Accurate Performance Prediction for Massively Parallel Systems and Its Applications

Jens Simon and Jens-Michael Wierum

Paderborn Center for Parallel Computing – PC^2
Fürstenallee 11, 33095 Paderborn, Germany
{jens, jmwie}@uni-paderborn.de
http://www.uni-paderborn.de/pcpc/

Abstract. A performance prediction method is presented, which accurately predicts the expected program execution time on massively parallel systems. We consider distributed-memory architectures with SMD nodes and a fast communication network. The method is based on a relaxed task graph model, a queuing model, and a memory hierarchy model. The relaxed task graph is a compact representation of communicating processes of an application mapped onto the target machine. Simultaneous accesses to the resources of a multi-processor node are modeled by a queuing network. The execution time of the application is computed by an evaluation algorithm. An example application implemented on a massively parallel computer demonstrates the high accuracy of our model. Furthermore, two applications of our accurate prediction method are presented.

1 Introduction

Performance evaluation is important at every stage in the life-cycle of a computing system: Computer architects, programmers as well as end-users are interested in obtaining realistic figures on the expected performance. Prediction techniques are used in the system development by comparing alternative designs without actually implementing all of them. In algorithm design, performance prediction does not yield just the complexity of the algorithm, but even the expected execution time on a real machine. The results can be used to improve algorithms, i.e. to obtain highly efficient implementations. In any case, an abstract model of the hardware and of the algorithm is desirable to analyze performance.

The first approach for modeling programs on parallel machines was the PRAM model [FJ78]. This model and the more detailed BSP and LogP models are popular in theoretical computer science, where they are used for the evaluation of computational complexity [Val90, CKP+93]. Some authors improved these models by extending them with further features [HK95, RR95]. As it turns out the models are still not suitable for an accurate performance prediction of today's parallel system architectures with their sophisticated hardware structures and applications with a high level of concurrency. The abstract machine is represented by only a few machine parameters describing the network (latency,

bandwidth) and the processor (operations per time). Main emphasis was laid on algorithmic aspects of the parallelism and the communication cost of the algorithm. But the node design, for example, which is the most relevant part of parallel systems in terms of performance, has not been modeled accurately enough.

Today's MPP systems are usually composed of standard RISC microprocessors, designed for the personal computer and workstation mass-markets. The most performance critical part of this processor architecture is its memory system. A model for multi-level memory hierarchies can precisely describe the data-movements in a single processor machine [ACFS94]. In addition the extended memory hierarchy of parallel machines with shared memory architectures can be modeled quite accurately [BCKL94, MB92, Zha91, ZYC95].

Here, we present a performance prediction method which accurately predicts the runtime of a parallel application using the message-passing model and asynchronous task programming paradigm. We focus on a distributed memory architecture with shared memory nodes. Our prediction method combines the features of (1) distributed memory models, (2) shared memory models, and (3) models of memory limited computation.

2 The Prediction Method

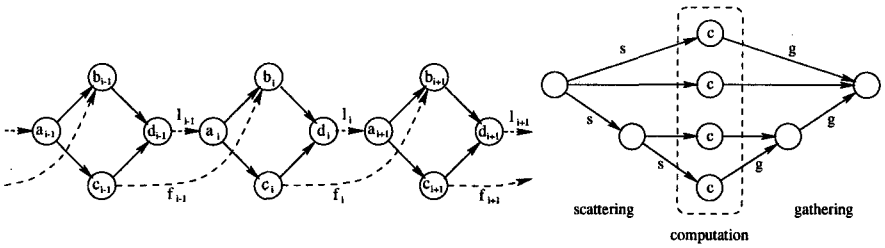
The different levels of a parallel system are modeled by several techniques: The *concurrency of the algorithm* is described by its task graph. *Communication* is modeled by functions depending on the message size and distance. *Multiprocessing on an SMP node* is modeled by an abstract scheduler with zero task switching time, fixed time-slices and a single priority level. The *resource-contention* on an SMP-node is estimated by a simple closed queuing network. An abstract model of a processor and the connection to the memory hierarchy is used to predict the sequential computation phases.

