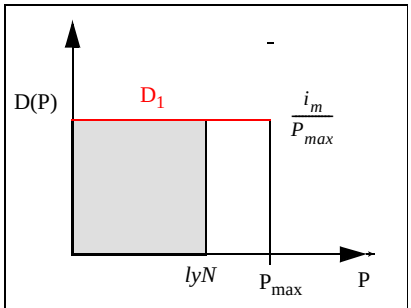



Fig. 5.44. $D_1$ with $P_{max} \ge lyN$

$$\sum_{P=1}^{lyN} D(P) = \frac{i_m}{P_{max}} \cdot lyN$$

$$\sum_{P=1}^{lyN} P \cdot D(P) = \frac{i_m}{P_{max}} \cdot lyN \cdot \frac{lyN+1}{2}$$

Using (Eq.5.19):

$$i^*_m \ge \frac{A}{N} + ly \cdot \frac{lyN \cdot i_m}{P_{max}} - \frac{1}{N} \cdot \frac{i_m}{P_{max}} \cdot lyN \cdot \frac{lyN+1}{2}$$

with $P_{max} \approx 2p$:

$$i^*_m \ge \frac{P \cdot i_m}{N} + \frac{ly^2 \cdot N \cdot i_m}{2p} - \frac{i_m \cdot ly \cdot (lyN+1)}{4p}$$

$$S = \frac{p \cdot i_m}{i^*_m} \; : \; S \le \frac{N}{1 + \frac{lyN(lyN-1)}{4p^2}}$$

An upper boundary for speedup using the distribution function $D_1$ can be made:

$$S_1 \le \frac{N}{1 + \left(\frac{lyN}{2p}\right)^2}, (D_1) \qquad \text{(Eq.5.24)}$$

for $p > \frac{lyN}{2}$ and $lyN \gg 1$ .

- **Case C: $P_{max} > lyN$ and distribution $D_2$**

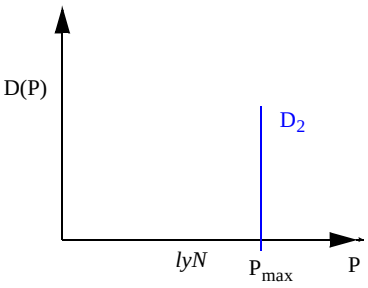



Fig. 5.45. $D_2$ and $P_{max} \geq lyN$

Using $\sum D(P) = 0$ results in:

$$\sum P \cdot D(P) = 0$$

With (Eq.5.19) an upper boundary for speed-up using the distribution function $D_2$ can be made:

$$S_2 \leq N \quad , (D_2) \qquad \text{(Eq.5.25)}$$

for $p > lyN$.

**Summary**

**For distribution $D_1$:**

$$\frac{N}{1+\frac{lyN}{p}} < S_1 \leq \begin{cases} \frac{p}{ly} & for\ p \leq \frac{lyN+1}{2} \\ \frac{N}{1+\left(\frac{lyN}{2p}\right)^2} & for\ p > \frac{lyN+1}{2} \end{cases}$$

**For arbitrary distributions:**

$$\frac{N}{1+\frac{lyN}{p}} < S_2 \leq \begin{cases} \frac{p}{ly} & p \leq lyN \\ N & p > lyN \end{cases}$$

The point $p = lyN$ in Fig. 5.46. characterizes the behavior of the workload (described by $p$) and the architecture (described by $lyN$). This point is called speedup-knee, for $p > lyN$ an efficiency $\varepsilon > 50\%$ is realized.

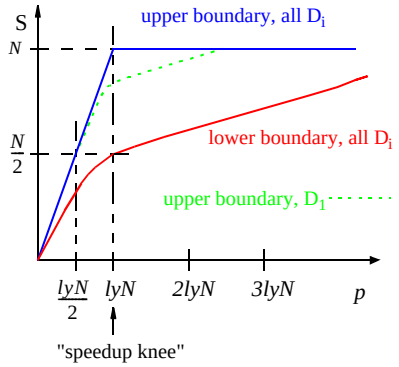
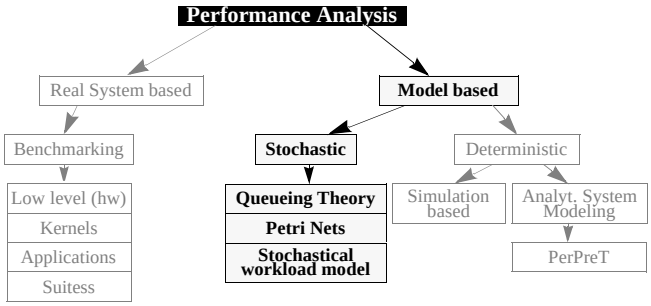



Fig. 5.46. Speedup-knee




## 5.4. Summary Performance Modeling

The development of a performance model for a computing system is a challenging task. As outlined in Fig. 5.47., the development of a computing system takes several steps from the high-level design of the system and its components down to the hardware development. Each level in this top down approach should be accompanied by appropriate performance evaluation techniques in order to optimize the system in terms of performance. Performance might have different meanings in this process, it can be speed, availability, fault tolerance and others. The performance measures are defined by the system designers.

The top down development process yields an increasing complexity from level to level. This is also valid for the performance evaluation techniques, they must be capable to deal with the increasing complexity. The problem is the lack of a common environment for both, the development and the modeling side in Fig. 5.47. There are tools for each step of the development and modeling, but there is no single common environment from top to bottom (vertical solid arrows in Fig. 5.47.). There is also a lack of interfaces between development and modeling tools (horizontal solid arrows in Fig. 5.47.). One way to overcome these lacks is a closer cooperation between developers and modelers.

The approach to use deterministic and stochastic Petri Nets to model performance as described in [Lin98] tries to establish a common environment to the first three steps of performance modeling in Fig. 5.47. In the author's opinion the Petri Net approach has two main advantages compared with other techniques:

- The computational effort to solve the resulting systems of equations could be reduced by using numerical methods.
- The graphical representation of the system as a Petri Net may provide additional insight to computer system designers.

The problems with this approach (as with others) are that there is no interface from the

modeling world to the developer world and that there is no interface to the deterministic evaluation which succeeds the stochastic modeling.

The dotted arrows in Fig. 5.47. indicate that performance results at any stage can be validated by using performance results of the next design stage.

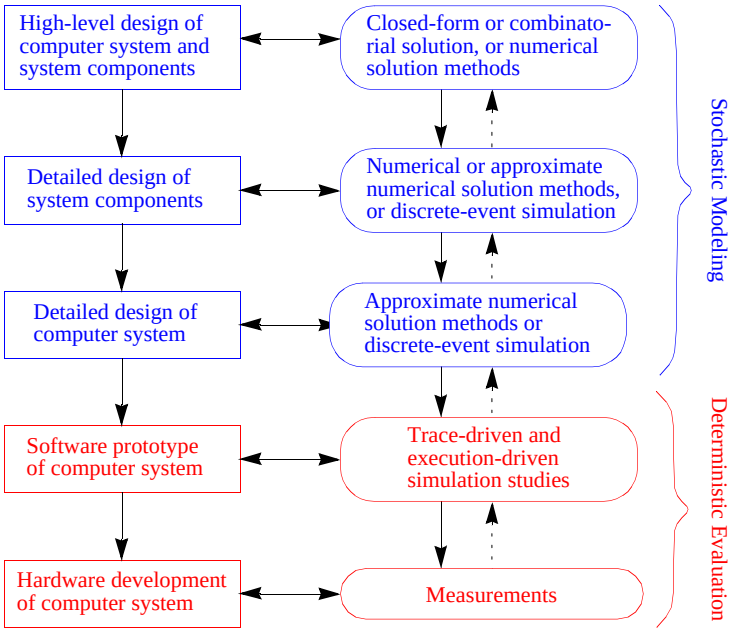



Fig. 5.47. Support of system design by modeling (from [Lin98])

In Fig. 3.17. an approach to build a validated system model is described. This technique is included in a modeling cycle described in [Laz84]. In many cases, the application of queueing network modeling techniques involves projecting the effect on performance changes to the configuration or workload of an existing system. Fig. 5.48. outlines the three phases that are revisited for such a study. Beginning with the measurements from the existing system a baseline model is defined and constructed in the first phase, the so called *validation* phase. The system workload measures are used to provide model input data and the model output is compared with the system performance measures. If discrepancies are detected in this phase, the model has to be redefined.

In the next phase, the so called *projection* phase, the derived model is used to forecast the effect on performance of the anticipated modifications. The model is then evaluated. The difference between the modified model outputs and the original model outputs is the projected effect of the modification.

In the last phase, the so called *verification* phase, the results from the second phase are

compared with measurements taken on the modified system. The last phase can only be carried out if the modified system is realized. In this phase the performance measures are compared to the model outputs and the workload measures are compared to the model inputs.

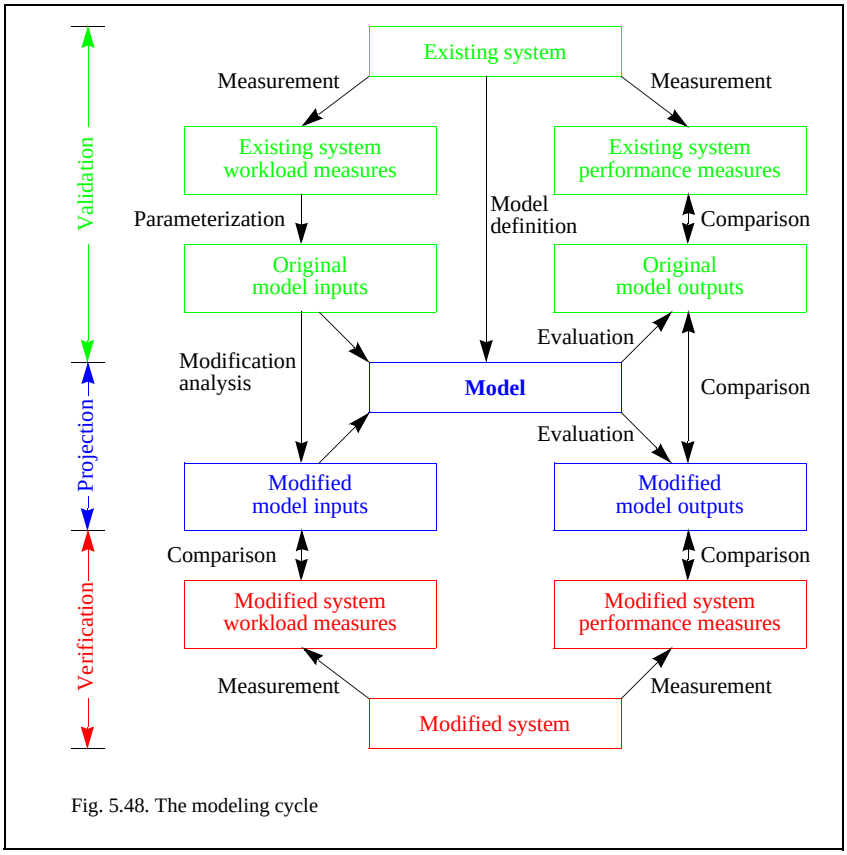
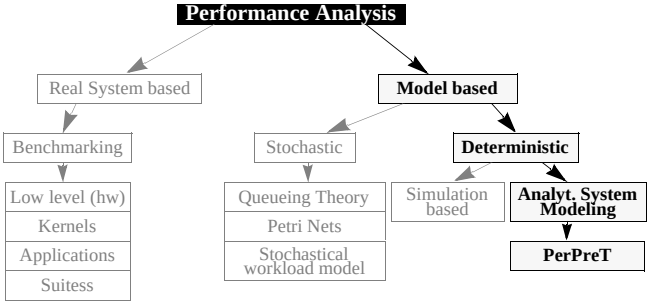



Fig. 5.48. The modeling cycle

Looking at both figures, Fig. 5.47. and Fig. 5.48., it becomes clear that the workload does not play an important role in conventional stochastic modeling techniques. In Fig. 5.47. it is not even mentioned, in Fig. 5.48. existing workload measures are used. The workload is abstracted on a very high level using simply parameters and probability distributions to describe its behavior. This is not surprising since the main goal of the performance modeling is to enhance and improve the capabilities of the system or system components, and not to change the workload itself. Of course, modifying and improving the system also has side effects on the workload, but the workload itself is not modified. In parallel systems, the problem is more complex, since the mapping of the workload onto the system is not as straight forward as in monoprocessor systems. In the next section, a modeling tool for performance prediction is presented that involves detailed workload and system models to predict performance measures.




# 6. Analytical Performance Modeling for Massively Parallel Systems

Problems in modeling complex architectures and applications using queueing based approaches were the start for the development of PerPreT (Performance Prediction Tool) [Bre95]. Before implementing a parallel application the programmer has to decide for a parallelization strategy for the problem. In general it is too time consuming to implement several parallelization strategies, measure their timing and use the best one. PerPreT provides the user with information for the expected overhead without the need to explicitly implement and monitor these strategies. PerPreT is specially designed for multiprocessors with large numbers of processing elements which are also called massively parallel systems.

In the last years, several articles with this goal were published [Tho86], [Wab94a], [Laz84], [Har94], and [Har95]. They are based on approaches using Petri Nets or Queueing Network Models. These approaches result in very accurate models of target systems and applications. Nevertheless there are problems to apply these approaches to massively parallel systems:

- The graphical representation of systems with hundreds or even thousands of processors that would be needed for these approaches is too complex.
- The application description and the mapping of the logical taskgraphs onto the target systems is too detailed.
- The systems of equations produced by the Petri Nets or Queueing Network Models for massively parallel systems are too complex to be solved in realistic time.

Typical applications for massively parallel systems use the single program multiple data (SPMD) Programming model. This section will show that modeling SPMD programs allows simplifications which do not or only slightly reduce the accuracy of performance predictions. The abstractions used to describe parallel applications do not necessarily mean a loss of accuracy for the prediction of speed-up, communication behavior, and execution time. But these abstractions can lead to models capable of handling complex applications for systems with large numbers of processors. The system and application description result in an analytical model for message passing multiprocessors. From this model performance numbers can easily be calculated. This

is the main advantage of PerPreT as compared to existing approaches based on Petri Nets or Queueing Network Models.

On the other hand, there are situations that cannot be modeled by the PerPreT approach because of its high level of abstraction. On the hardware side these are cache anomalies and network contention, on the software side these are applications with input dependent complexity like Traveling Salesman Problems. The Petri Net or Queueing Network Models based approaches are for these examples superior to PerPreT. Another restriction of PerPreT is its focus on the SPMD programming model.

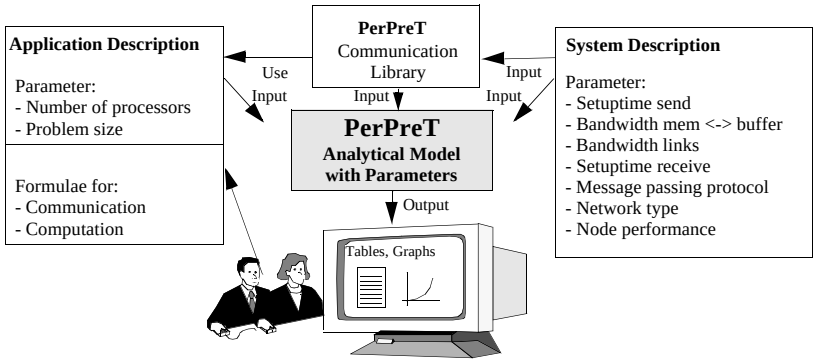



Fig. 6.1. The PerPreT modules

Fig. 6.1. summarizes the modular structure of PerPreT into application description, system description, and a communication library. These modules provide the modeling tool with parameterized analytical models which allow the calculation of expected speedup, time spent for communication and time spent for computation. If these numbers are available it is easy to calculate predictions for efficiency and total execution time.

The parameters used for the application description allow a scalable modeling. A PerPreT experiment may consist of predictions for several problem sizes or different data distributions (for different numbers of processors). The system description is independent from the application description. This is why one application description can be used for experiments with different systems and vice versa (for different application descriptions the system description has to be done once). The integer or floating point node performance for an application is the only parameter which affects both, application and system description.

The typical result graphs of PerPreT experiments are shown in Fig. 6.2. and Fig. 6.3. The left figure shows predicted times for communication and computation for a fixed problem size and different numbers of processors. The sum of communication and computation time defines the total execution time curve. This plot is also available as table showing predicted speedup and efficiency for the experiments. The right figure shows predicted times for a fixed number of processors and different

problem sizes. Again, this plot is available as table.

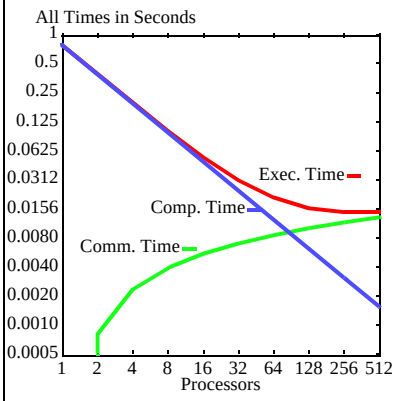



Fig. 6.2. All times plot of CG-Tree with psize 1024 on INTEL Paragon.

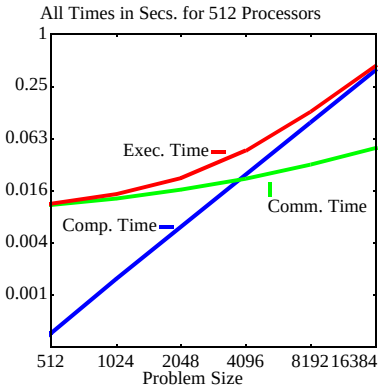



Fig. 6.3. All times plot of CG-Tree with nprocs 512 on INTEL Paragon.

## 6.1. System Description

The goal of the PerPreT system description is to describe the underlying hardware by its most important architectural parameters. For the performance of a multiprocessor system, the most important components are the communication units, the interconnection network structure and the node processors. This results in the parameters to describe the computation and communication behavior of the target system.

### 6.1.1. Communication Model

The time for a node to node communication of a message passing multiprocessor can be divided into five phases T1,...,T5:

T1:

Setup time for the send subroutine. This time is caused by the communication of the user process with the communication unit in order to initialize the message buffers and to transfer the control of the message exchange to the communication unit.

T2:

Time to copy the message to the message buffer. This time is only needed for systems with asynchronous communication using concurrently working communication units.

T3:

Message transfer time which is needed to copy the message from sender buffer to receiver buffer.

T4:

Setup time for the receive subroutine. This time is caused by the communication of the user process and the communication unit. The user process needs information where the message is stored.

T5:

Time to copy the message from message buffer to the user memory space. This time is obsolete for systems which provide the communication units with DMA (direct memory access) capabilities.

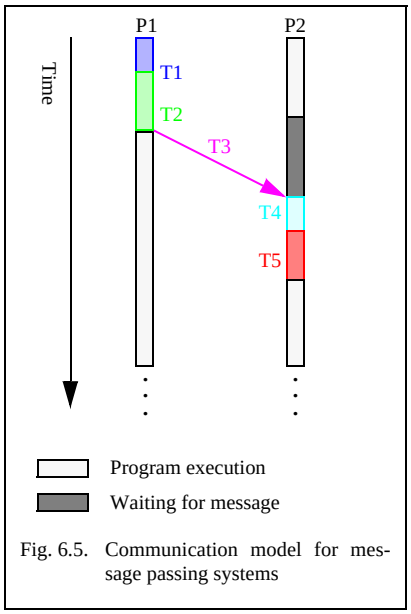



Fig. 6.5. Communication model for message passing systems

As already mentioned, depending on the system and message passing protocol, not all of the five phases have to be carried out. Trans-

puter for example use synchronous communication. In that case, the messages are directly copied from one sender user space to receiver user space. This saves phase T2 and phase T5. The five phase model realized for PerPreT offers enough flexibility to model most existing message passing technologies and protocols. The total time for one communication between two processors is the sum of the times T1,..., T5. Three of the times always are a function of message length (T2, T3, T5) the other two can be independent on the message length depending on the hardware.

If a full system specification is available, the times for T1 to T5 can be directly taken and used for the PerPreT communication library. Sometimes the hardware vendors do not provide the user with, or the information on the performance of the communication system differs from theory to real implementations, or additional overhead is created by the user through additional software layers for the communication (for example through using the communication library PICL instead of the native communication primitives). For all these cases, some test programs and measurement have to be carried out to find out formulae for the times T1 to T5.

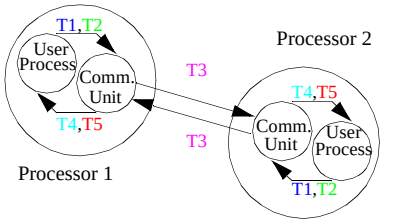



Fig. 6.6. Communication phases using message passing

### 6.1.2. Computation Model

The system model for the computational behavior of the nodes is based on their expected performance for the application. PerPreT uses the number of operations per time unit as measure (MIPS, Millions of instructions per second in case of Integer based applications, or MFLOPS, Millions of floating point operations per second in case of numerical applications). Many systems show a wide variation of the node performance for different applications. To find the correct performance measure for the PerPreT computation model a user has several options:

- Run the monoprocessor version of the program and calculate the performance for one node.
- Look whether performance measures exist for similar applications (vector codes, matrix codes, search codes,...) and use these values.
- A small kernel to simulate the profile of the application is implemented and the performance is measured.
- Use published benchmark results for comparable codes.

It is clear that the accuracy of the approach depends on the correct value for the node performance estimation. For complex codes it might be necessary to divide them into phases with different performance characteristics. In

that case a phase model which is presented in section 6.3.3. can be used.

### 6.1.3. System Examples

A description of the two multiprocessors (nCUBE/2 and INTEL Paragon) that are used to conduct the PerPreT experiments is given in section 4.1.2.1 and section 4.1.2.2 of this text.

**PerPreT System Description of nCUBE/2:**

The PerPreT system description for the nCUBE/2 consists of parameters for the computation and communication behavior of the system. The value for the node performance is the only parameter that connects system model and application model. All other parameters and formulae for system and application description are independent.The computation parameter for the application which were modeled for the nCUBE/2 (compare section 6.3.) was its performance of 0.43 MFLOPS for these applications. This number was taken for all applications from section 6.3. The parallel applications were programmed using the portable communication library PICL [PICL90]. For the communication primitives "send" and "receive" of PICL the following parameters were determined:

- Setup time send: 100 µsec
- Setup time receive: 73 µsec
- Link bandwidth: 1.67 MByte/s
- Bandwidth for copying message from user space to communication buffer: 5.35 MByte/s

**PerPreT System Description of INTEL Paragon:**

The PerPreT system description for INTEL Paragon consists of parameters for the computation and communication behavior of the system. The computation parameter for the application which were modeled for the INTEL Paragon needs some work to be determined. In contrast to nCUBE/2 the Paragon nodes did not show a nearly constant performance for the applications described in section 6.3. The performance even differed for different computing phases of one application. The variation is between 3 and 18 MFLOPS per CPU. To find out the correct performance number, the application was run and computing phases including their performance numbers were determined. These numbers were taken for the modeling experiments in the later section. The parallel applications were programmed using the portable communication library PICL [PICL90]. For the communication primitives "send" and "receive" of PICL the following parameters were determined:

- Setup time send: 45 µsec
- Setup time receive: 95 µsec
- Link bandwidth: 40 MByte/s
- Bandwidth for copying message from user space to communication buffer: 100 MByte/s

## 6.2. Application Description

The applications which can be modeled by PerPreT are parallelized using a SPMD programming model. The SPMD programming model and the resulting workload model are used to built a parameterized analytical model which represents the application.

