PerPreT - A Performance Prediction Tool for Massively Parallel Systems

Jürgen Brehm*
Universität Hannover, Institut für Rechnerstrukturen und Betriebssysteme
Lange Laube 3, 30159 Hannover
email: brehm@irb.uni-hannover.de

Manish Madhukar, Evgenia Smirni, Larry Dowdy**
Vanderbilt University, Department of Computer Science
Box 1679 B, Nashville, TN 37235
email: {manishm,esmirni,dowdy}@vuse.vanderbilt.edu

Abstract

Today's massively parallel machines are typically message passing systems consisting of hundreds or thousands of processors. Implementing parallel applications efficiently in this environment is a challenging task. The Performance Prediction Tool (PerPreT) presented in this paper is useful for system designers and application developers. The system designers can use the tool to examine the effects of changes of architectural parameters on parallel applications (e.g., reduction of setup time, increase of link bandwidth, faster execution units). Application developers are interested in a fast evaluation of different parallelization strategies of their codes. PerPreT uses a relatively simple analytical model to predict speedup, execution time, computation time, and communication time for a parametrized application. Especially for large processor numbers, PerPreT's analytical model outperforms traditional models (e.g., Markov based approaches such as queueing and Petri net models). The applications are modelled through parameterized formulae for communication and computation. Parameters include the problem size and the number of processors used to execute the program. The target systems are described by architectural parameters (e.g., setup times for communication, link bandwidth, and sustained computing performance per node).

Keywords: *workload characterization, workload modeling, performance evaluation, performance prediction*

1. Introduction

Advances in microprocessor technology and interconnection networks have made it possible to construct parallel systems with a large number of processors (e.g., INTEL Paragon, nCUBE Hypercubes, CM-5, multitransputer systems, workstation networks running PVM). Unfortunately, the application programs developed for conventional sequential systems or for pipelined supercomputers do not automatically run on these systems. There are few good compilers for efficient automatic parallelization of programs. There are also few useful tools to support the development of parallel programs. Before writing a program, the developer must identify a parallelization strategy. In many cases there are different options on how to distribute data and tasks onto the processors. Because it is too time consuming and expensive to implement several alternatives, it would be helpful for the programmer to be able to accurately predict the performance trade-offs of alternative strategies without resorting to implementation and measurement.

In the past, several approaches for the modeling of parallel systems using formal methods such as Petri nets or Markov models have been presented [Tho86], [Tri82], [Laz84], [Wab94], [Har94,95]. These approaches result in accurate models for the execution of tasks on parallel systems. Unfortunately, it is difficult to apply these methods to massively parallel systems for several reasons:

* While on leave at Vanderbilt University on a postdoc fellowship from the Alexander von Humboldt foundation.

**This work was partially supported by sub-contract 19X-SL131V from the Oak Ridge National Laboratory managed by Martin Marietta Energy Systems, Inc. for the U.S. Department of Energy under contract no. DE-AC05-84OR21400.

- The graphical representation of systems with hundreds or thousands of processors that would be needed for these approaches is too complex.
- The application description and the mapping of the applications onto the processors is too detailed.
- The resulting systems of equations from the Markov or Petri net models are too large to be solved efficiently.

PerPreT takes advantage of the fact that typical applications for massively parallel systems use the single program multiple data (SPMD) programming model. In this paper we show that SPMD programs allow simplifications. Abstraction of the application and system to be modeled can be made without a significant loss of accuracy for predicted values of speedup, communication time, computation time, and execution time. These simplifications make it possible to consider architectures with thousands of processors. The resulting analytical model of message passing architectures can be evaluated quickly and is the main advantage of PerPreT compared to Markov or Petri net model based approaches.

One disadvantage of PerPreT is the lack of modeling low level hardware features such as network contention. This requires a more detailed description of the hardware and the operating system. These features can be modeled using Petri nets or queueing network models. Another disadvantage is that non SPMD applications cannot be modeled using PerPreT. In these cases, the conventional approaches are more appropriate.

In Figure 1, the high-level modules of PerPreT (i.e., application description, system description, communication library, analytical model) are outlined. PerPreT uses parameterized system and application descriptions. The problem size for an application and the number of processors used to execute the SPMD program are free parameters. Therefore, PerPreT models a variety of alternative systems and applications. The system and application descriptions are kept independent of each other. In the case of complex node processor architectures, the sustained MFLOP/s (millions of floating point operations per second) rate is the only system variable that sometimes changes with different applications. PerPreT uses the system description parameters in Figure 1 and a communication library to model the communication and computation behavior of the target architecture. An SPMD application is reduced to formulae for computation (number of arithmetic statements) and communication (calls to the communication library).

