Predicting the Running Times of Parallel Programs by Simulation

Radu Rugina and Klaus Erik Schauser
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{rugina, schauser}@cs.ucsb.edu

Abstract

Predicting the running time of a parallel program is useful for determining the optimal values for the parameters of the implementation and the optimal mapping of data on processors. However, deriving an explicit formula for the running time of a certain parallel program is a difficult task.

We present a new method for the analysis of parallel programs: simulating the execution of parallel programs by following their control flow and by determining, for each processor, the sequence of send and receive operations according to the LogGP model. We developed two algorithms to simulate the LogGP communication between processors and we tested them on the blocked parallel version of the Gaussian Elimination algorithm on the Meiko CS-2 parallel machine. Our implementation showed that the LogGP simulation is able to detect the nonlinear behavior of the program running times, to indicate the differences in running times for different data layouts and to find the local optimal value of the block size with acceptable precision.

1 Introduction

When implementing a parallel algorithm, the designers need to know what granularity for computations and what data layouts yield the best running time of the program for a given number of processors. The prediction of running times is also useful for analyzing the scaling behavior of parallel programs.

Most of the previous work in modeling performance of parallel programs is based on analytical modeling techniques that abstract the features of parallel systems as a set of scalar parameters or parameterized functions. A survey of the current approaches to modeling is given in [14].

In the bulk-synchronous parallel (BSP) model [18] applications are expressed as sequences of computation steps separated by global synchronization. The model parameters are the number of time units of local computation at each step (L), and the communication bandwidth of the machine (g). The LogP model [6] and its successor, the LogGP model [2], extend BSP by taking into account the number of processors (P) and various communication costs: the latency (L), the overhead (o) and the gap (g) between consecutive messages, leading to more realistic predictions.

Other analytical models use parameterized functions to express the characteristics of parallel systems [8], [3] or use statistical modeling [1], [7]. The main disadvantage of functional and statistical models over the scalar parameter models is their high complexity which increases the computational costs of the prediction.

The analytical models can be extended by decomposing the parallel systems into several aspects and modeling each aspect separately [15], [17]. The Retargetable Program-Sensitive (RPS) model proposed by Stramm and Berman [17] uses mapping, program and architecture information to predict the running time. Program information includes flow of control, loop iteration, branching and synchronization, while architecture information comprises message transmission time, message startup overhead and instruction cycle time. Other approaches that use decomposition of parallel systems address both the communication and the memory hierarchy models via the LogP-HMM [11] or aim at decomposing the execution overheads due to parallelization [12], [4].

The work we present in this paper falls in this wide class of modeling by decomposition. The approach we propose is to simulate the program execution by following the control flow of the original program, to estimate the computation running time, and to determine the sequence of send and receive operations which is more likely to occur in the real execution, according to the LogGP model of parallel computing. We developed an algorithm which, given an input communication pattern, calculates the sequence of communication operations for each processor so that the resulting simulated execution complies with the LogGP model. Previous work on performance prediction used the LogGP model only to analyze regular communication patterns, where each

processor executes repeatedly a small communication pattern. The program running time was expressed using explicit formulas [9], [2] or was only given lower or upper bounds [16], [13]. Even for regular data layouts where all the processors got equal shares of data, and for the best-case algorithms, the resulting formulas were rather complicated. Our simulation algorithm is designed to deal with both irregular communication and irregular mapping patterns.

Our prediction approach takes into account only a restricted class of algorithms including oblivious algorithms for which the communication and computation steps alternate, hence being non-overlapping. We also consider that the whole volume of data the algorithm is working on is divided into equal-sized basic blocks that are spread among the processors such that each processor gets a certain number of such blocks. These basic blocks can be operated upon only by specific basic operations whose execution times are calculated separately, and which will be used to determine the computation time along the control flow path in the simulation algorithm. The restriction imposed by having equal-sized basic blocks makes it easy to extend our approach by including a memory hierarchy (cache) model in our decomposition.