### 6.2.1. Programming Model

Most massively parallel systems with up to thousands of processors are realized as message passing multiprocessors. Each processor has fast access to its own local memory. If data from memories of other computing nodes are needed a data transfer via message passing is initiated. Programs for these systems are mostly implemented using the SPMD programming model [Ser93].

Each processor executes the same program on different data. Data dependencies make communication and synchronization of the tasks necessary. The synchronization and communication of the tasks is realized on user level. In case of a massively parallel system the program has to be well structured and the communication topology is mostly regular to avoid load imbalances and communication deadlocks.This leads to a Program Task Graph (PTG) as shown in Fig. 6.7. SPMD programs are characterized by a sequence of computation and communication phases. The circles in Fig. 6.7. represent the computational tasks, the arrows represent communication.

An upper boundary for the time for a computational phase is $T_{CPi}$ time units ($i=1,2,..,7$) and an upper boundary for the time of a communication or synchronization phase is $T_{CMj}$ time units ($j=1,2,..,6$).

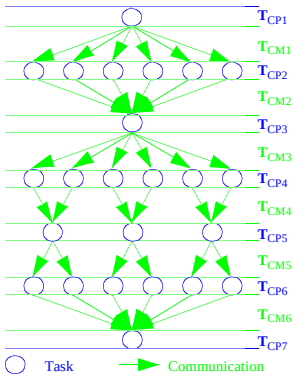



Fig. 6.7. SPMD Program Task Graph

### 6.2.2. Workload Model

Fig. 6.8. shows a mapping of the PTG of the SPMD program from Fig. 6.7. onto 6 processors. Using this mapping (and assuming there are enough processors for the tasks of the

computation phases), an upper boundary for the communication time of the application can be given:

$$\sum_j T_{CMj} \qquad \text{(Eq.6.1)}$$

Also, an upper boundary for the computation time of the application can be given:

$$\sum_i T_{CPi} \qquad \text{(Eq.6.2)}$$

Adding the times of (Eq.6.1) and (Eq.6.2) results in an upper boundary for the total execution time of the program:

$$\sum_i T_{CPi} + \sum_j T_{CMj} \qquad \text{(Eq.6.3)}$$

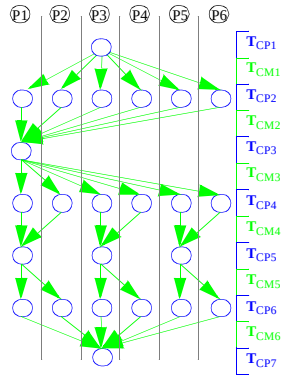



Fig. 6.8. Mapping of an SPMD program onto 6 processors

Experiments with measurements and validation using real applications running on real machines show that the calculated total execution time often is very close to the upper boundary. For more general PTGs the number of subtasks per phase is not necessarily constant. Also, the assumption that there are always enough processors to distribute one subtask of a phase on an individual processor is not always given. These problems result into different mappings with different timings ($T$ from Fig. 6.8. becomes $T^*$ in Fig. 6.9.), but they do not change the formulae for the boundaries of communication time and computation time. If the PTG of Fig. 6.7. is for example mapped onto 5 processors it will look like:

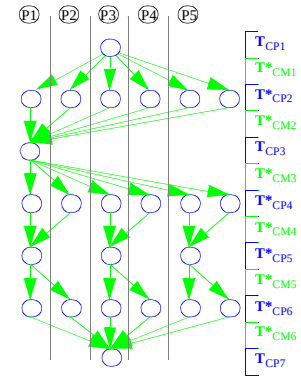



Fig. 6.9. Mapping of an SPMD program onto 5 processors

The communication phases of an SPMD application can be divided into local communications (only two or a subset of the processors are communicating during a phase) and global communications (all processors are communicating during a phase). Typical global communications are for example:

- Broadcast: one processor sends data to all other processors (compare phases CM1 and CM3 in Fig. 6.7.).
- Collect: all processors send data to one processor (compare phases CM2 and CM6 in Fig. 6.7.).
- Butterfly: all processors exchange data using a butterfly network configuration. Any other regular network configuration type is possible.

## 6.3. Application Examples

In order to validate the accuracy and usefulness of PerPreT, a set of case studies has been evaluated on an nCUBE/2 and on an INTEL Paragon running several parallel kernels from a LOOP benchmark suite [LOOP94]. All codes have been implemented using the PICL [PICL90] communication library. Similar kernels are also used in the Parkbench benchmark suite [Par94]. The execution times and speedups of the LOOP programs are compared with the predicted values using PerPreT. All experiments carried out on the nCUBE/2 are reported for 1 - 128 processors, since an 128 node nCUBE/2 is used for validation purposes. The experiments on the INTEL Paragon were carried out for various processor numbers. All curves in the presented figures with prefix *"PerPreT"* refer to PerPreT prediction results, while all curves with prefix "*<hostname>*" refer to execution times measured on the nCUBE/2 or on the INTEL Paragon. PerPreT provides predictions for execution time, communication time, and computation time in the ±10% accuracy range for most cases. For a few extreme cases (i.e., small problem size and large numbers of processors) the accuracy range is ±20%. Slowdowns, as shown in Fig. 6.22. and in Fig. 6.23., are also predicted correctly.

### 6.3.1. Matrix Multiplication

#### 6.3.1.1 PerPreT Application Description of MM

The multiplication of two matrices *A*B=C* is an often used workload for benchmarking systems. It is easy to program but still demanding with respect to processor performance and memory access of the target system. Using the parameter *psize* for the problem size results in the following program for the matrices *A, B,* and *C* of size (*psize* x *psize*):

```
for (i=0; i<psize; i++)
  for (j=0;j<psize; j++)
    for (k=0; k<psize; k++)
      C[i][j] += A[i][k] * B[k][j];
```

The number of arithmetic statements $st_s$ which have to be executed is:

$$st_s = 2 \cdot psize^3$$

and thus the formula for the computation description of the sequential version of the matrix multiplication is found. The parallel version will equally distribute all arithmetic operations onto all processors executing the program (number of processors = *nprocs*). In this case the PerPreT formula for the parallel computation of the matrix multiplication $st_p$ is:

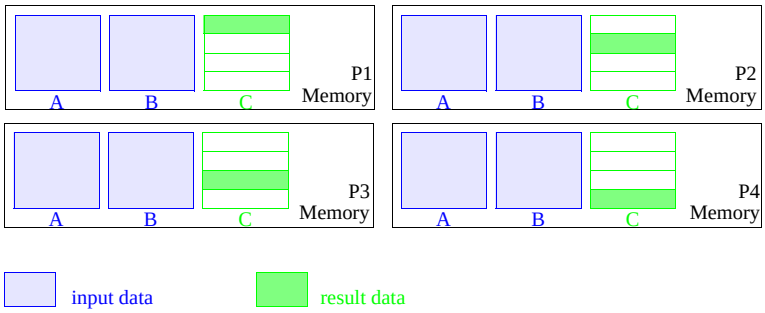
$$st_p = \frac{2 \cdot psize^3}{nprocs}$$

Besides the formula for computation a formula for the description of the communica-

tion is also part of the PerPreT application description. This formula depends on the parallelization strategy. For the matrix multiplication several strategies are possible. The main advantage of the parallel strategy for the matrix multiplication described in Fig. 6.10. is that during calculation of the result matrix *C* no communication or synchronization is necessary. This advantage is paid for by an enormous memory overhead since both input matrices *A* and *B* are copied to the memory of all processors in the beginning and thus have to be stored *nprocs* times (*nprocs* = number of processors). This is why mostly version 2 (compare Fig. 6.11. and Fig. 6.12.) of the parallel matrix multiplication is used.

**Version 1 of Parallel Matrix Multiplication:**

Copy matrices *A* and *B* to all nodes and compute (index driven) the corresponding fraction of result matrix *C* concurrently (Example for 4 processors):




The result matrix *C* will be distributed among the processors in row blocks (the same scheme is possible for column blocks)

Fig. 6.10. Version 1 of parallel matrix multiplication

Version 2 is executed in *nprocs* steps. In each step the row block of *A* is multiplied with the column block of *B*. This submatrix multiplication results in a small submatrix of result matrix *C*. After the computation of the submatrix a communication step is performed. The column blocks of matrix *C* are sent to the right processor neighbors using a ring configuration. After this communication, each processor has the next column block of *B* to be able to compute the next submatrix of *C*. After *nprocs* computing steps and *(nprocs-1)* communication steps the result matrix *C* is completed. The row blocks of *C* are equally distributed on all processors.

The main advantage of this version of the parallel matrix multiplication is that each matrix has to be stored only once and thus, the smallest memory usage is realized. Compared to the number of arithmetical operations that have to be performed, the communication effort for this version is acceptable. For some systems with DMA capability of the communication units the communication of this version can be overlapped with the computation.

The PerPreT formula for the parallel computation of version 2 of the matrix multiplication is the same as for version 1:

$$st = \frac{2 \cdot psize^3}{nprocs}$$

The PerPreT formula for the communication of version 2 of the parallel matrix multiplication is:

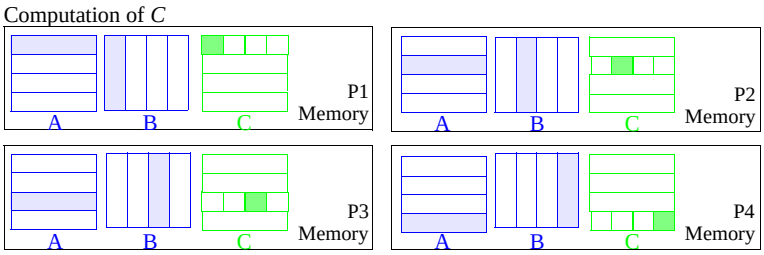
$$bytes = \frac{typesize \cdot psize^2}{npocs}$$

The parameter *bytes* is the number of bytes to be transmitted per message, the parameter *typesize* is the size of the matrix data type in number of bytes, the function *comm* calculates the time for a message transfer of length *bytes*. This function is part of the PerPreT communication library. The result of the communication formula *ct* is the estimated communication time for version 2 of the parallel matrix multiplication:

$$ct = (nprocs - 1) \cdot comm(bytes)$$

**Version 2, Step 1a of Parallel Matrix Multiplication (Computation):**

Distribute matrix *A* in row blocks, distribute matrix *B* in column blocks (each processor gets one column block and one row block) and compute the corresponding fraction of *C* concurrently (Example for 4 processors):




**Version 2, Step 1b of Parallel Matrix Multiplication (Communication):**

Send the column block of input matrix *B* to your right neighbor (assuming a ring configuration):

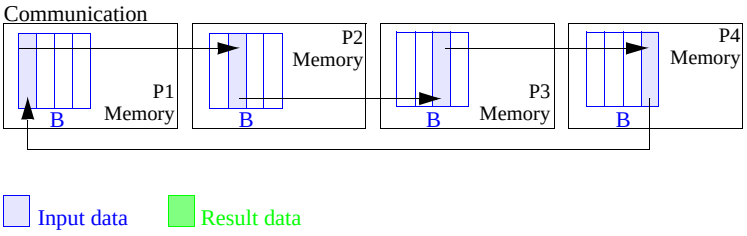
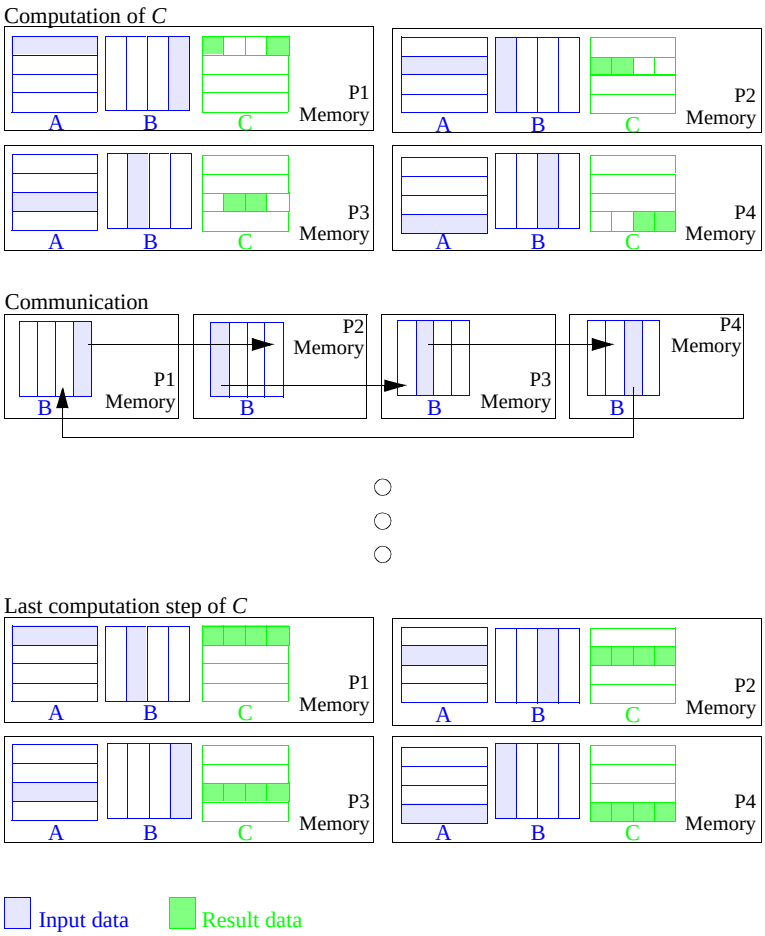



Fig. 6.11. Version 2 of parallel matrix multiplication

**Version 2, Steps 3 to nprocs of Parallel Matrix Multiplication:**

Repeat steps 1a and 1b *(nprocs-1)* times. For the last step, no communication is needed after computation:




The result matrix *C* will be distributed among the processors in row blocks (the same scheme is possible for column blocks)

Fig. 6.12. Version 2 of parallel matrix multiplication (steps 2 to *nprocs*-1)

#### 6.3.1.2 Variation of Problem Size

When the PerPreT application description and the PerPreT system description are defined, experiments with varying parameters such as problem size (*psize*) and number of processors (*nprocs*) can be executed. Experiments with varying problem size help to find the minimal problem size to be used for a given number of processors. In Tab. 6.1. the results for modeling version 2 of the parallel matrix multiplication are summarized. The system model is the INTEL Paragon as described in section 6.1.3. In this experiment 64 processors were modeled to execute version 2 of parallel MM with problem sizes varying from 64 to 512. Problem size 64 means that all three matrices *A*, *B*, and *C* are of size (64x64).

Computation for Parallel Matrix Multiplication
Version 2
Processors: 64 Problem Size: 64 ... 512
System: Paragon
Speedup estimate (all times in seconds):

| PSIZE | COMM | COMP | TOTAL | SP | EFF |
|---|---|---|---|---|---|
| 64 | 0.010272 | 0.002074 | 0.012345 | 10.75 | 0.168 |
| 128 | 0.014626 | 0.015945 | 0.030572 | 33.38 | 0.522 |
| 256 | 0.032044 | 0.222156 | 0.254200 | 55.93 | 0.874 |
| 512 | 0.101717 | 1.777247 | 1.878965 | 60.54 | 0.946 |

Tab. 6.1. PerPreT result table for parallel MM on INTEL Paragon (varying psize)

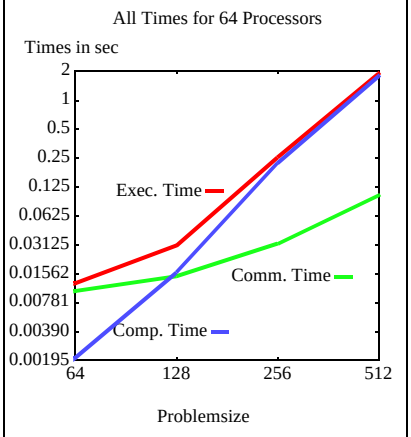



Fig. 6.13. Predicted time for Version 2 of parallel MM on INTEL Paragon with varying psize

For the smallest problem size of 64, Tab. 6.1. shows poor speedup (column *SP*) and thus poor efficiency (column *EFF*). Speedup and efficiency are defined in section 2.2.2. Looking at the time spent for communication (column *COMM*) and computation (column *COMP*) the bad efficiency can easily be explained. The parallel MM spends more time in communication than computation for problem size 64. The next problem size 128 shows significantly better performance. The speedup (33.38) is about half the number of processors (64) and thus, the efficiency is approximately 50%. The larger the problem size grows the better the multiprocessor performs for this program. In summary, the experiment tells the user not to use too many processors if dealing with small problem sizes.

Or, in case of the version 2 of parallel MM, the problem size should at least be twice the number of processors, if an efficiency better than 50% is desired. For easier interpretation of the result tables, PerPreT offers a graphical output feature. Fig. 6.13. shows the execution, communication, and computation time curves of Tab. 6.1.

#### 6.3.1.3 Variation of Number of Processors

Running experiments with varying numbers of processors will help the user to find out how many processors should be used to reach an optimum performance for a given problem size. For many parallel programs the effect of a slowdown is known which can occur if a too many processors are used to execute a program with a small problem size. It will be discussed in the next section (modeling of the Conjugate Gradient Method). The results in Tab. 6.2. clearly show that the parallel MM program speeds up to a number of 256 processors for problem size 256.

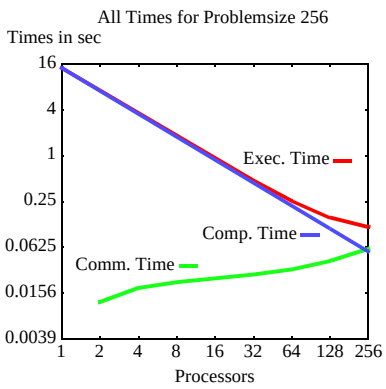



Fig. 6.14. Predicted time for Version 2 of parallel MM on INTEL Paragon

The curves for execution, communication and computation time in Fig. 6.14. show that the communication time slowly increases with the number of processors and that it crosses the curve for computation time at approximately 256 processors. When the curve of the communication time is above the curve of the computation time the resulting efficiency is less than 50%. Using modeling techniques such as PerPreT allows efficient use of multiprocessors undergoing the time consuming task of implementing and measur-

ing different problem sizes and numbers of processors.

Computation for Parallel Matrix Multiplication
Version 2
Processors: 1...256 Problem Size: 256
System: Paragon
Speedup estimate (all times in seconds):

| P | COMM | COMP | TOTAL | SP | EFF |
|---|---|---|---|---|---|
| 1 | 0.000000 | 14.217980 | 14.217980 | 1.00 | 1.000 |
| 2 | 0.011936 | 7.108990 | 7.120926 | 2.00 | 0.998 |
| 4 | 0.018115 | 3.554495 | 3.572610 | 3.98 | 0.995 |
| 8 | 0.021624 | 1.777247 | 1.798871 | 7.90 | 0.988 |
| 16 | 0.024218 | 0.888624 | 0.912842 | 15.58 | 0.973 |
| 32 | 0.027196 | 0.444312 | 0.471508 | 30.15 | 0.942 |
| 64 | 0.032044 | 0.222156 | 0.254200 | 55.93 | 0.874 |
| 128 | 0.041189 | 0.111078 | 0.152267 | 93.38 | 0.729 |
| 256 | 0.059201 | 0.055539 | 0.114740 | 123.91 | 0.484 |

Tab. 6.2. PerPreT result table for parallel MM on INTEL Paragon (varying nprocs)

#### 6.3.1.4 Validation of the PerPreT Formulae

The PerPreT system description of the nCUBE/2 and the INTEL Paragon and the application description of the parallel matrix multiplication derived in the previous sections are relatively simple. The question arises whether the results using these models are accurate enough.

**Parallel MM, 1 to 128 processors:**

| P | mod | exp | Dsec | D% |
|---|---|---|---|---|
| 1 | 78.033 | 77.963 | 0.069 | 0.09 |
| 2 | 39.273 | 39.506 | -0.232 | -0.59 |
| 4 | 19.894 | 19.895 | -0.001 | -0.01 |
| 8 | 10.204 | 10.125 | 0.079 | 0.79 |
| 16 | 5.361 | 5.216 | 0.144 | 2.77 |
| 32 | 2.941 | 2.774 | 0.166 | 6.01 |
| 64 | 1.735 | 1.593 | 0.141 | 8.90 |
| 128 | 1.141 | 1.160 | -0.019 | -1.67 |

Tab. 6.3. Validation of parallel MM using 1 to 128 processors on nCUBE/2

To validate the approach the results from modeling are compared with the execution times of the real applications on the real systems. Tab. 6.3. and Fig. 6.15. show the results

for the nCUBE/2 system. The accuracy of the prediction is always better than ± 10%.

**Parallel MM, 1 to 64 processors:**

| P | mod | exp | Dsec | D% |
|---|---|---|---|---|
| 1 | 14.217 | 14.182 | 0.035 | 0.25 |
| 2 | 7.120 | 7.162 | -0.041 | -0.58 |
| 4 | 3.572 | 3.594 | -0.021 | -0.61 |
| 8 | 1.798 | 1.811 | -0.013 | -0.72 |
| 16 | 0.912 | 0.901 | 0.011 | 1.23 |
| 32 | 0.471 | 0.439 | 0.031 | 7.17 |
| 64 | 0.254 | 0.251 | 0.002 | 1.01 |

Tab. 6.4. Validation of parallel MM using 1 to 64 processors on INTEL Paragon

The results for the validation on the INTEL Paragon are in the same range. For this algorithm the INTEL Paragon was approximately five times faster (unoptimized code) for the single processor version.

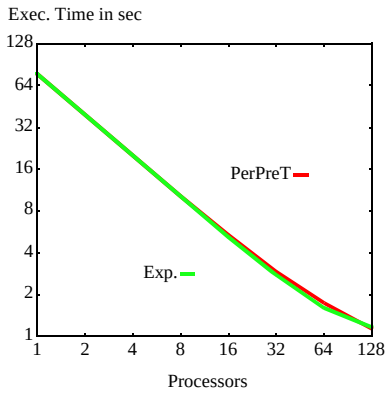



Fig. 6.15. Comparison of actual and predicted execution times of parallel MM for psize = 256 on nCUBE/2

The predicted execution times show an accuracy better than ± 10% (compare Tab. 6.4. and Fig. 6.15.). More experiments with different problem sizes and numbers of processors (compare Appendix-C and [Bre98] ) proved that the PerPreT application and system description allows to accurately predict execution and communication time of the parallel matrix multiplication.

If the difference between predicted time and real execution time is worse than 10% the following criteria should be checked:

- Is the problem size chosen large enough (very small execution times of less than 1 msec do not make sense)?
- Is there a memory hierarchy problem (cache influence)?
- Are the application and system models fine enough, or do they have to be refined?

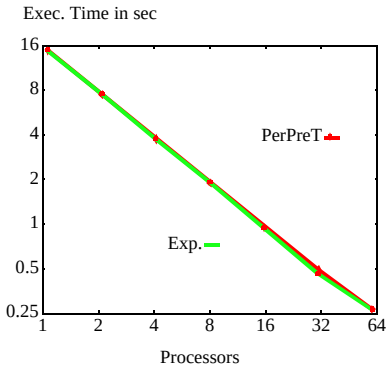



Fig. 6.16. Comparison of actual and predicted execution times of parallel MM for psize = 256 on INTEL Paragon

### 6.3.2. Conjugate Gradient Method

The parallel matrix multiplication from the previous section is a relatively simple algorithm. To show that the PerPreT approach can also be used for complex codes, further experiments for the Conjugate Gradient and for the Shallow Water Code have been carried out. In recent years, the conjugate gradient method for the solution of equation systems

$A*x=b$ ($A$ is matrix, $x$ and $b$ are vectors)

has become popular again. These methods are also better suited for SPMD programs than solvers based on Gaussian Elimination.

