Performance Models for the Processor Farm Paradigm

Alan S. Wagner, *Member, IEEE Computer Society*, Halsur V. Sreekantaswamy, and Samuel T. Chanson, *Member, IEEE Computer Society*

Abstract—In this paper, we describe the design, implementation, and modeling of a runtime kernel to support the processor farm paradigm on multicomputers. We present a general topology-independent framework for obtaining performance models to predict the performance of the start-up, steady-state, and wind-down phases of a processor farm. An algorithm is described, which for any interconnection network determines a tree-structured subnetwork that optimizes farm performance. The analysis technique is applied to the important case of *k*-ary tree topologies. The models are compared with the measured performance on a variety of topologies using both constant and varied task sizes.

Index Terms—Parallel programming paradigms, performance evaluation, processor farm, tree networks, message passing architecture, network flow, master-slave.

1 Introduction

The major problems in parallel computation revolve around questions of ease of use, reuse, and the seemingly contradictory goal of performance. One recent approach to reconciling ease of use and reuse with performance is the construction of software components (libraries, templates [23], skeletons [10], or runtime kernels) based on the parallel programming paradigms that have appeared in the literature [18], [8]. Software components based on these paradigms hide the complex distributed system code needed to implement the paradigm and allow the programmer to concentrate on the computation rather than the parallelization and the coordination of the multiple processes.

One paradigm that is commonly used on multicomputers[1] is the processor farm paradigm. The term processor farm was first used by May et al. [20] to describe a strategy for the parallel execution of programs that consist of a single task to be executed on a collection of initial data [16]. Descriptions of farms and their use can be found in several introductory books on parallel programming [9], [27], [7]. Examples of its use for simulation, biological sequence comparisons, and signal analysis can be found in [11], [24], [3].

One common approach to supporting the processor farm paradigm on multicomputers is to use a runtime kernel to handle the distribution of tasks and the gathering of results. The user creates a manager program to create the tasks and a worker program to execute tasks. This is compiled (or linked) with a kernel which creates the worker processes and, on demand, sends tasks and receives results on behalf of the manager program. All of the underlying process management and message-passing is hidden and independent from the application program.

Although this provides the user with a flexible, almost universal, strategy for parallelizing an application, performance varies greatly, and depends not only on the application, but also the runtime kernel and underlying machine. To effectively utilize the system, it is important to identify the key parameters which affect performance and to characterize the performance of the farm in terms of these parameters. Understanding the behavior and performance characteristics of the paradigm makes it possible to

1) predict performance and tune the program,
2) evaluate its use with regard to the limitations of the software and hardware, and
3) aid in the design of processor farms on different systems and topologies.

In this paper, we describe the design, implementation, and performance modeling of a runtime kernel to support the processor farm paradigm. Our main contribution is a topology-independent model that characterizes the performance of farms in terms of the application, runtime kernel, and machine.

In Section 2, we describe *Pfarm*—a runtime kernel for the processor farm paradigm—and compare it to other designs that have appeared in the literature. A general topology-independent framework for obtaining performance models for the communication and computation bound case is given in Section 3. In addition to the steady-state performance of the system, we also analyze the start-up and wind-down phases of the system and present an algorithm,

1. Processors, each with their own memory, interconnected by a message-passing network.

- *A.S. Wagner is with the Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada, V6T 1Z4. E-mail: wagner@cs.ubc.ca.*
- *H.V. Sreekantaswamy is with Nortel (Northern Telecom), PO Box 3511, Station C, Ottawa, Ontario, Canada, KIY 4H7.*
- *S.T. Chanson is with the Hong Kong University of Science and Technology, Kowloon City, Hong Kong.*

Manuscript received 10 Mar. 1994.
For information on obtaining reprints of this article, please send e-mail to: transtpds@computer.org, and reference IEEECS Log Number D95239.

which for any interconnection network determines a tree-structured subnetwork that optimizes farm performance. Section 4 applies our analysis technique to the case of *k*-ary tree topologies. In Section 5, the experimental setup is described and we compare the performance predicted by the models to that of the system. Results are given for five applications with both constant and varied task sizes on several different machine topologies. A synthetic benchmark is used to investigate the validity of the model over a wider range of parameter values. Examples of how the model can be used to optimize the topology or tune the application program are given in Section 6.

2 Software Overview

Consider an arbitrarily connected collection of processors. Designate one of the processors to be the manager and let the remaining processors be workers. Assume there is a link connecting the manager processor to the root of a tree spanning the worker processors.

The system software consists of a manager program, a worker routine, and a runtime kernel which we call *Pfarm*. The manager program is loaded onto the manager processor while the worker routine is linked with *Pfarm* and loaded onto each of the worker processors. *Pfarm* creates and initializes the processes on the worker, handles the distribution of tasks and gathering of results, and invokes the user defined worker routine to locally process a task. *Pfarm* uses nearest neighbor communication in a spanning tree of the underlying physical interconnection network to distribute tasks and gather results. Tasks enter the system from a single source, the link from the manager, and are either processed locally or forwarded, on demand, to a neighboring worker (i.e., one of its children).

The process structure for *Pfarm* running on each worker processor is shown in Fig. 1. Notice there is a communication process (`InLink/OutLink`) associated with every unidirectional link. These processes block on a send or receive and are dequeued, thus allowing the CPU to remain active while data is being transferred on the links. There is a separate `worker` process so that computation can be overlapped with communication.

The purpose of the two `manager` processes is to provide buffering for the `worker` process. Some buffering is necessary, since otherwise the processor may idle while waiting for another task or passing on results. However, as long as the time to receive a task from the parent is less than the time to process a task, a single buffer in the `task manager` suffices. It is important not to use more buffers, since it reduces the system's load-balancing ability by overloading processors farthest from the root (discussed in Section 3.3). Buffering also affects response time, which is important for applications that can make immediate use of the results or those applications with dependencies between tasks such as the use of farms to evaluate recursive functions [6]. As long as the result messages do not saturate the communication links it is not as important to restrict the number of buffers used to return the results.

Consider the total number of tasks in an N processor tree-structured system. Each node in the system has the process structure shown in Fig. 1, where each process to the right of the vertical line (including the `worker` process) has at most one task. Assuming the manager processor also has an `OutLink` process containing a task, there are at most two tasks at the ends of each `OutLink`→`InLink` arc. There are in total N arcs in the tree ($N-1$ in the tree and one arc connecting the host to the tree) or in total $2N$ tasks (one at each end of every arc). In addition, each node has at most one task in the `manager` process and the task being executed by the `worker` process. Therefore, an N processor system contains at most $4N$ tasks.

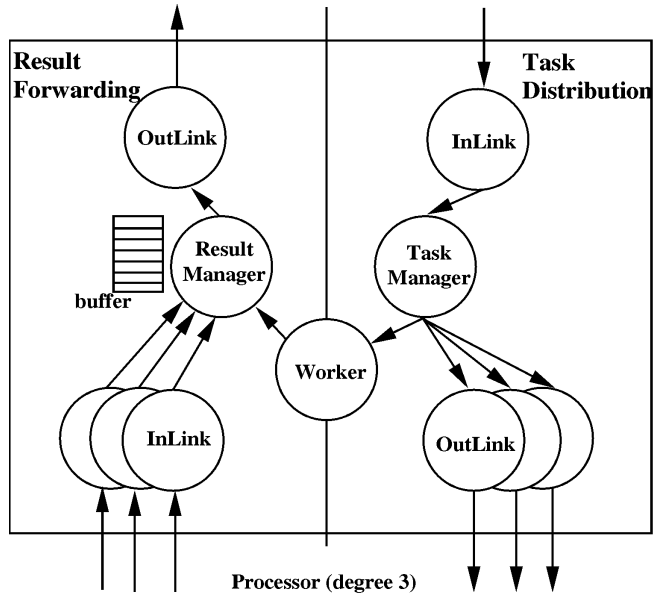

Fig. 1. *Pfarm* process structure for a single processor with four bidirectional links.

All communication processes, including the two manager processes, run at high priority while the worker runs at low priority. For scalability, it is important to expedite the forwarding of tasks and results. To avoid idle time, priority is given to assigning tasks to the `worker` process over the `OutLink` processes. Tasks are assigned to idle `OutLink` processes in round-robin fashion. The importance of round-robin scheduling of tasks to links is discussed in Section 3.3.1.

An aspect of the design that was influenced by the model (Section 3) was the assumption of a constant time overhead (per processor) for task forwarding and processing. By passing pointers, the system overhead can be made independent of task/result data sizes. However, because of the transputer ALT instruction, the overhead does vary slightly with degree. For small degree (2–5), the difference is small relative to the total overhead involved.

In describing the design, we have focused on the issues of parallelization, scheduling, and buffering, which are important to the implementation of a processor farm on most multicomputers. A discussion of more transputer specific design issues can be found in the literature. Day [13] measured the improvement in simultaneously using the links and compared the multiple process versus multithreaded versions of farms. Lau and Lau [19] has measured the performance of the occam ALT construct and has given a more

efficient implementation which would substantially decrease the difference between varying degree nodes in the processor tree. These enhancements would decrease the constants in *Pfarm* but would not change the overall model.

Thus far we have considered only the distributed manager organization and not the centralized organization in which requests for work are sent to a central manager process that assigns the tasks. Usually, as in Hey [16], these requests are routed through the network back to the manager. This type of system was analyzed by Pritchard [21] and Tregidgo and Downton [25]. An obvious disadvantage of the centralized scheme is that the manager can become the bottleneck. A second disadvantage is that the amount of buffering needed on each node varies according to the message latency between the manager and worker. Varied buffering complicates the start-up and wind-down analysis of the system. It also uses more buffers than the distributed control system, which, as shown in Section 3.3, detrimentally affects performance. The buffering problem can be avoided with the use of special hardware, like programmable crossable switches, to dynamically reconfigure the system to load and drain tasks from the workers [17]. However, the need for special hardware makes reconfigurable farm networks generally less applicable.

