Performance Prediction Modelling of Multicomputers

by
H.V. Sreekantaswamy
S. Chanson and
A. Wagner

Technical Report 91-27
November 1991

Department of Computer Science
University of British Columbia
Rm 333 - 6356 Agricultural Road
Vancouver, B.C.
CANADA V6T 1Z2

Performance Prediction Modelling of Multicomputers

H.V. Sreekantaswamy, S. Chanson and A. Wagner

Department of Computer Science,
University of British Columbia,
Vancouver, BC, Canada V6T 1W5.

Abstract

In order to effectively program Multicomputers users must be able to evaluate how well the system performs for a given application. In this paper we present an efficient execution model that can be used for tree structured computations. We provide a general framework for analyzing the performance of this type of computation for any given topology. This framework is used to derive models for two widely used parallel programming strategies: Processor Farms and Divide and Conquer. These models were validated on a large multicomputer and the accuracy of the model is such that they can be used to predict the performance of applications that use these strategies. We discuss how these models can be used to evaluate performance and how they could be used to restructure the application to improve performance.

1 Introduction

A multicomputer is a network of microprocessors, each with its own memory, with hardware support for message passing [9, 2, 8, 15]. The relative ease with which it is possible to build inexpensive, high-performance microprocessor-based systems has made multicomputers very popular. However, a major problem in this area is the difficulty in effectively programming these systems. The focus of much recent work in this area has been on identifying programming abstractions for managing and coordinating the activities of multiprocessor systems. Many existing abstractions are based on well-known strategies for parallelizing programs such as processor farms [14, 6], divide and conquer [4], compute and aggregate [13], and multipipelines [7]. Ultimately however, the success of these abstractions depend on the extent to

which they can make efficient use of the underlying architecture. It is therefore important to cost these abstractions. This can be done by providing users and system designers with analytic performance models that take into account both the hardware and software used to implement a given abstraction. These models can be used as the basis for comparing and evaluating different abstractions. It also provides insights into those parameters that most affect performance, potentially allowing them to be tuned to optimize performance.

In this paper we describe analytic models for two important parallel programming strategies: processor farm and divide and conquer. We describe a system that is simple, efficient and general enough to implement processor farms and tree-structured computation. Briefly, the system inputs a flow of tree-structured tasks that are distributed to the rest of the processors. The system dynamically, depending on its load, either executes the task, or splits and forwards it. This execution model is similar to one proposed by ZAPP[11] and by [14]. Our analytic models relate the speed-up of an N-processor system to the number of tasks, the size of these tasks, and the communication overheads of executing them in parallel.

This work differs significantly from other analytical models for parallel machines. First, it models both the hardware and software of the system. Many of the models for multi-computers are given only in terms of the hardware parameters and do not account for the software overhead of executing tasks in parallel [14]. Secondly, it differs from the more general analytic models that have appeared in the literature[5, 1]. Most of these models only consider parallelism and not the communication costs involved in exploiting this parallelism. As a result, these models cannot accurately predict the performance of programs for which there are substantial communication overheads. In some cases, it may be possible to model communication overheads as part of the computation. However, it is not clear where to include these overheads and what affect this has on the parameters to the model. Finally, it

also differs from more recent work on modelling processor farms and divide and conquer on shared memory multiprocessors. In this case the overheads being modelled are quite different and cannot be applied to the cases we have considered [10]. In addition, the analysis is not an asymptotic one like[4], and takes into account the actual overheads in the system.

The paper is organized as follows. A description of the system is given in Section 2. Section 3 begins with a general framework for analyzing tree structured computation. This is followed by an analysis of the processor farm and divide and conquer paradigms. Section 4 describes the system and the results of the experiments that were performed. Finally, in Section 5, we describe several practical applications of the use of this model.

2 The hardware and software system

In this section we describe the system that we are modelling.

2.1 Hardware parameters

There are a number of parameters that can be used to describe the performance of a multicomputer architecture. Following Stone [16], we can categorize the execution time of the program (i.e., cost of execution) into the computation time and the communication time. Communication can be further broken down into that part that can be overlapped with computation and that part that cannot be overlapped. Finally, execution costs can be further categorized by whether or not they are charged on startup or on a per/byte (or per/bit) basis.

