The LOOP Approach, a new Method for the Evaluation of Parallel Systems

Jürgen Brehm

Address until July 20th, 1995:

Department of Computer Science
Vanderbilt University
Box 1679, Station B
Nashville, TN, 37235
USA

Address after July 20th, 1995:

Institut für Rechnerstrukturen und Betriebssysteme
Universität Hannover
Lange Laube 3
30159 Hannover
Germany

e-mail: brehm@irb.uni-hannover.de

Abstract

The increasing number of different parallel computers requires a method to compare the performance of such systems. Values like MIPS and MFLOPS often used by computer vendors are normally of secondary value since such information says little about the behavior of real applications running on a certain system. This problem, well known from single processor systems, is even comlicated further in multiprocessor systems. Architectural features such as the arrangement of processors and the performance of interconnection networks significantly influence the overall system performance and cannot be described by these values.

This paper describes a new "LOOP" approach to benchmark message passing multiprocessor systems. The approach uses a description language for parallel workloads, a program generator for deadlock free message passing programs, and an interface to a visualization tool. The package using the approach has been was implemented on three different machines, the MEIKO Transputer system, the nCUBE/2 Hypercube, and the Intel Paragon. After a brief discussion of existing multiprocessor benchmarks the paper describes the new approach in detail and presents results for the MEIKO, nCUBE, and Paragon systems.

Keywords: *Benchmarking, Workload Characterization, Performance Evaluation*

The LOOP Approach, a new Method for the Evaluation of Parallel Systems

Jürgen Brehm

Institut für Rechnerstrukturen und Betriebssysteme
Universität Hannover,Lange Laube 3, 30159 Hannover
Germany
e-mail: brehm@irb.uni-hannover.de

Abstract

The increasing number of different parallel computers requires a method to compare the performance of such systems. Values like MIPS and MFLOPS often used by computer vendors are normally of secondary value since such information says little about the behavior of real applications running on a certain system. This problem, well known from single processor systems, is even comlicated further in multiprocessor systems. Architectural features such as the arrangement of processors and the performance of interconnection networks significantly influence the overall system performance and cannot be described by these values.

This paper describes a new "LOOP" approach to benchmark message passing multiprocessor systems. The approach uses a description language for parallel workloads, a program generator for deadlock free message passing programs, and an interface to a visualization tool. The package using the approach has been was implemented on three different machines, the MEIKO Transputer system, the nCUBE/2 Hypercube, and the Intel Paragon. After a brief discussion of existing multiprocessor benchmarks the paper describes the new approach in detail and presents results for the MEIKO, nCUBE, and Paragon systems.

Keywords: *Benchmarking, Workload Characterization, Performance Evaluation*

1. Introduction

Benchmarking computer systems is an important issue for both, computer architects and users. The purpose of benchmarking reaches from identifying architectural bottlenecks to determine purchase decisions. The number of multiprocessor architectures has substantially increased in the last years, so did the efforts to evaluate these machines. Unfortunately, none of the existing approaches is feasible for a wide range of existing multiprocessors or is adaptable to user defined workloads. This paper 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.

The rest of the paper is organized as follows. In section 2, a brief survey of standard benchmark approaches is given. Section 3 describes the approach of benchmarking parallel systems in general and section 4 explains the LOOP approach in detail. A set of parameterized parallel workloads that are used as standard benchmarks is described in section 5. Results for these workloads on the MEIKO,

nCUBE, and Paragon systems are provided in section 6. The final section with an outlook to future work concludes this paper.

2. Standard Approaches

This section describes several well known benchmark tests. An overview of existing benchmarks for single processor computers provides a useful perspective. Most benchmark programs consist of synthetic programs or real applications (uni- and multiprocessor applications).

Often, small benchmark programs are used to get a first impression of a system's performance. In general these programs are easy to port to another machine, but they typically measure only a single aspect of the machine, for example, the integer performance.

