JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 20, 317–329 (1994)

# Hierarchical Scheduling of Dynamic Parallel Computations on Hypercube Multicomputers

ISHFAQ AHMAD

*Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong*

ARIF GHAFOOR

*School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907*

AND

GEOFFREY C. FOX

*Northeast Parallel Architectures Center, 111 College Place, Syracuse University, Syracuse, New York 13244*

**In this paper a hierarchical task scheduling strategy for assigning parallel computations with dynamic structures to large hypercube multicomputers is proposed. Such computations represent a wide range of recursive and divide/conquer algorithms for which structure of the problem varies dynamically. To achieve load balancing and reduce processor contentions, the system is divided into multiple regions of processors for which the first level of scheduling is done by the host computer that spreads out the initial computations into these regions. The second level scheduling is done by a set of median processors of these regions which enable the processors of their regions to optimally balance the dynamically created load and to communicate with each other with reduced overhead. The results of an extensive simulation study are presented that exhibit the performance of the proposed strategy under different loading conditions, varying degrees of depth and parallelism, and communication costs. The proposed dual-level hierarchical scheduling is shown to outperform a well known distributed scheduling strategy.** © 1994 Academic Press, Inc.

## 1. INTRODUCTION

As parallel computers are becoming increasingly popular and a wide variety of parallel architectures are being proposed, an efficient use of these computers for running general-purpose problems has posed new challenges for researchers. This is largely due to the fact that an efficient use of a parallel processing system is a complex task, and the design and implementation of a supporting system needs to cope with a number of issues dealing with languages, compilers, parallelism detection techniques, grain-size determination procedures, data dependency analysis, interprocessor communication mechanisms, etc. The most critical considerations for an efficient use of a parallel system are to partition tasks and schedule them onto the processors. This can be accomplished by using a static scheduling strategy based on a priori knowledge of the problem structure that specifies the execution and communication requirements of subtasks. However, for a large class of problems, such a structure is not known in advance. For these problems, the workload is dynamically created as the execution of the program proceeds. In this environment the system must deal with dynamic partitioning of computational tasks and handle the load that can fluctuate with time across processors. Dynamically evolving computational problems can be found in many applications such as event-driven simulations, particle dynamics, searching of game trees, sorting, branch and bound algorithms, etc., [9, 15]. These problems require dynamic load balancing since tasks once assigned to processors can spawn more subtasks at the run time as the computation proceeds [17].

Considerable research has been done in both static [5] and dynamic load balancing paradigms [1, 3, 10, 11]. Static load balancing requires decomposing the application problem into multiple subtasks such that the communication cost among these subtasks is the minimum and the size of each subtask is approximately equal. This is an optimization problem which can be solved with various methods such as scattered decomposition, recursive bisection, simulated annealing [5], neural networks, genetic algorithms, and heuristics. On the other end of the spectrum, dynamic load balancing requires algorithms that are fast and are adaptive to the changes in the problem structure. Numerous dynamic load balancing algorithms have been proposed in the context of distributed systems [4, 17] that treat tasks as independent processing



0743-7315/94 $6.00

modules. However, for parallel systems, new solutions need to be found for supporting dynamic creation of workload and interprocessor communication [12]. A classification of load balancing techniques into four categories has been proposed in [18]. These categories are *by inspection*, *static*, *quasidynamic*, and *dynamic*. For the *by inspection* strategy, the problem structure is taken into account before writing the code. If the given program is partitioned before actually starting its execution, then load balancing is considered *static*. The difference between the *quasidynamic* and *dynamic* case is that in the former case load balancing is invoked infrequently during different computation phases, while in the latter case the load balancing algorithm is used frequently or continuously.

It is known that even simple dynamic load balancing algorithms provide better performance than not doing any load balancing [4, 19]. For interconnection-network-based multicomputer systems such as hypercube, distributed load balancing scheme based on task migration among nearest neighbors has gained considerable attention. In a number of studies [7, 11, 13], several variants of this strategy have been proposed and their effectiveness has been proven both by simulation [2] and experimentation [13]. Kalé [7] has compared one version of this strategy, known as *Contracting Within Neighborhood* (CWN), to the *Gradient Model* [8] and has shown that CWN spreads the load more quickly and performs better. In another study [13], the concept of load averaging among neighbors is introduced. The advantage of load averaging is that each node attempts to keep its own load equal to the average load among its nearest neighbors.

Unlike static tasks, scheduling of dynamically created tasks cannot be achieved by using a centralized control, especially if the number of processors are of the order of hundreds or thousands. A distributed scheduling algorithm performs better in such an environment. However, the performance of distributed scheduling algorithms cannot be improved beyond a certain level without increasing the complexity of collection and processing of load information that is required in order to make a better scheduling decision. An increase in this overhead can have a negative effect on the performance, since the scheduling of tasks must be done in real time. In a previous study [1], we have shown that both centralized and distributed algorithms are not always efficient, especially for a large system. In this paper we propose a dual-level hierarchical scheduling strategy for assigning parallel computation to large-scale hypercube systems.