A summary of these costs are shown in Table 1. Startup costs are expressed as a fixed time unit (microseconds) whereas byte costs are given as a computation or communication rate (bytes/second). The three parameters shown in the table are the primary costs that we investigate. In particular, we have found that non-overlapped start-up cost of communication

	Computation	Communication	
		overlapped	non-overlapped
startup	—	—	β
per/byte	e	τ	—

Table 1: Hardware performance parameters

is an important cost that can limit speed-up irrespective of the number of processors used. Where appropriate we indicate how the costs we have not considered affect the analysis.

2.2 Software System

The application software is represented as a collection of identical tree-structured task graphs. Figure 1 shows an example of the type of task graph executed by the system. Following the

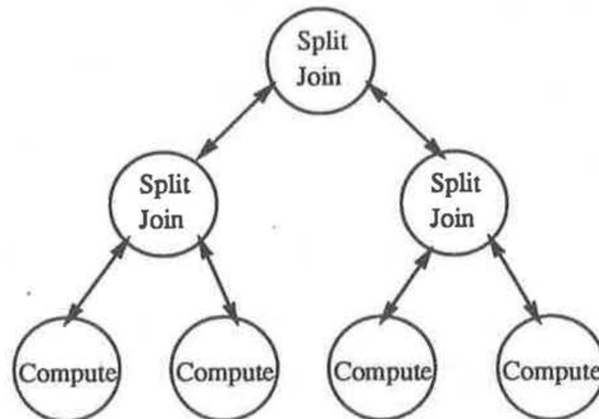

Figure 1: Software Structure

terminology of Cole [4] we can identify three generic computational processes for divide and conquer: compute, split, and join. The split and join processes occur at the internal nodes of the tree. They create subtasks and combine the results of subtasks. Tasks are completed by the compute process at the leaf nodes of the tree.

Tasks enter the system at a single root processor. The remaining processors request tasks

from its parent processor. The parent of a processor is assigned statically at compile time and may not correspond exactly to the physical configuration. On receipt of a task, a processor either processes the task locally until completion, or divides the task into subtasks that are then forwarded to its children processors. Subtasks are put on a single queue where requests from children processors compete for the tasks on this queue. Subtasks return their results to their parents. The three basic characteristics of the system are: there is a flow of tasks, tasks are dynamically scheduled, and tasks are distributed from a single source.

An analytic model of this system could be used in several ways. First to determine the peak operating point of the system [5]. This is the point beyond which adding more processors does not substanially increase performance. The steady state of this model is particularly important for applications, like computer vision, where there is a steady stream of tasks into the system. Secondly, the model can be used to determine the affect of granularity on performance. As we will show in Section 5, this model can be used to restructure the program so as to increase granularity and improve performance. Finally, the model can be used to compare different topologies and software structures for executing a given application.

3 Analytic models

In this section we present a general framework for analyzing the system introduced in the previous section. We then derive analytic performance models for processor farms and divide and conquer.

3.1 General framework

Let $T = (V, E)$ be a tree structured task graph and let M denote the total number of tasks to be executed. Let φ denote the parent relation of the topology (i.e., $\varphi(i) = j$ specifies

that processor j is the parent of i). Let T_s, T_e and T_j denote the computation time for splitting, executing and joining tasks, respectively. The model assumes that the split, join, and execution computation is identical for nodes on the same level of T. Later in Section 4.3, we give the results of experiments for when this is not the case. This permits us to write an expression for $T_{comp}(i)$, the execution time on ith processor, in terms of the number of splits, joins and executions done at this processor.

$$T_{comp}(i) = C_e(i)T_e(i) + C_s(i)(T_s(i) + T_j(i))$$

where $C_e(i)$ and $C_s(i)$ are the number of tasks processor i computes and splits, respectively.

Similarly, we can write an expression for $T_{comm}(i)$, the non-overlapped communication start-up costs, in terms of the number of tasks executed locally, the number of tasks forwarded and the respective overheads.

$$T_{comm}(i) = C_e(i)\beta_e + C_s(i)\beta_f$$

where β_e is the communication start-up overhead for executing the process locally and β_f is the start-up overhead for splitting a task and later joining its result.