Dhrystone:
A well known benchmark program is Dhrystone. This program was originally written in ADA by R. Weicker and was later implemented in C by R. Richardson. Dhrystone evaluates the performance of the CPU and the compiler. This synthetic benchmark program /Wei91/ generates a representative workload which is typical for single processor machines. By examining a large amount of program code and analyzing the type of operations, Weicker tries to recreate typical program behavior. The resulting benchmark is one where the amount of each operation type mimics that of the examined programs.

The performance of the CPU and the optimization features of the compiler are tested with this benchmark. It does not test floating point arithmetic nor does it stress the operating system. Due to its small size, it fits in almost every cache and may exaggerate the cache's effectiveness. Much work has been done to keep compilers from doing special optimizations specifically for Dhrystone.

Whetstone:
In order to test the performance of floating point operations, the Whetstone benchmark test was designed. Like the Dhrystone it is a synthetic benchmark where the procedures are designed to generate typical workload behavior rather than to execute a specific task.

SSBA:
SSBA is a benchmark suite assembled by the French UNIX user group (AFUU). It has especially been designed to test the performance of UNIX based systems. A recent version tests multiprocessor systems.

Linpack:
Another approach to benchmarking is to examine the performance of real applications. Compilers, databases and other programs are often used to simulate the overall system performance of a general purpose machine. Two famous suites of such programs are Linpack and SPEC.

The Linpack benchmark is widely used in scientific environments. It consists of several procedures which calculate problems such as the solution of large systems of linear equations, matrix multiplications, and dotproducts.

SPEC:
The SPEC benchmark suite consists of compilers, databases and other application programs that are typically found on general purpose machines /Spec91/. They are a typical mixture of floating point intensive, integer arithmetic intensive and memory bound applications. Large production code packages such as **SPICE** are used to create similar effects. In the newer versions of the SPEC benchmark suite some synthetical benchmarks can also be found.

SLALOM:
SLALOM is used to test parallel systems. The most unique feature in the SLALOM benchmark test is that instead of having a fixed problem-size and measuring its execution time, the execution time is fixed and the problem size is

chosen such that the benchmark completes in the allotted time. In /Sla90/ the algorithm, which calculates how a coupled set of diffuse surfaces emits and absorbs radiation, is introduced and it is shown how the problem size can be scaled.

SPLASH:
SPLASH is a set of several typical applications which are often used on parallel systems. These applications are well documented and thus can be used for benchmarking a system. This set of applications is described in /Sin91/.

3. Benchmarking Parallel Systems

3.1. The System Under Test

There are several approaches to the evaluation of systems, depending on the desired level of abstraction. Although there is a continuum of possible views, two examples of different abstraction levels are illustrated in Figure 1.

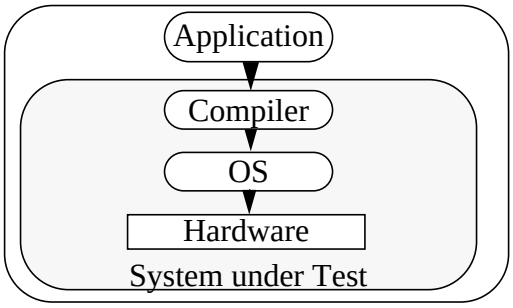
For the programmer of a high level application the system under test (SUT) includes several components such as the compiler, the operating system, and the underlying hardware. In a multiprocessor environment the interconnection network is also part of the SUT. The programmer might be interested in several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

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

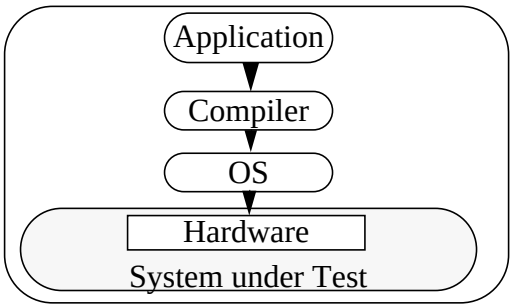
A different view of the same computer system is shown in Figure 1b. A hardware developer is normally less interested in the performance of the compiler or the operating system.Thus, the benchmarks of most use are specifically designed for the evaluation of certain components of interest. The features of most interest to the hardware developer include:

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

The system under test considered in this paper is the one of the application programmer, Figure 1a. 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 may be found.



a. SUT for application programmer



b. SUT for hardware developer

Figure 1: SUT depending on the level of abstraction

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, so 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.