We have not considered processor farms which can be dynamically created like those in the PUL project [14]. Dynamic farms are more flexible, but are more difficult to configure and, if not mapped to its own collection of processors, there is the added complication of understanding the behavior of two or more interacting systems. Restricting farms to dedicated, statically configured systems makes it easier to predict performance, thus making it easier to provide both guidance and some guarantee of performance.

We have also not considered the static case in which the problem can be decomposed and assigned to processors prior to execution. Although *Pfarm* can be used in this case,[2] the system overhead of *Pfarm* is likely larger than the overhead of a system specifically tailored to static decomposition. However, not all problems can, or should, be decomposed in this way. When workload varies, by decomposing the problem into more tasks than processors it is possible to better balance the load. In other cases, the tasks may not be initially available or memory constraints may make it necessary to use more tasks than processors. For these cases, a demand-driven task allocation strategy, such as that used in *Pfarm*, is an effective way to balance the workload.

The design of *Pfarm* is based on the following assumptions about the underlying processor. It is assumed that data can be transferred simultaneously on all links and data transfer can be overlapped with computation. In addition, it must be possible to preempt execution to service communication requests and avoid polling. Support for concurrent processes, process scheduling, and process priorities could be available in hardware or software.

In summary, *Pfarm* implements a distributed control, demand-driven, tree-structured processor farm with tasks entering and results returning to the root. The system is topology independent and can be loaded onto any fixed network. However, performance depends on the topology and the overhead of distributing tasks and gathering results. A model to describe this overhead and the affect of other system and machine parameters is presented in the next section.

2. When the number of tasks matches the number of workers, each worker executes exactly one task.

3 Modeling

The performance of the system cannot be described by a single model and depends on whether or not the system is communication bound or computation bound. The communication bound case is analyzed in Section 3.1 and the more interesting computation bound case is analyzed in Sections 3.2 and 3.3. The following variables are used to denote the different parameters of the system.

- M is the total number of tasks.
- β_f is the CPU overhead within a single node for forwarding tasks. It includes all the CPU overhead in receiving a task, forwarding it to a child processor, receiving the corresponding result and forwarding it to the parent processor.
- α is the local CPU time necessary to locally execute a task. α is further divided into T_e and β_e, the task execution time and the associated overhead time.
- D_t, D_r are the data size of a task and result, respectively.
- N is the total number of processing nodes.
- τ is the link transfer rate between neighboring processors.[3]

We assume a homogeneous, distributed memory, message passing architecture. It is assumed that there are M identical tasks available at the source that flow into the root of the system. Later, in Section 5.7, we show that the models remain robust when T_e is replaced by average task size. It follows from the design that there is a single source and sink for both tasks and results, and that tasks are dynamically distributed to processors in the manner outlined by the process structure in Fig. 1. Finally, as mentioned in Section 2, it is assumed that β_f and β_e are constants. The objective is to model *Pfarm* to predict the execution time as accurately as possible given M, β_f, and α on a tree of N processors.

3.1 Communication Bound Model

The system is communication bound when performance is limited because a critical communication resource is saturated. The two critical resources in *Pfarm* are the communication links which send and receive data from the root processor. Message traffic on the links is highest at the root and decreases towards the leaves of the tree.

Let C_R be the minimum time between successive receives of the `InLink` process and C_S be the minimum time between successive sends in the `OutLink` process. An upper bound on execution time is given by

$$T_{comm} = M \times \max\{C_R + \tau D_r, C_S + \tau D_t\}. \quad (1)$$

3. Only nearest neighbor communication is necessary since *Pfarm* can be used on any spanning tree of the network. Nonnearest neighbor communication can also be used, but the model does not account for delays due to distance or traffic.

In the case of nonidentical links, a modified shortest path algorithm can be used to locate the bottleneck to obtain a similar bound.

3.2 Steady-State Analysis

The steady-state phase begins once all the processors have a task to execute and ends when the last task enters the system. An important assumption of the analysis is that no processor idles during this phase—every processor is either processing a task or busy forwarding tasks and results. We also assume that $\alpha > \beta_f$, since otherwise it is faster to execute the tasks sequentially rather than in parallel.

Let T be a tree architecture with processors $p_1, \ldots, p_N$. Let $C(i) = \{j \mid p_j \text{ is a child of } p_i\}$ denote the children of p_i in T. Let V_i denote the number of tasks that visit p_i. Since no processor idles, the following condition holds for all processors in T,

$$T_{ss} = \alpha\left(V_i - \sum_{j \in C(i)} V_j\right) + \beta_f \sum_{j \in C(i)} V_j = \alpha V_i + \left(\beta_f - \alpha\right) \sum_{j \in C(i)} V_j \quad (2)$$

That is, the steady-state execution time (T_{ss}) equals the processing time with the associated overhead (α) for all the tasks executed locally plus the overhead (β_f) for all the tasks that were forwarded.

For a fixed T_{ss}, α, and β_f, these N conditions form a system of linear equations on N unknowns, $V_1, V_2, \ldots V_N$. By ordering the equations so that the parent of a processor in T appears before its children, it is easy to see that the system is in an upper triangular form. Thus, the N equations are linearly independent and there is a unique solution.

At the root of T, $V_1 = M$. Furthermore, it is easily verified, by back substitution, that V_1 is a linear function of T_{ss}. Thus T_{ss} can be expressed in terms of M, α, and β_f, which implies that given M, α, and β_f, we can solve for T_{ss} and V_1 to V_N. Because the topology is tree-structured, we can obtain these values in O(N) steps.

Although a solution to this system always exists, because of our assumption about idle time it may not be a feasible solution. In particular, to represent an actual flow, we must also have the condition that

$$V_i \geq \sum_{j \in C(i)} V_j. \quad (3)$$

When equality holds, the processor is said to be *saturated* (i.e., all of its CPU time is used to forward tasks). Since all tasks visit the root of the tree, if any node is saturated, then the root is also saturated. Since $\alpha > \beta_f$,

$$M = \frac{T_{ss}}{\beta_f}$$

is an upper bound on the maximum number of tasks that can be executed in time T_{ss} (i.e., maximum throughput is $1/\beta_f$).

In the actual system, processors in the subtree rooted at a saturated node may idle. In this case, the following procedure can be used to prune the tree to obtain the same performance using fewer processors.

Algorithm FindOptTree

1. **While** (there exists a saturated node)
2. Solve system (2) for V_i.
3. Using (3), find a saturated node u farthest distance from the root.
4. Remove a leaf in the subtree rooted at u.

The existence of a saturated node depends not only on β_f but also on α. As α increases, the overall system executes fewer tasks and therefore fewer tasks are forwarded. As a result, for any tree topology there exists a sufficiently large α/β_f such that no node is saturated. Again the critical value of α/β_f occurs at the root and can be found by solving the equation

$$V_1 - \sum_{j \in C(i)} V_j = 0$$

for α/β_f.

There remains the question of which spanning tree to use for *Pfarm*. We show that all shortest-path, demand-driven distribution schemes with overheads (β_e and β_f) equivalent to those in *Pfarm* have the same performance. In general, for nonidentical processors, determining the optimal subnetwork is a difficult problem related to the capacity assignment problem in data networks [2].

Let $T_{ss}(S)$ be the execution time of a shortest-path, demand-driven distribution scheme S with α and β_f equivalent to *Pfarm*.

THEOREM 1. *For any topology G, $T_{ss}(S)$ equals $T_{ss}(Pfarm(T))$ where $Pfarm(T)$ is Pfarm executing on T, a breadth-first spanning tree of G rooted at the source of the tasks.*

PROOF. Let $L(i)$ of G denote the set of processors that are at distance i from the source of the tasks. Since S and $Pfarm(T)$ are both shortest-path distribution schemes, tasks executed at a processor in $L(i)$ must have been forwarded from processors in $L(i-1)$. Let $s_i(S)$ and $s_i(T)$ denote the combined throughput of all the processors in $L(i)$ for scheme S and $Pfarm(T)$, respectively. We claim that for all i, $s_i(S) = s_i(T)$. It is initially true for n, the last level, since in both schemes the processors in $L(n)$ do not idle and can only execute tasks. In general, by induction on the level, the fact that processors do not idle and $s_i(S) = s_i(T)$ implies that both schemes must execute the same number of tasks on processors in $L(i-1)$. Thus $s_{i-1}(S) = s_{i-1}(T)$ and, in particular, $s_0(S) = s_0(T)$.

Therefore, for a fixed M, since the throughput for both the schemes is the same, $T_{ss}(S) = T_{ss}(Pfarm(T))$. □

It follows from Theorem 1 that for a fixed M there is only one value of T_{ss} that ensures that processors do not idle. There is the possibility that a nonshortest path scheme or a scheme that introduces idle time performs better. But, since this either increases the overhead to forward a task or reduces the computational power of a processor, it would be surprising if it outperformed the work efficient scheme we have analyzed.

3.3 Start-Up and Wind-Down Analysis

The start-up and wind-down time depends on the task distribution strategy and the hardware topology. For analyzing

start-up and wind-down costs, we consider the structure of the underlying process graph. Given a tree architecture, the *process graph* of the system can be constructed by replacing each processor by the process structure shown in Fig. 1. We remove the processes that gather the results and only consider the processes that are involved in task distribution: the **worker** process, the **manager** process, and the link processes. The process graph of the architecture given in Fig. 2a is shown in Fig. 2b. Notice that there are two types of edges in Fig. 2b: interprocessor communication edges and intraprocessor communication edges.