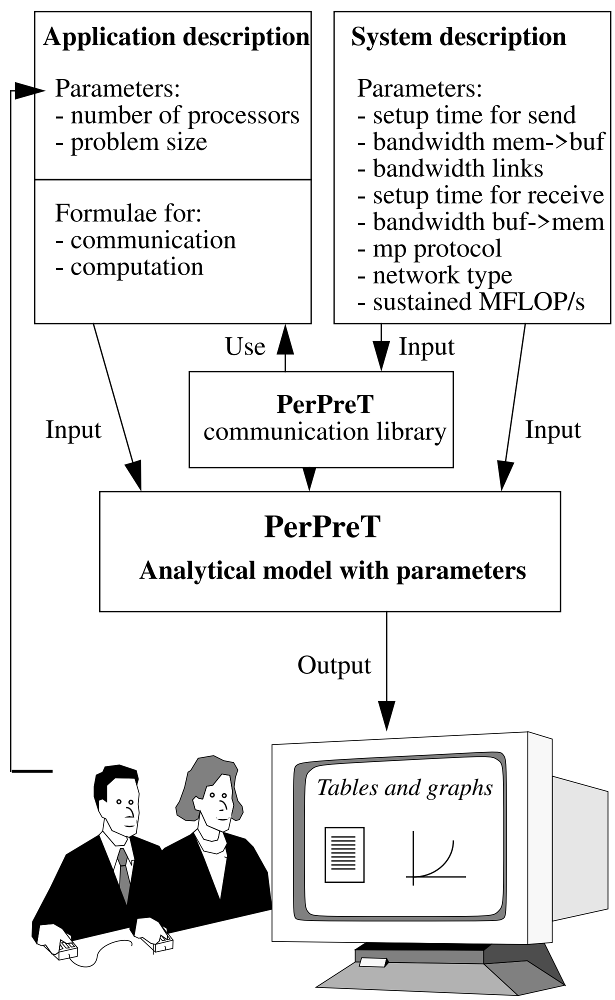


Fig. 1: The Modules of PerPreT

The rest of the paper is organized as follows. In Section 2, the application description and the programming model used for massively parallel systems are motivated and explained. Section 3 outlines the system description. PerPreT is described in Section 4. Case study applications that validate the usefulness and accuracy of PerPreT are presented in Section 5. Conclusions and future work are outlined in Section 6.

2. Application Description

2.1. Programming Model

In many massively parallel multiple instruction multiple data (MIMD) systems, each execution unit (i.e., processor) only has direct access to its own local memory. The communication between different execution units is realized using message passing.

Code for massively parallel systems is primarily written using the SPMD programming model [Ser93]. In this model the same code is loaded on all execution units to perform the same task on different sets of data. The synchronization and communication of the tasks is done at the user level. At the system level, each processor executes its own code. Because of data dependencies, the various tasks of an SPMD program may have to communicate during execution. In the case of up to several thousands of processors, the parallel codes have to be regular and well structured to avoid load balancing problems and remain deadlock free. Often, the codes have alternating phases of communication and computation.

In Figure 2, a typical SPMD program is outlined as a task graph. The circles represent the computational tasks and the arrows represent communication between tasks. A computation phase does not last longer than T_{CPi} time units (i=1,2,..,7) and a communication phase does not last longer than T_{CMj} time units (j=1,2,..,6). The assumption is that T_{CPi} and T_{CMj} are the maximum times for all tasks at levels i and j, respectively.

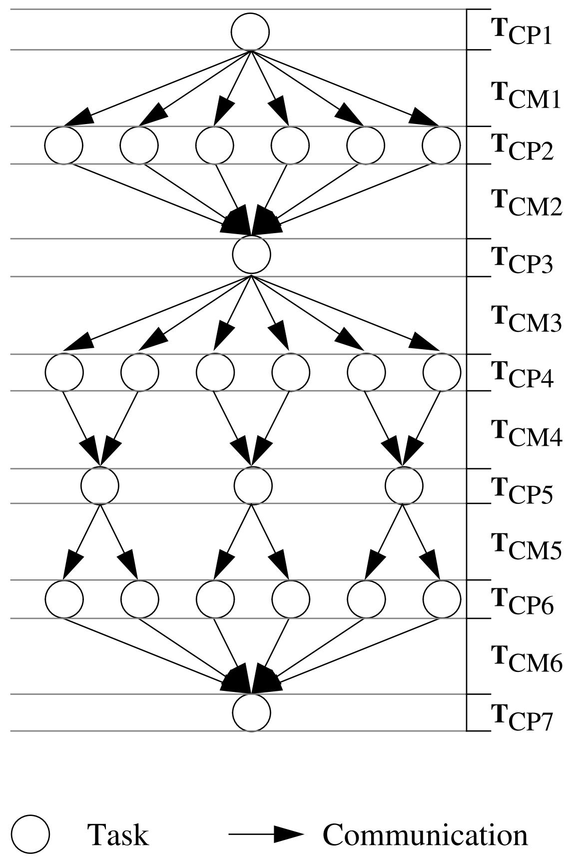


Fig. 2: Typical SPMD Program Task Graph

In Figure 3, a possible mapping of the tasks onto processors (P1,..,P6) is shown. An upper bound for the estimated communication time of this mapping is:

(1) $$\sum_j T_{CMj}$$

An upper bound for the estimated computation time is:

(2) $$\sum_i T_{CPi}$$

Thus, an upper bound for the total estimated execution time is:

(3) $$\sum_i T_{CPi} + \sum_j T_{CMj}$$

In Section 5, several case study applications show that the measured execution time is close to this upper bound.

For more general task graphs the number of subtasks per level, and thus the number of arrows per level, is not necessarily constant. The data parallelism often results in one subtask per processor for some of the levels. Thus, using the number of processors as a parameter for communication and computation. Using the number of processors as a parameter is a natural consequence of the implemented space sharing allocation policy on these machines. The problem size is the second parameter used. Clearly, the times T_{CPi} (determined by the number of statements to be executed) and T_{CMj} (determined by the message length) depend on these parameters, but they do not change the expressions for communication (1) and computation times (2).

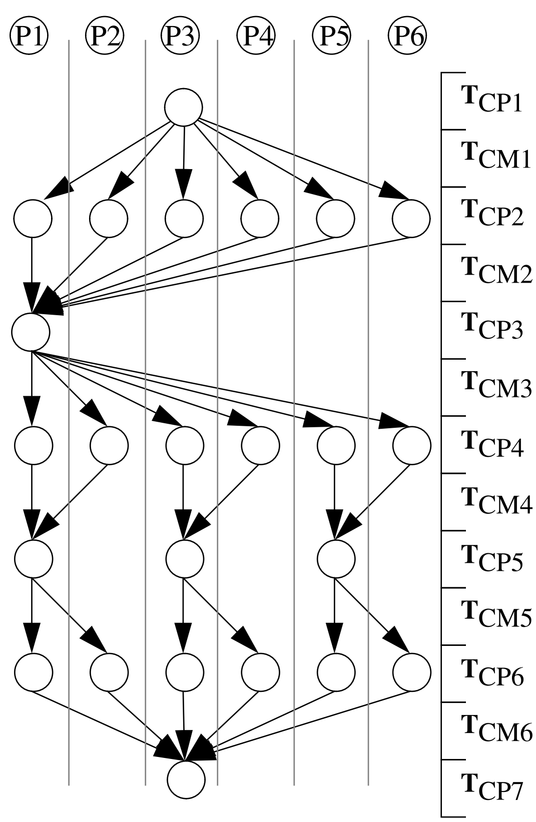


Fig. 3: Mapping of an SPMD program on 6 processors

Communication phases can be divided into global communications, where all execution units participate and local communications, where only a portion of the execution units participate. Typically, global communications are either:

1) broadcasts, where one processor broadcasts data to all other processors (e.g., CM1 and CM3 in Figure 3),
2) global collects, where one processor collects data from all other processors (e.g., CM2 and CM6 in Figure 3),
3) butterfly communication where all processors exchange data using a butterfly network configuration pattern, or
4) other regular patterns.