2.1 Task Graph

A parallel program can be seen as a collection of concurrent tasks with a certain control flow (Fig. 1). A task graph represents tasks and communication dependencies as nodes and edges which are labeled with computation and communication loads. A complete graph contains all tasks of a program which are spawned at execution time. We assume that the mapping of tasks to processors can be predetermined and does not change during the program runtime. A task is created and lives on a fixed processor until its termination. The load and location of a task frequently depends on the input data of a program (e.g. amount of work in computation-phases, the distance in communications). Estimations provided by mean-value analysis are substituted for problem dependent control flow.

Note that the size of the task graphs increases with the degree of concurrency, machine size, and problem size. The number of tasks in high performance implementations often exceeds the number of processors by a large factor [SW96b].

Creating such large graphs is quite effort-intensive and can be error-prone. Only for toy-problems on small parallel machines it is possible to generate and evaluate the complete task graph. For performance prediction of relevant problem and machine sizes, we need to reduce the task graph.



(a) (b)

Fig. 1. Example task graphs.

2.2 Relaxed Task Graph

A first approach to reduce the size of the task graph is to express the regularities in the graph by loops. The resulting cyclic task graphs have been studied by several authors [KME92, MST94]. Loops are represented by labeling loop edges with two parameters, the loop counter and the limit. Figure 2(a) shows a cyclic representation of Figure 1(a). The loop counter of loop edge l is i and the loop limit is n. Similar to a loop edge, a forward edge is labeled with a counter and a limit. Forward edges represent dependencies between tasks of different iterations. In our example f is such a forward edge. The size of task graphs becomes independent of the problem size with the introduction of loops, nevertheless it still depends on program size and machine size. A further graph reduction method is to group similar tasks on different nodes together [MNT93].

In our performance prediction method, we use a relaxed task graph for representing parallel programs. The large number of tasks of a massively parallel machine is reduced by eliminating negligible edges. Only the edges with high finishing times have to be considered in the set of incoming edges because of the maximum operation. Therefore, incoming edges with earlier finishing times can be omitted. Typically, this kind of edge reduction is possible within global communication operations and also in regular communication patterns on groups of processors. Figure 1(b) shows a typical global communication operation. Let us assume that only the labeled edges resp. nodes represent the relevant loads. It is obvious that the execution time of the graph is determined by the longest path (bottom). The resulting graph is depicted in Figure 2(b). This technique also works for a variable number of nodes. The number of nodes may depend on a probability function due to problem dependent control flow. In this case, the resulting load can be expressed by the mean-value. While these relaxations

potentially underestimate the runtime, our examinations in Section 4 show this to be negligible.

Simple paths (subsequent nodes without branches) of the resulting graph can be reduced further by replacing them by a single edge with empty nodes. The load of this new edge is the composition of functions of the eliminated nodes and edges. The resulting relaxed task graph is typically independent of the machine size (Fig. 2(c)). In the following, nodes and edges of the task graph are called *states* and *transitions*.

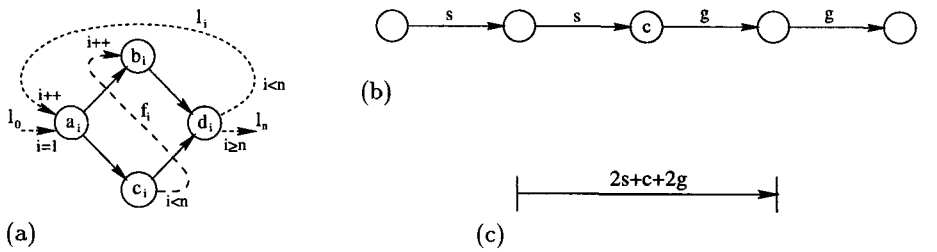


Fig. 2. Cyclic task graph (a) and relaxed task graphs (b,c).

2.3 Transitions in the Graph

The main part of the evaluation of the performance model is the determination of the computation and communication time functions. These functions characterize the architecture of the machine and the arguments specify the computational effort of the analyzed algorithm.