#### 6.3.2.1 PerPreT Application Description of CG

The basic algorithm consists of a matrix vector product and several scalar products. The PTG of Fig. 6.17. shows an SPMD version of a Conjugate Gradient (CG) Method. *N* represents the problem size and *P* represents the number of allocated processors. The circles contain the number of floating point operations performed by the specific subtask. The values at the arcs represent the number of data items that have to be transmitted during a communication. If a circle is empty, then no floating point operations have to be performed. The computation phases *(CPi)* and the communication phases *(CMi)* of the CG-SPMD program are:

*CP1:* Distributed computation of a scalar-vector product. *2N/P* statements are executed per processor.

*CM1:* Global collect of a distributed vector. Each processor sends *N/P* data items.

*CP2*: No computation is involved.

*CM2:* Global broadcast of the collected vector. One processor sends *N* data items to each processor.

*CP3:* Distributed computation of a matrix vector product ($2N^2/P$ statements) and a scalar product (*2N/P* statements).

*CM3:* Global sum. Each processor sends one data item.

*CP4:* Global sum built by one processor.

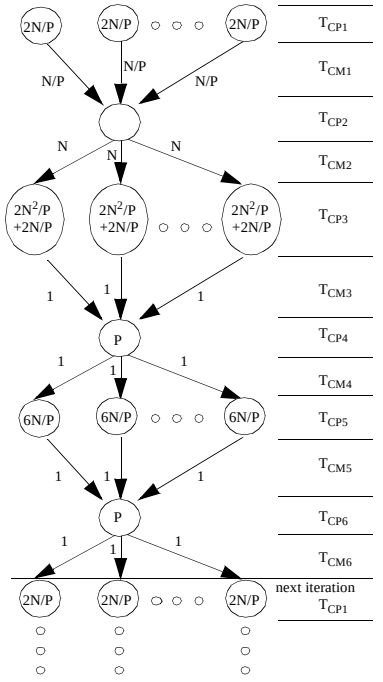



Fig. 6.17. PTG for parallel CG Method

*CM4:* Global sum. The processor that performed the sum in phase *CP4* sends the sum to every other processor.

*CP5:* Computation of two scalar-vector products (*2N/P* statements) and one scalar product (*2N/P* statements) per processor resulting in a total of *6N/P* statements.

*CM5, CP6, CP4:* same as *CM3*, *CP4*, and *CM4*, respectively.

For the CG-SPMD program, several global broadcast operations (CM2, CM4, CM6) and global collect operations (CM1, CM3, CM5) have to be performed with different message lengths. The PerPreT communication library contains routines that return the predicted communication times $T_{CMi}$ for i=1,...,6 (*simple_bcast* and *simple_collect*). The routines require the number of bytes to be transferred as an input parameter. They also have access to the global parameters *nprocs* (number of processors = P) and *psize* (problem size =N). *TYPE* is an indicator of the data type to be able to determine the number of bytes per data item.

**Data Distribution for Parallel Conjugate Gradient Method:**

Matrix A is distributed in row blocks, all vectors are distributed in blocks

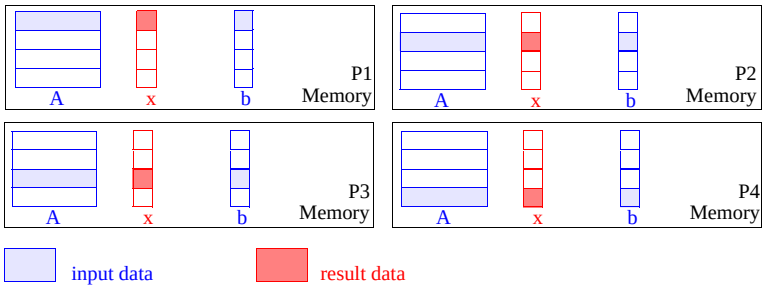



Fig. 6.18. Data distribution for parallel CG Method

**Version 1: PerPreT Communication Description of CG-Simple:**

Since the matrix data are distributed among all processors (compare Fig. 6.18.), the multiplying vector has to be copied and distributed. To build this vector, the routine *simple_collect* is used (phase *CM1* in Fig. 6.17.). To distribute the vector, the routine *simple_bcast* is used (phase *CM2* in Fig. 6.17.). Based on the above and the PTG shown in Fig. 6.17., the following communication description formula of the CG-SPMD program is used by PerPreT (the notation is derived from the C programming language):

```
bytes1 = sizeof(TYPE) * psize;
bytes2 = sizeof(TYPE) * psize/nprocs;
bytes3 = sizeof(TYPE);
comm_time += simple_bcast(bytes1);
comm_time += 2 * simple_bcast(bytes3);
comm_time += simple_collect(bytes2);
comm_time += 2 * simple_collect(bytes3);
```

**Version 2: PerPreT Communication Description of CG-Tree:**

The broadcast and collect operations are the most time consuming communications for the parallel CG-Methods. This is why besides the simple broadcast routines more efficient broadcast routines were implemented and modeled using PerPreT. The prefix *tree* of the *tree_collect* and *tree_bcast* routines indicates that a treelike topology is used to perform these communication operations. A more detailed description of the PerPreT communication routines can be found in appendix-C. The same routines are used to collect (*CM3* and *CM5* in Fig. 6.17.) and distribute (*CM4* and *CM6* in Fig. 6.17.) the global sum of the scalar products. The only difference between communication phases *CM3/CM5* and *CM4/CM6* is the amount of data to be transferred (parameters *bytes1*, *bytes2*, and *bytes3* in the formula given below). Thus, the communication description of CG-Tree is:

```
bytes1 = sizeof(TYPE) * psize;
bytes2 = sizeof(TYPE) * psize / nprocs;
bytes3 = sizeof(TYPE);
comm_time += tree_bcast(bytes1);
comm_time += 2 * tree_bcast(bytes3);
comm_time += tree_collect(bytes2);
comm_time += 2 * tree_collect(bytes3);
```

The two different versions of the conjugate gradient method (CG-Tree and CG-Simple) are used to find the number of processors where a treelike topology outperforms plain broadcast/collect routines (compare Fig. 6.22. and Fig. 6.23.).

**PerPreT Computation Description of the Parallel CG-Methods:**

One iteration of the examined CG-method involves two dotproducts, three scalar vector operations, and one matrix vector product. The calculation of one dotproduct requires *2*psize* floating point operations, the calculation of the scalar vector operations require *2*psize* floating point operations each, and the calculation of the matrix vector product requires $2*psize^2$ floating point operations. The number of floating point operations per processor for one iteration of the parallel CG Method is:

$$\frac{(10 \cdot psize + 2 \cdot psize^2)}{nprocs} + 2 \cdot nprocs$$

The measured and predicted results for problem size 512 are presented in Fig. 6.22. and in Fig. 6.23. The experiments included other problem sizes as well and consistently exhibit a good match between predicted and measured values.The predicted execution times show an accuracy better than ± 10%.

### 6.3.2.2 Variation of Number of Processors

The results of a typical PerPreT experiment are summarized in Tab. 6.5. The *psize* parameter was set to 1024, the number of processors varied from 1 to 512, the system description used was the INTEL Paragon, the application description used was the CG-Method with communication using treelike topologies.

Each row of the table is the result of one experiment. The result table includes five columns, the first column contains the number of processors, the second column the communication time (derived by the communication

description), the third column the computation time (derived by the computation description), the fourth column the estimated total execution time in seconds and the last column the expected speedup.

After defining both, the PerPreT application description and the PerPreT system description (compare section 6.1.), experiments to predict the runtime, speedup and efficiency of the parallel application can be carried out.

The PerPreT graphical representation of this experiment is shown in Fig. 6.2. The default scale for both axes is logarithmic. Execution and computation times decrease with increasing processor number. The communication time increases as the number of allocated processors increases. Between 64 and 128 processors the curves for communication time and computation time cross. After this point the execution time curve does not significantly decrease. In this case, the user may conclude that adding more processors will not significantly improve the execution time. This intersection point also indicates the 50% efficiency threshold. Beyond this point, more than half of the execution time is attributed to communication.

Computation for Conjugate Gradient CG
Communication for Tree Broadcast
Processors: 1...512 Problem Size: 1024
System: Paragon
Speedup estimate (all times in seconds):

| P | COMM | COMP | TOTAL | SP |
|---|---|---|---|---|
| 1 | 0.000000 | 0.771938 | 0.771938 | 1.00 |
| 2 | 0.000789 | 0.385969 | 0.386758 | 2.00 |
| 4 | 0.002285 | 0.192984 | 0.195269 | 3.95 |
| 8 | 0.003781 | 0.096492 | 0.100273 | 7.70 |
| 16 | 0.005277 | 0.048246 | 0.053523 | 14.42 |
| 32 | 0.006773 | 0.024123 | 0.030896 | 24.90 |
| 64 | 0.008268 | 0.012061 | 0.020330 | 37.97 |
| 128 | 0.009764 | 0.006030 | 0.015795 | 48.87 |
| 256 | 0.011260 | 0.003015 | 0.014276 | 54.07 |
| 512 | 0.012756 | 0.001507 | 0.014264 | 54.12 |

Tab. 6.5. PerPreT result table for modeling a parallel CG Method on Paragon

In Fig. 6.3. the same experiment is presented with problem size 4096 instead of 1024. In this example, the execution time continues to significantly improve for the entire processor allocation range. For 512 processors, the communication time and the computation time have reached similar values. This implies that for problem size 4096 the CG-Tree algorithm scales well up to 512 processors on the INTEL Paragon.

Since CG-Tree and CG-Simple only differ in calls to communication routines, the computation description for PerPreT is the same. For higher numbers of allocated processors, the CG-Simple workload shows significantly worse performance. This is due to the inefficient broadcast and collect operations. Using more than 32 processors, actually results in a slowdown for CG-Simple on both systems (see Fig. 6.22. and Fig. 6.23.)

.

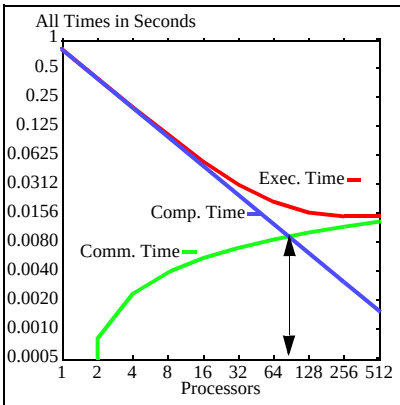



Fig. 6.19. All times plot of CG-Tree with psize 1024 on INTEL Paragon.

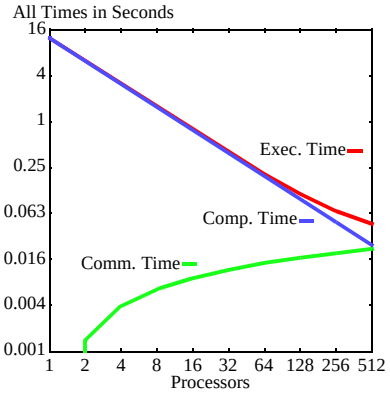



Fig. 6.20. All times plot of CG-Tree with psize 4096 on INTEL Paragon.

### 6.3.2.3 Variation of Problem Size

For another type of experiment, it is also possible to fix the number of processors and calculate the speedup over a range of problem sizes. The resulting table for 512 processors and problem sizes varying from 512 to 16384 is illustrated in Tab. 6.6., and graphically shown in Fig. 6.21. The first column in Tab. 6.6. contains the problem size (*PSZ*), the rest of the columns are self explanatory. The result of each experiment is summarized in two rows. The first row contains the execution time on one processor for the considered problem size. The second row contains the times and speedup for 512 processors.

As mentioned before, this type of experiment may help the user to decide on the minimum problem size for efficient use on a given (or constrained) number of allocated processors. For instance, Fig. 6.21. indicates that given 512 processors, a problem size of at least 4096 is needed to make efficient use of the system. For smaller problem sizes, a smaller processor partition would seem more appropriate.

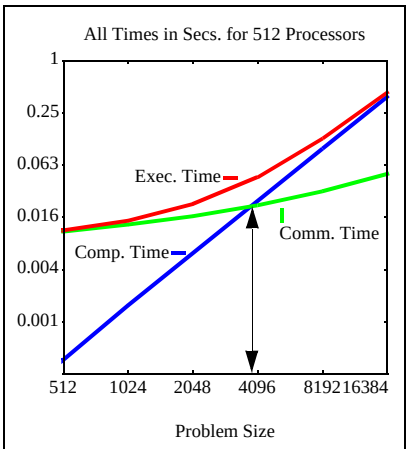



Fig. 6.21. All times plot of CG-Tree (512 processors, varying psize) on INTEL Paragon

Computation for Conjugate Gradient CG
Communication for Tree Broadcast
Processors: 512 Problem Size:512...16384
System: Paragon
Speedup estimate (all times in seconds):

| PSIZE | | |
|---|---|---|
| 512 | mono processor execution time.: | 0.180685 |
| 1024 | mono processor execution time.: | 0.771938 |
| 2048 | mono processor execution time.: | 3.080252 |
| 4096 | mono processor execution time.: | 12.306004 |
| 8192 | mono processor execution time.: | 49.194010 |
| 16384 | mono processor execution time.: | 196.716026 |

| PSIZE | COMM | COMP | TOTAL | SP |
|---|---|---|---|---|
| 512 | 0.01061 | 0.00035 | 0.01096 | 16.48 |
| 1024 | 0.01275 | 0.00150 | 0.01426 | 54.12 |
| 2048 | 0.01594 | 0.00601 | 0.02195 | 140.27 |
| 4096 | 0.02133 | 0.02403 | 0.04536 | 271.25 |
| 8192 | 0.03108 | 0.09608 | 0.12716 | 386.86 |
| 16384 | 0.04870 | 0.38421 | 0.43291 | 454.39 |

Tab. 6.6. Output Table for CG-Tree (varying psize, 512 processors) on Paragon

#### 6.3.2.4 Validation

The two parallel versions of the Conjugate Gradient Method were implemented on the INTEL Paragon and the nCUBE/2 using PICL. The execution times for varying numbers of processors were compared with the times predicted by PerPreT using the application model from the previous section and the system models described in section 6.1.3.

The results of the validation experiments for nCUBE/2 with problem size 1024 are summarized in Tab. 6.7. and Tab. 6.8., the results for this problem size are graphically displayed in Fig. 6.22. The accuracy of the prediction is always better than ± 10%.

**Parallel CG-Simple, 1 to 128 processors:**

| P | mod | exp | Dsec | D% |
|---|---|---|---|---|
| 1 | 4.900 | 4.874 | 0.026 | 0.55 |
| 2 | 2.463 | 2.442 | 0.020 | 0.85 |
| 4 | 1.243 | 1.232 | 0.010 | 0.89 |
| 8 | 0.642 | 0.634 | 0.008 | 1.39 |
| 16 | 0.360 | 0.351 | 0.008 | 2.51 |
| 32 | 0.252 | 0.242 | 0.010 | 4.16 |
| 64 | 0.263 | 0.252 | 0.011 | 4.41 |
| 128 | 0.396 | 0.383 | 0.012 | 3.37 |

Tab. 6.7. Validation of parallel CG-Simple on nCUBE/2 (psize = 1024)

**Parallel CG-Tree, 1 to 128 processors:**

| P | mod | exp | Dsec | D% |
|---|---|---|---|---|
| 1 | 4.900 | 4.874 | 0.026 | 0.55 |
| 2 | 2.464 | 2.442 | 0.021 | 0.87 |
| 4 | 1.245 | 1.232 | 0.013 | 1.06 |
| 8 | 0.640 | 0.640 | -0.000 | -0.01 |
| 16 | 0.341 | 0.344 | -0.003 | -1.09 |
| 32 | 0.195 | 0.201 | -0.005 | -2.95 |
| 64 | 0.125 | 0.132 | -0.007 | -5.34 |
| 128 | 0.094 | 0.102 | -0.007 | -7.60 |

Tab. 6.8. Validation of parallel CG-Tree on nCUBE/2 (psize = 1024)

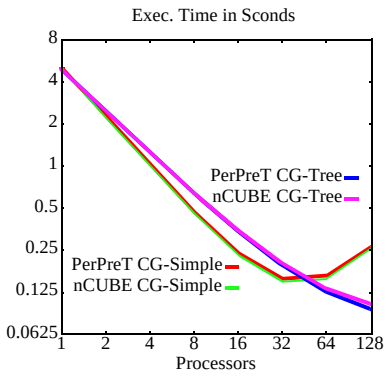



Fig. 6.22. Comparison of actual and predicted execution times of CG-Methods for psize = 1024 on an nCUBE/2

The results of the validation experiments for INTEL Paragon with problem size 1024 are summarized in Tab. 6.9. and Tab. 6.10., the results for this problem size are graphically displayed in Fig. 6.23. The accuracy of the prediction is always better than ± 10% except one measurement (64 processors) with a very small execution time.

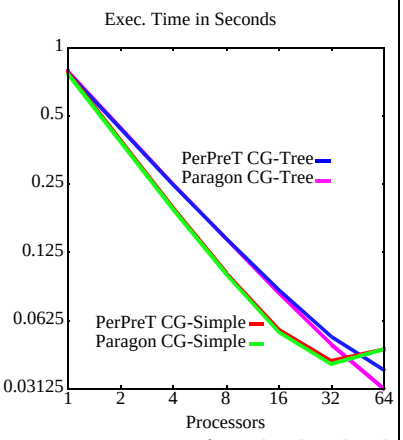



Fig. 6.23. Comparison of actual and predicted execution times of CG-Methods for psize = 512 on an INTEL Paragon

**Parallel CG-Simple, 1 to 64 processors:**

| P | mod | exp | Dsec | D% |
|---|---|---|---|---|
| 1 | 0.771 | 0.758 | 0.013 | 1.73 |
| 2 | 0.387 | 0.381 | 0.006 | 1.66 |
| 4 | 0.195 | 0.192 | 0.003 | 1.67 |
| 8 | 0.101 | 0.099 | 0.001 | 1.59 |
| 16 | 0.057 | 0.051 | 0.001 | 2.82 |
| 32 | 0.041 | 0.040 | 0.001 | 3.81 |
| 64 | 0.046 | 0.046 | -0.000 | -0.19 |

Tab. 6.9. Validation of parallel CG-Simple on INTEL Paragon (psize = 1024)

**Parallel CG-Tree, 1 to 64 processors:**

| P | mod | exp | Dsec | D% |
|---|---|---|---|---|
| 1 | 0.771 | 0.758 | 0.013 | 1.73 |
| 2 | 0.386 | 0.381 | 0.005 | 1.48 |
| 4 | 0.194 | 0.192 | 0.001 | 0.89 |
| 8 | 0.098 | 0.099 | -0.000 | -0.64 |
| 16 | 0.050 | 0.052 | -0.002 | -3.81 |
| 32 | 0.027 | 0.029 | -0.002 | -8.91 |
| 64 | 0.015 | 0.019 | -0.001 | -12.6 |

Tab. 6.10. Validation of parallel CG-Tree on INTEL Paragon (psize = 1024)

.

### 6.3.3. Shallow Water Code (PSTSWM)

PSTSWM is a message passing parallel program that solves the nonlinear shallow water equations on a rotating sphere using the spectral transform method. The program is used in the context of global climate modeling calculations. PSTSWM is an interesting case study in modeling for many reasons. It has numerous distinct phases, each with its own computation and communication rates and patterns. It has (static) load imbalances that change with the choice of parallel algorithm and logical processor mesh. It requires significant global communication during each timestep, divided into two collective operations that access the processors in different ways. Finally, PSTSWM is a representative member of an important class of simulation models. The goal of the following studies (see also [Bre98]) is to show that it is possible to build PerPreT models that are accurate enough to indicate which parallel algorithm is most efficient for a given problem size and number of processors on a given multiprocessor.

PSTSWM is written in Fortran 77 with VMS extensions and a small number of C preprocessor directives. Message passing is implemented using MPI [MPI94], PICL [PICL90], PVM [PVM94], or native message-passing libraries, with the choice being made at compile time. Optional performance instrumentation is implemented using the PICL trace and profile collection interface. PICL was used in the work described here, to collect performance data, but PICL simply represents a thin layer over the native NX message passing on the Paragon.

The shallow water equations in the form solved by the spectral transform method describe the time evolution of three *state* variables: vorticity, divergence, and a perturbation from an average geopotential. The velocities are computed from these variables. PSTSWM advances the solution fields in a sequence of timesteps. During each timestep, the state variables of the problem are transformed between the physical domain, where the physical forces are calculated, and the spectral domain, where the terms of the differential equation are evaluated. The physical domain for a given vertical level is a tensor product longitude-latitude grid. The spectral domain for a given vertical level is the set of spectral coefficients in a truncated spherical harmonic expansion of the state variables.

Transforming from physical coordinates to spectral coordinates involves performing a real fast Fourier transform (FFT) for each line of constant latitude, followed by integration over latitude using Gaussian quadrature (approximating the Legendre transform (LT)) to obtain the spectral coefficients. The inverse transformation involves evaluating sums of spectral harmonics and inverse real FFTs. The basic outline of each timestep is described as follows:

1. Evaluate non linear product and forcing terms.
2. Compute forward Fourier transform of non-linear terms.
3. Compute forward Legendre transforms.
4. Advance in time the spectral coefficients for the state variables.
5. Evaluate sums of spectral harmonics, simultaneously calculating the horizontal velocities from the updated state variables.
6. Compute inverse Fourier transform of state variables and velocities.

For more details on the steps in solving the shallow water equations using the spectral transform algorithm see [Hac92]. The parallel algorithms in PSTSWM are based on decompositions of the physical and spectral computational domains over a logical two-dimensional processor mesh of size **PX** x **PY**. Initially, the longitude dimension of the physical domain is decomposed over the processor mesh row dimension and the latitude dimension is decomposed over the column dimension. Thus, FFTs in different processor rows are independent, and each row of **PX** processors collaborates in computing a "block" of FFTs.

Similarly, the Legendre transforms in different processor columns are independent, and each column of **PY** processors collaborates in computing a "block" of Legendre transforms. The computation of the nonlinear terms at a given location on the physical grid is independent of that at other locations. The spectral domain decomposition is a function of the parallel algorithm used. In the version of PSTSWM used for modeling experiments, all computations on the spectral "grid" are likewise independent. Parallel efficiency is determined solely by the efficiency of the parallel algorithms used for the FFT and LT transforms and by any load imbalances caused by the choice of domain decomposition.

Two classes of parallel algorithms are available for each transform: distributed algorithms, using a fixed data decomposition and computing results where they are assigned, and transpose algorithms, remapping the domains to allow the transforms to be calculated sequentially. These represent four classes of parallel algorithms: distributed *FFT/distributed LT*, *transpose FFT/distributed LT*, *distributed FFT/transpose LT*, and *transpose FFT/transpose LT*. PSTSWM provides many parallel algorithms for each of the parallel algorithm classes [Wor92a]. These experiments are restricted to one transpose algorithm (for both FFT and LT), one distributed FFT algorithm, and two distributed LT algorithms, comprising the best parallel algorithms on the INTEL Paragon. These algorithms are briefly described below.

- **Transpose:**
  Assume that the transpose algorithm involves **Q** processors and that each processor contains **D** data to be transposed. Then every processor sends approximately **D/Q** data to every other processor, for a total of $\Theta$**(Q)** messages and a total per processor volume of $\Theta$**(D)**.

- **Distributed FFT:**
  Assume that the distributed FFT algorithm involves **Q** processors and that each processor contains **D** data to be transformed. Then each processor exchanges **D/2** data with its neighbors in a logical $\mathbf{(\log_2 Q)}$-dimensional hypercube, for a total of $\Theta$**(log Q)** messages and a total per processor volume of $\Theta$**(D log Q)**.