Typical local communications are point to point, where one processor sends or receives data to or from a subset of the processors (e.g., CM4 and CM5 in Figure 3).

2.2. Parameters

As outlined in the previous section, parallel SPMD applications running on multiprocessor systems are characterized by the problem size and the number of allocated processors as input parameters. Examples can be found in [LOOP94], [PAR94], [NAS93]. These architectures are scalable and are sold in different configurations. The problem size is an important parameter, because some of these machines do not support virtual memory (e.g., nCUBE/2, Transputer systems). If virtual memory is available, swapping should be avoided because it can significantly impact the performance of a parallel application.

PerPreT needs one formula for the computation and one formula for the communication of a parallel application as input. These formulae use the number of processors and the problem size as parameters. In the following sections it is demonstrated how to build these formulae using an example application.

2.3. Communication

PerPreT is equipped with a library to model typical local and global SPMD communications. Routines exist for several common communication patterns to compute the time required to perform such communication. The routines use the message size in bytes, the number of allocated processors, the problem size, and the system type as input parameters.

To date, the message passing protocol and the interconnection network of two system types, the Intel Paragon and the nCUBE, have been implemented. The structure of PerPreT allows the user to add defined system configurations. To model an SPMD program, PerPreT requires a high level communication description of the parallel program. This description can be derived in several ways. It can be given as a task graph, as LOOP language construct, or as a user defined description:

Using Task Graphs

The task graph of Figure 4 shows an SPMD version of a Conjugate Gradient (CG) Method. *N* represents the problem size and *P* represents the number of processors. The circles contain the number of floating point operations performed by the specific subtask. The values at the arrows represent the number of data items that have to be transmitted. If a circle is empty, then no floating point operations have to be performed. The phases 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.

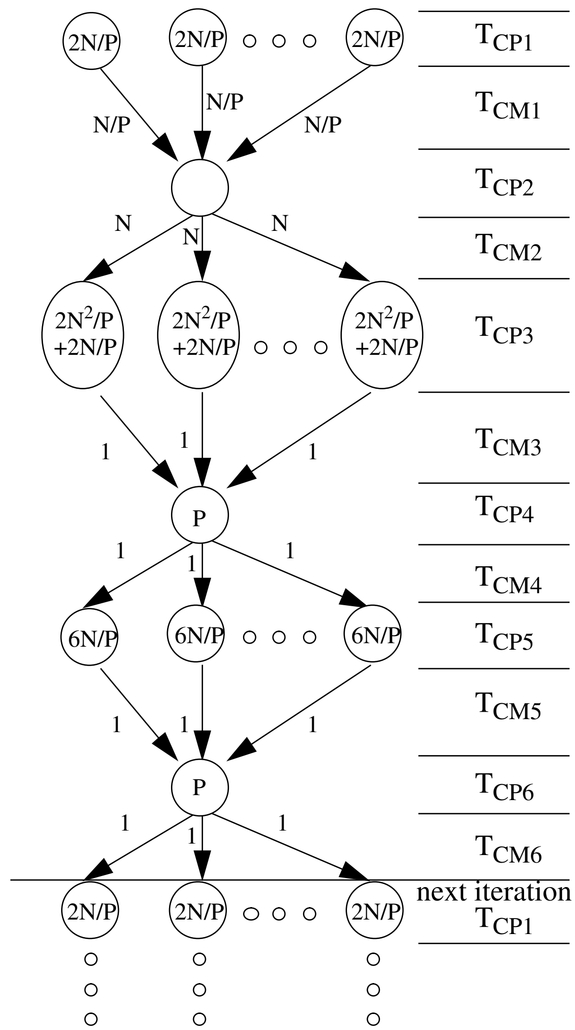


Fig. 4: Task Graph of CG-SPMD Program

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.

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 each) and one scalar product (2N/P statements) per processor resulting in a total of 6N/P statements.

CM5:
same as CM3

CP6:
same as CP4

CM6:
same as CM4

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 input parameter. They also have access to the global parameters *n_procs* (number of processors = P) and *p_size* (problem size =N). *TYPE* is an indicator of the data type to be able to determine the number of bytes per data item. The notation is derived from the C programming language. This leads to the following communication description formula of the CG-SPMD program that is used by PerPreT:

*bytes1 = sizeof(TYPE) * p_size;*
*bytes2 = sizeof(TYPE) * p_size/n_procs;*
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);*

Using the LOOP Language

The LOOP Language [LOOP94] is a high level language for SPMD programs. Similar to PerPreT, the LOOP Language provides a library for typical communication patterns. In Figure 5, the LOOP program for the CG-SPMD application is listed. The outer LOOP (lines 1 to 24) counts the iterations of the CG-solver. The variable *psize_node* has the value N/P (problem size divided by number of processors). The first inner LOOP (lines 3 and 4) results in 2N/P statements (CP1 in Figure 4). Then a *collect* and a *broadcast* of a vector are performed (CM1 and CM2 in Figure 4). Lines 8 to 11 contain a *matrix vector* product and a *scalar product* (CP3 in Figure 4). The rest of the code (lines 13 to 24) is self explanatory (*gsc* is a variable for the global scalar product, *t1* and *t2* are temporary variables).

```
1  \LOOP (ITERATIONS) iter {
2      beta = t1 / t2 ;
3      \LOOP (psize_node) i {
4          P[i] = beta * P[i] - RES[i] ; }
5      \SIMPLE_COLLECT_VEC(P);
6      \SIMPLE_BCAST(P,psize*sizeof(TYPE));
7      \LOOP i { H2[i] = P[i] ;}
8      \LOOP (psize_node) i {
9          H1[i] = (TYPE) 0;
10         \LOOP j {
11             H1[i] += MAT[i][j] * H2[j]; } }
12     \SCALPROD(&t1, P, H1, psize_node);
13     \SIMPLE_GLOBAL_SUM(&t1, &gsc);
14     \SIMPLE_BCAST(&gsc, sizeof(gsc));
15     alpha = t1 / gsc;
16     \LOOP (psize_node) i {
17         X[i] += alpha * P[i];
18         RES[i] += alpha * H1[i];}
19     t2 = t1;
20     \SCALPROD (&t1,RES,RES,psize_node);
21     \SIMPLE_GLOBAL_SUM(&t1, &gsc);
22     \SIMPLE_BCAST(&gsc, sizeof(gsc));
23     t1 = gsc;
24 }
```