The total time T_i spent by the ith processor assuming that the processor is either processing a task locally or splitting a task so that the subtasks can be forwarded is given by

$$T_i = T_{comp}(i) + T_{comm}(i).$$

Now, let $C_e(i) = f_iM$, where M is the total number of tasks and f_i is the fraction of the total tasks executed by the ith processor. Let $C_s(i) = F_iM$, where F_i is the fraction of the total tasks split and forwarded by the ith processor. Therefore,

$$T_i = f_iM\,(T_e(i) + \beta_e) + F_iM\,(T_s(i) + T_j(i) + \beta_f)\,.$$

Additionally there is the condition that

$$\sum_{i=1}^{N} f_i = 1.$$

For the steady state analysis, we assume an ideal task distribution in which processors are never idle as long as there are unprocessed tasks in the system. With the above assumption, all T_i's will be equal and can be set to T. The F_i's can be written as a linear combination of the f_i's. The two overheads for communication, β_e and β_f, depend on the underlying hardware and software system. For a given implementation, their values are assumed to be constant and can be measured experimentally. Altogether then we have a set of $N+1$ linearly independent equations with $N+1$ variables - T and the f_i's. Therefore this set of equations has a unique solution, specifying both the ideal distribution of tasks to processors and the total execution time of the system.

The previous analysis holds only when the transfer time across the communication links is not a limiting factor. In this case, assuming that τ is the same for all links, throughput is limited by the rate at which the the first processor can receive tasks, irrespective of the number of processors or their processing capabilities. The maximum throughput that can be obtained is

$$\frac{1}{T_\tau + \beta_e},$$

where T_τ is the transfer time for a task.

In the remainder of this section we apply this framework to the processor farm and divide and conquer paradigms. For the most part, a balanced binary tree topology is used for the analysis. The recursive structure of this topology allows us to avoid explicitly solving the system of equations that arise from the framework. However, in general, for different topologies or more irregular tree structured computation it is necessary to explicitly solve the

system of equations to obtain the distribution of tasks to processors.

For each class of programs we first analyse the steady state and then give an analysis for the start-up costs.

3.2 Processor Farm

One can view a processor farm as a degenerate case of divide and conquer where the tree structure now consists of only a single node. In this case, processors can choose whether to execute the task locally or forward it onto a child processor. Although we give the analysis for a binary tree topology it can easily be extended to linear chains and k-ary trees.

Since tasks in this model are never split, we have for all processors i, $T_e(i) = T_e$, where T_e is the processing time of the task, and $T_s(i) = T_j(i) = 0$.

3.2.1 Steady State

Assume that processors on the same level of the tree execute the same number of tasks and let f_i (for $i = 1 \dots N$) denote the fraction of the total number of tasks executed by the processors on the ith level of an N level tree. Levels are numbered from 1 at the leaves to level N at the root.

The total execution time of a processor on level i of the tree is given by

$$T_i = \frac{M}{2^{N-i}} \left[f_i \left(T_e + \beta_e\right) + \sum_{j=1}^{i-1} f_j \beta_f \right],$$

where f_i is the fraction of the tasks executed locally and $\sum_{j=1}^{i-1} f_j$ is the fraction of the M tasks forwarded to the next level of the tree. We assume that tasks are evenly distributed to the 2^{N-i} processors on the ith level of the tree.

Let

$$F_i = \frac{1}{2^{N-i}} \sum_{j=1}^{i} f_j, \tag{a}$$

where F_i represents the fraction of the total tasks processed by the processors in a subtree rooted at a processor in the ith level. Rewriting T_i in terms of F_i and F_{i-1},

$$\begin{aligned} T_i &= M(T_e + \beta_e)(F_i - 2F_{i-1}) + M\beta_f(F_i - (F_i - 2F_{i-1})) \\ &= F_i M(T_e + \beta_e) + 2F_{i-1} M(\beta_f - T_e - \beta_e). \end{aligned}$$

By rearranging the above,

$$F_i = 2F_{i-1}\left(\frac{T_e + \beta_e - \beta_f}{T_e + \beta_e}\right) + \frac{T_i}{M}\frac{1}{(T_e + \beta_e)}. \tag{b}$$

Let

$$S_i = \frac{MF_i}{T_i} \tag{c}$$

denote the throughput of a subtree rooted at one of the processors in the ith level. Substituting (c) in (b) with the condition that for all i, T_i's are equal, we have

$$S_i = 2S_{i-1}\left(\frac{T_e + \beta_e - \beta_f}{T_e + \beta_e}\right) + \frac{1}{(T_e + \beta_e)}. \tag{d}$$