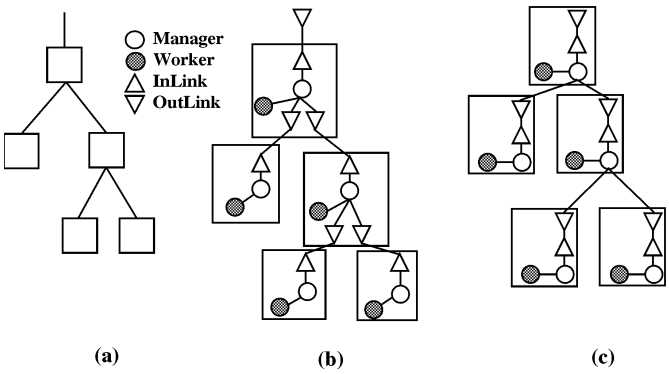

Fig. 2. (a) node graph, (b) process graph, (c) subtree decomposition.

Let T be the process graph of an N node architecture. We add as part of T an initial **OutLink** process, which we take as the root of T, so that in total T has $4N$ processes, each with a single buffer. Alternatively, T can be viewed as consisting of N subtrees (or nodes) of the type depicted in Fig. 2c.

3.3.1 Start-Up

Start-up begins when the first task enters the system, and ends when all the processors have received at least one task. The duration of the start-up phase depends on how the tasks are distributed to the processors. As mentioned in Section 2, the manager process gives priority to the worker process over the **OutLink** processes while allocating the tasks. Thus, when a worker process becomes free, it receives the next available task. When free, the **OutLink** processes receive tasks from the manager process. When more than one **OutLink** process is free, the manager allocates the tasks in a round robin ordering of the **OutLink** processes. Thus, as tasks enter the system, processors keep the first task they receive and then, as long as the **worker** process has not completed the current task, they distribute the incoming tasks to their children.

An upper bound on start-up time is obtained by dividing start-up into a sequence of steps. On each step, a task is transferred from an **OutLink** process of one processor to the **worker** process, an **OutLink** process, or when they all have tasks, to the last process in the node without a task. Furthermore, it is assumed that during the start-up phase no **worker** process finishes its first task. This is a reasonable assumption since the task forwarding link processes run at high priority while the worker process runs at low priority. Given a continuous flow of tasks into the system, we seek to bound the number of steps required before every worker process has a task.

For analysis purposes, the collapsed process graph shown in Fig. 2c is used. This graph is identical to the original architecture graph except that each node consists of the **InLink**, **manager**, and **worker** processes of a processor plus the corresponding **OutLink** process of the parent processor. In the tree, all the tasks are initially at the root and on each step every node with more than one task forwards, in round robin order, the last task it received to a child. In order to analyze the worst case, we assume that the tasks can be buffered at the manager process and thus each subtree has an infinite capacity to absorb tasks. We first obtain an upper bound on the number of steps to distribute at least one task to every node. This is equivalent to the procedure described above, except that it takes one additional step at the end to ensure that a leaf node forwards its task to the **worker** process.

Given a rooted oriented tree T, with child nodes numbered from left to right starting at one, let $c(v)$ be the child number of node v with respect to the parent of v, $p(v)$. Let $p^n(v)$ denote the nth ancestor of node v in T and let $\deg(v)$ be the down-degree of node v.

DEFINITION 1. *Let* $d_n(v)$ *equal* $\prod_{i=0}^{n} deg\left(p^i(v)\right)$.

LEMMA 1. *For any rooted oriented tree T, the first task received by node v is the*

$$1 + c\left(p^{n-1}(v)\right) + \sum_{j=0}^{n-2} c\left(p^j(v)\right) d_{n-j-2}\left(p^{j+2}(v)\right)$$

task at the node $p^n(v)$*, the root of T.*

PROOF. Let $s(v,i)$ be the number of the ith task received by node v. Using the fact that the child $c(v)$ receives every $\deg(p(v))$ task arriving at $p(v)$, except for the first which is kept by $p(v)$, we obtain the following recurrence

$$s(v, i) = s(p(v), 1 + c(v) + (i - 1)\deg(p(v))).$$

In general,

$$\begin{aligned}
s(v, 1) &= s\left(p^1(v), 1 + c\left(p^0(v)\right) + (i-1)d_0\left(p(v)\right)\right) \\
&= s\left(p^2(v), 1 + c\left(p^1(v)\right) + c(v)d_0\left(p^2(v)\right)\right. \\
&\qquad \left. + (i-1)d_0\left(p^1(v)\right)d_0\left(p^2(v)\right)\right) \\
&= s\left(p^2(v), 1 + c\left(p^1(v)\right) + \sum_{j=0}^{0} c\left(p^j(v)\right)d_{2-j-2}\left(p^{j+2}(v)\right)\right. \\
&\qquad \left. + (i-1)d_1\left(p(v)\right)\right) \\
&\ \ \vdots \\
s(v, i) &= s\left(p^n(v)\right), 1 + c\left(p^{n-1}(v)\right) + \sum_{j=0}^{n-2} c\left(p^j(v)\right)d_{n-j-2}\left(p^{j+2}(v)\right)\right. \\
&\qquad \left. + (i-1)d_{n-1}\left(p(v)\right)\right)
\end{aligned}$$

At the root $s(v, i) = i$. Thus, when $p^n(v)$ is the root,

$$s(v, 1) = 1 + c\left(p^{n-1}(v)\right) + \sum_{j=0}^{n-2} c\left(p^j(v)\right)d_{n-j-2}\left(p^{j+2}(v)\right). \quad (4)\ \square$$

The time step at which the worker process on node v receives task $s(v, 1)$ is $n + s(v, 1)$. This follows from the fact that task $s(v, 1)$ leaves the root after $s(v, 1) - 1$ steps and it takes $n + 1$ more steps for this task to reach the worker process on node v as this task gets forwarded in every step because $s(p^i(v), 1) < s(v, 1)$. The next theorem follows from these remarks.

THEOREM 2. *For any tree-structured process graph, after*

$$\max_{v \text{ a leaf}} \{n + s(v, 1)\}$$

steps, every worker process has a task to execute.

The time required for each step is determined by the communication cost to transfer a task from a processor to its child and the associated CPU overhead. Since β_f is the total overhead for forwarding a task and returning a result, $\beta_f / 2$ is a reasonable estimate for the cost of simply forwarding the task. Therefore, the estimated time for each step is $D_t\tau + \beta_f/2$.

From Theorem 2, it follows that

$$T_{su} = \left(D_t\tau + \beta_f / 2\right) \max_{v \text{ a leaf}} \{n + s(v, 1)\} \quad (5)$$

In an unbuffered system, the upper bound on the number of steps required for every worker process to have a task can be exponential in N. Consider, for example, a tree with a long path where each node on the path has large degree but all nodes off of the path are leaves. The sum of products in $s(v, 1)$ grows exponentially with respect to N. This is a result of our assumption that subtrees can always accept tasks. In practice, the upper bound cannot exceed the number of buffers in the tree, $4N$. But it is possible to come arbitrarily close to $4N$. As the upper bound is proportional to the number of buffers, for start-up, the number of buffers should be minimized. This is one of the reasons there is only one additional buffer on every worker processor.

Although the degree and depth of the tree is fixed it is possible to change $c(v)$. Theorem 2 provides a means for determining an orientation of the tree that minimizes start-up time and confirms that the best strategy is to always forward tasks along the longer paths first.

3.3.2 Wind-Down

Wind-down begins when the last task enters the system and ends when the last result leaves the system. The wind-down phase can be broken into two parts: the time to complete the remaining tasks in the system and the time to return results that are in the system after all the tasks have been executed.

Consider the state of the process graph when the last task enters the system. There are $4N$ tasks. In order to derive an upper bound, let us assume that N of the tasks have just started to execute. We will bound the maximum number of tasks executed by a single processor during the wind-down phase. This in turn is used to obtain an estimate for the wind-down time.

If all of the tasks have just begun execution, then after time α, each processor has executed one task. Since priority is given to distributing the tasks, some of the tasks buffered in each processor will be forwarded to the child processors. In the worst case, when α greatly exceeds β_f, the tasks will be forwarded as far as possible towards the leaves.

A worker process at a leaf can only execute those tasks available at its ancestor processes in the process graph. Therefore, we should consider the longest path from the root to a leaf in the tree (this is at most $3N + 1$, for example, see Fig. 2c) to derive an upper bound. Let m be the number of ancestor processes of the leaf process in the longest path, each of which initially contains a task. Starting at the root, every third process along the path is adjacent to a worker process. Therefore, after time α, when each of the worker processes have finished executing a task, each worker process adjacent to the path will receive a task from a manager along the path. The remaining tasks on the path shift down towards the leaves filling as many buffers as possible. This results in the following recurrence for $p(n)$, the length of the path,

$$p(0) = m$$

$$p(n) = p(n-1) - \left\lceil \frac{p(n-1)}{3} \right\rceil.$$

Solving this recurrence for $p(n) = 1$ shows that after $\lceil \log_{3/2} m \rceil + 1$ steps, all tasks have been processed. Thus the time taken for the first part of the wind-down phase is given by $\alpha(\lceil \log_{3/2} m \rceil + 1)$.