Fig. 5: LOOP Program for CG Method

Each LOOP communication function is available as a PerPreT communication function. The user has to specify how often a communication function is called and how many bytes are transferred. Lines 5, 13 and 21 contain the *simple_collects* and lines 6, 14 and 22 contain the *simple_bcasts* for each iteration.

User Defined

PerPreT allows the user to make predictions for a parallelization strategy without parallel code, task graph, or LOOP language representations. The user identifies the communication situations that are involved in the parallelization strategy for a given algorithm and replaces them by routines from the PerPreT communication library.

2.4. Computation

Besides communication, PerPreT also requires a description of the computations involved in the SPMD program. It consists of a simple algebraic expression for the number of arithmetic statements that have to be executed by each processor. If this number varies for different processors, the maximum number is used. The expression can be derived from task graphs, the LOOP language, or from user specified complexity measures.

Using Task Graphs

If task graphs of an SPMD application are available, each circle, representing a task, will have an algebraic expression (in terms of N and P) for the number of floating point operations associated with it. After mapping the tasks onto the processors, the largest sum of the T_{CPi} for any one processor determines the formula required for PerPreT. In most cases the number of arithmetic statements can be used to predict the computation time fairly accurately. For the example task graph of Figure 4, this method leads to the PerPreT computation description for one iteration of a parallel CG-method as:

$$(4)\ st = (10*N + 2*N^2) / P + 2*P$$

where N is the problem size and P is the number of processors. If several iterations are executed, the number of statements *st* has to be multiplied by the number of iterations.

Using the LOOP Language

The LOOP language is designed for SPMD programs. Its structure makes it easy to represent LOOP programs as task graphs. The computational complexity of a LOOP program can easily be determined. In Figure 5, there are global sums in lines 13 and 21 (P statements each), scalar products in lines 12 and 20 (2N/P statements each), vector-scalar operations in lines 4, 17 and 18 (2N/P statements each), and a matrix vector product in lines 8 to 11 ($2N^2/P$ statements). Therefore, the sum of the statements is identical to (4). Automatic representation of LOOP programs as task graphs is currently under study.

Complexity Measures

For many applications, the algorithmic complexity is known by the user. For instance, the complexity of a matrix multiplication is $O(2*p\_size^3)$, the complexity of one relaxation sweep of an iterative Gauss Seidel solver is $O(5*p\_size^2)$. In SPMD programs, the data is distributed across all processors, thus dividing the complexity by the number of processors. This often leads to user defined formulae needed for the computation description.

2.5. Scope of Modeled Applications

The previous sections demonstrate that it is possible to describe the communication and computation behavior of an application with PerPreT. Additional examples for these descriptions are given for three case study applications. It is easy to obtain these descriptions for most SPMD programs. The SPMD programming model was selected because it is the most promising strategy in which to implement programs on massively parallel systems. If the parallelization strategy and the complexity of a sequential algorithm are given, PerPreT allows the user to evaluate the strategy without explicitly realizing it as an SPMD program.

3. System Description

3.1. Communication

In most existing message passing systems, the time required for communication can be divided into five phases:

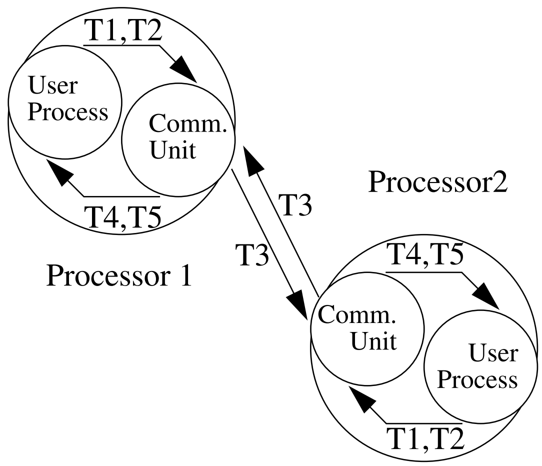


Fig. 6: Message Passing Communication

T1 Setup time for send routine:

This time is needed for the communication between the sender communication unit and the sender user process to initialize message buffers and transfer the control of the transmission to the communication unit.

T2 Copy message from user space to buffer

In the case of an asynchronous message passing protocol, the outgoing message is often copied to a buffer controlled by the communication unit.

T3 Message transmission time

This time is required to copy the message from the sender's communication unit to the receiver's communication unit.

T4 Setup time for receive routine

This time is needed for the communication of the receiver's user process with the receiver's communication unit. The receiver's user process is informed about the location of the message.

T5 Copy message from buffer to user space

In the case of an asynchronous message passing protocol, the incoming message is often copied from a buffer controlled by the communication unit to the receiver's user process space.