In Section 2 of this paper, we describe the type of workload supported by the proposed strategy. Section 3 discusses the hypercube topology along with the characteristics that are used to partition the systems for providing a hierarchical control. The details of scheduling and simulation methodologies are presented in Sections 4 and 5, respectively. Section 6 contains performance results, and Section 7 concludes the paper.

## 2. WORKLOAD FOR PARALLEL COMPUTATIONS

Parallel computations can be classified as synchronous, loosely synchronous, multilevel, multiphase, and pipelined. The class of parallel computation considered in this paper is viewed as a collection of computational modules that can run asynchronously. Each computational module, referred to as a task, is an atomic unit of computation that, once started, executes to completion without any preemption. A task may, however, communicate with another task at the beginning and/or at the end of their execution. The relationship among tasks is represented by a directed dependency graph where nodes represent tasks and edges represent the dependency constraints. A task $a$ is said to be a parent of task $b$ if there is a directed edge from $a$ to $b$, and $b$ is called the child of $a$. A task has no more than one parent, and therefore, each task graph has only one initial task, known as root. Each parent can have zero or more children. A task cannot start its execution until its parent has finished its execution. The task graph can consist of many levels. The level of a task is the length of the longest path (number of edges) from the root. We assume that there is no interaction among siblings, and the only communication is between parent and its immediate children. This kind of task graph is also known as *DAG* (Directed Acyclic Graph). A *DAG* represents the *Task Precedence Graph* of a parallel computations and contains no cycles, as opposed to a *Task Interacting Graph* that may contain cycles because of undirected edges.

Although a static *DAG* has been extensively used in the computing literature to represent parallel algorithms, our workload is different in that the problem graph is completely dynamic, that is, the structure of the graph is not known a priori. Task graphs based on this assumption represent a wide range of recursive algorithms. In such algorithms, creation and synchronization of tasks proceeds recursively and conditionally. After completion, a task can spawn a number of tasks that can be independently scheduled onto the processors. The parent task may pass some information and data to the newly created tasks. Therefore, interprocessor communication and synchronization needs to be supported. Furthermore, the results from the child tasks need to be propagated back to the parent tasks.

This type of workload needs a special consideration and simple methods of task scheduling algorithms cannot be applied in this case. The performance of these algo-

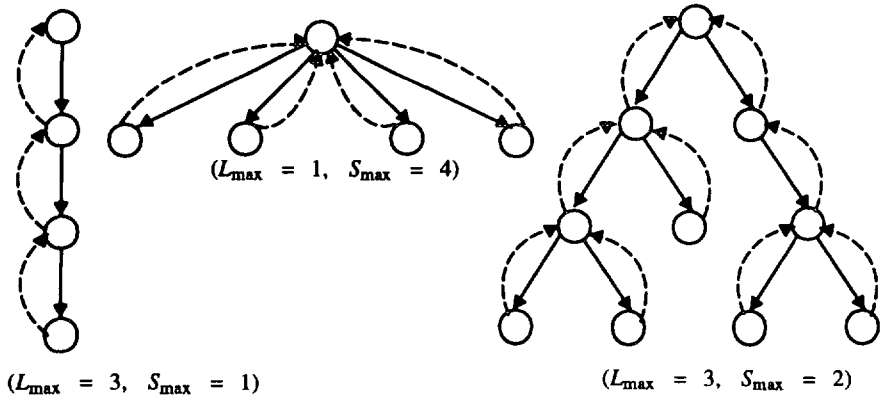


FIG. 1. Some of the possible graph structures representing parallel computations.

rithms have been thoroughly evaluated in the literature [9, 17]. However, most of these studies are based on simplified assumptions that ignore several aspects such as communication overhead and contention for resources.

We assume that the main task submitted to the system initially consists of a root task only. After its completion, it may generate a number of messages where each message results in creation of another task. The messages are assumed to carry the data from the parent to a child and other necessary information about the location of the parent, level number in the graph etc. The communication of these messages takes a certain amount of time. The task graph is characterized by two parameters, namely, the number of levels and the maximum number of children for each node, denoted as $L_{max}$ and $S_{max}$, respectively. For the workload, the number of children is generated randomly from a uniform distribution between zero and the maximum number of children allowed. After a task is executed, two possible actions can occur. First, if the task does not spawn more children, it sends the results back to its parent. Second, if it spawns more children, it gets suspended and waits for the results from its children. These assumptions are realistic and justifiable since the workload under these assumptions captures various types of algorithms mentioned above. Figure 1 illustrates some graph structures that can result by choosing different combinations of $L_{max}$ and $S_{max}$. The solid and dotted arrows show the messages for transferring data and results, respectively.

The workload considered here is similar to the *chare kernal*, a run-time system that supports the same type of computations mentioned above [14]. A number of problems such as N-Queens, Matrix Multiplication, Fibonacci, Iterative-Deepening A* Search, Prime Sieve, Gaussian Elimination and Romberg Integration, have been tested with the aid of chare kernel scheduler.

## 3. HYPERCUBE MULTICOMPUTERS

In this paper we evaluate the proposed scheduling strategy for message-passing MIMD multicomputers using binary $n$-cube network (hypercube). A number of hypercube network based MIMD and SIMD machines are commercially available such as the nCUBE, iPSC-2, iPSC-860, and the Connection Machine CM-2.