The second part of the wind-down cost is determined by the time required to forward the last result from the leaf process to the root. The transfer time is given by $D_r\tau$ and the estimated overhead per transfer is $\beta_f/2$, since β_f includes the overheads for both transferring a task to a child node and returning the corresponding result to the parent node. Therefore, the second part of the wind-down cost is $m/3 \times (D_r\tau + \beta_f/2)$ since there are $m/3$ processors in the path. If m is the length of the longest path, the wind-down cost is given by

$$T_{wd} = \alpha\left(\lceil \log_{3/2} m \rceil + 1\right) + \frac{m}{3}\left(D_r\tau + \beta_f / 2\right). \quad (6)$$

This analysis depends only on the depth of the tree and therefore gives the same bound for all breadth-first spanning trees of the topology. Note that the analysis is overly pessimistic since subtrees along a path from the root to a leaf also receive tasks from managers on the path. The actual wind-down time also depends on the number of nodes along the path with down degree greater than one. The fewer the number of nodes of degree one, the smaller is T_{wd}. Because of our round robin scheduling policy, any node of degree greater than one can only forward at most one task towards the leaf of the chosen path out of every three tasks that arrive at the node. Therefore, the number of tasks on the path decreases more quickly. If every processor on the longest path has at least two children, then the number of tasks executed by a leaf is bounded by $\max\{\lceil \log_3 m \rceil + 1, 4\}$. Leaves must execute at least four tasks, namely those in their local buffers.

4 BALANCED TREE TOPOLOGIES

In this section, we apply the framework described in the previous sections to obtain closed form solutions to k-ary tree topologies. These topologies are of interest because they achieve optimum performance for the following reasons.

1) A *k*-ary balanced tree has minimum depth among all degree *k* graphs, and therefore, by Theorem 1, has performance equal to the spanning tree of any *k*-degree network.
2) Both start-up and wind-down depend on the length of the longest path and the orientation of the tree. The symmetry in balanced tree implies that the orientation of the tree does not affect the start-up cost and, as mentioned, it has minimum depth over all *k* degree graphs.

As defined in Section 3, let α be the average processing time plus the overhead to execute a task locally and let β_f be the overhead for every task forwarded to a child processor. Consider a *D* level balanced *k*-ary tree with k^i processors on level *i*.

4.1 Steady-State Analysis

Consider (2), since all the processors on a level execute the same number of tasks, by summing over all the processors on level *i*, we have

$$\begin{aligned} k^i T_{ss} &= \alpha\left(\sum_i V_i - \sum_i \sum_{j\in C(i)} V_j\right) + \beta_f \sum_i \sum_{j\in C(i)} V_j \\ &= \alpha\sum_i V_i + \left(\beta_f - \alpha\right)\sum_{i+1} V_j. \end{aligned}$$

Let

$$L_i = \frac{1}{k^i}\sum_i V_i$$

be the number of tasks that visit a processor on level *i*. Therefore,

$$T_{ss} = \alpha L_i + \left(\beta_f - \alpha\right) k L_{i+1}.$$

By rearranging the above, we have

$$k\left(\alpha - \beta_f\right)L_{i+1} = \alpha L_i - T_{ss} \tag{7}$$

and $L_0 = M$.

Solving the recurrence

$$L_i = M\left[\frac{\alpha}{k\left(\alpha-\beta_f\right)}\right]^i - T_{ss}\left(\frac{\alpha^{i-1}}{\left[k\left(\alpha-\beta_f\right)\right]^i}\left[\frac{1-\left[\frac{k\left(\alpha-\beta_f\right)}{\alpha}\right]^i}{1-\frac{k\left(\alpha-\beta_f\right)}{\alpha}}\right]\right).$$

Let

$$a = \frac{\alpha}{k\left(\alpha-\beta_f\right)}.$$

Then

$$L_i = Ma^i - \frac{T_{ss}}{\alpha}a^i\left(\frac{1-1/a^i}{1-1/a}\right) = Ma^i - \frac{T_{ss}\left(a^i-1\right)}{\alpha\left(1-1/a\right)}.$$

Since $L_i = 0$ for $i = D$,

$$Ma^D = \frac{T_{ss}\left(a^D-1\right)}{\alpha\left(1-1/a\right)}$$

and

$$T_{ss} = \frac{M\alpha\left(1-1/a\right)}{\left(1-1/a^D\right)} = \frac{M\left[\alpha - k\left(\alpha-\beta_f\right)\right]}{1-\left(\frac{k\left(\alpha-\beta_f\right)}{\alpha}\right)^D}. \tag{8}$$

Our results for *k*-ary balanced trees subsumes those of Pritchard [21] and Tregidgo and Downton [25]. With proper substitutions[4] for $k = 1, 2, 3$, the formulas for throughput in [25] reduce to (8). Contrary to statements in [25], our model and consequently the model in [25] hold for distributed farms as well as small centralized farms.

4.2 Start-Up Analysis

Let *v* be the rightmost leaf node in a balanced *k*-ary tree topology. By (4),

$$s(v, 1) = 1 + k + k^2 + \ldots + k^{D-1} = \frac{k^D - 1}{k-1} = N.$$

Therefore, since $D + s(v, 1)$ is the maximum over all *v* (i.e., *v* is the last node to receive its first task), it follows from (5) that the start-up time for a balanced *D* level, *k*-ary tree is

$$T_{su} = (N + D - 1)(D_t\tau + \beta_f/2). \tag{9}$$

4.3 Wind-Down Analysis

The wind-down time is given by (6), which is reproduced below

$$T_{wd} = \alpha\left(\left\lceil \log_{3/2} m\right\rceil + 1\right) + \frac{m}{3}\left(D_r\tau + \beta_f/2\right),$$

where *m* is the length of the longest path in the process graph. For an *N* node linear chain topology, the wind-down analysis presented in Section 3.3.2 holds with $m = 3N$. The wind-down time is given by

$$T_{wd} = \alpha\left(\left\lceil \log_{3/2} 3N\right\rceil + 1\right) + N\left(D_r\tau + \beta_f/2\right).$$

As explained in Section 3.3.2, in a balanced *k*-ary tree, $k > 1$, the number of tasks in the longest path decreases more quickly. In this case, the number of tasks executed by the leaf node on the longest path is given by $\max\{\lceil \log_3 m\rceil + 1, 4\}$. For a *D* level *k*-ary balanced tree, $m = 3D$, and wind-down cost is given by

$$T_{wd} = \alpha\left(\left\lceil \log_3 D\right\rceil + 2\right) + D \times \left(D_r\tau + \beta_f/2\right). \tag{10}$$

In summary, the total execution time for processing *M* tasks on a *D* level *k*-ary balanced tree is

$$\begin{aligned} T_{total} &= T_{su} + T_{ss} + T_{wd} \\ &= (N + D - 1)\left(D_t\tau + \beta_f/2\right) + \frac{(M-4N)\left[\alpha - k\left(\alpha-\beta_f\right)\right]}{1-\left(\frac{k\left(\alpha-\beta_f\right)}{\alpha}\right)^D} \\ &\quad + \alpha\left(\left\lceil \log_3 D\right\rceil + 2\right) + D \times \left(D_r\tau + \beta_f/2\right). \end{aligned}$$

4. In [25], let $T_{CALC} = \alpha - \beta_f/2$ and $T_{SETUP} = \beta_f/4$.

5 Experimental Work

A variety of experiments were conducted to validate the models given in Sections 3 and 4. In Section 5.2, we describe five application programs, ranging from combinatorial/symbolic computation to numerical computation, used to test the accuracy of the model on real applications. In Section 5.4, a synthetic application with an easily adjustable workload is used to test the steady-state model over a wide variety of parameter values. The synthetic application is also used to validate the start-up and wind-down models from Section 3.3. Finally, in Section 5.7, we investigate *Pfarm*'s ability to dynamically balance the load with different task size distributions and schedules.

5.1 Experimental Setup

The experiments were conducted on a 75 node T800-based transputer system.[5] In addition, the system has ten 32×32 crossbar switches which were used to configure the system into a variety of different topologies. Processors were connected through at most one crossbar switch.

Pfarm was implemented in Logical Systems C (LSC)[6] and we have various versions running under the LSC toolset and Trollius [5]. Two versions of *Pfarm* were used in the experiments. The first uses fixed packet sizes and has a smaller β_f than the second version, which dynamically allocates message buffers. Similar results were obtained for both versions of *Pfarm*.

All timings were from the on-chip microsecond clock where the host machine was another transputer node. The synthetic workload used in the experiments was constructed by executing a tight loop to determine the number of iterations necessary to obtain T_es varying from one millisecond (ms) to 100 ms. The constant β_e was determined by measuring T, the time taken to complete M tasks of size T_e in a single worker system. We then solved for β_e in the equation $T = M(T_e - \beta_e)$. Similarly, given β_e, β_f can be determined by executing the two worker chain topology and solving for β_f in the expression

$$T = (T_e + \beta_f)(M - M_f) + M_f \beta_f,$$

where T is the measured execution time and M_f is the number of tasks that were forwarded.

We measured the performance of the system over a variety of topologies: chains (one to 64 nodes), binary trees (three to 64 nodes), ternary trees (four to 40 nodes), three BFSTs (breadth-first spanning tree) of the 3×8 mesh (see Fig. 3), one BFST of the 4×4 mesh and three BFSTs of the 8×8 mesh (similar to BFST1 and BFST3).

5.2 Application Programs

Five applications were developed and executed under a LSC version of *Pfarm*. A brief description of each application follows.