Communication: The communication time is a function of the message size and the communication distance represented by its source and target.

$$\tau_{comm}(size, dist) = \tau_{lat}(dist) + size \cdot \tau_{gap}(cont, dist) \quad (1)$$

Message size and communication distance have to be determined by an analysis of the algorithm. Low-level benchmarks are used to determine the hardware dependent functions startup time (lat) and the time interval between consecutive transmissions of words (gap). The reciprocal of τ_{gap} corresponds to the available communication bandwidth. Phases of intensive communication lead to contention of network resources (links, switchs, etc.). Communication contention is estimated during the evaluation of the task graph.

Computation: High performance implementations require an efficient utilization of the memory hierarchy of each node of the parallel machine. Usually arithmetic pipelines can process data in local memory faster than data can be

transferred between the different levels of the memory hierarchy. In order to design a simple model of memory-limited computation, we consider a single arithmetic pipeline connected to each level of the memory hierarchy (Fig. 3).

The execution time of scientific applications primarily depends on instructions on floating-point data including arithmetic and memory accesses. Here, we focus on algorithms allowing to hide integer instructions by floating-point instructions. The model includes three classes of floating-point instructions $\Im$: arithmetic operations and load and store instructions ($\odot$, $ld(l)$, and $st(l)$). Memory accesses are distinguished with respect to the level l of the memory hierarchy of the data. Level 0 corresponds to the register file. The cost of each instruction is independent of its execution context. The instruction stream $I = i(1), \ldots, i(k)$ of an algorithm is transformed into a stream of instructions $\tilde{I}$ of the model. $\tilde{I}$ must be determined by an analysis of the algorithm. The arithmetic operation $A = B \odot C$, for example, is modeled by the stream $ld(l_B)$, $ld(l_C)$, $\odot$, and $st(l_A)$. l_X specifies the level of the memory hierarchy of data X. In the following the frequency of such instructions is defined as μ_{instr}.

$$\tau(I) = \sum_{j=1}^{k} \tau_{i(j)} \quad \approx \sum_{instr \in \Im} \mu_{instr} \cdot \tau_{instr} = \tau(\tilde{I}) \tag{2}$$

The function $\tau(\tilde{I})$ defines the execution time in the model which approximates the execution time of I.

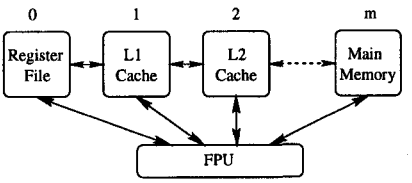


Fig. 3. Abstract model of processor and memory hierarchy.

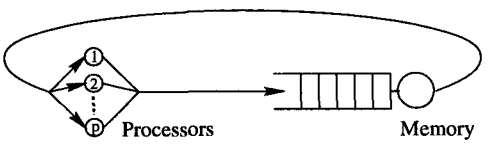


Fig. 4. Queuing model of an SMP-node.

Shared-Memory Model: Frequent simultaneous accesses to the shared memory lead to heavy contention on the memory bus in multi-processor systems. Mean value analysis of closed queuing networks gives an estimation on the increased response time [Lav83]. A node of the parallel machine is described by a single FCFS[1] queuing center. Apart from the memory queue and the corresponding server, the model contains p delay servers representing the processors (Fig. 4).

The time needed to execute of an arbitrary task T in the parallel system is given by $\tau(p) = R_{proc}(p) + R_{mem}(p)$, where R_{mem} and R_{proc} are the response

[1] FCFS: first-come first-serve

times of the memory system resp. processors. At first a one processor system is considered ($\tau(1)$, no contention). The execution time is divided into demands on the processor (τ_{proc}) and demands on the memory module (τ_{mem}). These correspond to the delay times of the servers. We assume that at any time the queuing network contains p tasks of the same type. This leads to a constant response time $R_{proc}(p) = R_{proc}(1) = \tau_{proc}$ of the processors. The response time of the memory module of a system with p processors can be described by the following recursion:

$$R_{mem}(1) = \tau_{mem},$$
$$R_{mem}(p+1) = \left[1 + \frac{p \cdot R_{mem}(p)}{\tau_{proc} + R_{mem}(p)}\right] \cdot \tau_{mem} \tag{3}$$

The previously described processor and memory hierarchy model defines the service demands. The delay time of the processors includes accesses to local memory levels and accesses to the shared memory define the delay time of the memory module.