Testing this new approach on the blocked parallel version of the Gaussian Elimination showed that this tool gives good indications on what the optimal layout and optimal block size is, out of a set of layouts and block sizes. The predicted running time values resulted in some differences due to the caching effects, especially for small block sizes. However, the rough simulation that takes into account only the computation and communication times and does not care about the caching effects was able to detect the non-linear shape of the running time variation with the block size. Including a model to simulate the caching effects, too, will improve the predicted values.

2 Restrictions on the analyzed algorithms

The approach we propose can be more easily studied if we restrict ourselves to a smaller class of algorithms that fulfill the following conditions:

- the communication pattern does not depend on the input;
- the whole volume of data is divided into equal-sized basic blocks that are assigned to processors;
- the basic blocks can only be operated upon by a finite set of basic operations;
- communication and computation steps do not overlap; they are alternating.

This class of algorithms matches the systolic matrix algorithms. For instance, Cannon's algorithm for matrix multiplication or the parallel Gaussian Elimination algorithm presented in the last sections are representative algorithms for this class. However, graph algorithms where several nodes are gathered in a single basic data block and assigned to a certain processor can be considered to fall in this class, too.

The condition of dividing data into basic blocks implies that the blocked versions of the algorithms will be used. The block size can be regarded as an input parameter, and finding the best value of this parameter is one of the goals of our approach.

One thing that can cause real problems when trying to derive an explicit formula is the fact the variation of the execution time with the block size can be (and usually is) non-linear. This may be normal since, for instance, in the case of matrix computations, where operations such as multiply grow with the cube of the block size. More interesting is the fact that one basic operation may be less expensive than another one for a certain block size and may become more expensive than that same operation for another block size. The basic operations used for Gaussian Elimination exhibit this type of behavior.

3 The LogGP model

This model of parallel computation presented in [3] abstracts the communication of arbitrary sized messages through the use of five parameters:

L: an upper bound to the latency of sending a message;

o: the overhead, defined as the length of time a processor is engaged in the transmission or reception of each message;

g: the gap between messages, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor;

G: the gap per byte for long messages, defined as the time per byte for a long message;

P: the number of processors/memory modules.

The model also assumes a single port model, in which at any given time a processor can either be sending or receiving a single message. The LogGP model does specify the gap between messages only for consecutive send or consecutive receive operations.

We assume that there is also a gap between consecutive send-receive or receive-send, too. Hence, after a send, the next consecutive receive begins g time units later, and after a receive, the next consecutive send begins $max(2o, g) - 2o$ time units later. These situations are shown in Figure 1.

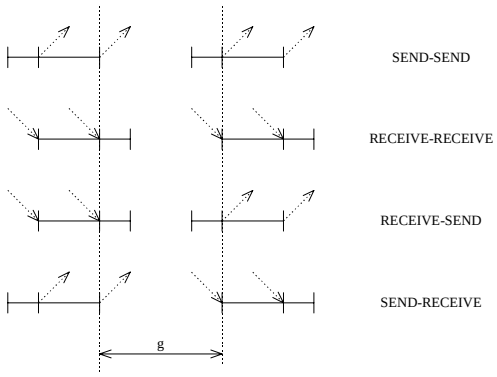

Figure 1. The gap in LogGP model

4 The communication simulation algorithm

In this section we present the algorithm for determining the execution time of communication steps that involve irregular send and receive operations. More precisely, given a communication pattern as input, our algorithm determines the sequence of send and receive operations performed by each processor, according to the LogGP model. The communication pattern is described by a directed graph where the nodes represent the processors involved in the communication step, the edges represent messages being transmitted and the costs of these edges represent the lengths of messages.

The algorithm that determines the sequence of transmitting messages also assumes that receive operations have priority over send operations, which means that whenever a processor wants to send a message and there is at least one message waiting to be received by that processor, the send operation will be postponed and the receive will be performed first. This supplementary assumption was introduced because our test case of the parallel Gaussian elimination was implemented in Split-C [5], whose active messages mechanism gives priority to receive operations.