- **Ssum.** Given a set of integers S and an integer s, the algorithm finds, if it exists, a subset of S summing to s. The subset sum problem is NP-complete and a brute force depth-first search algorithm is used. It is parallelized by generating tasks, at a predefined depth of the tree, to serially search subtrees. Task execution time varies since the search terminates whenever the partial sum exceeds s.
- **FFT.** A parallelized version of the numerical recipes algorithm [26] was implemented. After the initial permutation of the array, the worker processes compute the smaller FFTs on segments of the array, after which the manager performs the remaining higher level combines. Granularity is adjusted by varying the size of the array segments assigned to the workers.
- **Cepstral.** A Cepstral filtering algorithm for motion analysis [1] which, given two images of the same subject at different instances of time, determines the motion of the subject. The application was parallelized by partitioning the task into 64×64 bit blocks where a task consists of two corresponding blocks, one from each image, and returns the image with motion indicators. The worker code involves several computational steps.
- **Fact.** A simple algorithm to find all factors of an arbitrarily large integer N. The algorithm uses a symbolic `bignum` package and attempts to divide all odd numbers in the interval from three to $\sqrt{N}$ into N. It is parallelized by dividing the interval into segments ranging in size from five to 180. Task execution time varies depending on whether or not it finds factors.
- **Mult.** A dense block matrix multiply was implemented. Blocks (for both matrices) are sent and multiplied by the workers and results are accumulated by the manager. Granularity was adjusted by adjusting the block size.

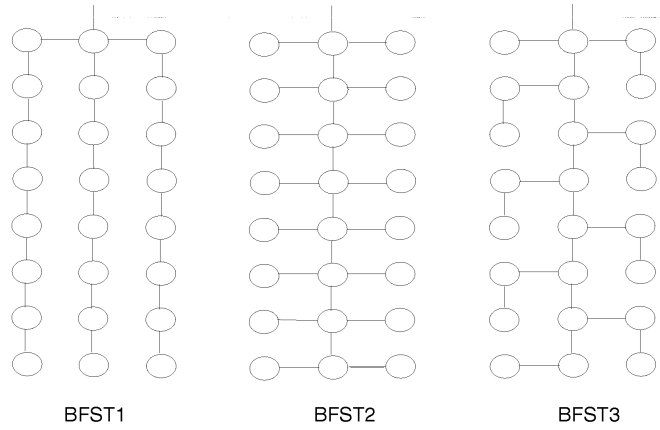

Fig. 3. Three breadth-first spanning trees of the 8×3 mesh.

In developing the applications, we started with the sequential program and reorganized it into a `manager` driver program and a `worker` routine. For example, the sequential FFT algorithm was parallelized by taking the outer loop that computes the transforms of length 2, 4, …, n and dividing into two loops, a loop from 2 to g and a final loop from $g + 1$ to n, where g was used to adjust the granularity. Each of the n/g segments of size g can be computed in parallel by a worker which performs the first loop. The manager code is almost identical to the modified program except that

5. 20 MHz T800s with 1 Mbyte of 0-wait state RAM, link speed set at 20 MHz.
6. V91.1.

TABLE 1
PARAMETER VALUES FOR THE FIVE APPLICATIONS

Application	T_e	M	D_t	D_r
Fact: segment size 40	30	1,639	36	184
FFT: 128K point, block size 1K	62	256	4K	4K
Mult: 512×512, block size 64×64	2,085	512	32K	16K
Ssum: $\|S\| = 26$	198.58	4,096	120	124
Cepstral: 512×512	178	961	818	72
Synthetic: small task size	9.91	10,000	4	4

T_e in milliseconds and data sizes (D_t, D_r) in bytes.

the first loop is replaced by calls to `dotask()` and `getresult()`. These two functions are part of the programming interface to *Pfarm* and handle the necessary packaging and unpackaging of tasks and results to and from the farm. Note that task decomposition, and hence task granularity, is done by the user.

No elaborate means were taken to parallelize these applications. In each case, we started with the sequential algorithm and sought only to achieve reasonable speedup on a modest size multicomputer.

5.3 Experimental Results

The values of the different parameters for each of the applications and the synthetic benchmark are shown in Table 1. Average T_e was determined by the formula $T_e = T/M + \beta_e$, using a one processor farm. For Ssum, FFT, Fact, and Mult, the system overheads in microseconds are $\beta_e = 190$ and $\beta_f = 125$. Cepstral used a different implementation with system overheads $\beta_e = 482$ and $\beta_f = 453$.

The results for each of the applications are given in Table 2. Table 2 shows only a representative set of results for a fixed size problem. The results for other experiments were similar, however, over the range of parameters that was tested there were more degenerate cases like those to be discussed.

For the most part, the models predict the execution time within a factor of 5% and the behavior of the applications closely matches that of the synthetic workload. We now discuss those cases in which the steady-state model did not apply or produced larger than expected errors.

The predicted time for two of the FFT results, Btree-31 and Mesh8x8, and the Cepstral result for Btree-63 were calculated using the communication-bound model given in Section 3.1. In this case, execution time is bound by the rate at which the manager processor generates tasks and collects results. For the FFT application, the measured rate at which the manager could process tasks was 274 tasks/second.

TABLE 2
PREDICTED TIME (IN SECONDS), % ERROR ((PREDICTED TIME – ACTUAL EXECUTION TIME) / ACTUAL EXECUTION TIME), AND OBSERVED SPEED-UP ($S_p = MT_e$/ACTUAL EXECUTION TIME) FOR THE APPLICATIONS ON VARIOUS TOPOLOGIES

FACTORING

Topology	N	Pred. Time	% Err	S_p
Ttree	4	12.4449	–0.09	3.95
Chain	7	7.3413	–0.12	6.69
Btree	7	7.1745	–0.18	6.84
Ttree	13	3.8943	0.03	12.64
Chain	15	3.6378	0.31	13.56
Btree	15	3.3921	–0.56	14.42
Mesh4x4	16	3.1871	–1.48	15.21
Chain	31	1.9682	–3.23	24.19
Btree	31	1.7095	0.53	28.93
Mesh8x8	64	0.9711	–8.87	46.17

FAST FOURIER TRANSFORM

Topology	N	Pred. Time	% Err	S_p
Ttree	4	4.0506	–0.32	3.91
Chain	7	2.6624	–0.07	5.96
Btree	7	2.4066	–1.89	6.47
Ttree	13	1.3541	–0.21	11.70
Chain	15	1.5740	–2.35	9.85
Btree	15	1.1915	–4.12	12.78
Chain	31	1.0877	4.35	15.24
Btree	31	0.7047	–21.77	16.19
Chain	63	0.9771	–6.26	15.23
Mesh8x8	64	0.5353	–31.35	11.67

BLOCK MATRIX MULTIPLY

Topology	N	Pred. Time	% Err	S_p
Ttree	4	269.1101	–0.05	3.97
Btree	7	156.7634	0.03	6.81
Ttree	13	86.3379	0.58	12.44
Btree	15	75.3852	–0.24	14.13
Mesh4x4	16	70.9355	–1.41	14.84
Chain	31	53.2567	2.84	20.62
Btree	31	40.7187	5.91	27.78
Btree	63	25.3040	–3.73	40.63
Chain	63	39.9272	–8.99	24.34
Mesh8x8	64	25.7479	–22.11	32.31

SUBSET SUM PROBLEM

Topology	N	Pred. Time	% Err	S_p
Ttree	4	203.8387	–0.05	3.99
Btree	7	116.8129	0.11	6.97
Ttree	13	63.0904	0.12	12.91
Btree	15	54.7536	0.14	14.88
Mesh4x4	16	51.3640	–0.65	15.73
Btree	31	26.9132	0.27	30.30
Chain	31	28.2995	–0.51	28.60
Chain	63	15.3597	–5.06	50.27
Btree	63	13.7544	0.36	59.35
Mesh8x8	64	14.5767	2.25	57.06

CEPSTRAL FILTERING PROBLEM

Topology	N	Pred. Time	% Err	S_p
Ttree	4	43.1405	–0.14	3.96
Btree	7	24.9490	0.46	6.89
Ttree	13	13.6062	0.16	12.59
Btree	15	11.8585	0.13	14.44
Chain	16	12.3503	–0.49	13.78
Btree	31	6.1163	–1.53	27.54
Chain	32	7.3517	–2.02	22.80
Ttree	40	4.8573	2.13	35.97
Btree	63	3.4469	12.28	38.08
Chain	64	4.8481	7.33	37.87

SYNTHETIC BENCHMARK WITH SMALL TASK SIZE

Topology	N	Pred. Time	% Err	S_p
Ttree	4	25.5073	0.05	3.89
Btree	7	14.7132	0.07	6.74
Ttree	13	7.9506	0.22	12.50
Btree	15	6.9489	0.09	14.28
Mesh4x4	16	6.5345	–0.10	15.16
Chain	31	3.9930	0.41	24.93
Btree	31	3.4211	0.42	29.10
Chain	63	2.4051	–0.94	40.83
Btree	63	1.7305	–0.06	57.26
Mesh8x8	64	1.8699	4.18	55.24