2.4 Microbenchmarks

We devised a simple set of low-level benchmarks to determine the system parameters of the performance prediction model. The benchmarks aim at measuring performance parameters that characterize the basic architecture of the system and the used compiler software.

Computation parameters are benchmarked with various synthetic algorithms whose execution patterns reflect real algorithms. These programs consist of mixes of load, store, and arithmetic operations on vectors of various lengths and investigate all levels of the memory hierarchy (register, caches, main memory, etc.). For example, in technical documentations of processors only the maximum pipeline through-put is reported. This peak performance is very often not realized in algorithms because data cannot be transferred to and from memory as fast as it is needed. The memory bottleneck and the stalls of pipelines are approximately modeled by a microbenchmark by considering different computational intensities (ratio of arithmetic operations to memory references).

The basic communication properties of the message-passing MIMD computer are measured by ping-pong and message-exchange benchmarks. Messages of variable length are transmitted between nodes to determine the communication parameters latency and bandwidth (resp. gap). The parameters are measured with respect to communication distance and network contention.

2.5 The Evaluation Algorithm

An evaluation of the relaxed task graph is necessary for the execution time forecast of the algorithm. The following scheme uses a transition based internal task graph representation. Starting with the activation of the initial state, our evaluation algorithm performs the following steps: Each active state becomes

inactive and all computation and communication functions of its transitions are computed according to the actual parameters. The flow control is carried out through updating the parameters of the loop edges and propagating time stamps to subsequent nodes. A time stamp is the sum of the activation time of its state and the execution times of the considered transition. The activation time of a state is determined by the maximum time stamp of its incoming edges.

3 An Example

In this section we apply the previously described performance prediction method to a parallel benchmark program. For our studies we choose the parallel Linpack program. A lot of algorithm research and performance analyses have focused on this popular benchmark. Runtimes of the program are measured on a parallel machine with distributed memory.

3.1 The Parallel System

The considered computer architecture is a massively parallel distributed memory system with multi-processor nodes. Each node of the system consists of two RISC microprocessors, local shared-memory, and communication engines connected to the global communication network. The communication network is a two-dimensional mesh. Measurements are done on the Parsytec GC/PowerPlus system with 192 PowerPC-601 processors. The peak floating-point performance per node is 160 MFLOPS while the communication network provides moderate 3.3 MByte/s unidirectional communication bandwidth per link. The two-processor node architecture enforce multi-threaded programming to achieve most efficient programs.

Hardware Parameters: The performance prediction model uses the following hardware parameters of the GC/PowerPlus. Some parameters are given by the architecture while others are obtained by running benchmark programs.

The architecture parameters are the number of processors per node (two), a description of the memory hierarchy (32 fp-register and 32 kByte cache per processor and 64 MByte main memory per node), and the cycle time of the machine (12.5 ns). The size of a user partition of the machine is specified by the parameters X and Y. The largest user partition is 12×8.

The set of low level benchmarks described in Section 2.4 was executed on the machine. Table 1 shows the hardware parameters which are determined by the microbenchmarks. Note that the arithmetic operations also include memory accesses to the register file (memory level 0). Memory level 1 is the cache (ld_c, st_c) and memory level 2 is the shared memory (ld_m, st_m).

instruction	$\tau_{instruction}$ (cycles)	remarks
add	1	floating-point addition
madd	2	floating-point multiply&add, multiplication
ld_c	0.46	register load from cache
st_c	3	register store in cache
ld_m	2.5	register load from main memory (full cache line)
st_m	8.5	register store in main memory (full cache line)
gap	220	communication time per double (bidirectional)
lat	16800	communication latency
ts	800000	scheduler time-slice

Table 1. Hardware parameters of a Parsytec GC/PowerPlus as determined by our microbenchmarks

3.2 The Parallel Application

The Linpack program solves a system of linear equations which is an important task in scientific computing. The main part of the Linpack program is the LU-decomposition algorithm. The coefficient matrix of size $N \times N$ is factorized into two permuted-triangular matrices. Numerical instability is avoided by partial pivoting based on the search in the column of an entry of largest magnitude. Operations on submatrices of size $B \times B$ allows efficient utilization of the memory hierarchy. In the following, the performance relevant features of our algorithm are outlined.