- **Distributed LT:**
  Assume that the LT is parallelized over **Q** processors and that each processor will contain **D** spectral coefficients when the transform is complete. Then the per processor communication costs for the two distributed LT algorithms can be characterized by
  - $\Theta$**(Q)** messages, $\Theta$**(DQ)** total volume
  - $\Theta$**(log Q)** messages, $\Theta$**(DQ)** total volume

  respectively. The $\Theta$**(Q)**-step algorithm works on a logical ring, each processor communicating only with its two neighbors. The $\Theta$**(log Q)**-step algorithm uses the same communication pattern as the distributed FFT algorithm.

These parallel algorithms for the FFT and LT generate the six parallel algorithms for the spectral transform method listed in Tab. 6.11.

There are many implementation variants possible for each of these algorithms, distinguished, for example, by the choice of communication protocol and the mapping of logical processors to physical processors. For

these experiments, those implementations are used that have proven most efficient on the INTEL Paragon. For details on the different implementation options, see [Wor92a].

| | |
|---|---|
| DH: | distributed FFT / Θ(logQ)-step distributed LT |
| DR: | distributed FFT / Θ(Q)-step distributed LT |
| DT: | distributed FFT / transpose LT |
| TH: | transpose FFT / Θ(logQ)-step distributed LT |
| TR: | transpose FFT / Θ(Q)-step distributed LT |
| TT: | transpose FFT / transpose LT |

Tab. 6.11.Candidate PSTSWM parallel algorithms

#### 6.3.3.1 PerPreT Application Description of PSTSWM

Assume that communication costs are negligible or scale linearly with the computation costs. Assume further that the computation rate varies in the same way across all algorithms as a function of the number of processors and of the problem size. Then a simple computational complexity analysis is sufficient to choose between the alternative parallel algorithms. If these assumptions do not hold or if runtime estimates are also needed, then it must be determined both, the computation and communication costs for a range of numbers of processors and of problem sizes.

Earlier research showed that different logical phases of a code may need to be modeled individually [Wal92]. Each phase has its own computation rate, depending on the amount of computation and the amount and pattern of memory accesses. As the number of processors and problem size change, the percentage of time spent in each phase changes. This changes the overall computation rate. In the following, models for important phases are identified and constructed. For brevity, only the phase models for algorithm *TH* (transpose FFT / Θ(logQ)-step distributed LT) are presented.

**Parameters**

As mentioned before, PerPreT expects one formula for the computation and one formula for the communication behavior of an application as input. These formulae use the number of processors and the problem size as parameters. For PSTSWM, the problem is specified by 8 parameters (to describe the physical grid and the number of timesteps to be executed): **DT**, **TAUE**, **MM**, **NN**, **KK**, **NLAT**, **NLON**, **NVER**, and specification of initial data and forcing function. The data and forcing function specification is fixed in these experiments and the following performance models are specific to the particular test case[1], representing the calculation of solid body rotation steady state flow [Wor92a]. **DT** is the length of the timestep and **TAUE** is the duration of the model run in simulated time. Thus, **TAUE/DT** is the number of timesteps in the simulation. For these experiments the number of timesteps is fixed at 108. **MM**, **NN**, and **KK** determine which spectral

1. Most of the other test cases differ only in calculation of the nonlinear terms, and only one phase model would need to be changed when changing cases.

coefficients are generated. The common choice of **MM**= **NN**=**KK** is used, which implies that **MM+1** Fourier coefficients are retained from the Fourier transform and **(MM+1)(MM+2)/2** spectral coefficients are used in the spectral representation. **NLAT**, **NLON**, and **NVER** define the tensor-product physical grid of size **NLON**x**NLAT**x**NVER**. These values are also a function of **MM** when the computational complexity is minimized subject to satisfying an anti-aliasing condition. The number of processors used is specified by the logical processor mesh **PX**x**PY**.

The costs associated with each phase of PSTSWM are functions of the domain decomposition relevant to the phase. There are two decompositions of the physical domain (**longitude** x **latitude** x **vertical levels**):

- **NLLON_P, NLLAT_P,** and **NLVER_P**, denoting the number of local longitudes, latitudes, and vertical levels assigned to a given processor during physical domain computations,
- **NLLON_F, NLLAT_F,** and **NLVER_F**, denoting the number of local longitudes, latitudes, and vertical levels assigned to a given processor during the Fourier transform phases,

and one decomposition of the Fourier domain (**wavenumber** x **latitude** x **vertical levels**):

- **NLMM_S, NLLAT_S,** and **NLVER_S**, denoting the number of local wavenumbers, latitudes, and vertical levels assigned to a given processor during the Legendre transform phases,

and one decomposition of the spectral domain (**spectral coefficients x vertical levels**):

- **NLSP_S, NCSP_S,** and **NLVER_S**, denoting the number of spectral coefficients assigned to a single processor and to a single column of processors, respectively, during computations in the spectral domain.

The values for these 11 parameters are functions of **MM**, **NN**, **KK**, **NLAT**, **NLON**, **NVER**, **PX**, **PY**, and the parallel algorithm being used. The values for parallel algorithm *TH* are as follows:

**NLLON_P** = $\lceil$**NLON/PX**$\rceil$
**NLLAT_P** = 2$\lceil$**NLAT/2PY**$\rceil$
**NLVER_P** = **NVER**
**NLLON_F** = **NLON**
**NLLAT_F** = 2$\lceil$**NLAT/2PY**$\rceil$
**NLVER_F** = $\lceil$**NVER/PX**$\rceil$
**NLMM_S** = **MM+1**
**NLLAT_S** = 2$\lceil$**NLAT/2PY**$\rceil$
**NLVER_S** = $\lceil$**NVER/PX**$\rceil$
**NCSP_S** = **(MM+2)(MM+1)/2**
**NLSP_S** = **NCSP_S**

These are maximum values across all processes, and load imbalance enters via the floor and ceiling functions in the expressions. The load imbalance varies with logical grid aspect ratio and parallel algorithm, and between the different computational domains.

### Computation Model

PerPreT requires a simple algebraic expression for the number of arithmetic statements executed by each processor. If this number varies for different processors, the maximum is used. To implement different models for different phases, a separate algebraic expression is generated for each phase. The computation model for the entire program is a weighted sum of the phase expressions, where the weights are the computation rates associated with the different phases.

Phases are included which involve only copying. In parallel codes, copying is often a significant cost. For example, for the transpose-based parallel algorithms the indices of the field arrays must be in a different order for the transposition than for the computation. This requires an explicit copy before and after the communication phases. The phase computation models in Tab. 6.12. for parallel algorithm *TH* were derived from the source

code and are of two types: number of floating point computations and number of bytes copied. For the purposes of these experiments, simple models that an industrious application developer would be willing to generate are used. Some phases are interleaved in time even for a single timestep, and a given phase model represents the sum of all calls to the relevant code during one time step. Later it will be examined whether this number of phases is necessary or sufficient. The phase models come in two forms: one-parameter (single rate) and two-parameter models. All of the phases show some performance sensitivity to problem size and aspect ratio, but many of the computational phases are relatively insensitive and a single rate is sufficient. (Accuracy issues are examined in detail in section 6.3.3.3) The variations in Tab. 6.12. between different phases arise from different access patterns to and from memory, and from differing amounts of computation per memory access.

| Phase | Model | Rate |
|---|---|---|
| | physical domain computation | (1/a, 1/b) |
| 1 | **12 · NLLON_P · NLLAT_P · NLVER_P** | 4.8 |
| | forward FFT | |
| 2 | **⌈(PX-1)/PX⌉ · 32 · NLLAT_P · NLVER_P · (a + b·NLLON_P)** | (4.5, 23.1) |
| 3 | **⌈(PX-1)/PX⌉ · 32 · NLLAT_P · NLVER_F · (a + b·NLLON_F)** | (17.7, 21.6) |
| 5 | **20 · NLLAT_F · NLVER_F · NLLON_F · (a + b·log$_2$(NLLON_F/4))** | (3.8, 24.0) |
| 6 | **64 · NLLAT_F · NLVER_F · (a + b·NLLON_F/4)** | (4.0, 15.2) |
| 7 | **144 · NLLAT_F · NLVER_F · (a + b·NLLON_F/4)** | (10.4, 19.8) |
| | forward LT | |
| 9 | **(PY-1) · 6 · NLVER_S · NCSP_S/PY** | 4.4 |
| 10 | **61 · NLVER_S · NLMM_S · NLLAT_S** | 10.0 |
| 11 | **(14 · NLLAT_S-1) · NCSP_S · NLVER_S** | 15.1 |
| | spectral domain computation | |
| 12 | **13 · NLSP_S · NLVER_S** | 11.5 |
| | inverse LT | |
| 13 | **17 · NCSP_S · NLVER_S** | 7.0 |
| 14 | **(14 · NCSP_S + 10 · NLMM_S) · NLLAT_S · NLVER_S** | 12.8 |
| 17 | **40 · NLLAT_F · NLVER_F · (a + b·(NLLON_F/2 - NLMM_S))** | (22.1, 36.8) |
| | inverse FFT | |
| 18 | **70 · NLLAT_F · NLVER_F · (a + b·NLLON_F/4)** | (8.8, 20.4) |
| 19 | **40 · NLLAT_F · NLVER_F · (a + b·NLLON_F/2)** | (2.8, 18.6) |
| 20 | **(25/2) · NLLAT_F · NLVER_F · NLLON_F · (a+b·log$_2$(NLLON_F/4))** | (3.8, 24.0) |
| 21 | **⌈(PX-1)/PX⌉ · 20 · NLLAT_F · NLVER_F · NLLON_F** | 10.2 |
| 22 | **⌈(PX-1)/PX⌉ · 20 · NLLAT_F · NLLON_P · (a·PX + b·NLVER_P)** | (15.2, 18.6) |

Tab. 6.12.Computational models and MFLOP/s or MByte/s rates for algorithm TH

In contrast, rates for phases with low computation to memory access ratios, like copy phases, vary significantly with aspect ratio and problem size. With a few exceptions, this variation is approximated reasonably well with the following two-parameter model: a rate for the total number of operations and a rate for the number of times that the inner loop is executed. The form of these models was derived empirically, but one justification is that it takes into account the additional cost

of crossing cache and page boundaries when accessing memory.

The phases requiring two-parameter models and the rates for all models were determined empirically. Timings were taken from a series of 8-processor runs using two different problem sizes, 32 bit precision, and all possible aspect ratios (1x8, 2x4, 4x2, 8x1). For one-parameter models, the maximum observed rates are used. This avoids contamination from atypical rates arising from inefficient memory alignments or poor cache performance. For the two-parameter models, typical or median values are used, giving preference to rates for the smaller problem when there is a significant discrepancy. The intent is to better capture the behavior when extrapolating to larger numbers of processors. If a rate for a phase showed variation but could not be accurately fit with the type of two-parameter models described above, a one-parameter model is used.

Interactions with the memory hierarchy are major determiners of computation and copy rates, and these change in a phase as the problem and algorithm parameters vary. Even one-parameter phase models that are highly accurate for the 8-processor calibration runs will be valid only for a range of problem and machine parameters. Consequently, there will be errors in the rates when extrapolating, and scalability will be a problem even in a phase model approach. Algebraic models that take into account memory access patterns are possible, but such models are unlikely to be developed by an application programmer and are not discussed here. The hope is that the range of validity of the rates is large enough or that the degradation affects all phases in a similar enough way that the algorithm comparisons will be reasonably accurate.

### Communication Model

PerPreT requires a high-level description of the communication in a parallel program. For PSTSWM, communication models are required for the two parallel FFT and for the three parallel LT algorithms. The detailed models are given in Tab. 6.13.

| Direction | Model |
|---|---|
| | Distributed FFT |
| forward | $\lceil \mathbf{(PX-1)/PX} \rceil \cdot (\mathbf{1 + \log_2(PX)}) \cdot \mathbf{comm(32 \cdot NLLAT\_P \cdot NLVER\_P \cdot \lceil NLLON\_P/2 \rceil}$ |
| inverse | $\lceil \mathbf{(PX-1)/PX} \rceil \cdot (\mathbf{1 + \log_2(PX)}) \cdot \mathbf{comm(20 \cdot NLLAT\_P \cdot NLVER\_P \cdot \lceil NLLON\_P/2 \rceil}$ |
| | Transpose FFT |
| forward | $\mathbf{(PX-1) \cdot comm(32 \cdot NLLAT\_P \cdot NLVER\_F \cdot NLLON\_P)}$ |
| inverse | $\mathbf{(PX-1) \cdot comm(20 \cdot NLLAT\_P \cdot NLVER\_F \cdot NLLON\_P)}$ |
| | Θ(Q)-step distributed LT |
| forward | $\mathbf{(PY-1) \cdot comm(24 \cdot NLVER\_S \cdot NLSP\_S)}$ |
| inverse | $\mathbf{(PY-1) \cdot comm(24 \cdot NLVER\_S \cdot NLSP\_S)}$ |
| | Θ(logQ)-step distributed LT |
| forward inverse | $\sum_{i=1}^{\log_2 PY} 2 \cdot \mathbf{comm(8 \cdot \lceil 3 \cdot NLVER\_S \cdot NCSP\_S/2^i \rceil)}$ |
| | Transpose LT |
| forward | $\mathbf{(PY-1) \cdot comm(64 \cdot NLLAT\_F \cdot NLVER\_S \cdot NLMM\_S)}$ |
| inverse | $\mathbf{(PY-1) \cdot comm(40 \cdot NLLAT\_F \cdot NLVER\_S \cdot NLMM\_S)}$ |

Tab. 6.13. Communication models for forward and inverse transforms

The **comm(mess_length)** function in Tab. 6.13. returns the time needed for one communication between two processors of the multiprocessor. The parameter **mess_length** is the message length in bytes. Contention for bandwidth and other network resources and distance in the network are ignored in these experiments. The models are parameterized solely by the number of messages and by the size of each message for a given processor.

Note that the nature of the communication varies significantly between the different algorithms. The distributed FFT and Θ(logQ)-step distributed LT use a butterfly pattern in their communication. In the transpose algorithm, each processor sends to every other processor, using an exclusive-OR ordering to avoid some contention. In the Θ(Q)-step distributed LT, each processor sends and receives from only two other processors, and the two processors are chosen to be neighbors in the physical network if possible. The Θ(Q)-step distributed LT also attempts to overlap the communication with computation. None of these differences are taken into account in these models, although they could be, and it is also examined whether more detailed models are needed. More detailed models of the communication cost are known to be necessary if poor communication algorithms or protocols are used. For example, a transpose algorithm in which all processors send to processor 0, then processor 1, etc., serializes the communication, and the maximum per processor number of messages and message volume will not represent the communication cost. The goal of the algorithm comparison is to compare good parallel implementations with the hope that more detailed communication models are not necessary.

#### 6.3.3.2 Variation of Problem Size and Processors

The performance models described in the previous sections are meant to be simple enough to be generated by the application developer, yet accurate enough to be used when scaling problem and machine parameters and when comparing alternative parallel algorithms. The approach taken here has been to construct the application model from a set of phase models.

After the application description in terms of computation and communication model is available, experiments with varying processor configurations and different input data sets are carried out. To give an impression of the form of experiments which are possible, Tab. 6.14. shows a result table for modeling PSTSWM with PerPreT. The goal of this experiment was to find out how the PSTSWM application scales for a given input data set considering different topologies.

**PerPreT results table for modeling PSTSWM for algorithm with TR communication (transpose FFT, ring LT), problem size set T85:**

| | Procs. | Comm. Time | Comp. Time | Exec. Time | SP | EFF |
|---|---|---|---|---|---|---|
| | 1 | 0.000000 | 3208.571642 | 3208.571642 | 1.00 | 1.000 |
| **w** | **2x1** | **33.154756** | **1778.032953** | **1811.187709** | **1.77** | **0.886** |
| **b** | **1x2** | **7.013710** | **1615.251173** | **1622.264883** | **1.98** | **0.989** |
| **w** | **4x1** | **24.934107** | **892.331102** | **917.265208** | **3.50** | **0.874** |
| | 2x2 | 20.114473 | 894.499153 | 914.613626 | 3.51 | 0.877 |
| **b** | **1x4** | **10.571524** | **818.590939** | **829.162463** | **3.87** | **0.967** |
| | 8x1 | 14.703656 | 49.480176 | 464.183831 | 6.91 | 0.864 |
| | 4x2 | 4.288521 | 448.906889 | 463.195410 | 6.93 | 0.866 |
| **w** | **2x4** | **13.642491** | **452.732252** | **466.374743** | **6.88** | **0.860** |
| **b** | **1x8** | **12.439284** | **420.259845** | **432.699129** | **7.42** | **0.927** |
| | 16x1 | 8.217158 | 228.054713 | 236.271871 | 13.58 | 0.849 |
| **b** | **8x2** | **8.360842** | **226.110757** | **234.471598** | **13.68** | **0.855** |
| | 4x4 | 9.012488 | 227.194782 | 236.207270 | 13.58 | 0.849 |
| **w** | **2x8** | **10.496287** | **231.848314** | **242.344601** | **13.24** | **0.827** |
| | 1x16 | 13.554605 | 221.094298 | 234.648903 | 13.67 | 0.855 |
| **w** | **32x1** | **8.959784** | **219.677039** | **228.636823** | **14.03** | **0.439** |
| **b** | **16x2** | **4.802086** | **114.712691** | **119.514777** | **26.85** | **0.839** |
| | 8x4 | 5.235494 | 114.426047 | 119.661542 | 26.81 | 0.838 |
| | 4x8 | 6.464724 | 116.338485 | 122.803210 | 26.13 | 0.816 |
| | 2x16 | 9.104625 | 121.406345 | 130.510969 | 24.58 | 0.768 |
| | 1x32 | 14.475145 | 121.511524 | 135.986669 | 23.59 | 0.737 |
| **w** | **64x1** | **10.056856** | **216.088023** | **226.144879** | **14.19** | **0.222** |
| | 32x2 | 5.415319 | 110.523854 | 115.939173 | 27.67 | 0.432 |
| **b** | **16x4** | **3.140260** | **58.041680** | **61.181940** | **52.44** | **0.819** |
| | 8x8 | 3.763308 | 58.583571 | 62.346878 | 51.46 | 0.804 |
| | 4x16 | 5.372283 | 60.910336 | 66.282619 | 48.41 | 0.756 |
| | 2x32 | 8.771674 | 66.185360 | 74.957034 | 42.81 | 0.669 |
| | 1x64 | 15.778748 | 71.721114 | 87.499862 | 36.67 | 0.573 |
| **w** | **128x1** | **12.056913** | **215.493157** | **227.550070** | **14.10** | **0.110** |
| | 64x2 | 6.447695 | 108.729346 | 115.177041 | 27.86 | 0.218 |
| | 32x4 | 3.688796 | 55.947261 | 59.636058 | 53.80 | 0.420 |
| | 32x4 | 3.688796 | 55.947261 | 59.636058 | 53.80 | 0.420 |
| **b** | **16x8** | **2.399950** | **29.706113** | **32.106064** | **99.94** | **0.781** |
| | 8x16 | 3.208654 | 30.662332 | 33.870986 | 94.73 | 0.740 |
| | 4x32 | 5.188942 | 33.196262 | 38.385204 | 83.59 | 0.653 |
| | 2x64 | 9.389745 | 38.575356 | 47.965101 | 66.89 | 0.523 |
| | 1x128 | 18.061229 | 71.733334 | 89.794563 | 35.73 | 0.279 |

Tab. 6.14. PerPreT result table for modeling PSTSWM on Paragon

Each row of Tab. 6.14. represents the model results for one run of the PSTSWM program.

- The first column of Tab. 6.14. contains the number of processors used for the experiment and their configuration (mesh). For a given number of processors larger than one, several configurations are possible. Eight processors e. g. can be configured as 8x1, 2x4, 4x2, or 1x8 mesh. The highest execution time for a configuration is marked with the letter w (worst) and the lowest execution time is marked with the letter b (best).
- The second column contains the estimated communication time.
- The third column contains the estimated computation time.
- The fourth column contains the estimated total execution time of the program.
- Taking the estimated execution time on one processor and the estimated execution time on *nprocs* processors, the speedup can be calculated in column five.
- Dividing the speedup by *nprocs* results in the efficiency in column six.

Six different versions of the PSTSWM code (compare Tab. 6.11.) are modeled with two input data sets each and processor numbers varying from 1 to 128 processors are modeled. Using PerPreT, 12 experiments (each algorithm with two input data sets) have to be carried out . The time to run a PerPreT experiment is approximately five seconds.

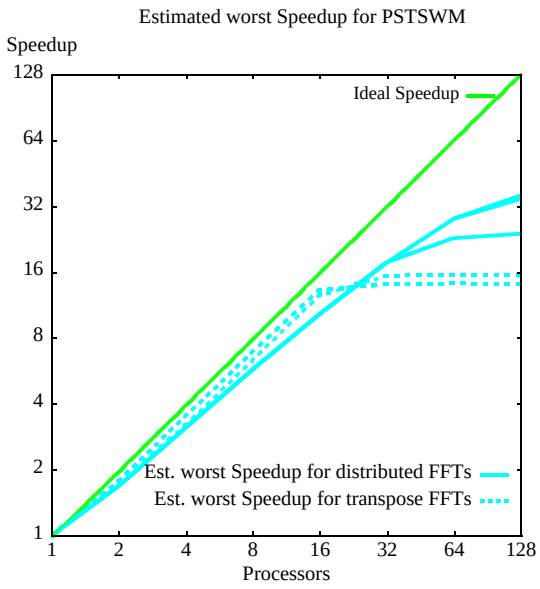



Fig. 6.24. Worst speedups for PSTSWM implementations

If these experiments are carried out using the implementation of PSTSWM on the real machine, 12*36=432 instrumented runs with execution times between 3200 seconds and several seconds would be necessary. All algorithms show good speedup behavior for the best topology for each number of processors. These tables also indicate that a non optimal

topology will result in high losses of efficiency. Fig. 6.24. summarizes the speedup behavior of the six PSTSWM algorithms for the choosing the topologies with the longest execution times. In contrast to the almost linear speedups of the optimal topologies, the speedups only reach approximately 16 for transpose FFTs and 32 for distributed FFTs for 128 processors.

#### 6.3.3.3 Validation

After the result tables with the predicted times are produced it is important to know the accuracy of the individual algorithm models. Then the models are used to investigate the following performance questions:

1. What is the best logical aspect ratio to use for a given parallel algorithm and for a given number of processors?
2. What is the best parallel algorithm to use for a given number of processors?
3. How long will the application take to complete a run?

Two problem sizes are investigated, denoted by T42 and T85:

| | T42 | T85 |
|---|---|---|
| **MM** | 42 | 85 |
| **NN** | 42 | 85 |
| **KK** | 64 | 85 |
| **NLAT** | 128 | 128 |
| **NLON** | 128 | 256 |
| **NVER** | 16 | 16 |

Tab. 6.15.Problem size parameters for PSTSWM

For the three performance questions, **P= 8,16,32,64,128,256,512** are discussed. The optimal logical aspect ratio is determined for each parallel algorithm. The optimal parallel algorithms are determined over all algorithms and aspect ratios. The estimation of runtimes is discussed in terms of the optimal parallel algorithms. Finally, the models are reexamined, evaluating the effectiveness and importance of the phase model approach in being able to answer the stated performance questions.