Thus, the task of the simulation algorithm was to find the send/receive sequence for each processor in order that the following constraints to be fulfilled:

- maintain the gap g between consecutive sends and receives
- send available messages as soon as possible
- give priority to receive operations over send operations

The algorithm is presented in Figure 2. Each processor is assigned one queue of messages to be send and one queue of messages to be received. The latter queue is a priority queue, ordered by the arrival times of the messages.

for all processors $ctime = 0$
while (there are processors that want to send)
  min_proc = the processor having minimum
   $ctime$ among the ones that want to send;
  for min_proc:
    if it has messages to receive
      assume that the next operation is receive and set
      $start\_recv =$ start time of the receive operation
    else
      $start\_recv = \infty$;
    assume that the next operation is send and set
    $start\_send =$ start time of the send operation
    if ($start\_send < start\_recv$)
      perform SEND and update $ctime$
    else
      perform RECEIVE and update $ctime$
for each processor
  while (there are still messages to receive)
    perform RECEIVE and update $ctime$

Figure 2. Communication simulation algorithm

The algorithm uses a variable $ctime$ for each processor to keep track of the current simulation time and associates a queue of received and a queue of sent messages. The main loop of the algorithm picks up the processor min_proc having the minimum current simulation time among the ones that still have something to transmit. If there are several processors with minimum $ctime$, one of them is chosen randomly. Then, processor min_proc chooses between the next message to send (the first one in its *send* queue) and the next message to receive, if any (the earliest received message, i.e. the first one in its *receive* queue). The choice between these two alternatives is made by computing the start time of these operations in the presumption that this operation will be executed next and then choosing the operation which yielded the smallest start time. The strict comparison sign gives priority of receives over send operations. If a send operation is performed, the time by which the message reaches the destination processor is computed and the message is inserted in the receive queue of that processor at a position given by this arrival time. Finally, there is a loop where there are no more messages to be send and all the processors process all the messages in their *receive* queue.

4.1 A sample communication pattern

We did run this algorithm on the specific communication pattern shown in Figure 3. This pattern is encountered in the Gaussian elimination algorithm in which the processors on several diagonals of the matrix are involved in each

communication step. All the transmitted messages have the same length, therefore they were not specified on the graph.

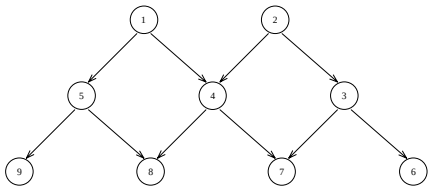

Figure 3. A sample communication pattern

Even though this seems to be a regular pattern, it is hard to get an explicit formula for this communication step for each processor. Things get more complicated if messages have different lengths.

In this example we considered that the machine has the following parameters: L=9 μs, o=2 μs, g=14 μs, G=0.03 μs, which are close to the Meiko CS-2 parameters. Messages being communicated have 101 bytes each. The results of the simulation algorithm are shown in Figure 4. It can be seen that the three conditions presented in the previous section are fulfilled by all the processors. For instance, processor 4 handles first the two receives (from processors 1 and 2) before sending its second message to processor 7, thus giving priority to receives. Processor 7 will terminate the last this communication step after 67 μs from the beginning of the step.