Solving the above recurrence, we obtain the steady state throughput of the system

$$S_N = \frac{1}{2\beta_f - \beta_e - T_e}\left[1 - \left(\frac{2(T_e + \beta_e - \beta_f)}{T_e + \beta_e}\right)^N\right]. \tag{e}$$

By substituting (a) in (b) and solving the recurrence we can calculate the fraction of tasks executed by the ith processor,

$$f_i = 2^{N-i}\left(\frac{2a^{i-1} - a^i - 1}{1 - a^N}\right),$$

where

$$a = \frac{2(T_e + \beta_e - \beta_f)}{T_e + \beta_e}.$$

In general, a similar analysis gives the throughput of any balanced k-ary tree to be

$$S_N = \frac{1}{(T_e + \beta_e) - k(T_e + \beta_e - \beta_f)} \left[1 - \left(\frac{k(T_e + \beta_e - \beta_f)}{T_e + \beta_e} \right)^N \right]. \quad \text{(f)}$$

The analytical results obtained are similar to the steady state results by Pritchard[14]. Our model differs from Pritchard's in that it considers the software overhead for scheduling and distributing tasks. Pritchard's model neglects this cost and is based only on the hardware communication overheads. Whereas there is only a single hardware overhead for setting up communication, we found that two overheads, β_e and β_f, had to be included in the model. Experimentation showed that these two overheads are distinct from each other and cannot be combined into a single cost.

3.2.2 Startup Cost

The analysis of the previous section is for steady state and holds only when no processor is idle. In particular, the analysis does not hold on system startup. For applications with large number of tasks, and in turn large execution time, startup time can be neglected compared to the total execution time. For the cases in which both the number of independent tasks and the total execution time are small, it is necessary to consider the startup time. Startup time is more difficult to predict because it is affected by the scheduling strategy used to distribute the tasks. Let us define the startup time to be the time it takes for the system to return the first result. The time it takes for the first task to be passed to the Nth level of the tree and then return the result is given by,

$$T_{startup} = (N-1)(2T_\tau + \beta_f) + T_e + \beta_e.$$

This assumes that T_e is sufficiently large that after $T_{startup}$ all processors have a task and that afterwards the tasks return at the rate given by the formula for steady state.

Therefore the execution time of an N level system is

$$T = T_{startup} + \frac{M-1}{S_N}. \qquad \text{(g)}$$

By substituting, we obtain the following expression for the speedup of a N level k-ary tree:

$$SP_N = \frac{MT_e(1-a^N)}{(M-1)(T_e+\beta_e) - k(T_e+\beta_e-\beta_f) + T_{startup}(1-a^N)}$$

where

$$a = \frac{k(T_e+\beta_e-\beta_f)}{T_e+\beta_e}.$$

3.3 Divide and Conquer Paradigm

We now consider the more general case of divide and conquer. Again we ignore additional routing overheads and assume that the processor topology is a tree with the same degree as the task structure graph. We again derive the throughput formula where the task graph is a binary tree. Once again, the analysis easily extends to arbitrary degree trees.

The model differs from many of the models that have appeared in the literature[13, 4] in that the internal processors of the tree topology also does some of the computing.

3.3.1 Steady State

Let N be the number of levels of the hardware topology. Let f_i represent the fraction of the total number of tasks processed by the processors at the ith level. As before, levels are numbered starting from 1 at the leaves. Now, we can write, in terms of f_i, a general expression for the execution time of a processor on level i of the tree.

$$T_i = f_i M\left[T_e(i) + \beta_e\right] + \left(\sum_{j=1}^{i-1} f_j\right) M\left[T_s(i) + T_j(i) + \beta_f\right],$$

where $\sum_{j=1}^{i-1} f_j$ is the fraction of the tasks forwarded to the next level. Let

$$F_i = \sum_{j=1}^{i} f_j,$$

where F_i represents the fraction of the total tasks executed by all the processors in levels 1 through i. Then rewriting T_i in terms of F_i and F_{i-1},

$$\begin{aligned} T_i &= (F_i - F_{i-1})M[T_e(i) + \beta_e] + F_{i-1}M[T_s(i) + T_j(i) + \beta_f] \\ &= F_i M[T_e(i) + \beta_e] + F_{i-1}M[T_s(i) + T_j(i) + \beta_f - T_e(i) - \beta_e] \end{aligned}$$