**Optimal aspect ratio**

The first performance question of interest for PSTSWM is how to allocate processors among the different parallel transforms to minimize execution time, i.e., for a given number of processors, what logical aspect ratio should be used. The relative accuracy of the execution time predictions is important here, not the absolute accuracy. Tab. 6.16.describes the true and predicted optimum for different numbers of processors *when they differ*, and the percentage loss from using the model results. The loss is measured in the following way. Let **PRED** represent the predicted optimal aspect ratio. Let **OPT** represent the true optimal aspect ratio. The percentage loss is defined as:

$$\frac{100 \cdot (predictedtime - truetime)}{truetime}$$

Only 17 of the 84 model predictions are incorrect, and only 4 of these result in errors in runtime of more than 5%. Performance on the Paragon is very consistent, but there is some small variation between runs. The 7 cases in which the "error" is less than 1% should probably be considered correct

| Procs. | T42 model results | T42 experimental results | T42 %error in runtime | T85 model results | T85 experimental results | T85 %error in runtime |
|---|---|---|---|---|---|---|
| DH (3 errors) | | | | | | |
| 32 | 4x8 | 2x16 | 0.5 | 4x8 | 4x8 | - |
| 512 | 16x32 | 32x16 | 8.5 | 16x32 | 32x16 | 0.2 |
| DR (2 errors) | | | | | | |
| 32 | 1x32 | 4x8 | 1.2 | 1x32 | 1x32 | - |
| 512 | 16x32 | 32x16 | 16.8 | 32x16 | 32x16 | - |
| DT (no errors) | | | | | | |
| TH (2 errors) | | | | | | |
| 32 | 16x2 | 8x4 | 0.2 | 8x4 | 16x2 | 0.3 |
| TR (4 errors) | | | | | | |
| 16 | 1x16 | 4x4 | 5.4 | 1x16 | 1x16 | - |
| 32 | 16x2 | 8x4 | 2.3 | 8x4 | 4x8 | 0.2 |
| 64 | 16x4 | 16x4 | - | 16x4 | 8x8 | 0.3 |
| TT (6 errors) | | | | | | |
| 16 | 61x1 | 16x1 | - | 16x1 | 1x16 | 5.6 |
| 32 | 1x32 | 4x8 | 4.4 | 1x32 | 1x32 | - |
| 64 | 16x4 | 8x8 | 2.6 | 16x4 | 8x8 | 3.4 |
| 128 | 16x8 | 8x16 | 0.9 | 16x8 | 8x16 | 2.5 |

Tab. 6.16.Error in choosing optimal ratio from model results

Tab. 6.17.-Tab. 6.19. are examples for validation tables of the *TR* algorithm (compare Tab. 6.11.). The predicted execution times (column two of the tables) is compared with the execution time of the real application running on the INTEL Paragon system. As in the previous examples an accuracy which is mostly better than ± 10% can be observed.

**8 Processors:**

| Procs. | mod | exp | Dsec | D% |
|---|---|---|---|---|
| 8 | 464.183 | 482.253 | -18.069 | -3.75 |
| 4x2 | 463.195 | 474.175 | -10.979 | -2.32 |
| 2x4 | 466.374 | 476.592 | -10.217 | -2.14 |
| 1x8 | 432.699 | 413.238 | 19.461 | 4.71 |

Tab. 6.17.Validation of PSTSWM (TR-T85) using 8 processors

**64 Processors:**

| Procs. | mod | exp | Dsec | D% |
|---|---|---|---|---|
| 64 | 226.144 | 226.451 | -0.306 | -0.14 |
| 32x2 | 115.939 | 114.010 | 1.929 | 1.69 |
| 16x4 | 61.181 | 58.206 | 2.975 | 5.11 |
| 8x8 | 62.346 | 58.012 | 4.334 | 7.47 |
| 4x16 | 66.282 | 61.562 | 4.720 | 7.67 |
| 2x32 | 74.957 | 70.744 | 4.213 | 5.96 |
| 1x64 | 87.499 | 84.818 | 2.681 | 3.16 |

Tab. 6.18.Validation of PSTSWM (TR-T85) using 64 processors

**128 Processors:**

| Procs. | mod | exp | Dsec | D% |
|---|---|---|---|---|
| 128 | 227.550 | 227.255 | 0.295 | 0.13 |
| 64x2 | 115.177 | 112.501 | 2.676 | 2.38 |
| 32x4 | 59.636 | 56.740 | 2.896 | 5.10 |
| 16x8 | 32.106 | 29.958 | 2.148 | 7.17 |
| 8x16 | 33.870 | 31.504 | 2.366 | 7.51 |
| 4x32 | 38.385 | 36.742 | 1.643 | 4.47 |
| 2x64 | 47.965 | 47.693 | 0.272 | 0.57 |

1x128 no experimental data available

Tab. 6.19.Validation of PSTSWM (TR-T85) using 128 processors

What is not indicated in Tab. 6.16. is how important it is to choose a good aspect ratio. The worst case aspect ratios are as much as ten times worse than the best case, primarily reflecting load imbalance.

Determining a good logical aspect ratio is important when implementing a parallel strategy. A parallel code could incorporate the flexibility to change at least some of these parameters at compile-time or runtime, in which case PerPreT simply makes this more convenient to determine. This convenience should not be underestimated. Determining the optimal aspect ratio experimentally requires access to the same number of processors as will be used in a production run and numerous, possibly expensive, experiments.

**Optimal parallel algorithm**

Determining the optimal parallel algorithm experimentally requires developing, tuning, and evaluating multiple parallel implementations. This is much more time consuming than determining the optimal aspect ratio experimentally, and there is much to be gained from using performance models to predict the optimal parallel algorithm. As before, relative accuracy in the predicted execution times is important. Tab. 6.20.indicates the true and predicted optimal parallel algorithm for different numbers of processors, and the percentage loss from using the model-identified algorithm, measured as in (4). The optimal aspect ratio was found for each parallel algorithm before being compared with the other parallel algorithms. The model results use the model-determined optimal aspect ratios. The empirical results use the experimentally-determined optimal aspect ratios.

The performance models correctly identify the optimal algorithm and aspect ratio in seven out of fourteen cases, and the correct algorithm (if not the optimal aspect ratio) in ten of the cases. The error in misidentifying the optimal algorithm was acceptable, especially for the "scaling" examples, P>8. The performance sensitivity of choosing the wrong algorithm (but with an optimum aspect ratio) is not as extreme as when choosing the aspect ratio, but worst case errors range as high as 85%. Note that when considering a larger sampling of interesting problem sizes, all of the parallel algorithms are optimal in some cases. It is not possible to eliminate any of the parallel algorithms *a priori*.

| | **T42** | | | **T85** | | |
|---|---|---|---|---|---|---|
| Procs. | model optimum | experimental optimum | %diff. in runtime | model optimum | experimental optimum | %diff in runtime |
| 8 | DR 1x8 | DR 1x8 | - | DT 1x8 | DR 1x8 | 6.2 |
| 16 | DT 1x16 | DT 1x16 | - | DT 1x16 | DR 1x16 | 1.8 |
| 32 | TR 8x4 | TR 8x4 | - | TR 16x2 | TR 4x8 | 1.5 |
| 64 | TR 16x4 | TR 16x4 | - | TR 16x4 | TR 8x8 | 0.3 |
| 128 | TH 16x8 | TR 16x8 | 1.1 | TT 16x8 | TT 8x16 | 2.5 |
| 256 | TH 16x16 | TT 16x16 | 3.7 | TT 16x16 | TT 16x16 | - |
| 512 | TH 16x32 | TH 16x32 | - | TT 16x32 | TT 16x32 | - |

Tab. 6.20. Error in choosing optimal alg. from model results instead of experimentally

**Runtime predictions**

When allocating resources, it is important to know how long a parallel job will take to run on a given number of processors. For example, runtime information is often required when submitting batch requests. This type of prediction requires a certain degree of absolute accuracy, but the degree needed is not great. (However, accurate predictions of runtime can be extremely important in real-time environments). Tab. 6.21. indicates how accurately the models predict the runtime for the model-determined "optimal" parallel algorithms (to pick particular examples). The percentage error is measured as in (4). With possibly one exception, the accuracy of these predictions is adequate for the determination of resource requirements. Note that similar accuracies hold for predicted speedup and parallel efficiency.

| | **T42** | | | **T85** | |
|---|---|---|---|---|---|
| Procs. | algorithm | predicted runtime | %error in prediction | algorithm | predicted runtime | %error in prediction |
| 8 | DR 1x8 | 79.8 | - 1.6 | DT 1x8 | 426.6 | - 2.8 |
| 16 | DT 1x16 | 40.9 | - 6.6 | DT 1x16 | 206.9 | - 8.4 |
| 32 | TR 8x4 | 23.0 | 2.2 | TR 16x2 | 118.6 | 0.7 |
| 64 | TR 16x4 | 12.2 | 1.7 | TR 16x4 | 60.6 | 4.3 |
| 128 | TH 16x8 | 6.7 | - 5.4 | TT 16x8 | 31.6 | 4.5 |
| 256 | TH 16x16 | 4.0 | - 11.1 | TT 16x16 | 16.8 | 1.8 |
| 512 | TH 16x32 | 2.6 | - 27.8 | TT 16x32 | 9.7 | - 5.8 |

Tab. 6.21. Error in predicting runtimes (seconds)

**Model accuracy requirements**

The previous results indicate that the accuracy of the phase model approach is adequate for algorithm tuning and comparison for this case study. Next it is discussed whether a simpler model might also suffice. There are numerous ways to simplify the current model. Here, only a few obvious alternatives are considered. First, the optimal algorithm is chosen on the basis of arithmetic complexity alone, ignoring copy phases, communication costs, and phase-dependent rates. (Including copy and communication complexity would require some sort of rate estimation to weight the different components of the model.)

Tab. 6.22. indicates the true and predicted optimal parallel algorithms using this simplified model, and the percentage loss from using the model-identified algorithm. These predictions are not as good as those from using a phase model. Depending on the application, the size of these errors may or may not be acceptable. But, since the error in the prediction is not known in practice, the wide and unpredictable variation in the error is worrisome.

| | T42 | | | T85 | | |
|---|---|---|---|---|---|---|
| Procs. | model optimum | experimental optimum | %diff. in runtime | model optimum | experimental optimum | %diff. in runtime |
| 8 | DT 1x8 | DR 1x8 | 6.6 | DT 1x8 | DR 1x8 | 6.3 |
| 16 | DT 1x16 | DT 1x16 | - | DT 1x16 | DR 1x16 | 1.8 |
| 32 | DT 2x16 | TR 8x4 | 17.3 | DT 2x16 | TR 4x8 | 10.9 |
| 64 | DT 4x16 | TR 16x4 | 22.3 | TT 16x4 | TR 8x8 | 7.5 |
| 128 | TT 16x8 | TR 16x8 | 2.7 | TT 16x8 | TT 8x16 | 25 |
| 256 | TT 16x16 | TT 16x16 | - | TT 16x16 | TT 16x16 | - |
| 512 | TR 16x32 | TH 16x32 | 45.1 | TT 16x32 | TT 16x32 | - |

Tab. 6.22. Error in choosing optimal algorithm from complexity analysis instead of experimentally

Runtimes cannot be predicted from the complexity analysis alone. The next models considered use the sustained computation rate for an 8-processor run for a given parallel algorithm to weight the corresponding arithmetic complexity model. Unlike for the phase models, a separate rate was determined for each problem size. Tab. 6.23. indicates how accurately these models predict the runtime for the above model-determined "optimal" parallel algorithms. For this type of model to be accurate requires that either copy and communication costs are negligible or they scale similarly with the computation costs, and that the rates are insensitive to scaling. It is clear from Tab. 6.23. that these conditions do not hold for PSTSWM.

| | T42 | | | T85 | | |
|---|---|---|---|---|---|---|
| Procs. | algorithm | predicted runtime | %error in prediction | algorithm | predicted runtime | %error in prediction |
| 16 | DT 1x16 | 41.2 | - 6.3 | DT 1x16 | 205.3 | - 9.1 |
| 32 | DT 2x16 | 20.8 | - 21.1 | DT 2x16 | 103.3 | - 20.9 |
| 64 | DT 4x16 | 10.6 | - 27.4 | TT 16x4 | 50.6 | - 18.6 |
| 128 | TT 16x8 | 5.5 | - 23.0 | TT 16x8 | 25.7 | - 15.1 |
| 256 | TT 16x16 | 3.0 | - 32.0 | TT 16x16 | 13.1 | - 20.8 |
| 512 | TR 16x32 | 1.6 | - 56.1 | TT 16x32 | 6.9 | - 33.1 |

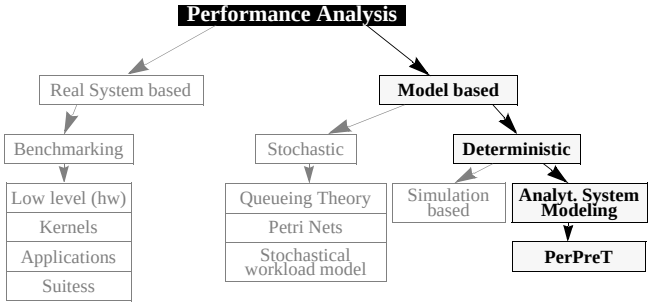
Tab. 6.23. Error in predicting runtime (seconds) using complexity based model

Our final simplified model includes terms for computation, copy, and communication costs, but does not take into account phase-specific rates. Instead average copy and computation rates determined from the 8-processor runs are used. As before, different rates are used for each parallel algorithm and problem size. Tab. 6.24. indicates how accurately this type of single-phase model predicts the runtime for the phase model "optimal" parallel algorithms (to allow direct comparison with the phase model results). With the exception of predictions for T42 for large numbers of processors, the single-phase model is

as accurate a predictor of runtime as is the (multiple-) phase model. So the question arises whether a phase model is required as long as the copy, computation, and communication costs are included in the model. A phase model does not appear to be required for accurate performance prediction for PSTSWM. However, the act of constructing the phase model was necessary. The error prone aspect of the phase model approach was in the generation of the phase model expressions. These same expressions are needed in a single-phase model (or in a complexity analysis). The additional step of calculating rates and validating the individual phase models also validates the expressions. Modeling phases can also identify performance "problems", for example, code that is overly sensitive to aspect ratio due to compiler peculiarities. Using average rates and a single-phase model removes the necessity of detailed profiling to determine individual phase model rates, but makes it more difficult to validate the model.

| | **T42** | | | **T85** | |
|---|---|---|---|---|---|
| Procs. | algorithm | predicted runtime | %error in prediction | algorithm | predicted runtime | %error in prediction |
| 8 | DR 1x8 | 79.8 | - 1.6 | DT 1x8 | 426.6 | - 2.8 |
| 16 | DT 1x16 | 40.9 | - 6.6 | DT 1x16 | 206.9 | - 8.4 |
| 32 | TR 8x4 | 23.0 | 2.2 | TR 16x2 | 118.6 | 0.7 |
| 64 | TR 16x4 | 12.2 | 1.7 | TR 16x4 | 60.6 | 4.3 |
| 128 | TH 16x8 | 6.7 | - 5.4 | TT 16x8 | 31.6 | 4.5 |
| 256 | TH 16x16 | 4.0 | - 11.1 | TT 16x16 | 16.8 | 1.8 |
| 512 | TH 16x32 | 2.6 | - 27.8 | TT 16x32 | 9.7 | - 5.8 |

Tab. 6.24. Error in predicting runtime (seconds) using single phase model




## 6.4. Summary PerPreT

For performance modeling with deterministic modeling techniques as described in this section, the system load or the system architecture, or both, are represented through a model.

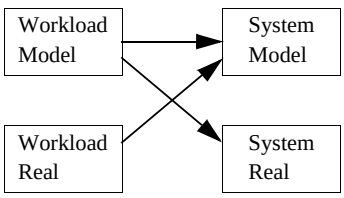



Fig. 6.25. Performance modeling

Deterministic models to analyze the performance of systems are cheaper and more efficient than performance measurement, because the implementation of parallel workloads and the monitoring of the workload execution on multiprocessors is a time consuming task.

Any representation of reality through models is an abstraction of reality and thus, suffers a loss of accuracy. The more detailed and the more complex a model is, the smaller these losses might be. It is important to make reasonable assumptions regarding the structure and detail of the models. The art of modeling is to find the perfect trade-off between needed accuracy and degree of abstraction.

In contrast to the stochastic modeling techniques presented in the previous section, PerPreT includes a detailed, scalable, parametrized modeling of the workload. The workload description is also independent of the system description and vice versa. Thus, each workload description and each system description can be combined to run performance prediction experiments. Since problem size and numbers of processors used to execute the workload are parameters in both, the workload and system description, series of experiments with varying processor num-

ber or varying problem sizes can be carried out for one workload model and one system model.

In terms of efficiency, the experiments can help to find:

- the optimal *aspect ratio* of a processor configuration,
- the optimal *processor topology* for an application,
- the optimal *parallelization strategy*,
- the optimal *number of processors* for a given problem size,
- the optimal *problem size* for a given number of processors.

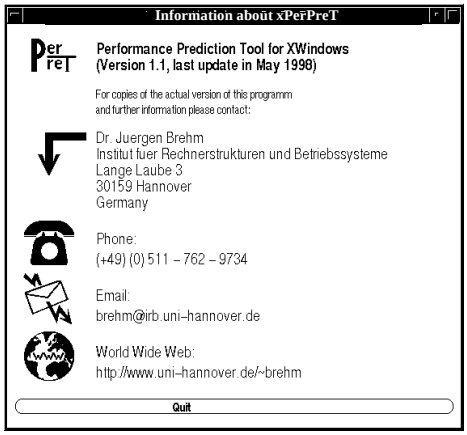
For the system designer, PerPreT can be helpful to:

- identify *system bottlenecks*,
- determine optimal *system parameters* (such as setup times, bandwidth, arithmetic performance) for given workloads
- build fictive systems by varying the system parameters and evaluate their behavior.

Additionally, PerPreT descriptions of workloads combined with PerPreT system descriptions can be used as benchmarks to compare systems facing particular workloads.

In summary, PerPreT is a tool to tune parallel applications and compare systems. Unlike some other modeling techniques, PerPreT can also be used for massively parallel systems running complex real applications.

The graphical user interface and its graphical output features make is easy to use. A description of the PerPreT software is given in the next section.




# 7. The PerPreT Software

This section describes the PerPreT software which can be used to execute performance prediction experiments for parallel applications running on message passing multiprocessor systems. The software is freely available by contacting the address given above.

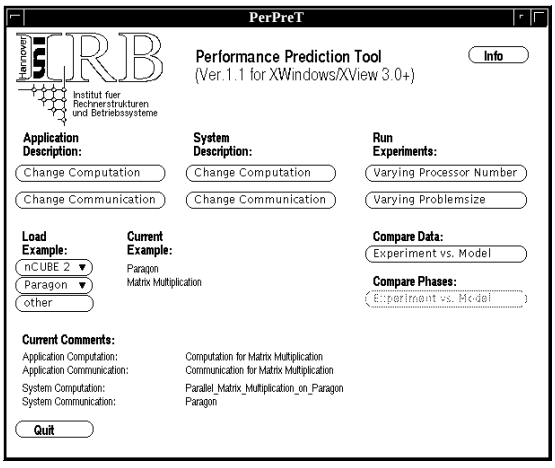



Fig. 7.1.PerPreT main window

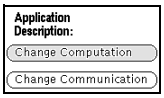
PerPreT is implemented using the C-Programming language [Ker88] and can thus be run on any hardware platform equipped with a C-Compiler. PerPreT can be executed in ASCII mode using menus. If an X-Windows environment with XView libraries is avail-

able, the graphical user interface of PerPreT makes it easier to run experiments using different application and system descriptions.

Starting PerPreT opens the main windows as shown in Fig. 7.1. This window reflects the PerPreT structure and modules which are *Application Description*, *System Description*, *Run Experiments*, *Compare Data*, and *Compare Phases*. Each of these modules is implemented through one or more subwindows described in the rest of this section. Additionally, PerPreT includes a set of example applications (*Conjugate Gradient*, *FFT*, *Matrix Multiplication*, *Red-Black Relaxation*, *PSTSWM*) and example system descriptions (*nCUBE/2*, *INTEL Paragon*).

# 7.1. Application Description

## 7.1.1. Computation Description (without phases)




Following the SPMD programming model as described in the previous section, the computation description of an application for PerPreT can be derived calculating the sum for the number of statements which have to be executed. The term statements may refer to arithmetic statements for numerical applications, to instructions, or to transactions for data base applications. If in a computation phase of the SPMD application the number of statements to be executed differs for some processors, the maximum number is taken for the PerPreT formula.

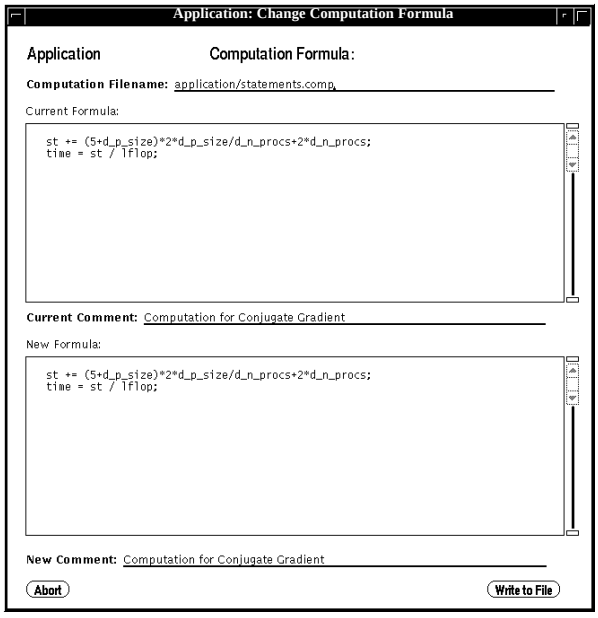



Fig. 7.2. PerPreT application computation description window

The PerPreT application computation description subwindow is shown in Fig. 7.2. It can be activated using the *Change Computation* button of the PerPreT main window. The application description files are stored in a subdirectory *application* of the PerPreT main

directory, the default filename for the application computation description is *statements.comp*.

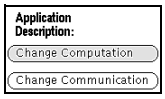
The upper part of the window shows the computation description formula. It is expressed in C-program statements. The variables *st* (for number of statements), *d_p_size* (for problem size), *d_n_procs* (for number of processors), *time* (for time in seconds) and *lflop* (for node performance) are of data type *double* and predeclared in the PerPreT environment. The variables *st* and *time* are result variables for the experiment output and initialized with the value zero, the variables *d_p_size*, *d_n_procs*, and *lflop* are runtime variables and predefined for every experiment.

To form a valid PerPreT formula to describe the computation behavior of an application, a sequence of C-statements containing the predefined variables is used. It is possible to declare additional variables if necessary. As a result of the computation description formula, the variable *st* should contain the total number of statements that have to be executed and the variable *time* is the value of the variable *st* divided by the value of the variable *lflop*. The derivation of the *lflop* value is described in section 7.3.1.

The PerPreT application computation descriptions for all examples (*Conjugate Gradient, FFT, Matrix Multiplication, Red-Black Relaxation, PSTSWM*) are automatically loaded with the examples.

The lower part of the subwindow shown in Fig. 7.2. can be used to change existing descriptions to new descriptions and store them using new filenames or to overwrite existing descriptions. Each description can be equipped with a comment which is also stored in the description file.

### 7.1.2. Computation Description (with phases)



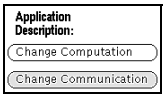

Complex applications like the PSTSWM code might consist of several computation phases with significantly different performance for the target node. If these phases are of different weight for the execution time and if these phases do not scale equivalent with the problem size parameters, using a mean value to describe the system computation performance is not enough to achieve accurate predictions. In that case, it is possible to use more than a single variable to describe the computation performance for a given problem size. The two-dimensional data array *mflop[index][phase]* is a predefined variable of type double which may be used to store performance measures. This data array is automatically initialized using the system computation description (compare section 7.3.2.).