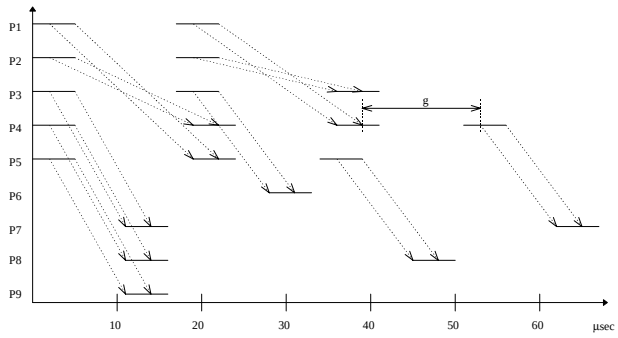

Figure 4. Sequence of send-receives in the simulation algorithm for the task graph from Figure 3

However, if only one message arrives a bit later than the LogGP model expected to, the whole sequence of send/receive operations for all the processors can be completely changed. Such situation can easily happen in real execution since the LogGP model gives an average behavior of the transmission of messages over the network, and not a precise one. Therefore it is probably better to find a scenario that is closer to the worst case (since deciding which is the worst case is again a difficult problem).

4.2 The overestimation simulation algorithm

In order to get a worse situation than the one predicted by the LogGP model, and thus to have a overestimation of the communication time, we modified the algorithm presented in Section 4 such that each processor first waits for all the messages it has to receive and only afterwards it starts transmitting its own messages.

This algorithm assumes that each processor knows at the beginning how many messages it has to receive by means of a *messages_to_receive* counter. At each iteration, each processor that has no more messages to receive starts sending all its messages, decrementing the counters at destinations. In the second part of the iteration, each destination processor performs the corresponding receive operation.

Running this algorithm on the communication pattern presented in the previous section yields the send-receive sequence shown in Figure 5. The same LogGP parameters and the same message length as in the previous case were used. The execution time of the communication step has increased now to 76 μs and is given by processors 6, 7, and 8, which finish the last receive operation concurrently. It can be seen that processor 8 receive the messages from processors 4 and 5 concurrently, but processing the second receive is delayed in order to fulfill the gap requirements between consecutive receives.

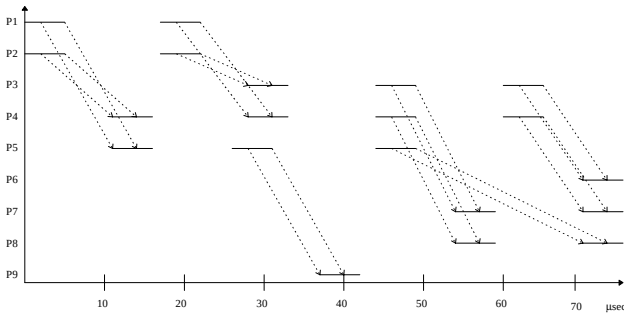

Figure 5. Sequence of send-receives in the overestimation algorithm for the task graph from Figure 3

This algorithm cannot take place in real execution since processors usually do not know how many messages they are expected to receive, as in the case when source processors use `store` Split-C instructions based on active messages for transferring data to destination processors, which are not aware in the program when the receives are taking place. Even if the programmer sets for each processor the number of messages to receive, it is sure that he will try to

perform the send instructions as soon as possible, without waiting for the receives. Therefore this algorithm was introduced only to give an upper bound to the communication running time under LogGP model.

We must say that if there are cycles in the communication pattern, then the above algorithm will produce a deadlock since any processor belonging to the graph cycle will never start sending any message because it first waits to receive some message from some other processor in the cycle. Therefore, in such cases, the algorithm performs randomly some message transmissions in order to break the deadlock.

5 The parallel Gaussian Elimination algorithm

For testing our performance prediction approach, we used the Gaussian Elimination algorithm without pivoting. The parallel version of the algorithm is presented in [10], and is based on the observation that each iteration of the sequential algorithm can be regarded as a diagonal wave traversing the matrix from the upper left corner to the lower right corner. Thus, several diagonals of elements can be made active at the same time, resulting in a parallel computation.