By rearranging the above,

$$F_i = F_{i-1}\left(\frac{T_e(i) + \beta_e - T_s(i) - T_j(i) - \beta_f}{T_e(i) + \beta_e}\right) + \frac{T_i}{M}\frac{1}{(T_e(i) + \beta_e)} \qquad \text{(h)}$$

We have

$$S_i = \frac{MF_i}{T_i}, \qquad \text{(i)}$$

where S_i represents the throughput of a subtree consisting of the processors from levels 1 to i, and $S_0 = 0$. By using relation (i) in (h) with the condition that all T_i's are equal and rearranging we obtain,

$$S_i = S_{i-1}\left(\frac{T_e(i) + \beta_e - T_s(i) - T_j(i) - \beta_f}{T_e(i) + \beta_e}\right) + \frac{1}{(T_e(i) + \beta_e)}. \qquad \text{(j)}$$

We do not have a closed form solution to this recurrence. The throughput of the complete tree can be obtained by recursively evaluating the S_i's up to level N.

Eventually, for sufficiently large N, throughput is limited by the rate at which subtasks can be distributed to the leaf processors. This throughput is given by the inverse of the maximum time spent by a task on any intermediate node before reaching the leaf nodes, i.e., the maximum of the sum of $T_s(i)$, $T_j(i)$ and β_f among all the nodes in levels 2 to N. Let

$$t_{max} = \max_{2 \leq i \leq N} \left(T_s(i) + T_j(i)\right).$$

Then the throughput limit, S_{max}, is given by the following expression

$$S_{max} = \frac{1}{(t_{max} + \beta_f)}. \tag{k}$$

If S_1 obtained by equation (j) is less than S_{max}, then S_N obtained by the same equation is the actual throughput that can be achieved. Otherwise, the throughput is S_{max} and in this case, the intermediate nodes should not be used for processing subtasks as this will not increase the throughput.

3.3.2 Startup cost

For an N level tree, the startup cost

$$T_{startup} = \sum_{i=2}^{N} (2T_\tau(i) + T_s(i) + T_j(i) + \beta_f) + T_e(1) + \beta_e,$$

where $T_\tau(i)$ is the actual transmission time of data for a subtask at the ith level. The speedup of an N level tree is given by,

$$SP_N = \frac{MT_e}{T_{startup} + \frac{M-1}{S_N}},$$

where S_N is the steady state throughput as given by equation (j).

4 Validation of the model

The analytic models in Section 3 were validated by experiments on a 74 node T800 transputer based multicomputer system. The system has several programmable crossbar switches that are used to configure the system into different topologies. Each transputer consists of a processor with its own local memory and four bidirectional bit-serial communication links. Data transfer on the links can be overlapped with the computation, but link setup time can not be overlapped and requires at least 50 clock cycles. The T800 has a microcoded scheduler

with both a low-priority and a high-priority process queue. Experiments were done using Logical Systems C[12] and the Trollius Operating System[3].

To validate our performance model, experiments were conducted with various model parameter values. In our performance models we have assumed that the tasks are distributed to the processors such that no processors will be idle during the steady state, i.e., a new task is always immediately available as soon as a processor finishes its current task. To emulate this ideal task distribution situation, we have used a "flood-fill" technique which is described in the next section.

4.1 Task Distribution Strategy

The analysis in section 3 determines the fraction of the total number of tasks to be executed on each processor. However, it does not determine an exact schedule. There is however an efficient implementation of the system that is closely approximated by the analytic model.

The optimal task distribution can be achieved by flood-filling the system and using buffers to control the rate at which tasks flow into the system. The split and join processes execute at high priority while the execute process is a low priority process. Thus, creating and forwarding new tasks is always given higher priority over executing the tasks locally. Each processor is provided with a queue for storing unprocessed tasks. These buffers are in addition to the task being processed locally and the task (or tasks) being forwarded to the successor node (or nodes). As soon as a task has been executed or forwarded, the execute or split process takes another task from the queue. A processor accepts new tasks as long as the queue is not full, when full it blocks, waiting for a free buffer. The size of this queue indirectly controls the flow of tasks into the system. If the queue is large, then the processors away from the root receive a large number of tasks, and thus the processors closer to the root may become

idle at the later stages of the processing. So, to ensure that no processor idles, the size of the queue must be large enough to supply a steady stream of tasks yet small enough so as not to overload the leaf processors. For sufficiently large M, a queue size of one or two was enough to ensure that the execution model closely approximated the analytical model.