TABLE 3
COMPARISON OF PREDICTED AND MEASURED RESULTS FOR A VARIETY OF TOPOLOGIES

Configuration	T_e	N	Predicted Time	Measured Time	% Error
Chain	0.001	8	47.890	46.796	2.284
		32	45.300	45.593	–0.647
		64	45.300	45.598	–0.658
	0.005	8	90.917	90.427	0.539
		32	48.365	47.354	2.09
		64	45.366	45.618	–0.555
	0.020	8	276.633	276.392	0.087
		32	88.655	88.364	0.328
		64	59.539	59.124	0.697
Binary Tree	0.010	7	159.486	159.525	–0.021
		31	45.300	45.595	–0.651
		63	45.300	45.578	–0.614
	0.020	7	302.117	302.251	–0.044
		31	70.922	70.934	–0.017
		63	45.300	45.583	–0.625
	0.040	7	587.952	588.119	–0.028
		31	135.371	135.453	–0.061
		63	67.313	67.412	–0.147
Ternary Tree	0.005	4	146.462	146.262	0.137
		13	48.421	48.532	–0.229
		40	45.262	45.576	–0.694
	0.010	4	270.943	271.090	–0.054
		13	86.602	86.688	–0.099
		40	45.282	45.622	–0.751
	0.020	4	520.917	521.082	–0.032
		13	163.412	163.522	–0.067
		40	54.235	54.275	–0.074
BFST1 (8 x 3)	0.01	24	5.288	5.273	0.284
	0.04		17.918	17.870	0.268
	0.08		34.816	34.796	0.057
BFST2 (8 x 3)	0.01	24	5.288	5.265	0.435
	0.04		17.918	17.798	0.670
	0.08		34.816	34.564	0.724
BFST3 (8 x 3)	0.01	24	5.288	5.272	0.303
	0.04		17.918	17.797	0.675
	0.08		34.816	34.642	0.500
BFST1 (8 x 8)	0.03	64	5.457	5.398	1.081
	0.06		10.346	10.275	0.686
	0.09		15.241	15.077	1.076
BFST2 (8 x 8)	0.03	64	5.457	5.372	1.557
	0.06		10.346	10.220	1.218
	0.09		15.241	14.991	1.640

β_e = 482 Microseconds, β_f = 453 Microseconds.

This is less than the rate at which Btree-31 and Mesh8x8 could execute tasks, 363 and 478, respectively. Similarly, for Cepstral, tasks can be generated at a rate of 248 tasks/second, yet, according to the steady-state model, Btree-63 can process tasks at a rate of 349 tasks/second.

The large error for FFT Btree-31, Mesh8x8, and Cepstral Btree-63 is partially due to the wind-down phase which was not included in the calculation. Start-up and wind-down is difficult to accurately predict in the communication-bound case since it depends on how far tasks are pulled towards the leaves. These degenerate cases are less interesting since the obvious solution is to use fewer processors, which increases both efficiency and performance. For example, in the FFT application, Btree-15 outperforms Mesh8x8 and is close to that of Btree-31.

The error for Chain-63 and Mesh8x8 of the matrix multiplication application exceeds the 5% or less error present in most of the experiments. However, as described in Section 3.3, the pessimistic bound used for wind-down makes the models for this phase less accurate than that of the steady-state model. In the case of matrix multiplication, half of the tasks ($512 - 4 \times 64$) are part of start-up and wind-down. The relatively large start-up and wind-down phases also affect speedup since processors are not fully utilized during those phases. For example, Chain-63 has a speed-up of only 24.34 on 63 processors. As predicted by the models, the speed-up for Chain-63 is less than that of Btree-63 and Mesh8x8.

5.4 Steady-State Experiments

In order to determine the accuracy of the steady-state models, we conducted experiments with large M equal to 100,000 (10,000 for the BFST experiments). This value of M was sufficient to ensure that start-up and wind-down was insignificant in comparison to the steady-state time. T_e ranged from one millisecond to 90 milliseconds. One-millisecond tasks were small enough to obtain close to the maximum throughput of the system (i.e., $1/\beta_f$ tasks/second)

and 90-millisecond tasks, or smaller, were large enough to achieve close to linear speed-up.

A sample of these results are shown in Table 3. This table includes instances of the worst case error among the data that was collected. The value of $M \times \beta_f = 45.3$ seconds is a lower bound on the execution time of the system. For increasing N, the results clearly show the execution time asymptotically approaching the limit of 45.3 seconds. Note that the model accurately predicted the performance of the system (within 3%).

5.5 Communication-Bound Model

The communication bound model comes into effect once (1),

$$T_{comm} = M \times \max\{C_R + \tau D_r, C_S + \tau D_t\}$$

exceeds the execution time given by the steady-state model. Table 4 shows the percentage error between the predicted and measured total execution time on the chain and binary tree configurations for experiments in which the task data size and result size is now 1,000 bytes.

TABLE 4
COMPARISON OF PREDICTED AND MEASURED TIME FOR A COMMUNICATION BOUND *Pfarm* EXECUTING 10,000 TASKS ON A CHAIN AND BINARY TREE

Configuration	N	T_e = 5 ms		
		Predicted Time	Measured Time	%Error
Chain	1	54.845	55.183	–0.616
	2	28.604	28.873	–0.940
	4	15.532	15.795	–1.693
	8	9.092	9.341	–2.739
	16	6.913	8.320	–20.353
	32	6.913	8.301	–20.078
Binary Tree	1	54.845	55.183	–0.616
	3	19.347	19.585	–1.230
	7	8.844	9.009	–1.865
	15	6.913	8.172	–18.212
	31	6.913	8.169	–18.169

In Table 4, the measured execution time is approaching a limit, the communication bound of the system. For small values of N, the system is computation bound and the execution time was predicted using the steady state model. The errors for these entries are similar to those in Table 3. The predicted execution time of the last few entries in Table 4 is the bound given by (1) calculated as 6.913 where $C_R = \beta_f/4$ (β_f = 453 microseconds) and τ = 1.76 Mbytes/sec. The error in this case is approximately 20%, which is due to τ. We have used the hardware value of τ which led to an optimistic upper bound on performance. The value of τ can be measured. However, for the transputer, the transfer rate varies according to the presence or absence of bidirectional communication, which in turn depends on C_r and C_t, the message injection rates. From the predicated execution times in Table 4, a value of τ = 1.4 Mbytes/sec is a better estimate of the actual value.

5.6 Start-Up and Wind-Down

The combinatorial analysis given in Section 3.3.1 for the start-up model was verified by running a number of experiments with different T_e and M and observing the task number of the first task processed by each processor. In all the cases, the task number of the first task executed at a node was the same as that predicted by the model given in Section 3.3.1. The wind-down model was validated by running experiments with $M = 4N$ and observing the number of tasks executed by the leaf node in the longest path. In the case of wind-down, the maximum number of tasks executed by any single processor did not exceed the bound given by (6). As explained in Section 3.3.2, (6) is overly pessimistic and in many cases fewer tasks were executed leading to better performance.

Table 5 compares the predicted execution times to the actual time taken to execute $M = 4N$ tasks. 4N, with sufficiently large T_e, fills all the buffers in the system and ensures that the system just reaches steady-state. Thus, the time to execute $4N$ tasks is a reasonable approximation to the combined time of start-up and wind-down. The errors in Table 5 are due to the overly pessimistic bound on the maximum number of tasks executed by the leaf farthest from the root. For example, in the 4×4 and 8×8 meshes, the bound gives nine and 10, whereas the actual number of tasks is six and eight. Note the error is proportional to the size of the difference (e.g., for the 4×4 mesh the error is about 9-6/6). When the actual number of tasks is used in the model the error is less than 5%. The dynamic nature of wind-down makes it difficult to obtain a completely accurate estimate. The actual number of tasks executed by a processor can depend on the outcome of a single scheduling decision. Table 5 also shows advantage of the bound for balanced trees from Section 4.3 versus the more general bound from Section 3.3.

In general, when $M < 4N$, it is difficult to predict the number of tasks executed by a single processor. It depends on M and the way tasks are pulled towards the leaves. In the special case of $M = N$, each processor receives exactly one task and the execution time is estimated by α plus start-up plus the time to return the result from the leaf farthest from the root.

5.7 Robustness

The experiments with the synthetic benchmark given in the last three sections used a constant value for T_e. In practice, however, T_e varies from task to task. Although several of the application programs did have varying T_e, to test the robustness of using average task size for prediction, we experimented with uniform and bimodal task size distributions (i.e., execution times). The large variance for these distributions makes them a good test of the dynamic load-balancing ability of *Pfarm* and the use of average T_e. Our experiments using the bimodal distribution show that as long as the task sizes are "well-mixed," the behavior of the synthetic benchmark with constant T_e matches that of the benchmark with varying task sizes using average task size for T_e.

In the case of uniformly distributed task sizes, we used 10,000 tasks and varied T_e from one to 19 ms (average T_e = 10 ms) for one set of experiments and one to 40ms (average T_e = 20 ms) for a second set of experiments. Table 6 compares the predicted and measured total execution time for these tasks executed on a chain topology. As Table 6 shows, the errors are all within 2%.