5.1 The blocked version

The blocked parallel version of the algorithm increases the granularity of the basic processing elements, which we call basic blocks. The communication pattern remains the same, but the operations to be performed on these basic blocks are more complex than in the non-blocked version, requiring upper triangularization, inversion and multiplication of matrices. More precisely, the blocked GE algorithm uses four basic operations to operate on basic blocks: Op1 = $\{ B_1^U, B_2^U B_3^{-1}, (B_4^U)^{-1} \}$, Op2 = $\{ B_1 B_2, B_3 = 0 \}$, Op3 = $\{ B_1 B_2 \}$, Op4 = $\{ B_1 - B_2 B_3 \}$. Our implementation considered a 960×960 matrix that was divided into blocks having values in the set {10, 12, 16, 20, 24, 30, 40, 48, 60, 64, 80, 96, 120, 160}. We implemented the basic block operations presented above and we measured the running time of each operation for different block sizes. As we said in Section 2, it is expected that these operations result in nonlinear behavior relative to the block size, and this is the case for the actual computed values presented in Figure 6. Moreover, this figure shows that the "most expensive operation" changes with the block size: for small blocks Op1 is the most expensive, for blocks of about 20×20 elements all the operations take about the same amount of time ($\approx$1700 μs), while for large blocks (160×160 elements) the multiplication involved in Op2-3 takes about twice the time needed for Op1.

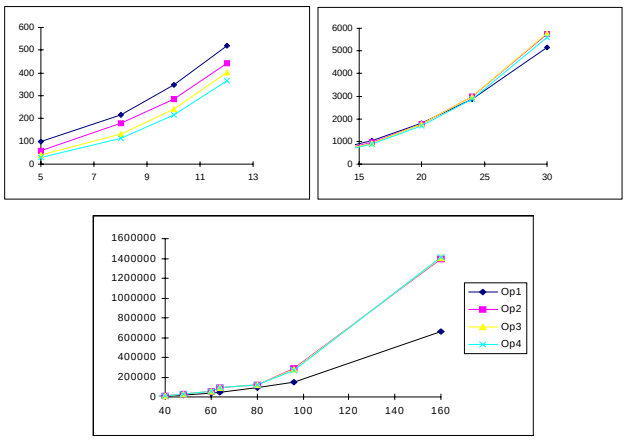

Figure 6. Dependence of the basic operations running time (μs) on the block size

5.2 Data layout

In our implementation we used and compared two layouts: the *row stripped cyclic* and the *diagonal* mapping. In the first one processors are assigned cyclically rows of blocks, hence the row-wise propagation of data does not involve any message transfer. This mapping produces a non-uniform load distribution and increases the computation time. The diagonal mapping assigns the blocks of each diagonal to different processors, ensuring a uniform load on the diagonal bands that decreases the computation time. However, there is a small probability that row- or column-adjacent blocks are mapped on the same processor. This leads to an all-to-all broadcast situation, increasing the communication time.

5.3 Numerical results

We implemented the parallel blocked version of the Gaussian elimination algorithm for matrices of 960×960 elements divided in blocks ranging from 10×10 elements to 160×160 elements using row stripped cyclic layout and diagonal layout on 8 processors on the Meiko CS-2 parallel machine. The measured running time values were close to the predicted values for large block sizes. The differences between predictions and measurements for small block sizes are due to the cache effects: when processors are assigned many non-adjacent small blocks, the cache miss rate increases.

Therefore, we introduced some dummy instructions to bring the necessary blocks in the cache and we timed this section separately. Figure 7 shows the predicted values computed with each of the two algorithms as well as the measured values with and without the extra caching section.