For small M, even a queue size of one can overload the leaf processors and lead to a suboptimal distribution of tasks. In this case, we can use the $f_i M$'s obtained from the model to control the flow of tasks. Processor i forwards tasks as long as the number of forwarded tasks does not exceed $F_{i-1}M$.

4.2 Experiments

In order to compare the analytical model with the actual execution it is necessary to determine β_e and β_f. These constants depend on the underlying machine and system software. They were determined experimentally, for several sets of test parameters, by substituting the measured throughput in the model and then solving for β_e and β_f. The values obtained were consistent over the different test parameters used.

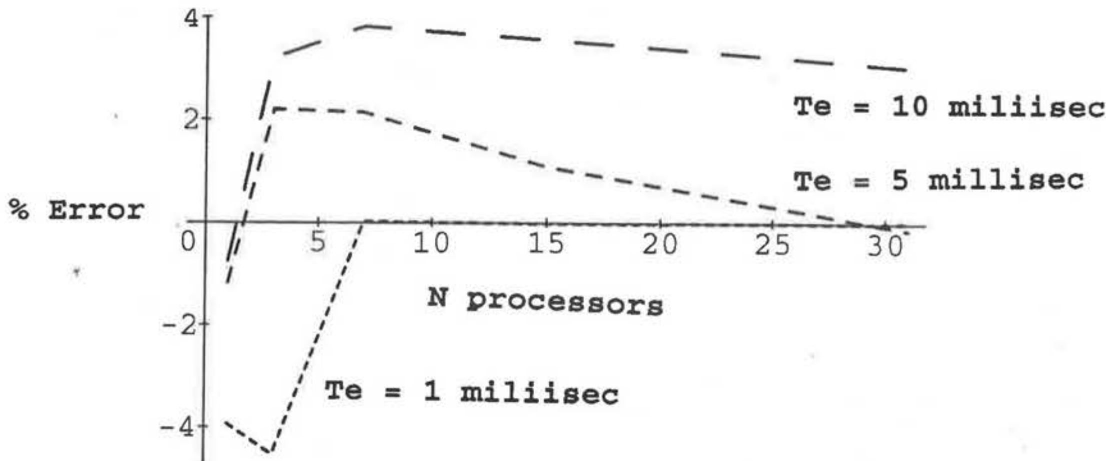

Figure 2: Comparison of predicted and measured speedup

Having determined β_e and β_f we ran several experiments to validate the analytic model. Figure 2 shows the percentage difference in speedup for the cases with T_e = 1, 5, and 10 milliseconds for the processor farm paradigm running on a binary tree. As shown in Figure 2, the difference between the measured speedup and the speedup predicted by the model does not exceed 5%. The experimental results for other cases such as processor farm running on linear chain and divide and conquer running on a binary tree exhibited similar behaviours for a range of input parameters.

4.3 Robustness

An assumption of our model is that T_e is constant. To test the robustness of the model with respect to this parameter, we tested two different distributions of task size, uniform and exponential, against the results obtained by the model using the average task size. The percentage difference between the measured and analytic speedup for exponential and uniform distributions with average T_e = 10 msec for different number of processors is given in Figure 3. Errors were again under 5% for both exponential and uniform distributions.