- load balancing:
  - static on machine level (two-dimensional block-cyclic matrix distribution)
  - dynamic on node level (scheduling of two computing tasks)
- communication hiding:
  - concurrent pivoting
  - pendent row communication
  - pendent blocked elimination

For a detailed description of the high performance implementation of the parallel LU-decomposition we refer to [SW96b].

Task Graph: The relaxed task graph of the parallel LU-decomposition algorithm resulting from the relaxations is depicted in Figure 5. The graph shows three main paths of pivoting, row communication and blocked elimination. Pivoting consists of the two parts node-level pivoting and global pivoting. Row communication includes the elimination and broadcast of the pivot row and the communication of the exchange row. The update of the submatrix is represented by the blocked elimination path. The outer loop iterates the blocked operations. The inner loop iterates pivoting and row communication of a block. A forward edge shows the dependencies between two iterations.

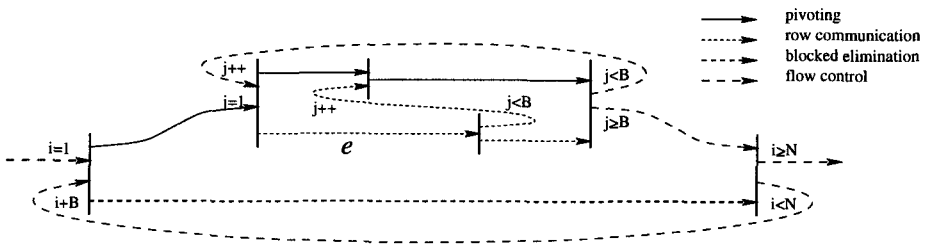


Fig. 5. Relaxed task graph of blocked LU-decomposition.

Transitions: As an example we describe the transition e of Figure 5. The edge represents two subparts of the LU-algorithm. The first term of the maximum operation of function 4 is the update of the block row which includes the pivot, the copy of the pivot row in a communication buffer and $j-1$ eliminations of the pivot row. The second term is the copy of the first part of the pivot row in a buffer and the sending of the buffer.

$$e = Max\{\alpha \cdot update\_row + copy_2 + row\_elim, copy_1 + send\_row\} \quad (4)$$

At first a detailed description of the function *update_row* is given. Due to the pending block elimination we need α to describe the probability that the considered row is not yet computed. In the following N_x and N_y represent the size of the submatrix which must be updated on a node. In this case $N_x = \lceil \frac{N-B\cdot(i-1)}{X\cdot B} \rceil$ and N_y is equal to one. Table 2 includes all necessary implementation details of the blocked elimination routine of the LU-decomposition (register blocking of size 2×12, 12×2, 2×2 resp.).

	type of op.	number of operations	remarks
proc	*madd*	$B^2 \cdot N_y \cdot N_x \cdot B$	arithmetic op.
	ld_c	$B^2 \cdot N_y \cdot (\lceil \frac{B}{2} \rceil N_x - 1 + (\lceil \frac{B}{12} \rceil - 1) \cdot N_x)$	cache loads
	st_c	$B^2 \cdot N_y \cdot ((\lceil \frac{B}{12} \rceil - 1) \cdot N_x)$	cache stores
mem	*ld_m*	$B^2 \cdot N_y \cdot (2 \cdot N_x + 1)$	memory loads
	st_m	$B^2 \cdot N_y \cdot N_x$	memory stores

Table 2. Number of operations occurring in the block elimination routine

$$\tau_{proc} = \frac{1}{2} \cdot (\mu_{madd} \cdot \tau_{madd} + \mu_{ld\_c} \cdot \tau_{ld\_c} + \mu_{st\_c} \cdot \tau_{st\_c}) \quad (5)$$

$$\tau_{mem} = \frac{1}{2} \cdot (\mu_{ld\_m} \cdot \tau_{ld\_m} + \mu_{st\_m} \cdot \tau_{st\_m}) \quad (6)$$

$$update\_row = \tau_{proc} + \left[1 + \frac{\tau_{mem}}{\tau_{proc} + \tau_{mem}}\right] \cdot \tau_{mem} \quad (7)$$

Function 5 specifies the number of cycles of arithmetic operations and cache accesses on a single processor. The number of cycles needed for data movement between memory and processors can be determined with function 6. The update of a block row on a two processor node is described by function 7. The increased response time of the memory module derived from bus contention is estimated by the queuing model described in Section 2.3.