3.2. Using Parameters

All of the existing benchmark tests described in section 2 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. The *LOOP* Method

In the LOOP approach /BBS94a,Schl93/, 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 Section 5 can be used to get a first impression of the system without the need to fully implement a user specific application.

In Figure 2, 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.

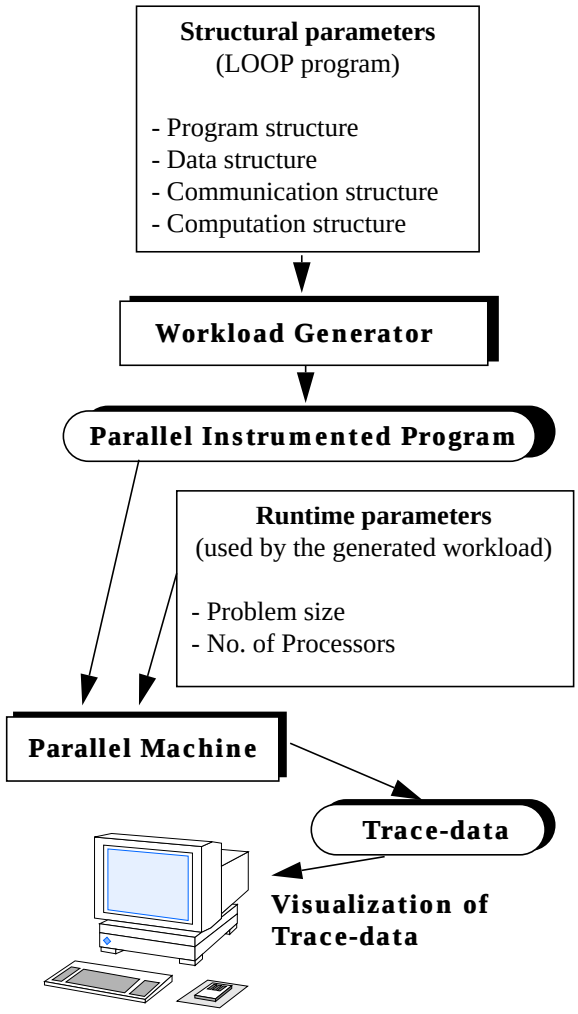


Figure 2: Generating Workloads using the *LOOP-method*

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

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 sections 5 and 6.

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 of 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 /Schl93/. *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.

4.1. 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.

4.1.1. 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.

4.1.1.1. Programs and Data Structures

Considering problems normally solved on massively parallel systems, numerical applications are arguable 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 Figure 3, a LOOP program abstraction of a simple parallel matrix multiplication is shown. The use of the construct \LOOP and the decla-

ration 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.

```
int main(void){
    int nodes, me, host, prob_size, amount;
    \OPEN0(&nodes, &me, &host, &prob_size,
                TRACE_BUF_SIZE);
    /* Declarations: */
    \Mat(double) Mat1, Mat2, Mat3;
    /*random init */
    \INIT_ARRAY(Mat1);
    \INIT_ARRAY(Mat2);
    \INIT_ARRAY(Mat3, 0);
    amount=sizeof(double)*prob_size*
          prob_size/nodes;
    \LOOP i3 {
        \LOOP i2 {
            \LOOP i1 {
            Mat3[i3][i2] += Mat1[i3][i1]*Ma-
t2[i1][i2];
            }
        \COMMUNICATE(amount,1,1);
    }}
    \CLOSE0();
}
```

Figure 3: Complete program for a parallel Matrix-Matrix multiplication

The construct

\LOOP [iterator]

in Figure 3 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 code given in Figure 3 is the complete input for the generator. The result of the generator is a parallel instrumented program, PIP (see Figure 2). 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.

4.1.1.2. 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.

4.1.1.3. 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.