TABLE 5
COMPARISON OF PREDICTED AND MEASURED TOTAL TIME (START-UP AND WIND-DOWN) FOR *Pfarm* RUNNING ON VARIOUS TOPOLOGIES

Configuration	N	T_e = 10 ms			T_e = 40 ms		
		Upper Bound	Measured Time	% Error	Upper Bound	Measured Time	% Error
Chain	4	0.097	0.086	11.340	0.367	0.326	11.17
	16	0.159	0.145	8.80	0.579	0.535	7.60
	32	0.202	0.186	7.92	0.712	0.626	12.08
	64	0.268	0.257	4.10	0.818	0.726	11.25
Binary Tree	7	0.066	0.065	1.51	0.246	0.245	0.41
	15	0.078	0.067	14.10	0.289	0.247	14.53
	31	0.083	0.068	18.07	0.293	0.251	14.33
	63	0.092	0.080	13.04	0.302	0.256	15.23
Ternary Tree	13	0.067	0.056	16.42	0.247	0.206	16.60
	40	0.075	0.072	4.00	0.255	0.248	2.745
4 x 4 Mesh	16	0.095	0.063	49.62	0.368	0.242	51.84
8 x 8 Mesh	64	0.117	0.097	20.70	0.421	0.340	23.77

TABLE 6
COMPARISON OF PREDICTED AND MEASURED TOTAL EXECUTION TIME FOR UNIFORM TASK DISTRIBUTION FOR PROCESSOR FARM RUNNING ON LINEAR CHAIN

N	T_e = 10 ms			T_e = 20 ms		
	Predicted Exec. Time	Constant % Error	Uniform % Error	Predicted Exec. Time	Constant % Error	Uniform % Error
1	104.880	0.010	–0.400	204.941	0.013	–0.453
2	53.602	0.000	–0.396	103.625	0.000	–0.458
4	27.986	0.000	–0.382	52.978	–0.028	–0.525
8	15.225	–0.110	–0.512	27.677	–0.188	–0.755
16	9.020	0.610	0.455	15.235	0.407	–0.217
32	6.047	0.496	–0.050	9.018	0.044	0.011
48	5.186	0.174	–0.116	7.019	–0.527	–1.225
64	4.828	–0.249	1.263	6.068	–0.906	–1.269

In the case of a bimodal distribution of task sizes, we had 5,000 tasks of one ms duration and another 5,000 of 20 ms duration. In these experiments, M was 10,000 and the values of T_e were 1, 5, 10, 20, and 40 ms. Here we describe a set of experiments in which we used 5,000 tasks of one ms duration and another 5,000 of 20 ms duration. In the bimodal distribution case, the order of arrival of the tasks into the system affects the performance. Experiments were conducted with four different arrival patterns:

1) Both one ms and 20 ms tasks arrive with equal probability.
2) 20 ms tasks arrive at a probability of 0.75, until all 5,000 of them are processed.
3) One ms tasks arrive at a probability of 0.75, until all 5,000 of them are processed.
4) All the one ms tasks arrive before any 20 ms tasks.

In Fig. 4, we have plotted the percentage error between the predicted and measured total execution times for these experiments on a linear chain topology. For prediction, we have used an average value of T_e = (1 × 5,000 + 20 × 5,000)/10,000 = 10.5 ms in the model. As we can observe from the figure, the errors are small (< 3%) for all the four cases when N is less than eight. For larger N, the prediction is accurate for the first case but the error increases for the other three cases. The maximum error observed varies from around 6.5% in the second case to around 10.25% in the third case and is highest at around 15.0% for the fourth case. Note that as long as the tasks are "well-mixed," the farm balances the load.

The errors depend on the extent to which the average T_e reflects the actual computation requirements of the tasks. As long as N is smaller than the optimal value corresponding to the smaller T_e of the two, the average value works well for all the cases. However, for larger values of N, the average value works well only when the two kinds of tasks are well mixed with respect to arrival order, as in the first case with equal probability. Among the other three cases, tasks are mixed for a larger portion of the total execution time in the second case than in the third case, and there is no mix at all in the fourth case. In these cases, there is a corresponding increase in the error, with the largest errors occurring for the fourth case. Obviously, in this case, it is more appropriate to view the execution as occurring in two distinct phases of computation, the first consisting of all one ms tasks and the second with all 20 ms tasks. For this case, it is better to use the model twice, predicting the execution time for one and 20 ms tasks separately and adding them together for the total time.

6 USE OF THE MODEL

The validation experiments from the previous section show that the performance of *Pfarm* can be characterized by a model of the form

$$\mathcal{M}(T_{total}, M, T_e, D_r, D_t, \beta_e, \beta_f, topology, N, \tau).$$

In this section, we explore the relationships between the different parameters and describe how $\mathcal{M}$ can be used to improve speed-up and efficiency. $\mathcal{M}$ has three types of parameters: M, T_e, D_r, and D_t, which depend on the application; β_e and β_f, which depend on the implementation of *Pfarm*; and *topology*, N, and τ, which depend on the underlying architecture. Note that for a given machine and implementation of *Pfarm*, β_f, β_e, and τ are constants.

The model can be used to evaluate the use of *Pfarm* for a particular application. For a fixed size problem with M tasks and data sizes D_r and D_t, the communication bound model and the steady state bound of $1/\beta_f$ give an upper bound on performance. However, in order to predict performance, one needs an estimate of T_e, the task size. T_e can often be determined from the sequential program; for example, in the FFT application, the worker routine consists of the inner part of a loop, which can be estimated by timing the sequential program on a single node of the target machine. Given T_e, the model can be used to predict performance over a range of N and/or different topologies.

One way to improve the performance of an application is to increase the task size (e.g., g in the FFT application); however, increasing the granularity will in general decrease M, or increase D_r and D_t; either of which will eventually degrade performance. The model can be used to find the optimal sized granularity by solving the model for various values of T_e, M, D_r, and D_t. For example, T_e for a 1K point FFT was 62 milliseconds, hence, for a problem of size n, we have for a particular g

$$T_e = 6.05 \times g \log_2(g) \qquad D_r = D_t = 4g \qquad M = n/g$$

We can now solve the model for different values of n, g, and N to improve performance or determine whether it is possible to achieve the desired performance. It is, for example, not possible to effectively use 32 nodes to compute a 128K FFT. For small g the system is compute bound whereas for larger g there are too few tasks to achieve steady-state. On a 16 node binary tree, for g = 128, 256, 512, 1,024, 2,048 the model gives speedups of 12.66, 13.52, 13.15, 12.23, 10.65, 5.52, respectively. Thus a granularity of g = 256 points gives the best performance.

The architecture dependent parameters, *topology* and N, can also be used to improve performance. As shown in Section 3.2, for a fixed interconnection network, it is best to use a BFST. If the system is saturated, it is possible to improve system efficiency by pruning the tree with algorithm **`FindOptTree`**.

A measure of the effect of topology on performance is shown in Fig. 5, which shows the performance of *Pfarm* on a linear chain, binary tree, and ternary tree. In low degree networks (e.g., meshes), the performance of an arbitrary subtree lies somewhere between the curves for a chain and a binary tree. Between these two curves it is possible to significantly improve performance by choosing the best subtree. For high degree networks (e.g., hypercubes), any tree with logarithmic depth is likely to perform well. The behavior of the system for $k = 1$ versus $k > 1$ occurs as a result of the term $1/a$ in (8) which becomes less than one between $k = 1$ and $k = 2$.

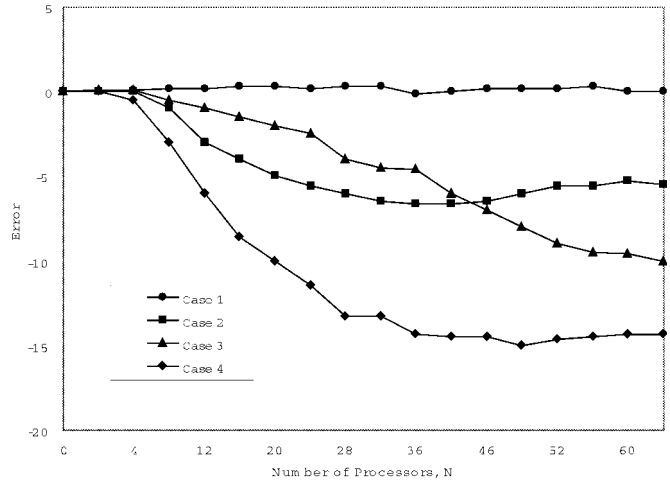

Fig. 4. Error graph for processor farm on linear chain with tasks of bimodal distribution.

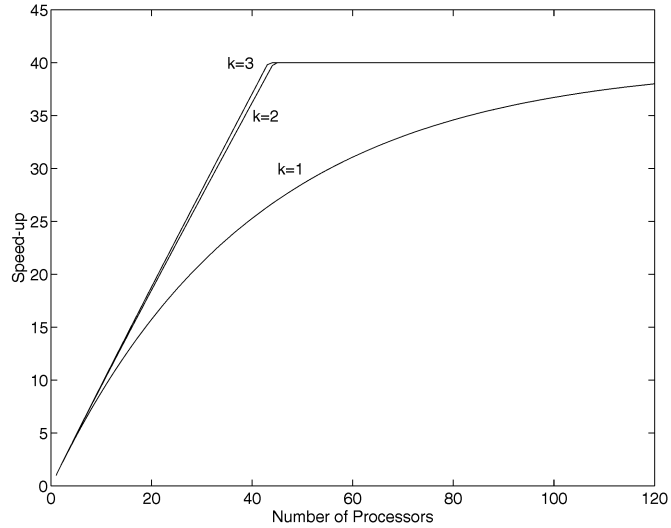

Fig. 5. Comparison of speed-ups on a linear chain, binary tree, and ternary tree topology for t_e = 40 ms and M = 10,000.

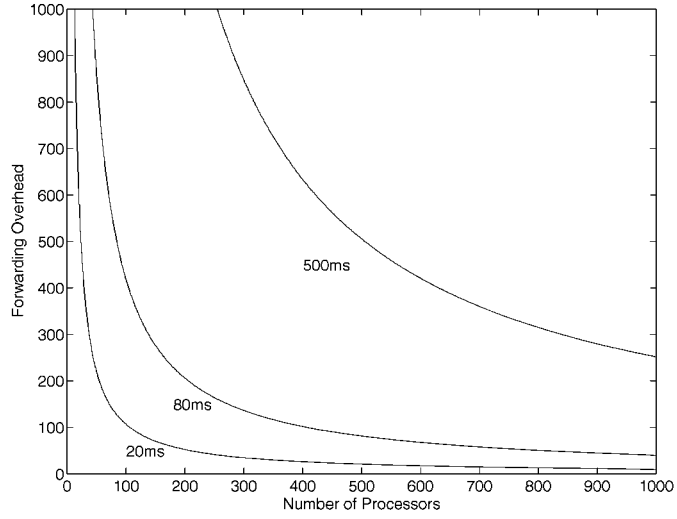

Fig. 6. The saturation point for varying β_f on a binary tree with 20, 80, and 500 millisecond task sizes.