Figure 6 outlines the five phases for communication. Depending on the message passing protocol of the underlying hardware, one or more of the phases may or may not exist. For instance, Transputers use synchronous message passing where the messages are directly copied from the user space on a processor to the user space on another processor. In this case it is not necessary to copy the messages from user space to the communication buffer and vice versa. The PerPreT approach is general enough to model a wide variety of existing message passing protocols. The time for communication in message passing system normally follows the simple formula:

Tc = T1+T2+T3+T4+T5

where Tc is the communication time. Some of the phases (e.g., T2, T3, T5) depend on the message size. If a complete system specification is available, these times can be directly used by the PerPreT communication library routines. However, users often do not have access to a detailed specification or else the vendor provided times tend to be "optimistic", by reporting best case times. These reported times may not be valid if third party or other non native communication routines are used. For instance, if a program uses a non native portable communication library such as PICL [PICL90], the times are slightly higher because of the overhead of an additional software layer. The experiments reported in the next section used the PICL communication library. This library is portable across a variety of message passing systems including nCUBE Hypercubes and the Intel Paragon. The times

T1,..,T5 were determined for the PICL *send0* and *receive0* message passing calls by experimentation [Smi95]. These times are used as input parameters for the routines of the PerPreT communication library.

3.2. Computation

The computation description of an application yields a formula for the number of arithmetic statements that have to be executed per processor. If the vendor specified computation performance is close to the sustained node performance, the specified MFLOP/s rate can be used. Unfortunately the difference between the specified performance and the observed sustained performance is often significant. To provide PerPreT with realistic node performance values, the sustained MFLOP/s rate from the sequential program (without communication) on a single node is used.

The nCUBE nodes do not show significantly different performance for the case studies described in the next section. The sustained MFLOP/s rate is 0.43 and it is independent of the problem size. Therefore, this value is used for all the nCUBE experiments.

The i860 node processor of the Intel Paragon is more complex and performs differently for various case studies. The problem size parameter also affects the sustained MFLOP/s rate. In order to obtain accurate PerPreT predictions, the computation description of the Paragon consists of a table of measurements of the MFLOP/s rate for the sequential algorithm for each case study example. More accurate performance values yield more accurate predictions. If a user wants a fast investigation of a parallelization strategy, and the sequential program to measure the single node performance is not available, it is possible to estimate the expected MFLOP/s rate by looking at sustained MFLOP/s rates for reference applications.

3.3. Limitations of the System Model

PerPreT is designed for massively parallel MIMD message passing architectures. The model described in Section 3.1 handles many existing message passing architectures. Since a simple analytical model for communication and computation is used rather than running a more complex simulation, it is difficult to model hardware level phenomena such as network contentions. PerPreT routines calculate the communication time by taking the number of messages into account that can be sent or received simultaneously from an execution unit. They also include the message passing protocol and the implemented routing strategy. Experiments with parallel benchmarks validate PerPreT for SPMD applications as described in the next section.

4. Modeling using PerPreT

In this section, the output tables and graphs, generated by PerPreT, are presented. As in the previous sections, the parallel Conjugate Gradient method is used as an example.

Figure 7 shows the output table of an example experiment. The first column contains the number of processors (P), the second column contains the communication times (COMM), the third column contains the computation time (COMP), the fourth column contains the total execution time (TOTAL), and the last column contains the speedup (SP). Speedup is defined as the ratio of the single processor execution time to the multiple processor execution time.

The application examined is a Conjugate Gradient Method and the system modeled is the INTEL Paragon with 512 processors. PerPreT requires the application description, the system description, the parameter values for problem size, and the range of processors as input. PerPreT generates the output table shown in Figure 7.

============= R E S U L T S ===============

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	0.0482461	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

============= R E S U L T S ===============

Fig. 7: Output Table for CG-Tree (Problem Size 1024, Varying Processors) on an Intel Paragon

The user can repeat the experiment with different values of the problem size and processor range without changing the system or application description. In Figure 7, the problem size has the value 1024, and the number of processors varies from 1 to 512. This type of experiment may help the user to decide how many processors to allocate to the application. PerPreT also provides graphical representations for:

- speedup curve,
- execution, communication and computation time curves in a single diagram, and
- separate diagrams for the individual time curves.

The PerPreT graphical representation of the previous experiment is shown in Figure 8. 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 further. 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.

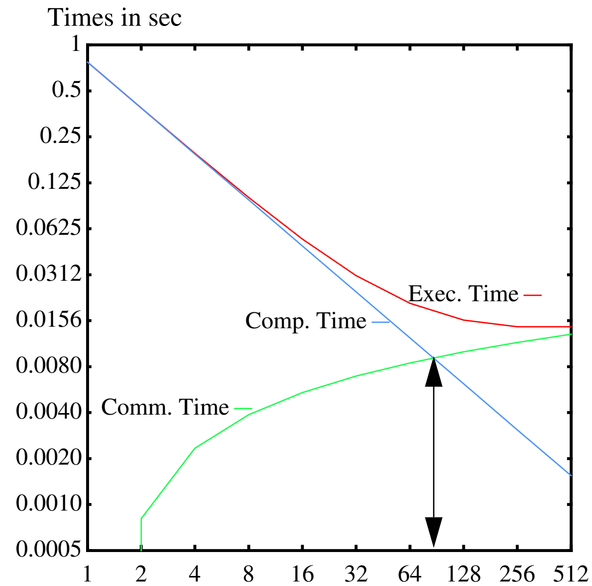


Fig. 8: All Times Plot of CG-Tree with Problem Size 1024 on an Intel Paragon

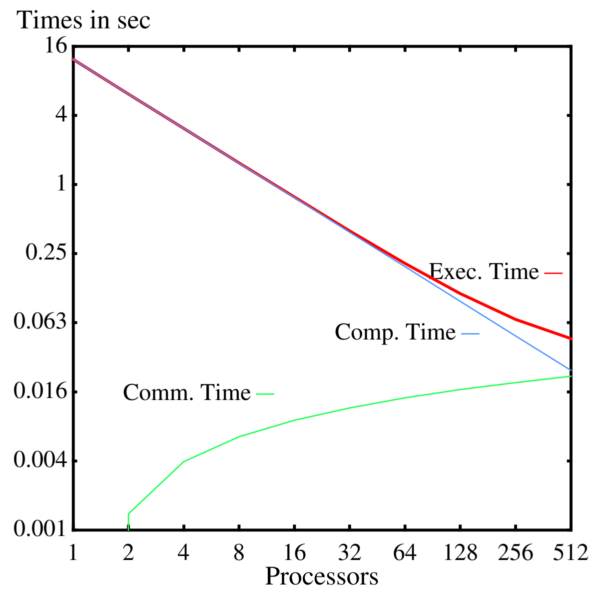


Fig. 9: All Times Plot of CG-Tree with Problem Size 4096 on an Intel Paragon.