Function *send_row* is the time for the communication of the first part of the pivot row which needs no further update.

$$send\_row = \tau_{gap} \cdot \left\lceil \frac{B \cdot (i-1)}{X} \right\rceil + \tau_{lat} \cdot \frac{Y^2 - 1}{3Y} + \beta \cdot \frac{\tau_{ts}}{3} \tag{8}$$

The last term is the average waiting time of a communication thread being executed on one of the two processors under the assumption that two computation threads are active ($\beta = 1$). If no computation task is active, β is equal to zero. The factor of τ_{lat} is the average distance between any two processor rows.

4 Model Validation

The performance prediction model developed in Section 3 is validated by comparing the predicted numbers with already reported performance results [SW96b]. Several numbers of processors and problem sizes are considered. We use system sizes between 2 and 192 processors and problem sizes from 352 to 26400 variables. The overall runtimes lie in the wide range of 0.6 to 1700 seconds. In particular, applications with small problem sizes are hard to analyze because of their short runtimes. Figure 6 depicts the measured and predicted runtimes implicitly expressed by the performance per node. Four classes of problem sizes

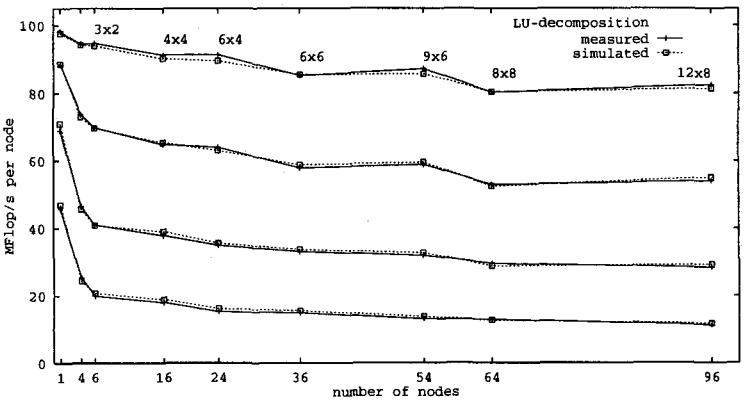


Fig. 6. Comparison of measured and predicted performance for different processor numbers and problem sizes (top to bottom: 100%, 25%, 6%, and < 2% mem. util.).

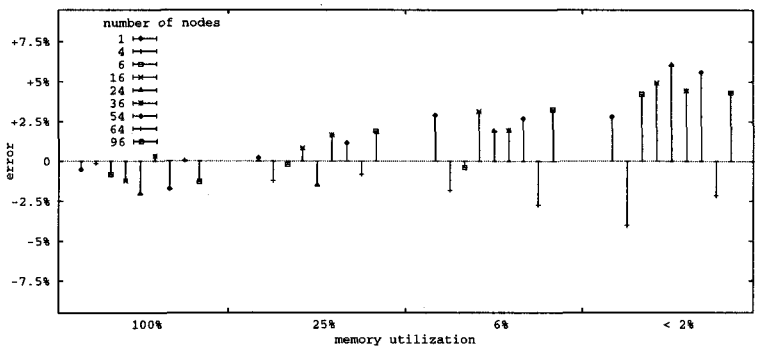


Fig. 7. Relative error of simulation for different processor numbers and problem sizes

are considered, leading to 100%, 25%, 6%, and 2% memory utilization. The diagram shows that our predicted results conform well with the experimentally obtained results. Also on small problem sizes the absolute difference between the two values is negligibly small.

A closer look at the quality of the model is shown in Figure 7. The relative error of the prediction is presented relative to the measured values. Problem sizes which utilize more than 25 percent of the memory lead to relative errors between ± 2 percent for all machine sizes. This is a very accurate result. Also the prediction of runtimes of very small problems on large machines is quite good (± 6.5 percent).