4.1.1.4. 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, s simple parallel matrix multiplication is considered. First, the basic algorithm is described in Figure 4.

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

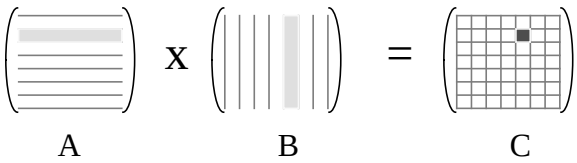


Figure 4: Structure of a parallel matrix-matrix multiplication

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 Figure 3, the communication can be modelled 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 Figure 3 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.

4.1.1.5. Information Exchange

Similar to the \COMMUNICATE construct, the \EXCAHANGE 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. 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 we consider the principles of a parallel red-black relaxation algorithm.

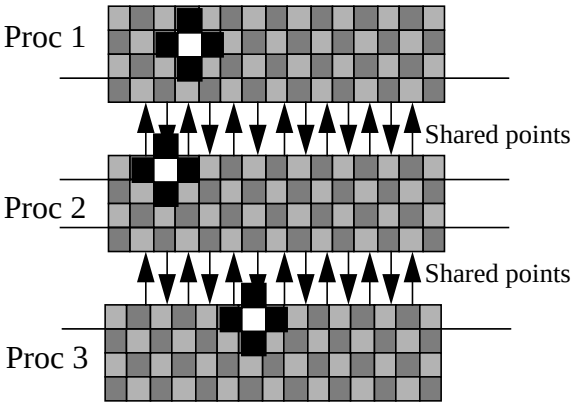


Figure 5: Parallel red-black relaxation

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 Figure 5. 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,
    psize /* problemsize */, nodes /* nodes */;
\MAT (TYPE: MY_TYPE, S1: psize_node) Mat ;
\OPEN0(&nodes, &me, &host, &psize, T_BUF);
int psize_node = (int) ( psize / nodes ) ;
\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) ;
}
\CLOSE0();
return EXIT_SUCCESS;
}
```

Figure 6: LOOP program for parallel red_black relaxation

A *LOOP* program which is capable to model this algorithm is shown in Figure 6. 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 Figure 6 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.

4.1.1.6. 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.

4.1.2. Runtime Parameters

In the preceding, the structural parameters needed to generate a certain type of workload are 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 efficient 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. 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 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 paper it is not possible to describe all LOOP statements, a complete description can be found in /BBS94a/.

5. 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, 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.

5.1. The 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

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

reference information for a first glance comparison between different machines. As shown in the annotated version of the space time diagram in Figure 8, 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 Figure 7 the LOOP source code for the *Fingerprint* workload is given.

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

Figure 7: *LOOP* source code for *Fingerprint*

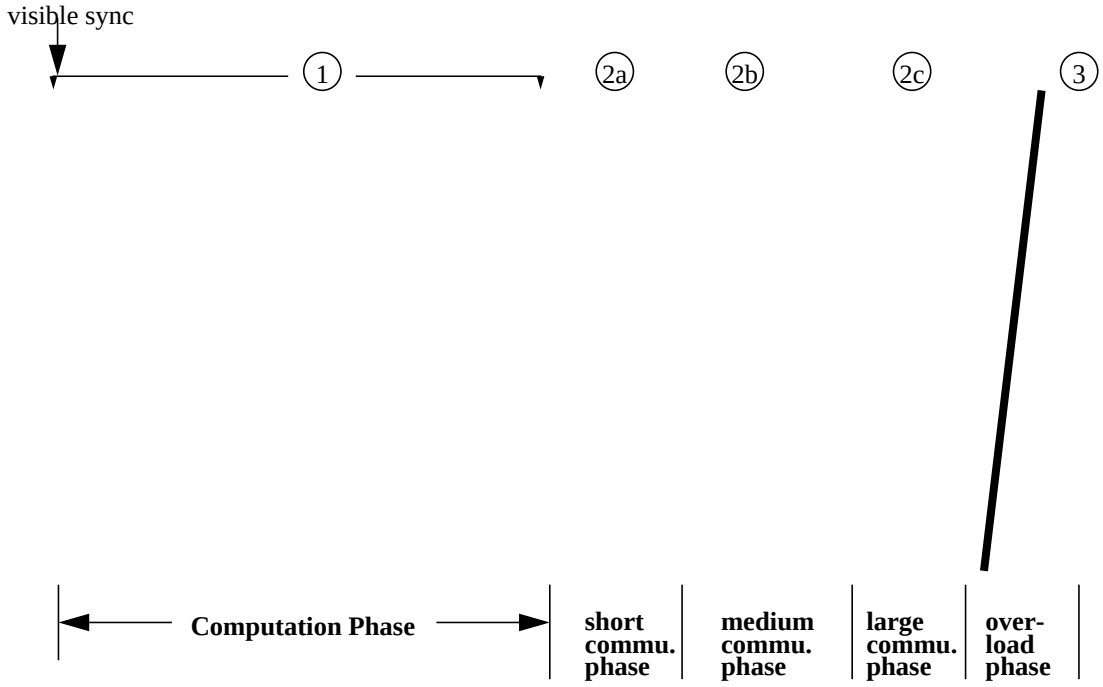


Figure 8: Space time diagram for a typical 16 processor fingerprint execution

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.

5.2. 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).

1. Using a virtual ring topology

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 Figure 6.

6. Results