Fig. 7.3. shows a small fraction of the PSTSWM computation description. The PerPreT predefined variables *st* and *time* are used as before, the variable *lflop* contains different values for different computation phases and is thus initialized for each phase using the *mflop* data array.

```
/* calculate RHS */
st = NLLON_PHY * NLLAT_PHY * NLVER_PHY * 12;
st *= TIME_STEP; lflop = mflop[pc][i]; pc++;
time += st / lflop;
/* sequential forward FFT - phase 5 */
/* cost of first step of sequential forward block fft */
st = (8 * NLLAT_FOU * NLVER_FOU * NLLON_FOU/4 * 10);
st *= TIME_STEP; lflop = mflop[pc][i]; pc++; time += st / lflop;
/* cost of remaining steps of sequential forward block fft */
st = (8 * NLLAT_FOU * NLVER_FOU * NLLON_FOU/4 * ld (int_nlon/4) *10);       st *= TIME_STEP;
lflop = mflop[pc][i]; pc++; time += st / lflop;
```

Fig. 7.3. Extract of PSTSWM computation description

### 7.1.3. Communication Description




The communication phases of an SPMD program result in the PerPreT communication description. The PerPreT communication library offers several functions to estimate the timing of typical communication patterns. The communication library functions use the PerPreT system description to calculate the time estimates. For communications which involve all processors, the runtime parameter *number of processors* is used by these functions. The currently implemented communication functions are:

**communicate**

The function *communicate(bytes)* calculates the timing of a node to node communication. The parameter *bytes* is used for all functions of the PerPreT communication library. It contains the amount of data which has to be transferred from sender node to receiver node. The result of the function *communicate(bytes)* is the estimated time in seconds for a communication of length *bytes* from sender node to receiver node.

**exchange**

The function *exchange(bytes)* calculates the timing of a node to node data exchange. A pair of nodes send and receive the same amount of data (message length is given by parameter *bytes*) from each other. The result of the function *exchange(bytes)* is the estimated time in seconds for this operation.

**simple_bcast**

The function *simple_bcast(bytes)* calculates the timing of a broadcast operation. One node sends a message of length *bytes* to all other nodes. The function *simple_bcast(bytes)* mimics a simple broadcast scheme which assumes that the sender node sends the messages sequentially to all receiver nodes. The re-

sult of the function *simple_bcast(bytes)* is the estimated time in seconds for this operation.

**simple_collect**

The function *simple_collect(bytes)* calculates the timing of a collect operation. All nodes send a message to one target node. The function *simple_collect(bytes)* mimics a simple collect scheme which assumes that the target node sequentially receives the message from all sender nodes. The result of the function *simple_collect(bytes)* is the estimated time in seconds for this operation.

**tree_bcast**

The function *tree_bcast(bytes)* calculates the timing of a broadcast operation. One node sends a message of length *bytes* to all other nodes. The function *tree_bcast(bytes)* mimics a broadcast scheme which assumes that the sender node starts sending the messages using a binary tree topology. In a binary broadcast tree, each node waits until a message arrives from its father node and then sends the message to its two son nodes. Compared with the *simple_bcast* function, the number of steps for the broadcast operation decreases from *O(nprocs)* to *O(ld(nprocs))*, with *nprocs = number of processors*. The result of the function *tree_bcast(bytes)* is the estimated time in seconds for this operation.

**tree_collect**

The function *tree_collect(bytes)* calculates the timing of a collect operation. All nodes send a message to one target node. The function *tree_collect(bytes)* mimics a collect scheme which assumes that the messages are sent to the target node using a binary tree topology. In a binary collect tree, each node waits until it gets the messages of its two sons and then sends to message to its father. The target node is the root of the binary tree. The result of the function *tree_collect(bytes)* is the estimated time in seconds for this operation.

**butterfly**

The function *butterfly(bytes)* calculates the timing for a data exchange of messages with length bytes between all processors. The function *butterfly(bytes)* assumes a message exchange using a butterfly network topology. The result of the function *butterfly(bytes)* is the estimated time in seconds for this operation.

The PerPreT application communication description subwindow is shown in Fig. 7.4. It can be activated using the *Change Communication* button of the PerPreT main window. The application description files are stored in a subdirectory *application* of the PerPreT main directory, the default filename for the application communication description is *statements.comm*.

The upper part of the window shows the communication description formula. It is expressed in C-program statements. The variables *bytes* (for message lengths in number of bytes), *d_p_size* (for problem size), *d_n_procs* (for number of processors), and *comm_time* (for time in seconds) are of data type *double* and predeclared in the PerPreT environment. The variable *comm_time* is a result variable for the experiment output and initialized with the value zero, the variables *d_p_size* and *d_n_procs* are runtime variables and predefined for every experiment.

To form a valid PerPreT formula to describe the communication behavior of an application, a sequence of C-statements containing the predefined variables and the functions of the PerPreT communication library are used. It is possible to declare additional variables if necessary. At the end, the variable *comm_time* should contain the estimated ex-

ecution time for the communication operations for the experiment.

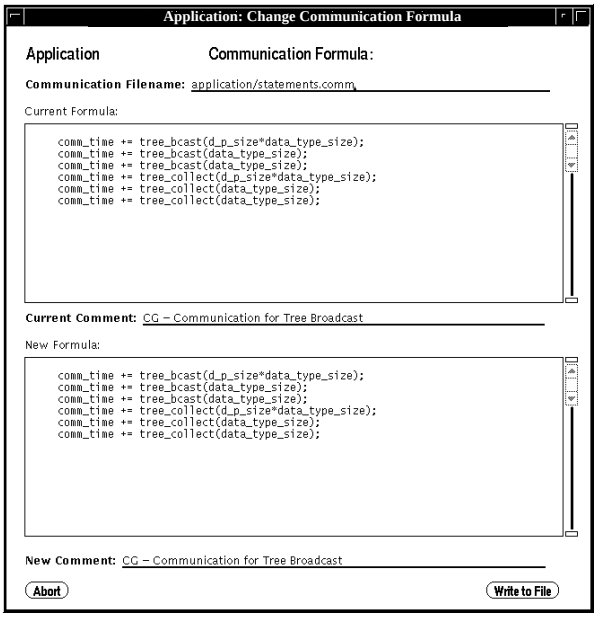



Fig. 7.4. PerPreT application communication description window

The PerPreT application communication descriptions for all examples (*Conjugate Gradient, FFT, Matrix Multiplication, Red-Black Relaxation, PSTSWM*) are automatically loaded with the examples.

The lower part of the subwindow shown in Fig. 7.4. can be used to change existing descriptions to new descriptions and store them using new filenames or to overwrite exiting descriptions. Each description can be equipped with a comment which is also stored in the description file.

## 7.2. Application Examples


The PerPreT software package includes application descriptions for a set of example applications (*Conjugate Gradient, FFT, Matrix Multiplication, Red-Black Relaxation, PSTSWM*). The two versions of the Conjugate Gradient algorithm and the Matrix Multiplication algorithm are explained in sections 6.3.2. and 6.3.1. respectively. The Red-Black Gauss Seidel Relaxation algorithm is explained in section 4.2.2.4 and the different versions of the PSTSWM algorithms are described in section 6.3.3.1. The FFT algorithm is a simple parallel two-dimensional Fast Fourier transformation with butterfly communication pattern.

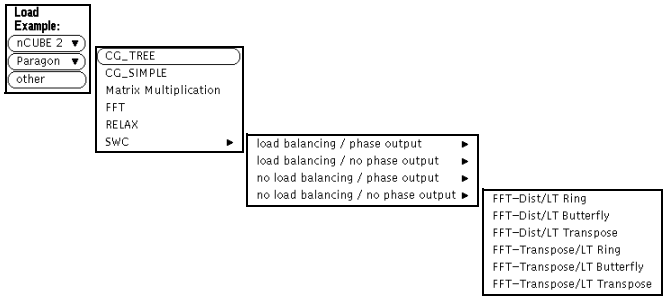



Fig. 7.5. PerPreT application examples

Using the pull down menu from the PerPreT main window the application descriptions for the examples can be loaded for two different multiprocessors, namely nCUBE/2 and INTEL Paragon. The user can easily add system descriptions to run the experiments for other machines (compare Fig. 7.6.). Using the *other* button from the *Load Example* menu the *Select Configuration Files* subwindow is activated. The application descriptions for computation and communication are stored in separate files (default subdirectory *application*). These files can be loaded together with the files containing the system communication and computation descriptions (default subdirectory *system*).

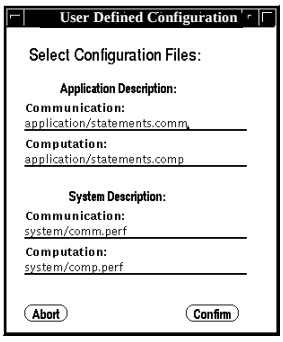
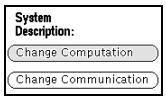



Fig. 7.6. Load user defined examples

## 7.3. System Description

### 7.3.1. Computation Description (without phases)




The PerPreT description of the computation performance of a system is stored in a file (default subdirectory *system*, default filename *comp.perf*). The first line of the file contains a comment, the second line the data type of the problem to be solved, the rest of the lines contain information on the system performance for different problem sizes. If a system's computation performance for an application is independent of the problem size for the application, only one line is needed to express the expected system performance for this application. If a system's performance varies with the problem size, each line following the first two lines can contain a problem size and the corresponding performance. The different problem size lines have to be ordered from smallest to largest.

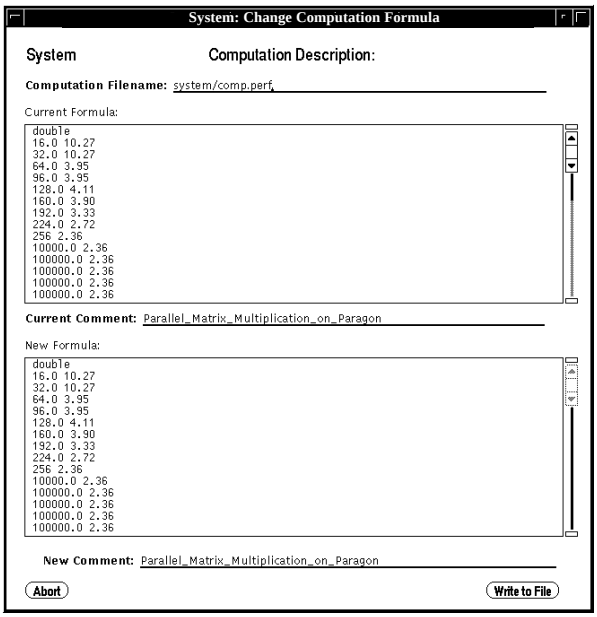
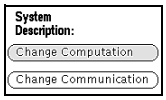



Fig. 7.7. PerPreT system computation description window

The upper part of the system computation description subwindow (compare Fig. 7.7.) which is activated by the *Change Computation* button of the system description in the main window shows the currently loaded computation description. For editing purposes this description is copied to the lower part of the window. Changes to the comment, the data type and the computation performance data can now be mode in the lower part of the window. The resulting computation description is written to a new file if the given filename is changed or the currently loaded file is overwritten if the filename stays unchanged.

### 7.3.2. Computation Description (with phases)




Complex applications like the PSTSWM code might consist of several computation phases with significantly different performance for the target node. If these phases are of different weight for the execution time and if these phases do not scale equivalently with the problem size parameters, building a mean value to describe the system computation performance is not enough to achieve accurate predictions. In that case, it is possible to use a different system computation description format which supports variable computation phase rates (compare section 7.1.2.).

The system computation description window for phase rates is outlined in Fig. 7.8. As before, the upper half of the window contains the currently loaded system computation description file which can be changed through edits in the lower half of the subwindow. The format of the description file is different using phases, the first line contains the data type of the problem to be solved, the second line a input parameter identifier. The rest of the lines contain two performance numbers for a phase and a comment which can be used to identify the phase. The resulting computation description is written to a new file if the given filename is changed or the currently loaded file is overwritten if the filename stays unchanged.

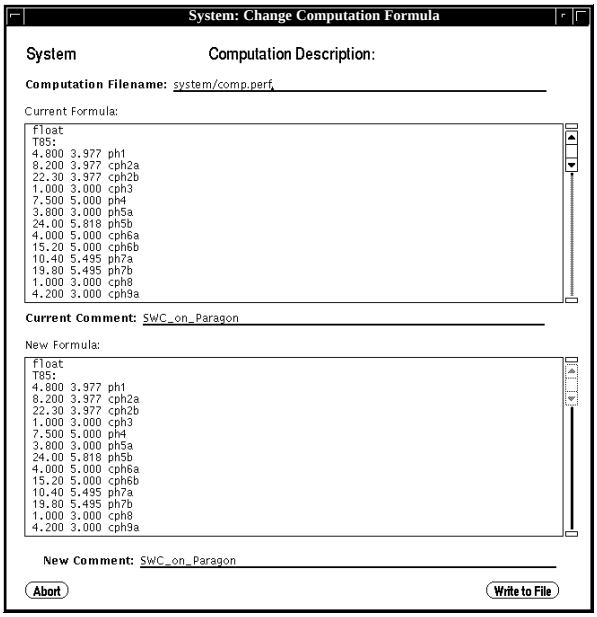
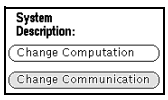



Fig. 7.8. PerPreT system computation description window (phase oriented)

### 7.3.3. Communication Description




The PerPreT description of the communication performance of a system is stored in a file (default subdirectory *system*, default filename *comm.perf*). The first line of the file contains the bandwidth for copy operations from the main memory of the target system to a communication buffer. This bandwidth is given in MBytes/second. For some systems which support direct memory access(DMA) for the communication units, this parameter is ignored. The second line of the file contains the bandwidth for copy operations from communication buffer to the main memory of the target system. This bandwidth is also given in MBytes/second and unnecessary for systems with DMA of the communication

units. The next lines describe the timing for sending messages. Each line consists of six values. The first, third and fifth value are message length parameters, the second value is the setup time for sending a message associated with the preceding message length, the fourth value is the setup time to receive a message with the preceding message length, and the sixth value is the bandwidth to transfer a message with the preceding message length from sender node to receiver node.

Using the *Change Communication* button from the system description menu of the main window, the subwindow shown in Fig. 7.9. is opened.

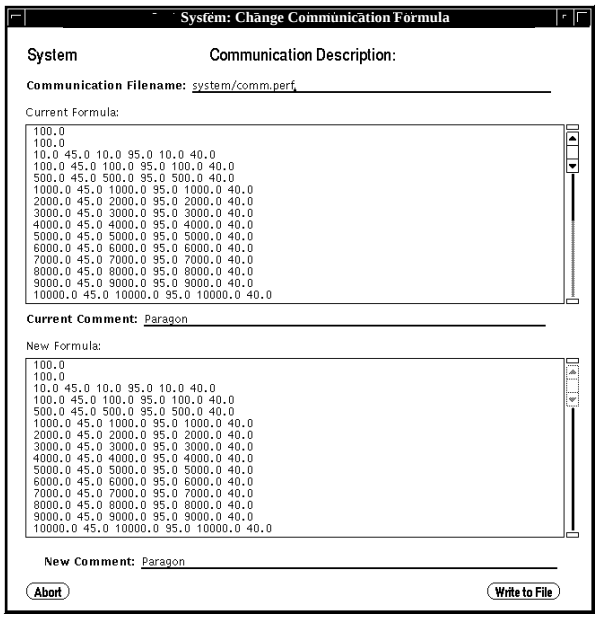
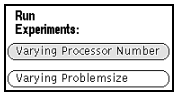



Fig. 7.9. PerPreT system communication description window

As before, the upper half of the window contains the currently loaded system communication description file which can be changed through edits in the lower half of the subwindow.

# 7.4. Experiments

## 7.4.1. Experiments with Varying Processor Number




The parameterized PerPreT application and system description are now used to execute performance prediction experiments. Two kinds of experiments are possible, either the *problem size* is fixed and the *number of processors* are varied or vice versa. The *Run Experiment* subwindow for varying processor number is activated using the *Varying Processor Number* button of the PerPreT main window.

Fig. 7.10. shows the subwindow to control experiments with varying processor numbers. The upper half of this window is used for the scale parameters. The menu *First Processor* inputs the smallest number of processors to be used to execute the experiment, the menu *Last Processor* inputs the largest number of processors. The *Processor Increment* menu inputs a linear processor increment if a positive number is given or an exponential increment if a negative number is given. In the example of Fig. 7.10. experiments for 1, 2, 4, 8, 16, 32, 64, 128, and 256 processors are carried out using problem size 1024 from the *Problem size* menu.

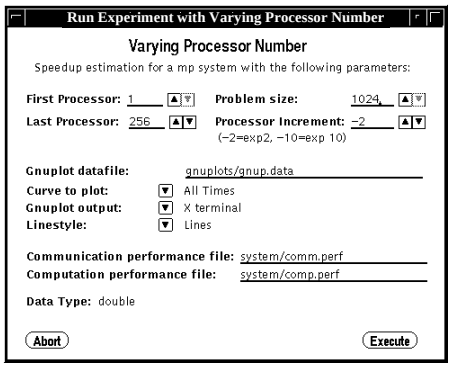



Fig. 7.10. PerPreT experiment window (varying number of processors)

The *Execute* button starts the experiments for varying processor number. For each number of processors the total execution time, the communication time, and the computation time are calculated using the problem size, the application description and the system description. Fig. 7.11. shows the resulting output table. The first column contains the number of processors, the second column the communication time, the third column the

computation time, the fourth column the total execution time. Using the total execution time for a single processor run, the performance measures speedup and efficiency (compare columns five and six) can be calculated for each processor number.

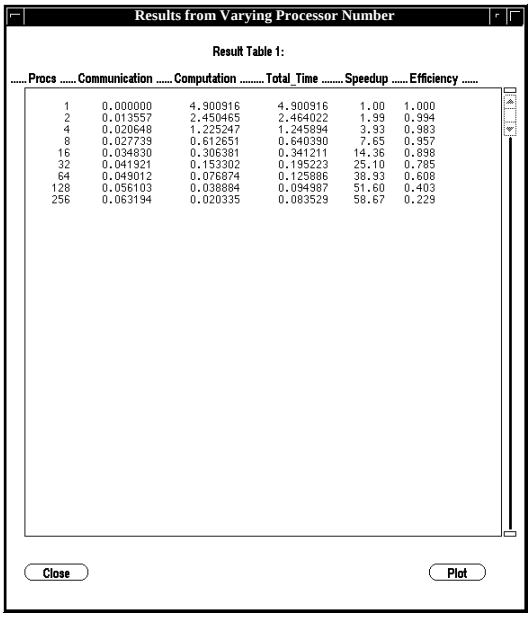
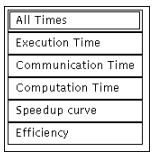



Fig. 7.11. PerPreT experiment table window (varying number of processors)

The middle section of the *Run experiment* subwindow in Fig. 7.10. determines the gnuplot output parameters. Gnuplot is a command-line driven interactive function plotting utility for UNIX, MSDOS, and VMS platforms. The software is copyrighted but freely distributed[1]. It was originally intended as a graphical program which would allow visualization of mathematical functions and data. Gnuplot supports many different types of terminals, plotters, and printers (including many color devices, and pseudo-devices like LaTeX) and is easily extensible to include new devices. Gnuplot handles both curves (2 dimensions) and surfaces (3 dimensions). For 2-d plots, there are many plot styles, including lines, points, lines with points, error bars, and impulses (crude bar graphs). Graphs may be labeled with arbitrary labels and arrows, axes labels, a title, date and time, and a key.

The results of an experiment are written to files. Using a gnuplot command file, the results can be visualized. The default gnuplot command file created by PerPreT is in the subdirectory *gnuplots*, the filename is *gnup.data*. This filename can be changed by the user. The first gnuplot parameter *Curve to plot* contains the following options:

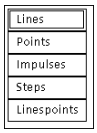
1. Gnuplot is available at ftp.dartmouth.edu, directory /pub/gnuplot




*All times* means that the plot contains the curves for communication, computation and total execution time. The options *Execution Time*, *Communication Time*, and *Computation Time* select the corresponding curve to plot. Instead of the absolute time plots, the relative performance measures *Speedup* and *Efficiency* can also be plotted using the corresponding options. The *Gnuplot* subwindow in Fig. 7.12. is activated through the plot button of the *Result table* subwindow, in this example it shows an *All Times* plot.

Gnuplot offers various output formats. PerPreT users can select one of the following options:


Gnuplot offers various line styles. PerPreT users can select one of the following:



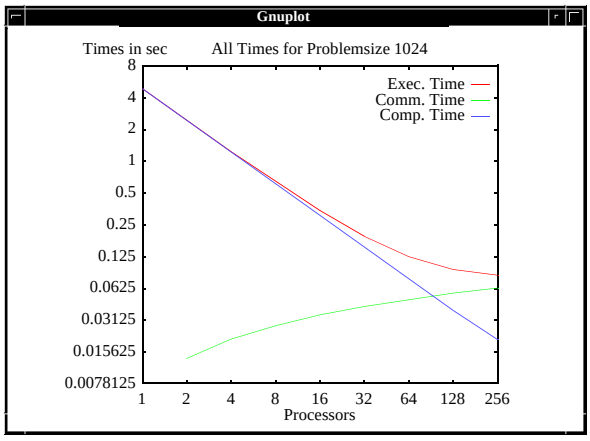




Fig. 7.12. PerPreT gnuplot window (varying number of processors)

Complex applications like PSTSWM which use several parameters to describe the problem size can be controlled through a slightly different *Run Experiment* subwindow displayed in Fig. 7.13. Instead of one number for the problem size, an input data filename is given. The file contains all relevant problem size parameters.

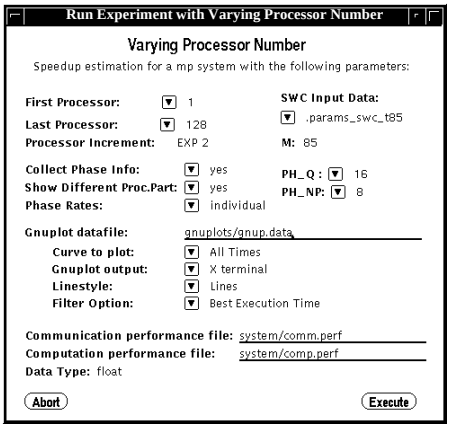



Fig. 7.13. PerPreT-PSTSWM experiment window (varying number of processors)

The resulting output table:

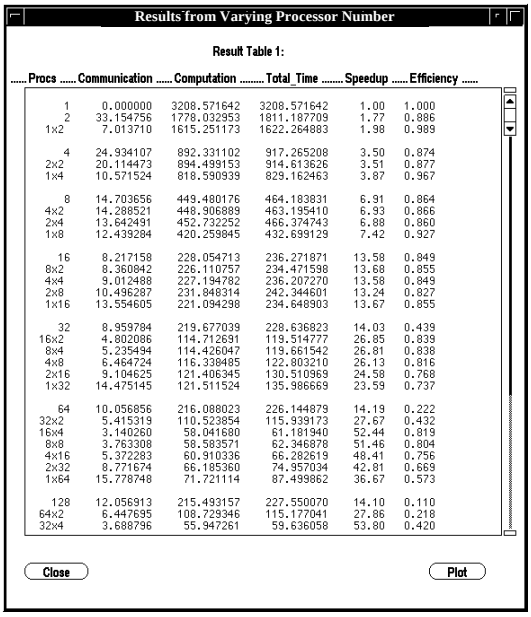
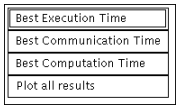



Fig. 7.14. PerPreT-PSTSWM experiment table (varying number of processors)

For applications which use various topologies for one number of processors, the gnuplot output options include one additional filter menu:




*Best Execution Time* plots one curve for the best execution time for each number of processors. It is also possible to filter for *Best Communication Times* or *Best Computation Times*. The *Plot all results* option results in an interval for each of the times and each number of processors (compare Fig. 7.16.), since the various topologies for one number of processors result in different times.