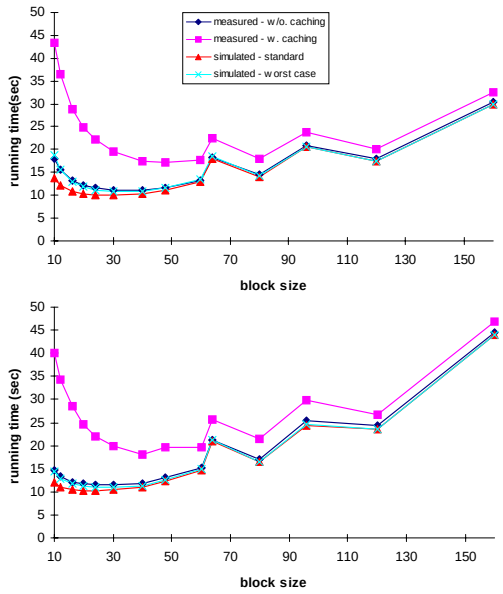

Figure 7. Total running time. top: diagonal mapping, bottom: stripped cyclic mapping

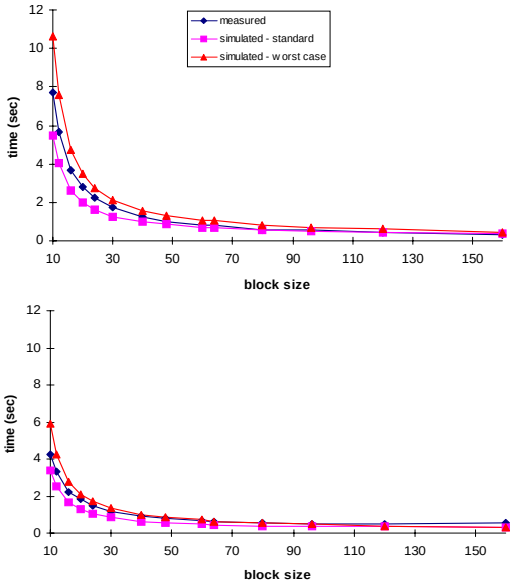

Figure 8. Communication time. top: diagonal mapping, bottom: stripped cyclic mapping

The new simulation approach gives accurate estimations if no cache effects are taking place. Figure 7 shows that even with this rough estimation that do not take into account the cache effects, the predicted values follow the saw-tooth behavior showed by the real execution for block sizes greater than 60 elements.

The distorting cache effects introduce differences in the predicted optimal block size values: 30 elements instead of 48 for diagonal mapping, and 24 elements instead of 40 for the stripped cyclic mapping. But these roughly predicted best block sizes yield real running times that are not far from the real minimum times.

Another purpose of the simulation approach was to determine differences in running times for different layouts. The simulation predictions indicated that the diagonal mapping works better, especially for large block sizes, which is exactly the same result as the one obtained from the real execution comparison. The caching overhead is almost the same for the two data layouts, so that the comparison between layouts is not strongly influenced by cache effects.

When comparing the communication running time alone, the measured values fall between the simulated values used the algorithms presented in Section 4 (standard) and Section 4.2 (worst-case) for either layout, as shown in Figure 8. The predicted values are expected to be under the measured ones since our simple simulation does not take into account the message transfers from one processor to itself, which are local memory transfers in real execution.

For the computation running times, the simulation predicts values that are very close to the measured ones. Differences are introduced here by the overhead of iterating through the all blocks each processor is assigned to, which is not taken into account by our simple simulation. For small block sizes, each processor is assigned a larger number of blocks, so that the overhead of iterating through these blocks will be greater in this case, as Figure 9 indicates.

The under-estimations of the computation times are compensated by the over-estimation of the worst-case communication times, so that Figure 7 shows a better prediction of the worst-case algorithm compared to the normal one.

Hence, testing our approach with a simple LogGP simulation that takes into account only the effective computations performed and the messages transferred across processors, disregarding the caching overhead, the local data transfers and the iteration overhead, we obtained accurate enough information about the variation of the running time with the block size, about the block size value that yields the minimum running time and about the best data layout.