For the comparison of a MEIKO (a 64 node T800 based multiprocessor), an nCUBE/2 (a 128 node hypercube connected multiprocessor), and an Intel Paragon (a 512 node i860 based mesh connected multiprocessor), 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 /BBS94b/. The timings are given in Table 1. On the nCUBE/2 and the Paragon the workloads are also executed with 64 and 128 nodes. The results are shown in Table 3. The results tables are organized as follows. First, the name for the LOOP workload program is given. The first parameter is the numbers of allocated processors, the second parameter is the problem size, and the third parameter is the number of iterations (if applicable).

6.1. Execution Times

Table 1 shows the results of the LOOP benchmarks executed on the different systems.

	Runtime in seconds		
	MEIKO	nCUBE	Paragon
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
red_black 16/1024/5	6.885	4.439	0.567
red_black 32/1024/5	3.621	2.210	0.284

Table 1: Execution times for the LOOP benchmarks

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.]

To see the results from a relative viewpoint, the times are converted to “paragon seconds“ (see Table 2). 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.

	Slowdown		
	MEIKO	nCUBE	Paragon
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
mmm_s 16/256	7.956	5.074	1.0
mmm_s 32/256	8.800	5.246	1.0
mmm_a 16/256	8.085	4.986	1.0
mmm_a 32/256	8.941	5.045	1.0
red_black 16/1024/5	12.121	7.743	1.0
red_black 32/1024/5	12.750	7.782	1.0

Table 2: Slowdown against Paragon

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.

	Runtime in secs.		Slowdown
	nCUBE	Paragon	nCUBE
fp 64/100	0.181	0.072	2.613
fp 128/100	0.380	0.145	2.621
cg 64/1024/8	1.129	0.189	5.974
cg 128/1024/8	0.829	0.145	5.717
mmm_s 64/256	2.968	0.484	6.132
mmm_s 128/256	2.598	0.327	7.945
mmm_a 64/256	2.816	0.484	5.818
mmm_a 128/256	2.468	0.328	7.524
red_bl. 64/1024/5	1.078	0.143	7.531
red_bl. 128/1024/5	0.544	0.074	7.351

Table 3: Runtimes for the LOOP benchmarks on 64 and 128 nodes

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).

6.2. 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]. An example for the standard result sheets for a conjugate gradient LOOP workload is provided as Appendix.

The PICL tracefiles contain all the information on communication and computation events. Paragraph offers a wide variety of displays to visualize these events. Thus, the user can go into as much detail as desired.

7. Conclusions and Future Work

A primary goal of the LOOP approach is to provide the user with a set of parallel workloads which can be used to compare various aspects of different systems quickly. A second goal is to offer a convenient way to implement user defined workloads on parallel systems. There are a variety of LOOP statements (not all of which are described in this paper) for typical communication and computation loads. LOOP programs can be complemented with C statements. Once the description of a workload is complete, the remaining steps are automatic. The user only has to specify runtime parameters (problemsize, number of processors, and number of iterations). The program is then executed and a tracefile is generated. ParaGraph can be used to visualize the trace data.