In Figure 9 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 up to 512 processors on the Intel Paragon.

============ R E S U L T S ==============

Computation for Conjugate Gradient CG
Communication for Tree Broadcast
Processors: 512 Problem Size:512...16384
System: Paragon
Speedup estimate (all times in seconds):

PSZ	COMM	COMP	TOTAL	SP
512		Single proc. t.:	0.180685	1.00
	0.010611	0.000353	0.010964	16.48
1024		Single proc. t.:	0.771938	1.00
	0.012757	0.001508	0.014265	54.12
2048		Single proc. t.:	3.080252	1.00
	0.015943	0.006016	0.021959	140.27
4096		Single proc. t.:	12.306004	1.00
	0.021332	0.024035	0.045367	271.25
8192		Single proc. t.:	49.194010	1.00
	0.031080	0.096082	0.127162	386.86
16384		Single proc. t.:	196.716026	1.00
	0.048708	0.384211	0.432919	454.39

============ R E S U L T S ==============

Fig. 10: Output Table for CG-Tree (Varying Problem Size, 512 Processors) on an Intel Paragon

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 Figure 10, and graphically shown in Figure 11. The first column in Figure 10 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. 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, Figure 11 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..

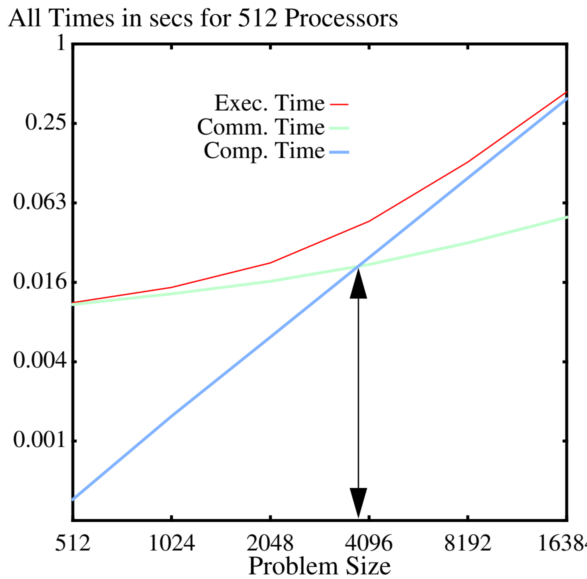


Fig. 11: All Times Plot of CG-Tree (512 Processors, Varying Problem Size) on an Intel Paragon

5. Case Study Applications

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. Since the nCUBE/2 available for the experiments consisted of only 128 processors, all results are plotted up to this processor number. 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 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 processor number) the accuracy range is ±20%. Slowdowns as shown in Figure 12 and in Figure 13 are also predicted correctly.

5.1. Conjugate Gradient Methods

CG-Tree:

In recent years, the conjugate gradient method for the solution of equation systems has become popular again. These methods are better suited for SPMD programs than solvers based on Gaussian elimination. The basic algorithm consists of a matrix vector product and several scalar products. Since the matrix data are distributed among all processors, a copy of the multiplying vector has to be built and distributed. To build this vector, the routine *tree_collect* is used (CM1 in Figure 4). To distribute the vector, the routine *tree_bcast* is used (CM2 in Figure 4). The prefix *tree* indicates that a treelike topology is used to perform these communication operations. The same routines are used to collect (CM3 and CM5 in Figure 4) and distribute (CM4 and CM6 in Figure 4) the global sum of the scalar products. The only difference is that the amount of data to be transferred (parameter *bytes3* in the formula given below) is a single data item.

Communication description of CG-Tree:

*bytes1 = sizeof(TYPE) * p_size;*
*bytes2 = sizeof(TYPE) * p_size / n_procs;*
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);*

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*p_size* floating point operations, the calculation of the scalar vector operations require *2*p_size* floating point operations each, and the calculation of the matrix vector product requires $2*p\_size^2$ floating point operations. The number of floating point operations per processor is:

$$(10*p\_size+2*p\_size^2)\,/\,n\_procs + 2*n\_procs$$

Seven iterations were modeled resulting in the following computation description of CG-Tree:

iter = 7;
$st = iter*(10*p\_size+2*p\_size^2)\,/\,n\_procs + 2*n\_procs;$

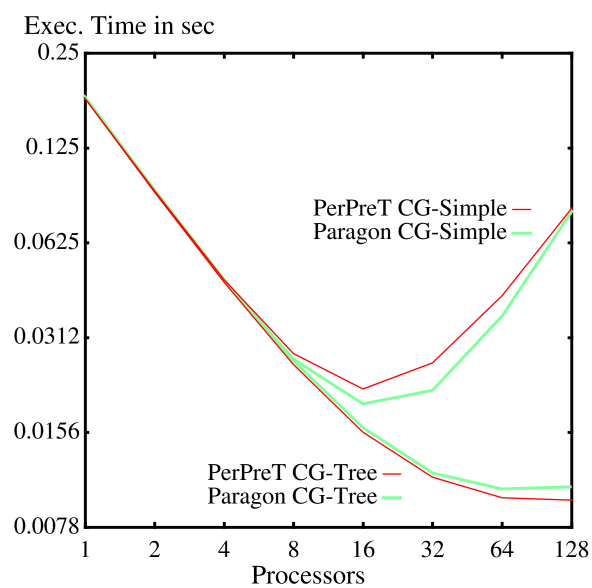


Fig. 12: Comparison of Actual and Predicted Execution Times of CG-Methods for Problem Size 512 on an Intel Paragon

The measured and predicted results for problem size 512 are presented in Figure 12 and

Figure 13. The experiments included other problem sizes as well and consistently showed a good match between predicted and measured values.