6 Conclusions and future work

We presented a new approach to predict the running time of parallel programs which simulates the execution of the programs and determines the LogGP sequence of send and receive operations. The measurements performed for the blocked parallel version of the Gaussian Elimination algorithm showed that this approach can be used to get accurate

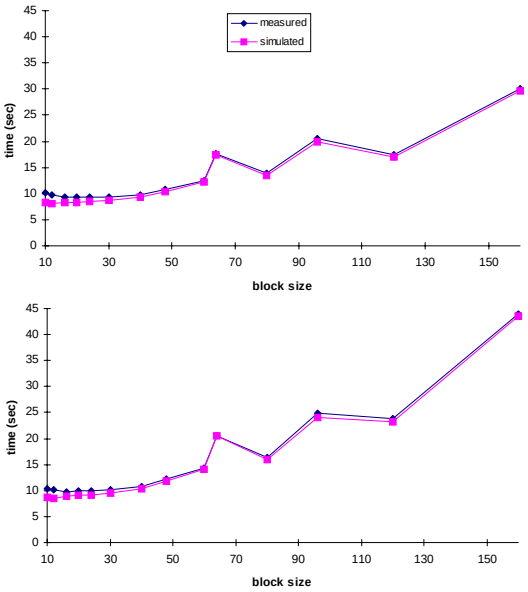

Figure 9. Computation time. top: diagonal mapping, bottom: stripped cyclic mapping

indications about the dependence of the running time on the block size, and about the locally optimal block size value and layout from a certain set of block sizes and data layouts.

The measured running times showed that cache effects have a great importance and therefore a model to simulate caching behavior must be incorporated in the simulation algorithm, beside the models for communication and computation. Further, future work may be done to automatically determine these optimal values from the predicted running times. This reduces to a search problem and therefore some heuristics have to be used. Analyzing the program simulation for overlapping communication and computation steps or for variable-sized blocks are also subjects for future development of this approach.

References

[1] V. D. Agrawal and S. T. Chakradhar. Performance estimation in a massively parallel system. In *Proceedings of Supercomputing*, pages 306–313, 1990.

[2] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. In *Proceedings of the 7th Annual Symposium on Parallel Algorithms and Architectures*, Santa Barbara, CA, July 1995.

[3] E. A. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[4] M. E. Crovella and T. J. LeBlanc. Parallel performance prediction using lost cycles analysis. In *Proceedings of Supercomputing*, Washington, D.C., November 1994.

[5] D. E. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing*, pages 262–273, November 1993.

[6] D. E. Culler, R. M. Karpand, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[7] R. T. Dimpsey and R. K. Iyer. A measurement-based model to predict the performance impact of system modifications: A case study. *IEEE Transactions on Parallel and Distributed Systems*, 6(1):28–40, 1995.

[8] M. A. Driscoll and W. R. Daasch. Accurate predictions of parallel program execution time. *Journal of Parallel and Distributed Computing*, 25:16–30, 1995.

[9] M. Karp, A. Sahay, E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *Proceedings of the 5th Symposium on Parallel Algorithms and Architectures*, June 1993.

[10] Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[11] Z. Li, P. H. Mills, and J. H. Reif. Models and resource metrics for parallel and distributed computation. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, 1995.

[12] D. R. Liang and S. K. Tripathi. Performance prediction of parallel computation. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 625–629, 1994.

[13] W. Lowe and W. Zimmermann. Upper time bounds for executing PRAM-programs on the LogP-machine. In *Proceedings of Supercomputing*, pages 41–50, 1995.

[14] W. Meira. Modeling performance of parallel programs. In *Technical Report 589*, Rochester University, 1995.

[15] G. R. Nudd, E. Papaefstathiou, Y. Papay, T. J. Atherton, C. T. Clarke, D. J. Kerbyson, A. F. Stratton, R. Ziani, and M. J. Zemerly. A layered approach to characterization of parallel systems for performance prediction. In *Performance Evaluation of Parallel Systems*, U. Warwick, U.K., 1993.

[16] E. Santos. Solving triangular linear systems in parallel using substitution. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, 1995.

[17] B. Stramm and F. Berman. Predicting the performance of large programs on scalable computers. In *Proceedings of the Scalable High Performance Computing Conference*, pages 22–29, 1992.

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