Regarding future work on the LOOP approach, two extensions are planned:

1) The use of LOOP programs for automatic workload characterization, and
2) the use of the LOOP language for fast prototyping of message passing programs.

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)

8. References

/BBS94a/ J. Brehm et.al.:
A Multiprocessor Benchmark, User's guide and refernce manual, ESPRIT III technical report, available via ftp.irb.uni-hannover.de

/BBS94b/ J. Brehm et.al.:
A Multiprocessor Benchmark, Appendix D, machine evaluations, ESPRIT III technical report, available via ftp.irb.uni-hannover.de

/Gol83/ Gene H. Golub et al.:
Matrix Computation, North Oxford Academic, Oxford 1983

/Jain91/ R. Jain:
The art of computer systems performance analysis: techniques for experimental design, measurement, simulation and modeling,
John Wiley&Sons, New York 1991

/Ker88/ B. W. Kernighan; D. M. Ritchie:
The C Programming Language
Prentice-Hall, 1988

/Kil94/ U. Killermann:
Implementierung der parallelen Programmierumgebung PPRC auf nCube
Diplomarbeit IRB, Hannover 1994

/PARA91/ M. T. Heath; J. E. Finger:
ParaGraph: A Tool for Visualizing Performance of Parallel Programs, IEEE Software, 8(5), September 1991, pp. 29-39

/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

/PICL90/ G.A. Geist; M.T. Heath; B.W. Peyton; P.H. Worley:
PICL - A Portable Instrumented Communication Library, Technical Report, Oak Ridge National Laboratory, Oak Ridge, July 1990

/PICL90a/G.A. Geist; M.T. Heath; B.W. Peyton; P.H. Worley:
PICL - A Portable Instrumented Communication Library, C Reference Manual, Oak Ridge National Laboratory, Oak Ridge, July 1990

/Pre88/ H. Press et. al.:
Numerical Recipes in C - The Art of Scientific Comptuing,
Cambridge University Press, New York 1988

/Schl93/ T. Schlemeier:
Entwicklung eines Generators für parallele Benchmarkprogramme
Diplomarbeit IRB, Hannover 1993

/Sin91/ Singh; Weger; Gupta:
SPLASH: Stanford Parallel Applications for Shared-Memory
Stanford University, CA 94305, Technical Report CSL-TR-91-469

/Sla90/ J. Gustafson et. al.:
SLALOM: The First Scalable Supercomputer Benchmark
Supercomputing Review, November 1990, pp.56-61

/Spec91/ SunTech Journal:
SPECulations (Defining the SPEC Benchmark)
January 1991

/Wei91/ Reinhold Weicker:
Benchmarking: Status, Kritik, Aussichten, Proceedings zur 6. GI/ITG Fachtagung Messung Modellierung und Bewertung von Rechensystemen,
p. 259-277, Springer-Verlag, Berlin 1991

Appendix: Example for the standard result sheets for a conjugate gradient workload

Hardware	**Evaluated Problem**
nCUBE/2	Fingerprint(1,500,1000)
Nodes: 128 at 20MHz Network: Hypercube	No. nodes: 16 Problemsize: 100 Iterations: 100

Main Results

Execution time:	0,0728 sec
Percent Processors Busy:	51.32 %
Percent Processors Overhead:	43.37 %
Percent Processors Idle:	5.32 %
Avg Time Send (usec):	4378
Avg Time Rcvd (usec:	4378

Appendix: Example for the standard result sheets for a conjugate gradient workload

Statistics evaluated by ParaGraph (Scalingfactor for times: 100):

Appendix: Example for the standard result sheets for a conjugate gradient workload