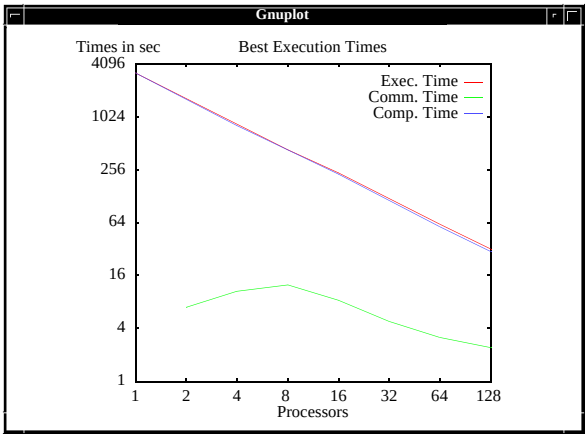



Fig. 7.15. PerPreT-PSTSWM gnuplot window (varying number of processors)

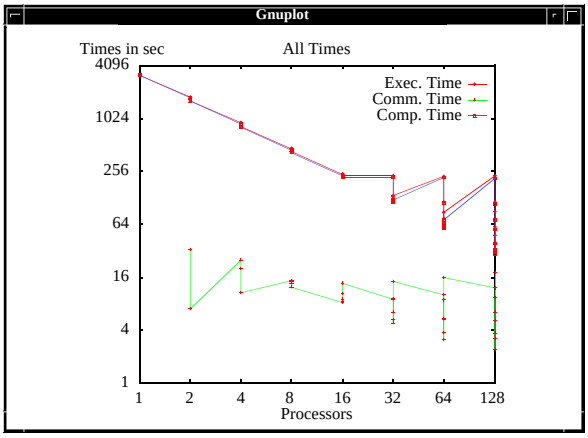
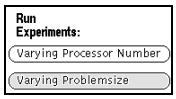



Fig. 7.16. PerPreT-PSTSWM gnuplot window (varying number of processors)

### 7.4.2. Experiments with Varying Problem Size




Besides predicting execution time, speedup, and efficiency for a varying number of processors, it is also important to consider these performance measures for varying problem sizes. Fig. 7.17. shows the subwindow to control experiments with varying problem size. It is activated using the *Varying Problemsize* button of the PerPreT main window. The number of processors can be arbitrarily chosen, but it is fixed during the experiments.

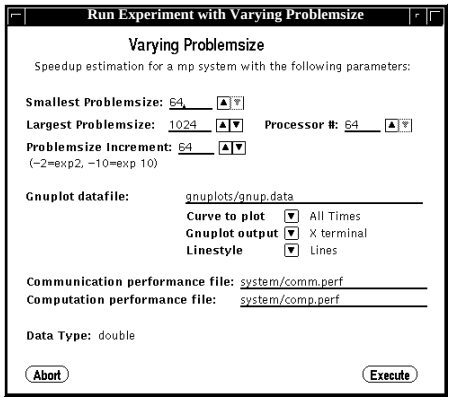



Fig. 7.17. PerPreT experiment window (varying problem size)

The menu *Smallest Problemsize* inputs the smallest value for a problem size to be used to execute the experiment, the menu *Largest Problemsize* inputs the largest problem size value. A series of experiments is conducted with problem sizes varying from the smallest to the largest value using the increment given by the Problemsize Increment. A linear increment is used if a positive number is present, an exponential increment if a negative number is used. In the example of Fig. 7.17. experiments for problem sizes varying from 64 to 1024 using a 64 increment are carried out for a fixed number of 64 processors.

As before the gnuplot output options can be used to determine the performance measure to be plotted (*Curve to plot*), the output format (*Gnuplot output*) and the line style (*Linestyles*).

The default files for the system description are in the subdirectory system. They are *comm.perf* for communication description and *comp.perf* for system description. These files contain the parameters for computation and communication performance. Varying these parameters using different files offers a new way to compare systems. The files can easily be changed by using the *Communication performance file* or *Computation performance file* menu.

The resulting output table for Fig. 7.17. is:

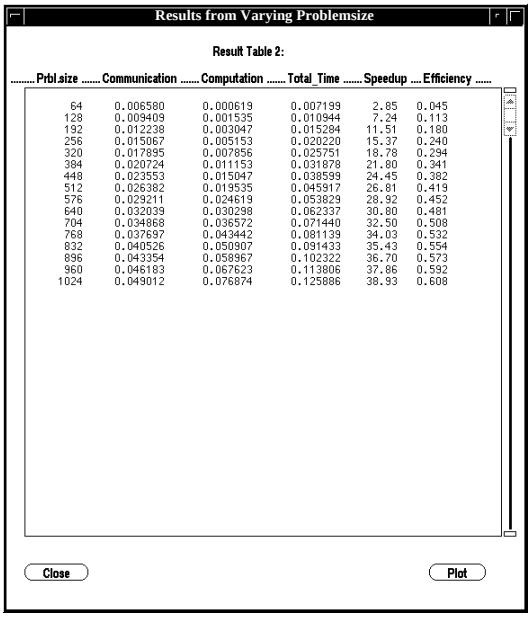



Fig. 7.18. PerPreT experiment table window (varying problem size)

The resulting plot for varying problem size:

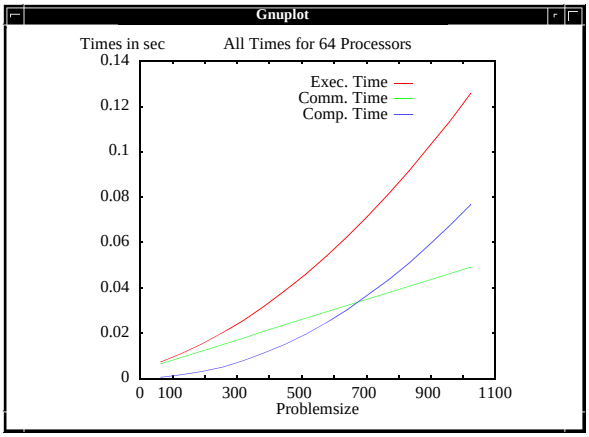
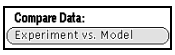



Fig. 7.19. PerPreT experiment gnuplot (varying problem size)

## 7.5. Validation

### 7.5.1. Compare Model and Experimental Data




The PerPreT example applications (*Conjugate Gradient, FFT, Matrix Multiplication, Red-Black Relaxation, PSTSWM*) were also run on nCUBE/2 or INTEL Paragon or on both machines. The execution time, communication time and computation time was measured and the data were collected. These results can be compared against actual model predictions using the *Compare Experiment vs. Model Data* subwindow. It can be activated using corresponding button in the PerPreT main window.

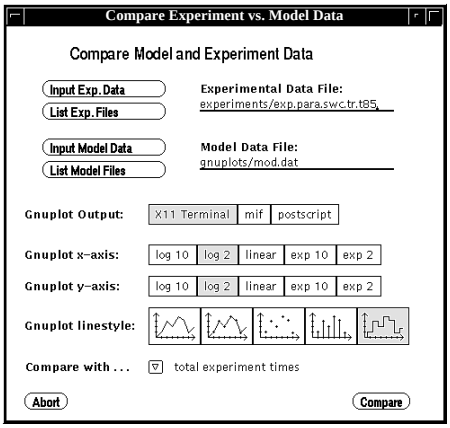



Fig. 7.20. PerPreT validation window

The *compare* button of the validation window starts a comparison of the data given in the *Experimental Data File* against the data given in the *Model Data File*. The output of this operation is another subwindow as shown in Fig. 7.21. It contains the validation table. Each row of the table contains a comparison for one experiment. The first column shows the number of processors used for the experiment, the second column shows the time predicted by PerPreT, the third column shows the time measured on the real system, the fourth column shows the difference between measured and predicted time in seconds, and the last column shows the difference between measured and predicted time in percent.

The middle part of the PerPreT validation window allows to determine output format (postscript, mif, X11) of plots that can be displayed for the comparisons. Besides the line style, it is also possible to select axis with exponential increment.

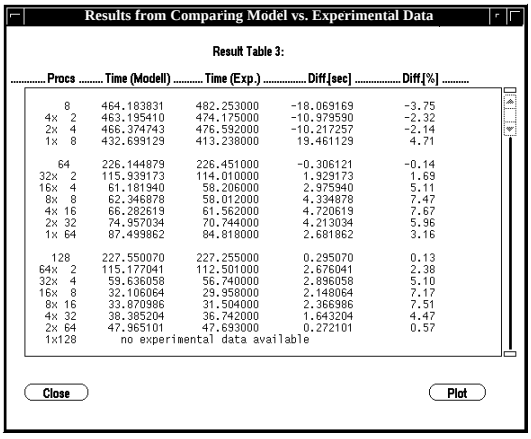
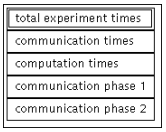



Fig. 7.21. PerPreT validation table

Depending on the experiment and the available timings, total execution times, communication times, or computation times can be compared. The *Compare with* popup menu from the validation window offers these options:




The upper part of the PerPreT validation window (compare Fig. 7.20.) allows to select the files containing the data of the real and model experiments. Using the *Input Exp. Data* button or the *Input Model Data* button opens another subwindow (compare Fig. 7.22.) which allows to input result data to be stored in files. These results can than be used for comparisons. The *List Exp. Files* and *List Model Files* buttons of the validation window activate subwindows which show all existing files containing data from experiments on real systems (compare Fig. 7.23.) or previously run prediction experiments (compare Fig. 7.24.).

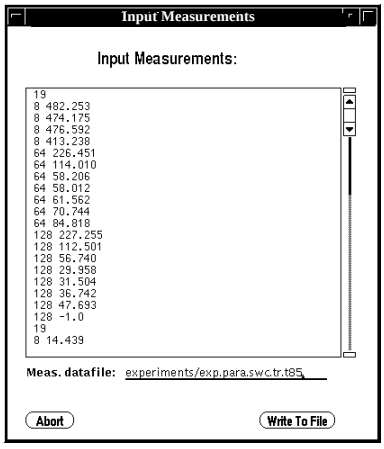



Fig. 7.22. PerPreT input measurement data

By clicking on the corresponding filenames the files are selected for comparison. After

comparison, the results are displayed in the output table of Fig. 7.21. Using the *plot* button of this figure, a gnuplot subwindow with a plot of the results is activated.

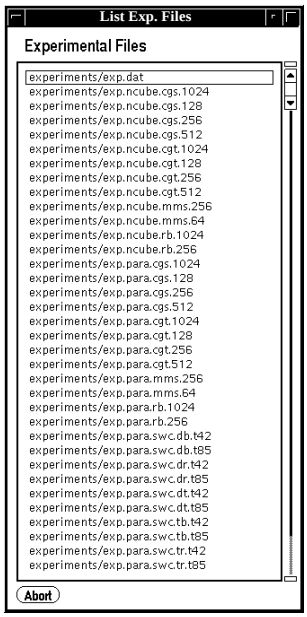



Fig. 7.23. PerPreT experimental data file selecti

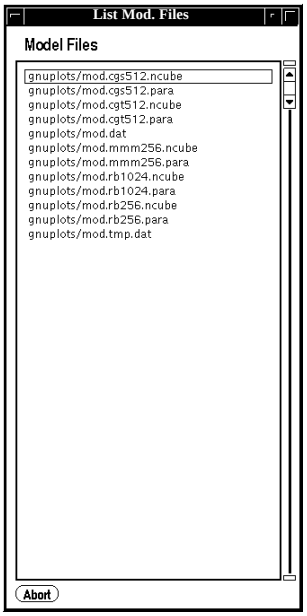
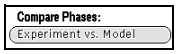



Fig. 7.24. PerPreT model data file selection

## 7.5.2. Compare Model and Experimental Phases




Complex applications like PSTSWM might require a computation description consisting of different phases with different performance characteristics (compare section 6.3.3.1). For the modeling of PSTSWM a graphical interface for validation of computing phases was implemented. This interface can also be adapted for other applications using computation phases for the PerPreT model.

PSTSWM assumes an array configuration of the processors. The *phase validation* subwindow as shown in Fig. 7.25. allows to select the processor whose data are to be validated. The selection can be one of the corner processors or all four of them. The model data from the last experiment are the compared against data from a data base containing all PSTSWM results. The *compare* button of the phase validation window activates the subwindow containing a phase validation table

as shown in Fig. 7.26. If experimental phase data are available they are compared with the model data phase by phase. Sums of the experimental data and the model data are formed and compared at the bottom of the table. The communication model results are also compared with experimental results.

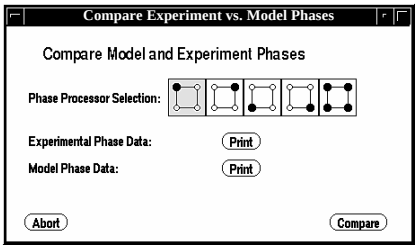
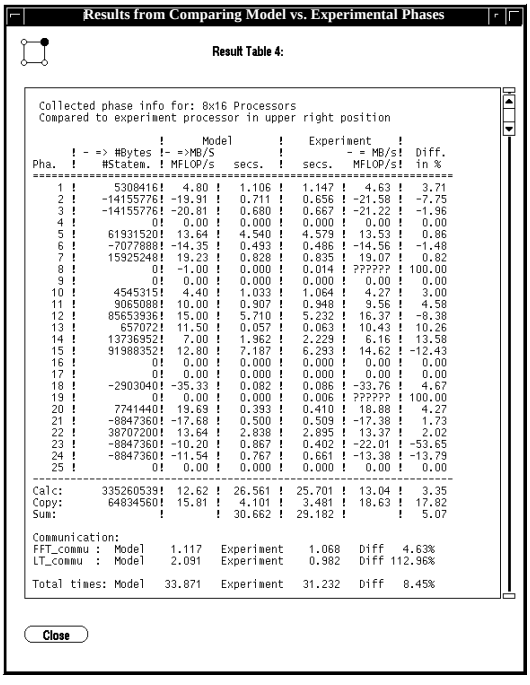



Fig. 7.25. PerPreT phase validation window

The resulting validation table:




| Pha. | Model: #Bytes (=> MB/S) | Model: #Statem. | Model: MFLOP/s | Model: secs. | Experiment: secs. | Experiment: MFLOP/s | Experiment: MB/s | Diff. in % |
|---|---|---|---|---|---|---|---|---|
| 1 | 5308416 | 4.80 | 1.106 | 1.147 | 4.63 | | | 3.71 |
| 2 | -14155776 | -19.91 | 0.711 | 0.656 | -21.58 | | | -7.75 |
| 3 | -14155776 | -20.81 | 0.680 | 0.667 | -21.22 | | | -1.96 |
| 4 | 0 | 0.00 | 0.000 | 0.000 | 0.00 | | | 0.00 |
| 5 | 61931520 | 13.64 | 4.540 | 4.579 | 13.53 | | | 0.86 |
| 6 | -7077888 | -14.35 | 0.493 | 0.486 | -14.56 | | | -1.48 |
| 7 | 15925248 | 19.23 | 0.828 | 0.835 | 19.07 | | | 0.82 |
| 8 | 0 | -1.00 | 0.000 | 0.014 | ?????? | | | 100.00 |
| 9 | 0 | 0.00 | 0.000 | 0.000 | 0.00 | | | 0.00 |
| 10 | 4545315 | 4.40 | 1.033 | 1.064 | 4.27 | | | 3.00 |
| 11 | 9065088 | 10.00 | 0.907 | 0.948 | 9.56 | | | 4.58 |
| 12 | 85653936 | 15.00 | 5.710 | 5.232 | 16.37 | | | -8.38 |
| 13 | 657072 | 11.50 | 0.057 | 0.063 | 10.43 | | | 10.26 |
| 14 | 13736952 | 7.00 | 1.962 | 2.229 | 6.16 | | | 13.58 |
| 15 | 91988352 | 12.80 | 7.187 | 6.293 | 14.62 | | | -12.43 |
| 16 | 0 | 0.00 | 0.000 | 0.000 | 0.00 | | | 0.00 |
| 17 | 0 | 0.00 | 0.000 | 0.000 | 0.00 | | | 0.00 |
| 18 | -2903040 | -35.33 | 0.082 | 0.086 | -33.76 | | | 4.67 |
| 19 | 0 | 0.00 | 0.000 | 0.006 | ?????? | | | 100.00 |
| 20 | 7741440 | 19.69 | 0.393 | 0.410 | 18.88 | | | 4.27 |
| 21 | -8847360 | -17.68 | 0.500 | 0.509 | -17.38 | | | 1.73 |
| 22 | 38707200 | 13.64 | 2.838 | 2.895 | 13.37 | | | 2.02 |
| 23 | -8847360 | -10.20 | 0.867 | 0.402 | -22.01 | | | -53.65 |
| 24 | -8847360 | -11.54 | 0.767 | 0.661 | -13.38 | | | -13.79 |
| 25 | 0 | 0.00 | 0.000 | 0.000 | 0.00 | | | 0.00 |
| Calc: | 335260539 | 12.62 | 26.561 | 25.701 | 13.04 | | | 3.35 |
| Copy: | 64834560 | 15.81 | 4.101 | 3.481 | 18.63 | | | 17.82 |
| Sum: | | | 30.662 | 29.182 | | | | 5.07 |

Communication:
FFT_commu : Model 1.117 Experiment 1.068 Diff 4.63%
LT_commu : Model 2.091 Experiment 0.982 Diff 112.96%

Total times: Model 33.871 Experiment 31.232 Diff 8.45%

Fig. 7.26. PerPreT phase validation table

## 7.6. Summary PerPreT Software

The benefits of using PerPreT for performance prediction experiments have been summarized at the end of the previous chapter. This chapter described how to get and how to use PerPreT. Through the graphical interface it is easy to use and for UNIX/SOLARIS like platforms *makefiles* are available. PerPreT can be downloaded free of charge after contacting the author:

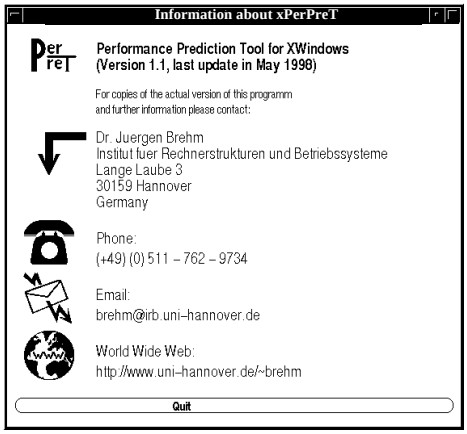
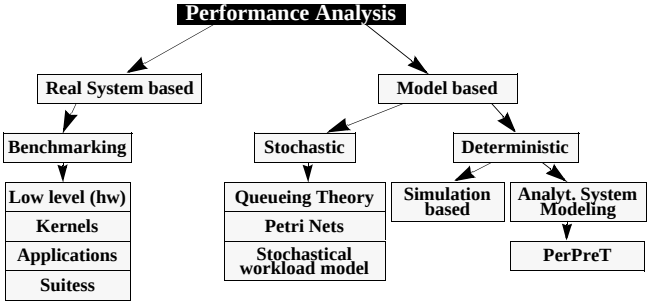



Fig. 7.27. PerPreT contact information




# 8. Open Problems in Performance Analysis

## 8.1. Parallel Applications with Irregular Topologies

The starting point of the development of PerPreT was that the existing modeling techniques are too complex to be applied to massively parallel systems. PerPreT solved this problem for SPMD applications. To describe parallel applications that do not follow the SPMD programming model (e.g. parallel branch and bound problems, parallel adaptive methods) is an open problem. Models with stochastic workloads could be integrated in PerPreT and then be used for an evaluation of non SPMD programs. The results of models with stochastic workloads are statistical by nature, i.e. single runs of a parallel program cannot be modeled or evaluated in detail.

Another problem are parallel applications with irregular communication topologies. In these cases, it is not easy or even impossible to find a formula needed for the PerPreT application communication description.

Simulation based techniques might be able to handle the problems non deterministic program behavior and irregular communication topology, but the complexity (runtime of the simulation programs, size of traces) for massively parallel systems will still be too high.

The question is, whether it makes sense to try to model these kind of applications on massively parallel systems. Even for running the real application on the real system, the results

are often not reproducible (because of communication race conditions).

## 8.2. Workstation Clusters

The existence of fast communication devices (high speed ethernet, SCI-cards) made it possible to use workstation clusters as multiprocessor systems. Communication libraries as PICL, MPI or PVM transform a workstation cluster to a virtual multiprocessor. The modeling of such a system causes new problems compared with the modeling of a tightly coupled system. The nodes of a workstation cluster usually are heterogeneous in the sense of the processors, the memory capabilities, the communication hardware, and sometimes the machines of ones cluster run different operating system. From the modeling point of view it is no longer possible to describe all the nodes of the system by one set of hardware parameters, but each node has its own set of parameters. The interconnection network might also be heterogeneous depending on the locality and communication hardware of the nodes. Last but not least, the behavior of a parallel application running on a workstation cluster depends on the load of the nodes which are multiuser time shared systems. This is the main reason why SPMD programs will usually run with low efficiency on these systems. Instead of data parallel applications a task partitioning strategy can lead to better results. The granularity of parallelism must be coarse because of the relatively high and unpredictable communication latency of a workstation cluster. In summary, the modeling of a workstation cluster used as multiprocessor is an open problem which will be hard to solve.

## 8.3. Performance Analysis of Software

The question of the system under test is discussed in chapter 2.2.4. For the application developer, the system under test consists of compiler, operating system and hardware. The performance of an application on the system under test depends on all these components. The results usually do not show where the losses of efficiency occur. If the real performance is compared with the theoretically available performance, the difference usually tends to be rather large, especially in the case of multiprocessor systems. Different sources for the loss of efficiency can be identified:

- Parallelization strategy:
  Is an optimal parallelization of the tasks implemented, or do load imbalances cause losses of efficiency? Is the mapping of the task onto the system optimal?
- Compiler:
  Is the resulting machine code optimized (especially for vector or super scalar architectures)?
- Operating system:
  Is the potential performance of the interconnection network hardware optimally used?

For the user, it is almost impossible to identify the quantities of efficiency loss for the re-

spective sources. In general a system under test as described in Fig. 2.9.a is used. Compared with the system under test in Fig. 2.9.b the performance will show large differences. The fast growing speed of the systems more than equalizes the inefficiency of the software, but the question “Do I really need a 450 MHz Pentium II microprocessor and 128 MByte of main memory to write a letter using a word processing program?“ should be discussed more seriously in the future.

## 8.4. Embedded Control Systems

Large networks of embedded control systems are of growing importance. More demanding applications, greater functionality of hardware and system software and the usage of distributed subsystems, require an analysis in each stage of development. In [BMS97] an approach to simulate the behavior of embedded control systems is presented. The main advantage of this simulator called ClearSim which is based on execution and event driven methods is the high simulation speed. The components of the system supported by the ClearSim modeling and simulation method are application processes, RTOS (real time operating system), processor, peripheral devices, communication links, and the physical environment. The ClearSim approach is promising, but extensions have to be made to prove its usability for modern microprocessor architectures. Hardware features such as pipelining, caches, and branch prediction have to be included in the processor model. More details on ClearSim can be found in [BMS97].