5 Applications

In this section we present two applications of our performance prediction method: optimizing parallel algorithms and evaluation of alternative designs for parallel computer architectures. The first one is useful in early stages of algorithm design and implementation while the second application is important for architecture designers interested in the influence of improved hardware parameters on the runtime of real parallel programs.

5.1 Optimizing Algorithms

When prediction tools allow to model precisely, successful program tuning is more likely to be achieved. Exact performance prediction may avoid the time consuming and error-prone process of implementation of different program versions. Furthermore the modeling and simulation prevent usage of expensive resources of large parallel machines.

In the following the optimization of the block size of the LU-decomposition is considered. Standard prediction methods assume a constant number of cycles for arithmetic operations regardless of the hierarchical memory system. The results of our performance prediction and a standard prediction method are compared

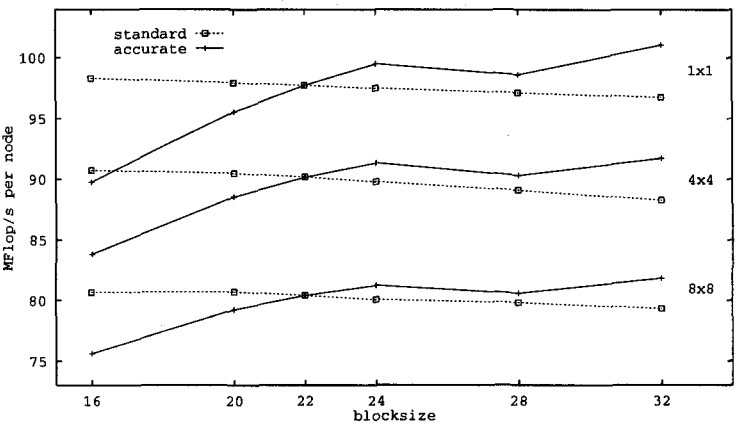


Fig. 8. Simulated performance of LU-decomposition on different machine sizes versus block size.

for different machine sizes (Fig. 8). The standard prediction is not able to show the expected behavior of the performance curve. In contrast, our method yields the same dependency between block size and performance as the experimental results [SW96b]. For all machine sizes our prediction conforms well with the measured performance.

5.2 Valuation of Architecture Designs

A further application of our performance prediction method is the valuation of architecture designs. During system development this technique can be used to compare a number of alternative designs to find the best one without realizing all different variants. In the following experiment, we study how the four main classes of machine parameters influence the runtime of the example application.

The parameter classes are computation (*add*, *madd*, *ld_c* and *st_c*), communication (*gap* and *lat*), load&store (*ld_m* and *st_m*) and the general data movement (*lat*, *gap*, *ld_m* and *st_m*). A parameter class is improved by the factor of two realized by halving the cycle numbers of all hardware parameters.

Figure 9 shows the obtained speedups resulted from the individual improvement of each parameter class in relation to the machine size. Here, the problem size scales with the machine size. With larger machines the node performance becomes less important while the influence of communication performance increases. Additionally to these already known tendencies our prediction method allows to determine real speedup numbers to quantify this behavior. The improvement of the memory bandwidth has the least effect on the speedup of all considered parameter classes (from +13% to +4%). This relative low speedup is due to the optimal utilization of the memory hierarchy of a node derived by the blocked algorithm. Its influence on the runtime decreases slowly with the machine size. For large machines a similar behavior can also be observed

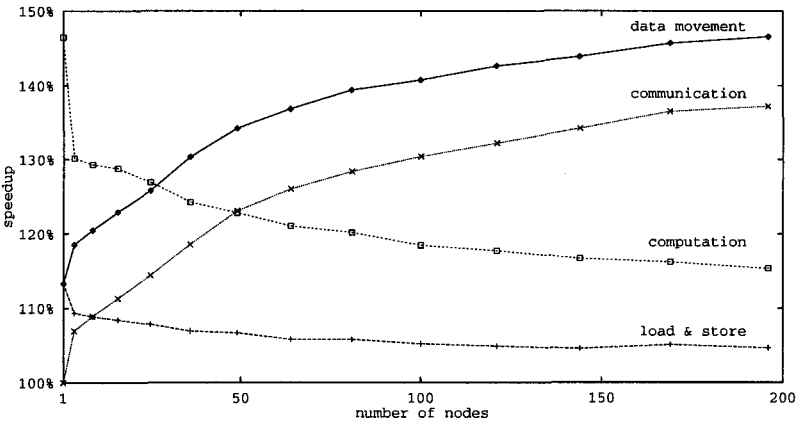


Fig. 9. Speedup in relation to the number of nodes of the machine (mem. util. of 25%).