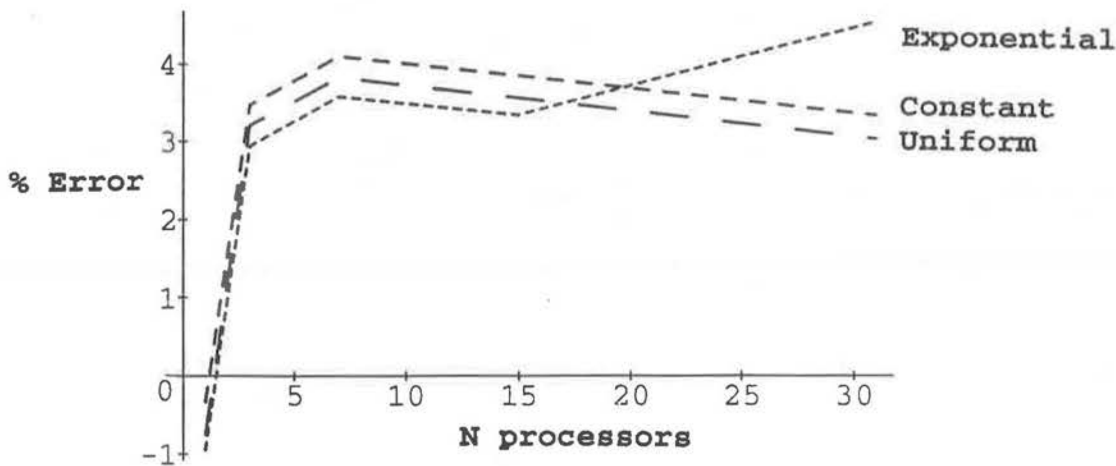

Figure 3: Comparison of predicted and measured speedup

For the divide and conquer case, we have assumed in the model that the tasks were split into equal size subtasks. To test the robustness of the model we conducted experiments in which the tasks were split randomly into two parts. For this case also, the error did not exceed 5%.

5 Discussion

In this section, we give two ways in which the models we have developed can be used.

The first way to use the model is to obtain the peak operating point of the system. Figure 4 shows a graph of speedup as a function of N, the number of processors connected as a linear chain, for three values of T_e, the processing time.

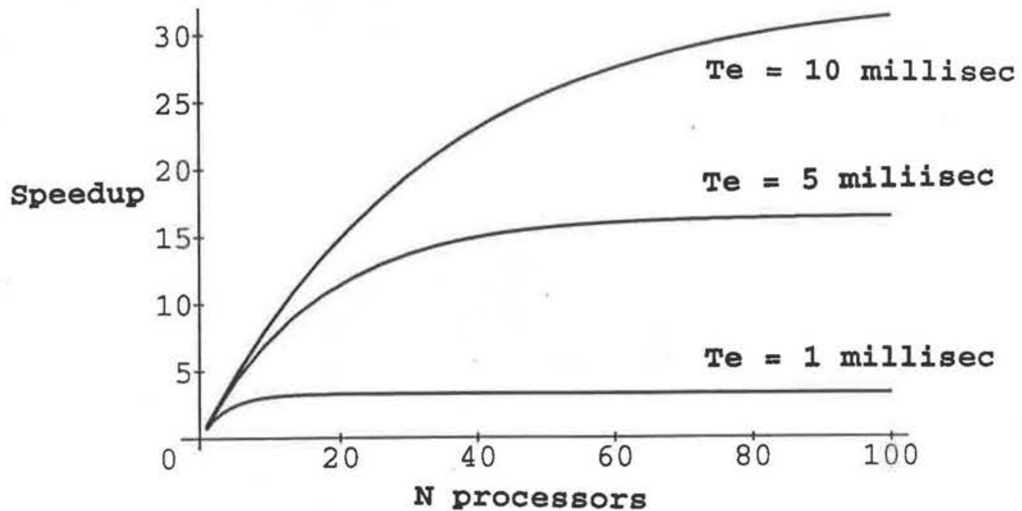

Figure 4: Speedup for various size T_e and N

Observe that for fixed T_e, the speedup increases with N and remains relatively constant after some value of N. This can be used to determine the optimal size of the system to be used for a given application. If more physical nodes are available than the optimal N for a given application, the application can be split into two or more parts and run simultaneously on two or more separate topologies. This would improve overall performance by effectively

making use of all the nodes available.

A second way to use the model is for adjusting the granularity of the application. Observe from figure 4 that as computation time T_e increases the optimal value of N increases. This is due to the fact that with larger T_e, the ratio of computation to non-overlapped communication overhead increases making it possible to effectively use a larger number of nodes. Thus, by packetizing a number of tasks into one message and sending it as a single unit it is possible to reduce the overall non-overlapped overheads and thereby increase the throughput. There is however a limit to the number of tasks that can be packed together. The message transmission time increases with the number of packets per message and at some point it may no longer be possible to overlap communication with computation. Larger messages also increase start-up costs. Figure 5 shows a graph of speedup, with startup included, as a function of granularity and N. Here granularity is the number of original tasks put into a message.