CG-Simple:

This version of the conjugate gradient method is similar to the previous one. The only difference is that the topology used for the communication operations is not treelike. The routine *simple_bcast(bytes)* calculates the time for a broadcast where one processor sequentially sends a message of length *bytes* to every other processor. The routine *simple_collect(bytes)* calculates the time that is needed for one processor to receive a message of length *bytes* from every other processor. The two different versions of the conjugate gradient method (CG-Tree and CG-Single) were used to find the number of processors where a treelike topology outperforms plain broadcast/collect routines. The PerPreT communication description of CG-Simple is:

*bytes1 = sizeof(TYPE) * p_size;*
*bytes2 = sizeof(TYPE) * p_size / n_procs;*
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);*

Since CG-Tree and CG-Simple only differ in calls to communication routines, the computation description for PerPreT is the same:

iter = 7;
$st = iter * (10*p\_size + 2*p\_size^2) / n\_procs$
$2*n\_procs;$

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 Figures 12 and 13).

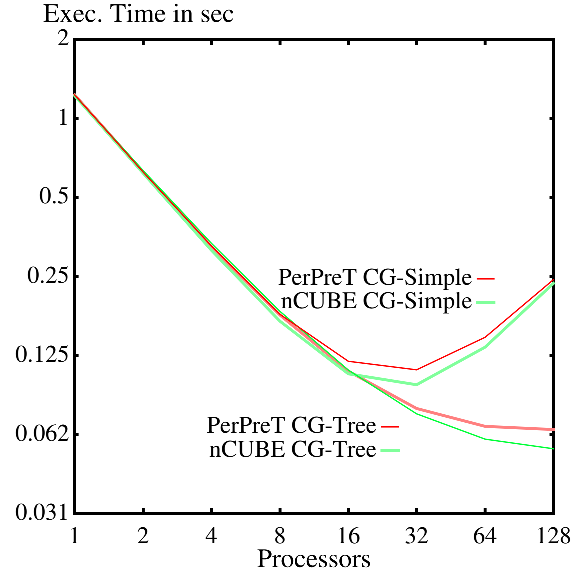
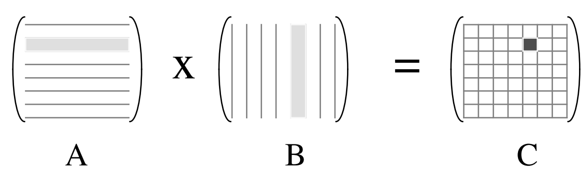


Fig. 13: Comparison of Actual and Predicted Execution Times of CG-Methods for Problem Size 512 on an nCUBE

5.2. Matrix Multiplication

Parallel matrix multiplication (PMM), algorithms are often used to evaluate the performance of multiprocessor systems. The basic algorithm is described in Figure 14.



Each node performs the following steps in a loop on the number of processors:

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

Fig. 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, 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. The routine *communicate(bytes)* calculates the time needed to send a message of length *bytes* to another processor and to receive a message of the same length from a third processor. Thus, the PerPreT communication description of PMM is:

*bytes = sizeof(TYPE) *p_size^2 / n_procs;*
comm_time =
*(n_procs-1) * communicate(bytes);*

The complexity of the sequential matrix multiplication algorithm is $O(2*N^3)$. With N being the problem size (*p_size*) and the data distribution onto *n_procs* processors the PerPreT computation description of PMM is:

*st = 2 * p_size^3 / n_procs;*

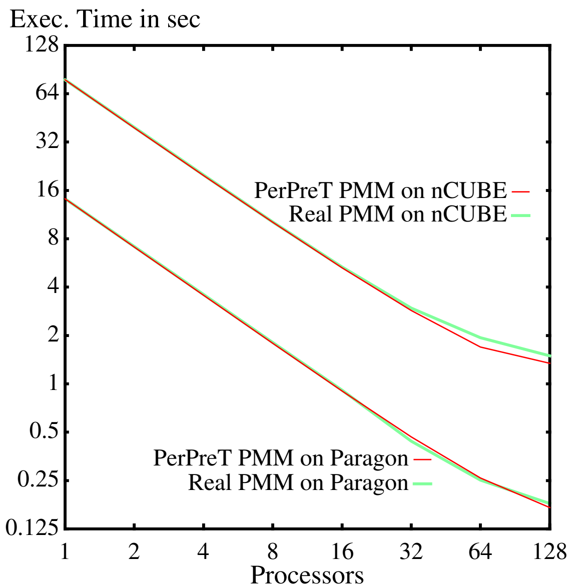


Fig. 15: Comparison of Actual and Predicted Execution Times of Matrix Multiplication with Problem Size 256 on an nCUBE and an Intel Paragon

The results for PMM presented in Figure 15 show a close match of predicted and measured values for a problem size of 256 on an nCUBE and an Intel Paragon. Additional experiments were carried out with different problem sizes. The difference between PerPreT values and measured values is always less than 10%.

5.3. Relaxation

Kernels in computational fluid dynamics (CFD) codes [NAS93] often use iterative solvers such as the Gauss Seidel relaxation method. Other iterative solvers, such as the successive overrelaxation method or multigrid methods, exhibit similar computational behavior. Red Black (RB) is a parallel version of the Gauss Seidel relaxation algorithm. To apply the starlike solver in parallel, all elements of the matrix are marked in a chess-board like manner using the colors red and black. The computation of a new iterative solution for the gridpoints of one color is only dependent on the gridpoints of the other color. Thus, it is possible to compute the iteration in parallel. For the SPMD program, the matrix is distributed as shown in Figure 16.