## 8.5. Summary Open Problems

Due to the complexity of modern hardware, the importance of modeling and simulation techniques will grow. The whole spectrum from real system based techniques such as benchmarking to model based techniques using stochastic or deterministic approaches will be necessary to keep the pace of the hardware development of the last decades.

# 9. References


**[Amd67]** G.M. Amdahl: *Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities, i*n AFIPS Conference Proceedings, vol. 30, pp. 483-485, AFIPS Press, Reston, Va. 1967

**[Bai90]** D. H. Bailey: *FFT's in External or Hierarchical Memory*, Journal of Supercomputing 4 (1), pp. 23-35, March 1990

**[Bai91]** D. Bailey et. al.: *The NAS Parallel Benchmarks*, International Journal of Supercomputer applications, Vol. 5, No. 3, pp. 63-73, 1991

**[Bai94]** D. Bailey et. al.: *The NAS Parallel Benchmarks*, RNR Technical Report RNR-94-007, March 1994

**[Bai95]** D. Bailey et. al.: *The NAS Parallel Benchmarks 2.0*, Report NAS-95-020, December 1995

**[Bar93]** T. Kauranne and S. R. M. Barros: *Scalability estimates of parallel spectral atmospheric models*, in Parallel Supercomputing in Atmospheric Science: Proceedings of the Fifth ECMWF Workshop on Use of Parallel Processors in Meteorology, G.-R. Hoffman and T. Kauranne, eds., World Scientific Publishing Co. Pte. Ltd., Singapore, 1993, pp. 312-328.

**[Bar94]** S. R. M. Barros and T. Kauranne: *On the parallelization of spectral weather models*, Parallel Computing, 20 (1994), pp. 1335-1356.

**[Bea76]** R. Beam, R. Warming: *An Implicite Finite Differnec Algorithm for Hyperbolic Systems in Conservative Law Form*, Journal on Computational Physics, 22:87, 1976

**[Ble91]** G. E. Bleloch et. al.: *A Comparison of Sorting Algorithms for the Connection Machine CM-2*, Proceedings of the Symposium on Parallel Algorithms and Architectures, pp. 3-16, July 1991

**[BMS97]** J. Bruns, C. Müller-Schloer, S. Scherber: *Workstation-based HW/SW-Cosimulation for the Performance Analysis of Embedded Control Systems*, Processdings APS'97, Koblenz, 1997

**[Bra77]** Achim Brandt: *Multi-Level Adaptive Solutions to Boundary-Value Problems*, Mathematics of Computation 31 (138), pp. 333-390, 1977

**[Bre94]]** J. Brehm et. al.: *A Multiprocessor Communication Benchmark, User's Guide and Machine Evaluation*, Abschlußbericht des ESPRIT/OMI Projektes Benchmarking - No. 6271, Institut für Rechnerstrukturen und Betriebssysteme, Universität Hannover 1994

**[Bre95]** J. Brehm et. al: *PerPreT - A Performance Prediction Tool for Massively Parallel Systems*, Lecture Notes in Computer Science 977, Springer Verlag, Heidelberg 1995

**[Bre98]** J. Brehm, P. Worley, M. Madhukar: *Performance Modeling for SPMD Message-Passing Programs*, Concurrency: Practice and Experience, Vol. 10(5), pp. 333-357, John Wiley, April 1998

**[Cas95]** Castelli, G., Ragazzini, G.: *EOS: A Real Time Operating System Adapts to Application Architectures*, IEEE Micro - Special Issue on Embedded Control System, Oct 1995, pp. 41-49

**[Chr86]** Z. Christdis: *Hydrodynamic Mesoscale Modeling of Atmospheric Transport and Pollutant Deposition in the Vicinity of a Lake*, PhD thesis, Atmospheric and Oceanic Science Department, University of Michigan, Ann Arbor, 1986

**[Chr87]** Z. Christdis, V. Sonnad: *Parallel implementation of a Pseudospectral Method on a Loosely Coupled Array of Processors*, IBM Kingston, Technical Report KGN-143, 1987

**[CMS93]** Müller-Schloer, C., Spitzkowsky, J.: *Verhaltensvorhersage für parallele Programme durch ausführungsgesteuerte Simulation*, Proc. Arbeitsplatzrechensysteme (APS) 93, VDE Verlag, 1993, p. 11

**[CMS97]** C. Müller-Schloer: *Hochleistungsrechner*, Skript zur Vorlesung Hochleistungsrechner/Parallelrechner, Institut für Rechnerstrukturen und Betriebssysteme, Lange Laube 3, 30159 Hannover, 1997

**[Cur84]** J. Curry et. al.: *Order and Disorder in Two- and Three Dimensional Bemard Convection*, Journal on Fluid Mechanics, 174:1, 1984

**[Den68]** P. J. Denning: *The Working Set for Program Behavior*, Communications of the ACM, 11(5), pp. 323-333, 1968

**[Den78]** P. J. Denning, J. P. Buzen: *The Operational Analysis of Queueing Network Models*, Computing Surveys, Vol. 10, No. 3, pp. 225-261, September 1978

**[Den90]** D. Dent: *A Modestly Parallel Model*, in The Dawn of Massively Parallel Processing in Meteorology, G.-R. Hoffman and D. K. Maretis, eds., Springer-Verlag, Berlin, 1990, pp. 21-31.

**[DNS95]** T.A.Diep, C. Nelson, J.P. Shen: *Performance Evaluation of the Power PC620 Microarchitecture*, Proc. 22nd. International Symposium on Computer Architecture, Santa Magherita Ligure, Italy, pp.163-174, 1995

**[Don79]** Jack Dongarra et. al.: *LINPACK - User's Guide*, SIAM, Philadelphia, PA, 1979.

**[Don97]** Jack Dongarra: *Performance of Various Computers Using Standard Linear Equations Software*, ORNL report, CS-89-85, June 10th, 1997.

**[Dup87]** M. Dupuis, J. Watts: *Towards Efficient Parallel Computation of Correlated Wave Functions - Implementation of the Two-Electron Integral Transformation on the LCAP Parallel Supercomputer*, IBM Kingston Technical REport KGN-100, 1987

**[Dup88]** M. Dupuis et. al.: *HONDO Version 7.0 Documentation*, IBM Kingston Technical Report KGN-169, and Quantum Chemistry Program Exchange Bulletin 8:2, 1988

**[Far95]** T. Fahringer: *Estimating and optimizing performance for parallel programs*, IEEE Computer, 28 (1995), pp. 47-56.

**[Fos92]** I. Foster, W. Gropp, and R. Stevens: *The parallel scalability of the spectral transform method*, Mon. Wea. Rev., 120 (1992), pp. 835-850.

**[Fos93]** I. T. Foster, and P. H. Worley: P*arallelizing the spectral transform method: A comparison of alternative parallel algorithms*, in Parallel Processing for Scientific Computing, R. F. Sincovec, D. E. Keyes, M. R. Leuze, L. R. Petzold, and D. A. Reed, eds., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1993, pp. 100-107.

**[Fos94]** I. T. Foster, and P. H. Worley: *Parallel algorithms for the spectral transform method*, Tech. Report ORNL/TM-12507, Oak Ridge National Laboratory, Oak Ridge, TN, May 1994.

**[Fos95]** I. T. Foster, B. Toonen, and P. H. Worley: *Performance of parallel computers for spectral atmospheric models*, Tech. Report

ORNL/TM-12986, Oak Ridge National Laboratory, Oak Ridge, TN, April 1995.

**[GEN91]** C.A. Addison, V.S. Getov, A.J.G. Hey, R.W. Hockney and I.C. Wolton: *The GENESIS Distributed-memory Benchmarks, Computer Benchmarks*, J.J. Dongara & W. Gentzsch (Eds), Advances in Parallel Computing, Vol 8, Elsevier Science Publications, BV (North Holland), Amsterdam, The Netherlands, p 257 - 271, 199

**[Gre87]** Leslie Greengard: *The Rapid Evaluation of Potential Fields in Particle Systems*, ACM Press, 11987

**[Gol83]** Gene H. Golub et al.: *Matrix Computation*, North Oxford Academic, Oxford 1983

**[Gol90]** Goldsmith, D.: *Tango Introduction and Tutorial*, Technical Report CSL-TR-90-410, Computer Science Laboratory, Stanford University, Jan. 1990

**[Got88]** T. Gottschalk: *Concurrent Multiple Target Tracking*, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, ACM, New York, 1988

**[Gus88]** John L. Gustafson: *Reevaluating Amdahl's Law*, Communications of the ACM, Vol. 31, No. 5,pp. 532-533, May 1988

**[Gus91]** J. Gustafson, D. Rover, S. Elbert, and M. Carter: *SLALOM: The First Scalable Supercomputer Benchmark*, Parallelogram, February 1991

**[Gus92]** . Gustafson: *The Consequences Of Fixed Time Performance Measurement*, Proceedings of the Twenty-Fifth Hawaii International Conference on system Sciences, Vol. III, Kauai, Hawaii, January 7-10, 1992

**[Hac92]** J. J. Hack and R. Jakob: *Description of a global shallow water model based on the spectral transform method*, NCAR Tech Note NCAR/TN-343+STR, National Center for Atmospheric Research, Boulder, CO, February 1992.

**[Hei83]** P. Heidelberger and K. S. Trivedi: *Analytic queuing models for programs with internal concurrency*, IEEE Trans. Comput., c-32 (1983), pp. 73-82.

**[Har94]** Günter Haring, Harald Wabnig: *PAPS - The Parallel Program Performance Prediction Toolset*, Proceedings of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, LNCS 794, pp. 284-304, Springer Verlag, 1994.

**[Har95]** Günter Haring, Gabriele Kostis: *Workload Modeling for Parallel Processing Systems*, Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 95), IEEE Computer Society Press, pp. 8-12, Durham, NC 1995.

**[Her90]** Herring, C.: *ModSim: A New Object-Oriented Simulation Language*, Proc. SCS Multiconference on Object-Oriented Simulation, Jan. 1990, pp. 55-60

**[HSA91]** P. Hanrahan, D. Salzman, L. Aupperle: *A Rapid Hierarchical Radiosity Algorithm*, Proceedings of SIGGRAPH, 1991

**[Jai91]** R. Jain: *The art of Computer Systems Performance analysis*, John Wiley, New York, 1991.

**[Jad89]** Jade Simulations International Corp.: *Sim++: A Discrete-Event Simulation Language.* Release 2.0, Calgary, Jade Simulations Int., 1989

**[Jam83]** A. Jameson: *Solution of the Euler quations for a Two-Diemsional Transonic Flow by a Multigrid Method*, Applied Mathematics and Computations, 13:327, 1983

**[Kat96]** Ralf Kattner: *Simulationsbasierte Optimierung statischer paralleler Abläufe*, VDI Fortschrittberichte, Reihe 20, Nr. 217, VDI Verlag 1996

**[Ker88]**/ B. W. Kernighan; D. M. Ritchie: *The C Programming Language*, Prentice-Hall, 1988

**[Kel76]** T. W. Keller: *Computer System Models with Passive Resources*, Ph.D. Dissertation, University of Texas at Austin, 1976

**[Kos95]** G. Kostis: *Workload Modeling for Parallel Processing Systems*, Dissertation, University of Vienna, Austria, 1995

**[Laz84]** E. D. Lazowska et. al.: *Quantitative System Performance: Computer System Analysis using Queueing Network Models*, Englewood Cliffs, NJ, Prentice Hall, 1984.

**[Lie86]** G. Lie, E. Clementi: *Molecular Dynamics Simulation of Liquid Water with an ab initio Flexible Water-Water Interaction Potential*, Physical Reviews, A33:2679, 1986

**[Lin85]** David Gelernter: *Parallel Programming in LINDA*, Technical Report 359, Yale University Department of Computer Science, Jan. 1985

**[Lin98]** C. Lindemann: Performance Modelling with Deterministic and Stochastic Petri Nets, John Wiley and Sons, ISBN 0471976466, 1998

**[Lit61]** J. D. C. Little: *A Proof of the Queueing formula $L = \lambda W$*, Operations Research, Vol. 9, pp. 383-387, 1961

**[Mar84]** M. Ajmone Marsan, G. Balbo, G. Conte: *A Class of Gereralized Stochastic Petri Nets for the Performance Analysis of Multiprocessor Systems*, ACM Transactions on Computing Systems, 2, pp.93-122, 1984

**[Mar87]** M. Ajmone Marsan, G. Chiola: *On Petri Nets with Deterministic and Exponentially Distributed Firing Times*, in: Advances in Petri Nets, Lecture Notes in Computer Science 266, pp. 146-161, Springer 1987

**[Mar95]** M. Ajmone Marsan et. al.: *Modeling with Generalized Stochastic Petri Nets*, John Wiley & Sons, 1995

**[Meh94]** Pankaj Mehra et. al.: *A Comparison of Two Model-Based Performance Prediction Techniques for Message Passing Parallel Programs*, Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Nashville, TN, May 1994.

**[Men94]** D. A. Menasce, V. Almeida, L. W. Dowdy: *Capacity Planing and Performance Modeling: From Mainframes to Client-Server Systems*, Prentice Hall, New Jersey, 1994

**[MNS86]** E. A. Mc Nair, C. H. Sauer: *Elements of Practical Performance Modeling*, Prentice Hall, 1986

**[Mol82]** M. K. Molloy: *Performance Analysis Using Stochastic Petri Nets*, IEEE Transaction on Computers, 31, pp. 913-917, 1982

**[MPI94]** MPI Committee, *MPI: a message passing interface standard*, Internat. J. Supercomputer Applications, 8 (1994), pp. 165-416.

**[MPI95]** Message Passing Interface Forum: *MPI - A Message Passing Interface Standard*, Postscript document available through http://www.mcs.anl.gov/mpi/index.html, University of Tennessee at Knoxville, 1995

**[MPI96]** W. Gropp, E. Lusk, N. Doss, and T. Skjellum: *A high performance, portable implementation of the MPI message-passing interface standard*, Tech. Report ANL/MCS-P567-0296, Argonne National Laboratory, February 1996.

**[Nag75]** L. Nagel: *SPICE2 - A Computer Program to Simulate Semiconductor Circuits*, Memorandum ERL-M520, Electronics research Laboratory, College of Engeneering, University of California, Berkeley, 1975

**[NAS93]** D.H. Bailay et. al.: *NAS Parallel Benchmarks Results, Parallel and Distributed Technology*, Vol. 1, IEEE, February 1993.

**[New83]** R. Newton, S. Vincentelli: *Relaxation Based Circuit Simulation*, IEE Transation on ED, ED-30:9:1184, 1984

**[Mis86]** Misra, J.: *Distributed Discrete Event Simulation*, Computing Surveys, vol. 18, no. 1, March 1986, pp.39-65

**[NiL92]** J. Nieh, M. Levoy: *Volume Rendering on Scalable Shared-Memory MIMD Architecturea*, Proceedings of the Boston Workshop on Volume Visualization, October 1992

**[Noo85]** A. Noor, J. Peters: *Model Size Reduciton Techniques for the Analysis of Symmetric Anisotropic Structures*, Eng. Comp., 2:4:285, 1985

**[Ott87]** S. Otto et. al.: *Lattice Gauge Theory Benchmarks*, Caltech report, $C^3$P-405R, 1987

**[Par94]** D. Walker et al.: *Public International Benchmarks for Parallel Computers*, Report of the ParkBench Committee, available on www: http://www.epm.ornl.gov/~walker/report.html.

**[Par96]** M. Parashar and S. Hariri: *Compile time performance prediction of HPF/Fortran 90D*, IEEE Parallel and Distributed Technology, 4 (1996), pp. 57-73.

**[Para92]** M. T. Heath; J. E. Finger: *ParaGraph: A Tool for Visualizing Performance of*

*Parallel Programs*, User Guide, Oak Ridge National Laboratory, Oak Ridge, October 1992

**[Pat92]** Peter G. Harrison, Naresh M. Patel: *Performance Modelling of Communication Networks and Computer Architectures*, Addison-Wesley Publishing Company, 1992

**[PCB94]** D. Kuck et. al: *The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers*, CSRD report available at ftp://csrd.uiuc.edu/pub/CSRD_Reports/ July, 1994

**[Pel93]** R. B. Pelz and W. F. Stern: *A balanced parallel algorithm for spectral global climate models*, in Parallel Processing for Scientific Computing, R. F. Sincovec, D. E. Keyes, M. R. Leuze, L.R. Petzold, and D. A. Reed, eds., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1993, pp. 126-128.

**[Pet62]** C. A. Petri: *Communication with Automatas*, Ph.D thesis, Universitaet Bonn, Germany, 1961

**[Pet81]** J. L. Peterson: *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, N.J. 1981

**[PICL90]** P.H. Worley et. al.: *PICL - A Portable Instrumented Communication Library*, Technical Report, ORNL/TM-1130, Oak Ridge National Laboratory, Oak Ridge, July 1990.

**[Poi93]** L. Pointer: *Performance Evaluation for Cost-Effective Transformations, Report 2*, Technical Report #964, Center for Supercomputer Research and Development (CSRD), Urbana Champaign, 1993, CSRD report available at ftp://csrd.uiuc.edu/pub/CSRD_Reports

**[Pre88]** H. Press et. al.: *Numerical Recipes in C - The Art of Scientific Comptuing*, Cambridge University Press, New York 1988

**[Pul85]** T. Pulliam, J. Steger: Recent Improvements in Efficiency, Accuracy, and Convergence for Implicite Approximate Factorization Algorithms, AIAA-85-0360 23rd Aerospace Science Meeting, January 14-17, Reno, Nevada, 1985

**[PVM94]** G. A. Geist, A. L. Beguelin, J. J. Dongarra, W. Jiang, R. J. Manchek, and V. S. Sunderam, *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing*, MIT Press, Boston, 1994.

**[Res89]** M. Reshef, D, Kessler: P*ractical Implementation of Three-Dimensional Post-Stack Depth Migration*, Geophysics, March 1989

**[RSG93]** E. Rothberg, J. Singh, A. Gupta: *Working Set, Cache Sizes, and Node Granularity Issues for Large Scale Multiprocessors*, Proceedings of teh 20th International Symposium on Computer Architecture, pp. 14-25, May 1993

**[Sar95]** S. R. Sarukkai, P. Mehra: and R. J. Block: *Automated scalability analysis of message-passing parallel programs*, IEEE Parallel and Distributed Technology, 3 (1995), pp. 21-32.

**[Saw87]** K. Swamy, E. Clementi: *Hydration Structure and the Dynamics of B- and Z-DNA in th ePresence of Counterions via Molecular Dynamics Simulations*, IBM Kingston Report KGN-94, 1987

**[Sch93]** T. Schlemeier: *Entwicklung eines Generators für parallele Benchmarkprogramme*, Diplomarbeit IRB, Lange Laube3, 30159 Hannover, 1993

**[Sel80]** J. Sela: S*pectral Modeling at the National Meterological Center*, Monthly Waether Revies, 180:279, 1980

**[Sel82]** J. Sela: *The NMC Spectral Model*, NOAA Technical Report NWS 30, May 1982

**[Ser93]** Maria Calzarossa, Giuseppe Serazzi: *Workload Characterization - A Survey*, Proceedings of the IEEE, 81(8), pp. 1136-1150, August 1993.

**[Sev81]** K. C. Sevcik, I. Mitrani: *The Distribution of Queing Network States at Input and Output Instants*, Journal of the ACM, Vol. 28, No. 2, pp. 358-371, April 1981

**[SGL94]** J. P. Singh, A. Gupta, M. Levoy: *Parallel Visualization Algorithms: Performance and Architectural Implications*, IEEE Computer27 (7), pp. 45-55, July 1994

**[Smi95]** E. Smirni et. al.: *Thread Placement on the Intel Paragon: Modeling and Experimenation*, Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems

(MASCOTS 95), IEEE Computer Society Press, pp. 226-231, Durham, NC 1995.

**[SPEC95]** R. Giladi, N. Ahituv: *SPEC as a Performance Evaluation Measure*, Computer, Vol. 28, Nr. 8, pp. 33-42, August 1995

**[Spi93]** Spitzkowsky, J., Müller-Schloer, C., *Semipessimistic Prediction of Parallel Program Behaviour*, European Simulation Symphosium (ESS) 93, Oct. 1993, pp. 25-28

**[SPL91]** J. P. Singh, W. Weber, A. Gupta: *SPLASH: Stanford Parallel Applications for Shared-Memory*, Computer Architecture News, vol. 20, no. 1, pages 5-44, 1991

**[SPL95]** S. C. Woo, M. Ohara, E. Torrie, J.P. SIngh, A. Gupta: *The SPLASH-2 Programs: Characterization and Methodological Considerations*, Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 24-36, June 1995

**[Sym78]** F.J.W. Symons: *Modeling and Analysis of Communication Protocols Using Numerical Petri Nets*, Ph.D. Dissertation, University of Essex, Great Britain, 1978

**[Tho86]** A. Thomasian, Paul F. Bay: *Analytic Queueing Network Models for Parallel Processing of Task Systems*, IEEE Transaction on Computers, Vol. C-35, No.12, December 1986.

**[Tri82]** K. S. Trivedi, P. Heidelberger: *Queueing Network Models for Parallel Processing with Asynchronous Tasks*, IEEE Transactions on Computers, Vol C-32, pp.15-31, January 1982.

**[Ull87]** A.V. Aho, J. E. Hopcraft, J.D. Ullman: *Data Structures and Algorithms*, Addison Wesley, 1987

**[Ver87]** M. K. Vernon, J. Zarhojan, E. D. Lazowska: *A Comparison of Petri Nets and Queueing Network Models*, Proc. 3rd. International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Paris France, pp.181-192, 1987

**[Wab94a]** H. Wabnig, G. Haring: *Performance Prediction of Parallel Systems with Scalable Specifications - Methodology and Case Study*, Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, pp. 288-289, Nashville, TN, 1994.

**[Wab94b]** H. Wabnig and G. Haring, *PAPS - the parallel program performance prediction toolset*, in 7th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, 1994, pp. 284-304.

**[Wal92]** D. W. Walker, P. H. Worley, and J. B. Drake, *Parallelizing the spectral transform method. Part II*, Concurrency: Practice and Experience, 4 (1992), pp. 509-531.

**[Wei88]** Reinhold Weicker: *Dhrystone: a synthetic systems programming benchmark*, SIGPLAN Notices, Vol. 23 No. 8, 1988

**[Wil92]** D. L. Williamson, J. B. Drake, J. J. Hack, R. Jakob, and P. N. Swarztrauber, *A standard test set for numerical approximations to the shallow water equations on the sphere*, J. Computational Physics, 102 (1992), pp. 211-224.

**[Wor90]** P. H. Worley and M. T. Heath: *Performance characterization research at Oak Ridge National Laboratory*, in Parallel Processing for Scientific Computing, J. Dongarra, P. Messina, D. C. Sorenson, and R. G. Voigt, eds., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990, pp. 431-436.

**[Wor92a]** P. H. Worley: *Phase modeling of parallel scientific code*, in Proceedings of the Scalable High Performance Computing Conference SHPCC-92, J. Saltz and R. Voigt, eds., IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 322-327.

**[Wor92b]** P. H. Worley and J. B. Drake: *Parallelizing the spectral transform method*, Concurrency: Practice and Experience, 4 (1992), pp. 269-291.

**[Wor94]** P. H. Worley, I. T. Foster:*Parallel Spectral Transform Shallow Water Model: A Runtime-Tunable Parallel Benchmark Code*, Proceedings of the SHPCC'94, IEEE Computer Society, pp. 207-214, 1994

**[Wor95]** P. H. Worley and B. Toonen: *A user's guide to PSTSWM*, Tech. Report ORNL/TM-12779, Oak Ridge National Laboratory, Oak Ridge, TN, July 1995.

# 10.Index