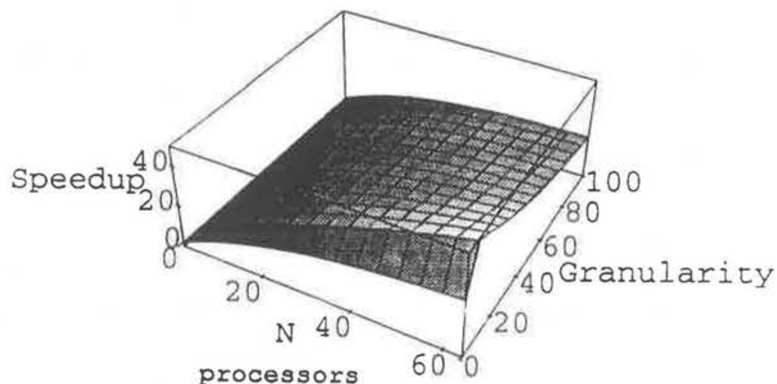

Figure 5: The affect of granularity on speedup

From this figure, observe that overall speedup starts to decrease after a certain value of N, as granularity increases. The optimal operating point of the system is a complex function of N, T_e and the granularity.

6 Conclusions

Our goal is to obtain accurate performance prediction models for various classes of applications on multicomputers. We have presented an efficient execution model and have developed analytic performance prediction models for two paradigms with this execution model. We have validated these models with a large number of experiments covering a broad range of values for the model parameters. Startup costs are important for small applications and have to be included in the performance model. We have shown how these models can be used to obtain the peak operating point of the system and to improve performance by adjusting the granularity of the application. The results of the model are also used to achieve the optimal distribution of tasks to processors when the total number of tasks is small. The same system software used for validating the model has been used to implement a computer vision application and a second numerical application. Further work will focus on extending this work to other parallel programming paradigms and integrating performance prediction models into our existing parallel program development environment.

References

[1] G. M. Amdahl. Valildity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*. AFIPS Press, Reston, Va., 1967.

[2] William C. Athas and Charles L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, August 1988.

[3] Moshe Braner, Gregory D. Burns, David L. Fielding, and James R. Beers. Trollius OS for Transputers. In D. Stiles, editor, *Transputer Research and Applications (NATUG 1)*, 1989.

[4] Murray Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, Cambridge, Massachusetts, 1989.

[5] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, March 1989.

[6] Anthony J.G. Hey. Experiments in MIMD parallelism. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE: Parallel Architectures and Languages Europe*, pages 28–42. Springer-Verlag, New York, 1990. Lecture Notes in Computer Science 366.

[7] H. T. Kung. Computational models of parallel computers. In R. J. Elliot and C.A.R. Hoare, editors, *Scientific applications of multiprocessors.* Prentice Hall, 1989.

[8] H.T. Kung et al. iWARP: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88, Orlando, FL.* IEEE Computer Society Press, 1988.

[9] INMOS limited. *Transputer development system.* Prentice Hall, New Jersey, 1988.

[10] Sridhar Madala and James B. Sinclair. Performance of synchronous parallel algorithms with regular structures. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):105–116, January 1991.

[11] D. L. McBurney and M. R. Sleep. Transputer-based experiments with the ZAPP architecture. In J. W. de Bakker et al, editor, *Lecture notes in Computer Science.* 1988.

[12] Jeffrey Mock. Process, Channels and Semaphores (version 2). Logical Systems C Programmers Manual, Logical Systems, 1988.

[13] P. A. Nelson. *Parallel Programming Paradigms.* Ph.D. thesis, Department of Computer Science , University of Washington, Seattle, WA, 1987.

[14] David J. Pritchard. Performance analysis and measurement on transputer arrays. In Aad van der Steen, editor, *Evaluating Supercomputers.* Chapman and Hall, 1990.

[15] Daniel A. Reed and Richard Fujimoto. *Multicomputer Networks: Message Based Parallel Processing.* MIT Press, 1987.

[16] Harold Stone. *High Performance Computers.* Addison-Wesley, 1988.