if the computation parameters are improved. A doubled processor clock speed (faster pipelines and cache memory) improves the program execution time on a single node by 46%. But on a four node system the profit drops dramatically to 30%. The improvement of the communication leads up to 37% on the largest machine. This curve compared to the data movement (communication + load&store) shows the effect of the complex concurrencies in the considered application. The resulted function is not the sum of the two underlying curves. These complex relations between hardware characteristics and runtimes of parallel programs can only be analyzed with such an accurate prediction model.

A similar complex behavior occurs if the speedup is related to the problem size [SW96a]. Smaller problem sizes lead to an increasing of the influence of the communication parameters on the runtime. On the other hand, the influence of the computation parameters decreases.

6 Conclusion

We presented a performance prediction model that is appropriate for distributed memory architectures with multi-processor nodes. The underlying programming paradigms are message-passing and multi-threading. The complex memory hierarchy of nodes is described by a special processor model and a queuing model. We introduced several techniques to reduce the large task graphs of real parallel programs. As shown, mean-value analysis can result in accurate representations of problem dependent applications. The relaxed task graphs and the efficient evaluation algorithm give quick runtime forecasts. Both features, accuracy and fast evaluation of our method, open new applications of performance prediction of parallel programs.

References

[ACFS94] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:72–109, 1994.

[BCKL94] R. Bianchini, M.E. Crovella, L. Kontothanassis, and T.J. LeBlanc. Alleviating memory contention in matrix computations on large-scale shared-memory multiprocessors. *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 56 – 65, October 1994.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. van Eicken. LogP: Towards a realistic model of parallel computation. *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[FJ78] S. Fortune and J.Wyllie. Parallism in random access machines. *Proceedings of the 10th Annual Symosium on Theory of Computing*, pages 114–118, 1978.

[HK95] S.E. Hambrush and A.A. Khokhar. C3: A parallel model for coarse-grained machines. Technical report, Purdue, University, January 1995.

[KME92] A. Kapelnikov, R.R. Muntz, and M.D. Ercegevac. A methodology for performance analysis of parallel computations with looping constructs. *Journal of Parallel and Distributed Computing*, 14(2), February 1992.

[Lav83] S.S. Lavenberg. *Computer Performance Modeling Handbook*. Academic Press, New York, 1983.

[MB92] D.A. Menasce and L.A. Barroso. A methodology for performance evaluation of parallel applications in shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 14(1), January 1992.

[MNT93] D.A. Menasce, S.H. Noh, and S.K. Tripath. A methodology for performance prediction of massively parallel applications. *Proc. of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 250–257, 1993.

[MST94] Hermann Mierendorff, Helmut Schwanborn, and Maurizio Tazza. Performance modelling of grid problems – a case study on the SUPRENUM system. *Parallel Computing 20*, pages 1527–1546, 1994.

[RR95] T. Rauber and G. Rünger. A computation model for the parallel solution of differential equations. *Proceedings of the 5th Workshop on Compilers for Parallel Computers*, pages 294–306, June 1995.

[SW96a] J. Simon and J.-M. Wierum. On accurate performance prediction for massively parallel systems and its applications. Technical report, Paderborn Center for Parallel Computing, April 1996.

[SW96b] J. Simon and J.-M. Wierum. Sequential performance versus scalability: Optimizing parallel LU-decomposition. *Proc. of HPCN'96 in Lecture Notes in Computer Science 1067*, pages 627 – 632, 1996.

[Val90] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Zha91] X. Zhang. Performance measurement and modeling to evaluate various effects on a shared memory multiprocessor. *IEEE Transactions on Software Engineering*, 17(1):87–93, 1991.

[ZYC95] X. Zhang, Y. Yan, and R. Castaneda. Comparative performance evaluation of hot spot contention between min-based and ring-based shared-memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):872 – 886, 1995.