The implementation dependent parameter β_f limits the steady-state performance of the computation bound model which ultimately limits the number of processors that can be effectively used to solve the problem. Fig. 6 is a plot of β_f versus the maximum number of processors that can be effectively used to solve the problem, for various different task sizes on a binary tree. Fig. 6 shows the critical nature of β_f, particularly for smaller task size. Once beyond the knee of the curve, a small increase in β_f can dramatically increase the number of processors that can be used effectively.

As shown in Fig. 5, *Pfarm* gives close to linear performance before reaching the saturation point. This implies that for problems which fit the model and users satisfied with speed-ups within the limit imposed by β_f, there is no reason to use a more elaborate parallelization technique. Decreasing β_f increases the applicability of processor farm by increasing the range of N which can be effectively used. The overall efficiency of the system is limited by β_e to β_e / α. Reducing β_e in effect increases the computation rate of the nodes and again serves to increase the applicability of farms. For example, the performance (speedup and efficiency) of the matrix multiplication application from Section 5.2 begins to degrade at $N = 16$. On a different system, a smaller β_e and β_f would make it possible to decrease the grain size and effectively use more processors.

7 Conclusions

In this paper, we have described the design and implementation of a demand-driven, distributed control processor farm. We have modeled and analyzed its behavior and performance. To summarize, the design should

1) overlap communication with computation,
2) minimize, to the extent possible, the number of task buffers, and
3) have a constant overhead for forwarding and executing tasks.

For arbitrary topologies the analysis indicates that a breadth first spanning tree should be used and the start-up and wind-down analysis gives criteria that can be used to chose the best such spanning tree. In the computation bound case when the tree is saturated, algorithm `FindOptTree` can be used to chose a subtree which maximizes efficiency. The models were validated in Section 5 and their accuracy shows that they can be used to predict performance.

These models could also be directly incorporated into compilers and programming environments for parallel machines [15]. Recent work in the functional parallel programming community has focused on the identification and compilation of program templates or skeletons, processor farm being one of them [12], [4], [22]. There are also programming environments like Enterprise [23], which have embedded in them a farm-like processing system.

Acknowledgments

We would like to thank the referees for their helpful comments. This work was supported by the Natural Sciences and Engineering Research Council of Canada. Halsur V. Sreekantaswamy acknowledges the financial support provided by the Canadian Commonwealth Scholarship and Fellowship Administration.

References

[1] E. Bandari and J.J. Little, "Multi-Evidential Correlation & Visual Echo Analysis," Technical Report TR 93-1, Dept. of Computer Science, Univ. of British Columbia, Vancouver, Canada, Jan. 1993.
[2] D.P. Bertsekas and R.G. Gallager, *Data Networks*. Prentice Hall, 1987.
[3] R.D. Beton, S.P. Turner, and C. Upstill, "A State-of-the-Art Radar Pulse Deinterleaver—A Commercial Application of Occam and the Transputer," *Occam and the Transputer—Research and Applications*, C. Askew, ed., pp. 145–152. IOS Press, 1988.
[4] T.A. Bratvold, "A Skeleton-Based Parallelising Compiler for ML," *Proc. Fifth Int'l Workshop Parallel Implementation of Functional Languages*, The Netherlands, Sept. 1993.
[5] G.D. Burns, A.K. Pfiffer, D.L. Fielding, and A.A. Brown, "The Trillium Operating System," *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, ACM Press, Jan. 1988.
[6] D. Busvine, "Implementing Recursive Functions as Processor Farms," *Parallel Computing*, vol. 19, pp. 1,141–1,153, 1993.
[7] N. Carriero and D. Gelernter, *How to Write Parallel Programs*. MIT Press, 1990.
[8] K. M. Chandy and C. Kesselman, "Parallel Programming in 2001," *IEEE Software*, pp. 11–20, Nov. 1991.
[9] R.S. Cok, *Parallel Programs for the Transputer*. Prentice Hall, 1991.
[10] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, Mass.: MIT Press, 1989.
[11] I. Cramb and C. Upstill, "Using Transputers to Simulate Optoelectronic Computers," *Tools and Techniques for Transputer Applications (OUG 12)*, S.J. Turner, ed. IOS Press, Apr. 1990.
[12] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While, "Parallel Programming Using Skeleton Functions," technical report, Imperial College London, London, UK, 1993. Internal Report.
[13] W. Day, "Farming: Towards a Rigorous Definition and Efficient Transputer Implementation," *Transputer Systems—Ongoing Research*, A.R. Allen, ed. IOS Press, 1992.
[14] "EPCC. PUL-TF Prototype User Guide," Technical Report EPCC-KTP-PUL-TF-PROT-UG-1.4, Edinburgh Parallel Computing Center, 1992.
[15] D. Feldcamp, H.V. Sreekantaswamy, A. Wagner, and S. Chanson, "Towards a Skeleton-Based Parallel Programming Environment," *Transputer Research and Applications 6 (NATUG 6)*, A.M. Veronis and Y. Paker, eds., pp. 104–115. IOS Press, Apr. 1992.
[16] A.J.G. Hey, "Experiments in MIMD Parallelism," *PARLE: Parallel Architectures and Languages Europe*, E. Odijk, M. Rem, and J.-C. Syre, eds., pp. 28–42. *Lecture Notes in Computer Science 366*. New York: Springer-Verlag, 1990.
[17] D. Homeister, "An Adaptive Granularity Scheduler for Multiprocessors," *Transputer Research and Applications 6 (NATUG 6)*, S. Atkins and A. Wagner, eds. IOS Press, May 1993.
[18] H.T. Kung, "Computational Models of Parallel Computers," *Scientific Applications of Multiprocessors*, R.J. Elliot and C.A.R. Hoare, eds. Prentice Hall, 1989.
[19] S.W. Lau and F.C.M. Lau, "A Efficient and Flexible Implementation of Alt," *Transputer Research and Applications (NATUG 3)*, A. Wagner, ed., 1989.
[20] D. May, R. Shepherd, and C. Keane, "Communicating Process Architectures: Transputers and Occam," *Future Parallel Computers, Lecture Notes in Computer Science*, P. Treleaven and M. Vanneschi, eds., vol. 272, pp. 35–81. Springer-Verlag, 1987.
[21] D.J. Pritchard, "Performance Analysis and Measurement on Transputer Arrays," *Evaluating Supercomputers*, A. van der Steen, ed. Chapman and Hall, 1990.
[22] F.A. Rabhi, "Exploiting Parallelism in Functional Languages: A 'Paradigm-Oriented' Approach," *Abstract Machine Models for Highly Parallel Computers*, P. Dew and T. Lake, eds. Oxford Univ. Press, 1993.
[23] A. Singh, J. Schaeffer, and M. Green, "A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 1, pp. 52–67, Jan. 1991.

[24] S.S. Sturrock and I. Salmon, "Application of Occam to Biological Sequence Comparisons," *Occam and the Transputer—Current Developments*, J. Edwards, ed., pp. 181–190. IOS Press, 1991.
[25] R.W.S. Tregidgo and A.C. Downton, "Processor Farm Analysis and Simulation for Embedded Parallel Processing Systems," *Tools and Techniques for Transputer Applications (OUG 12)*, S.J. Turner, ed. IOS Press, Apr. 1990.
[26] W.T. Vetterling, W.H. Press, and B.P. Flannery, *Numerical Recipes in C*. Cambridge Univ. Press, 1992.
[27] G. Wilson, *Practical Parallel Programming*. MIT Press, 1995.

Alan S. Wagner received the BSc degree in mathematics from Dalhousie University in 1977, the MSc degree in computer science from the University of Alberta in 1983, and the PhD degree in computer science from the University of Toronto in 1987. He is currently an assistant professor in the Department of Computer Science at the University of British Columbia.

Dr. Wagner's research interests include parallel computer architecture, parallel programming environments, and the graph theoretic properties of interconnection networks.

Halsur Sreekantaswamy received the BE degree in electronics and communications from the University of Mysore, Karnataka, India, in 1980, the MTech degree in electrical engineering from the Indian Institute of Technology, Kanpur, India, in 1983, and the PhD degree in computer science from the University of British Columbia, Vancouver, Canada, in 1993. From 1983 to 1986, he worked as an assistant executive engineer (R & D) at the Indian Telephone Industries, Bangalore, India.

Dr. Sreekantaswamy is currently working as a member of the scientific staff at Nortel (Northern Telecom), Ottawa, Canada. His research interests are in the areas of network management and parallel and distributed systems.

Samuel T. Chanson received his PhD degree in electrical engineering and computer sciences from the University of California, Berkeley, in 1975. He was a faculty member at the School of Electrical Engineering, Purdue University, for two years before joining the Department of Computer Science at the University of British Columbia, where he became a full professor and director of its Distributed Systems Research Group. In 1993, Professor Chanson joined the Hong Kong University of Science & Technology as professor and associate head of the Computer Science Department. He has consulted widely for industry and government institutes in Canada, the United States, China, Korea, Taiwan, and Japan on communication technologies, and has served on the program committees of many international conferences on distributed systems and computer communications. He was the program co-chair of the 1996 IEEE International Conference on Distributed Computing Systems.

Dr. Chanson's research interests include computer communications (particularly protocols and high speed networks), multimedia communication, parallel software, and Internet technologies. He is the director of the Cyberspace Centre at HKUST.