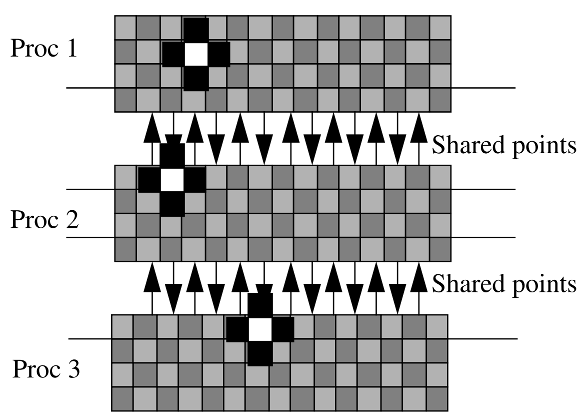


Fig. 16: Parallel Red-Black Relaxation

Each processor gets a row block of the matrix. 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. For each point, the following statement has to be calculated:

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

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

In the communication description for PerPreT, *iter* is the number of iterations, *neighb* is the number of neighbors of each processor, and *col* is the number of colors (two in the case of red black coloring). The routine *exchange* calculates the time needed to send a message of length *bytes* to another processor and to receive a message of the same length from that processor. This leads to the following PerPreT communication description of RB:

iter = 10;
neighb = 2;
col = 2;
*bytes = sizeof(TYPE) * p_size;*
comm_time =
*iter * neighb * col * exchange (bytes);*

The starlike gridsolver involves five floating point operations per gridpoint and cycle. The grid of dimension *p_size*p_size* is distributed across all processor, The resulting PerPreT computation description of RB is:

iter = 10;
*st = iter * 5 * p_size * p_size / n_procs*

Experiments executed for an nCUBE and an Intel Paragon with different problem sizes indicate a good match between PerPreT and measured values (see Figure 17 and Figure 18). Even for the relatively small problem size of 256 for 128 processors, the PerPreT results and the measured execution times are within 15% accuracy range.

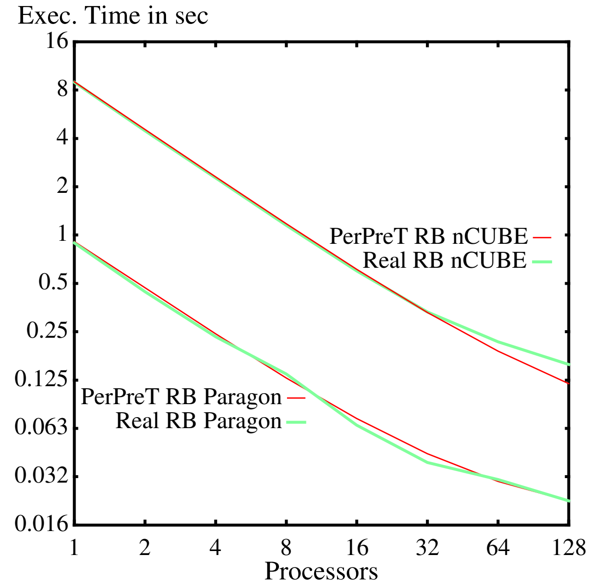


Fig. 17: Comparison of Actual and Predicted Execution Times of RB-Relaxation for Problem Size 256 on an Intel Paragon and an nCUBE.

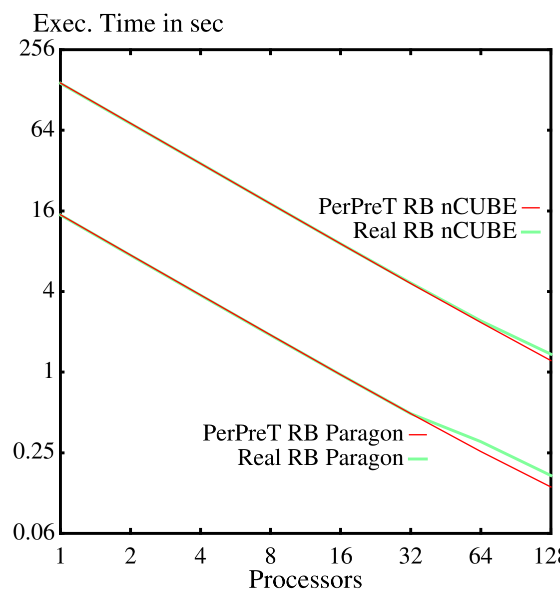


Fig. 18: Comparison of Actual and Predicted Execution Times of RB-Relaxation for Problem Size1024 on an Intel Paragon and an nCUBE

6. Conclusions and Future Work

This paper introduces PerPreT, a performance prediction tool for massively parallel systems. Several case studies for parallel application kernels used for the validation of PerPreT show that the predictions are accurate compared with the measured values on an nCUBE and an Intel Paragon.

The scalability of PerPreT with respect to the problem size and number of allocated processors is helpful when evaluating a range of applications and system configurations. Due to the modularity of the application and the system description, PerPreT can be used for fast evaluation of parallelization strategies and for fast evaluation of different systems.

Currently a more complex workload[1] is being implemented on the Intel Paragon that will be used for further validation of PerPreT. The PerPreT communication library is also being extended to architectures others than nCUBE and Paragon. A graphical user interface for PerPreT will be available soon.

Acknowledgements

The authors would like to express their thanks to Patrick Worley and the Mathematical Sciences Research Section at the Oak Ridge National Laboratory for providing access to the Intel Paragon systems and also to the Department of Computer Architecture and Operating systems of the University of Hannover for providing access to the nCUBE/2 system.

1. Shallow Water Code [Wor94]

References

[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

[Laz84] E. D. Lazowska et. al.:
Quantative System Performance: Computer System Analysis using Queueing Network Models, Englewood Cliffs, NJ, Prentice Hall, 1984

[LOOP94] J. Brehm et. al.:
A Multiprocessor Communication Benchmark, User's Guide and Reference Manual, Public Report of the ESPRIT III Benchmarking Project, 1994, available at: ftp@irb.uni-hannover.de

[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

[NAS93] D.H. Bailay et. al.:
NAS Parallel Benchmarks Results, Parallel and Distributed Technology, Vol. 1, IEEE, February 1993

[Ser93] Maria Calzarossa, Giuseppe Serazzi:
Workload Characterizatio - A Survey, Proceedings of the IEEE, 81(8), pp. 1136-1150, August 1993

[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

[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

[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

[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

[Wab94] Harald Wabnig, Günter 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, May 1994

[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