The hypercube network topology, which we denote as $Q_n$, consists of $2^n$ processors, each one represented as a binary codeword of length $n$. Two processors with codewords $x$ and $y$ are connected if and only if the Hamming distance, $H_{xy}$, between them is 1. It is known that for such a network the physical distance between any two processors $x$ and $y$, denoted as $L_{xy}$ is equal to $H_{xy}$. The diameter, $k$, of such a network which is the maximum of all the shortest distances in $Q_n$ is equal $n$. $Q_n$ is a distance-regular graph, that is the number of processors at distance $i$ from any processor is independent of the processor, and is given by the $i$th valency $v_i = \binom{n}{i}$ for $i = 0, 1, 2, \ldots, n$.

For the proposed hierarchical scheduling strategy, we partition a hypercube system into independent regions (spheres), centered at some set ($C$) of processors. These processors act as medians for maintaining the load information of their respective spheres. The selection of set $C$ and the size of each sphere are determined by the *graphical covering radius* $r$ in $Q_n$, which is defined for given a set $C$ as $r = Max_{i \in U}(Min_{j \in C}(L_{ij}))$, where $U$ represents the set of all the processors in the hypercube. This radius affects the amount of communication overhead incurred during task scheduling and the number of processors shared by competing tasks. For an arbitrary set $C$, finding $r$ in an arbitrary graph is an NP-hard problem [1].

There are a number of considerations involved in choosing the set $C$. The first one is the size of each sphere. Let the size of a sphere assigned to a processor

$$M = \begin{matrix} 0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,0\,1\,1\,1 \\ 0\,0\,1\,0\,1\,1\,1\,0 \\ 0\,1\,0\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,0\,0\,1 \\ 0\,1\,1\,1\,0\,0\,1\,0 \\ 0\,1\,1\,0\,0\,1\,0\,1 \\ 0\,1\,0\,0\,1\,0\,1\,1 \end{matrix} \qquad M^C = \begin{matrix} 1\,1\,1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,0\,1\,0\,0\,0 \\ 1\,1\,0\,1\,0\,0\,0\,1 \\ 1\,0\,1\,0\,0\,0\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,0 \\ 1\,0\,0\,0\,1\,1\,0\,1 \\ 1\,0\,0\,1\,1\,0\,1\,0 \\ 1\,0\,1\,1\,0\,1\,0\,0 \end{matrix}$$

FIG. 2. A Hadamard matrix and its complementary matrix.

$x \in C$ be denoted by $|S_i(x)|$, where $i$ is the radius of this sphere. We can notice the total size of the sphere is given as $|S_i(x)| = \sum_{j=0}^{i} v_j$. For partitioning of the hypercube, we need a $\delta$-uniform set $C$ (for some $\delta$ to be determined) which is the maximal set of processors in $Q_n$, such that the graphical distance among the medians processors is at least $\delta$ and $|S_i(x)|$ is constant $\forall x \in C$, where $i$ is the covering radius of $C$. The size, $|C|$, depends on the selection of $\delta$. Intuitively, larger $\delta$ yields smaller $|C|$, but it results in spheres with larger size. It can also be observed that reducing $|C|$ increases the sphere size and vice versa. Since a median processor needs to maintain the load state information of all the processors within the sphere (discussed in the next section), the diameter of the sphere should be as small as possible. Another consideration is that the whole network should be uniformly partitioned into spheres which should be symmetric and equal in size thus leading to a symmetric algorithm for load balancing.

For $Q_n$, we propose an efficient solution for finding the set $C$ using a combinatorial structure called *Hadamard matrix* [6]. For this purpose the set $C$ is selected from the code generated by the rows of a Hadamard matrix $M$ (which is a square matrix of order $n$ with entries $\pm 1$; see [1]) and its complement $M^C$. Figure 2 shows the Hadamard matrix of order 8 and its complement. Each row of set $C$, also called *Hadamard code*, represents the binary address of a median node. When $n$ is a multiple of 4, $|C| = 2n$ for $Q_n$. For all other values of $n$, a Hadamard matrix can be modified to generate the set $C$ [1].

For $Q_8$ network, the spheres for the processors having binary addresses 00000000 and 11111111 are shown in Fig. 3. The covering radius $r$ in this case is equal to 2 and the valencies $v_0$, $v_1$ and $v_2$, $\forall x$, have values 1, 8 and 28, respectively, corresponding to a total volume of the sphere $|S_2(x)|$ equal to 37. The rest of the 14 ($=2n - 2$) medians with codewords corresponding to the rows of set $C$ are located midway between processor number 0 and 255, that is at a distance equal to the radius. A similar configuration can be visualized for each of the remaining 14 medians, such that from each median there is one median at distance 8, and 14 medians at distance 4.

There are a number of reasons for choosing Hadamard code for the set $C$. The noted among them is the fact that the set $C$ provides the maximal $k/2$ (radius)-*uniform set* for $Q_n$ [1]. A Hadamard matrix can be generated by various ways (see, e.g., [6]).

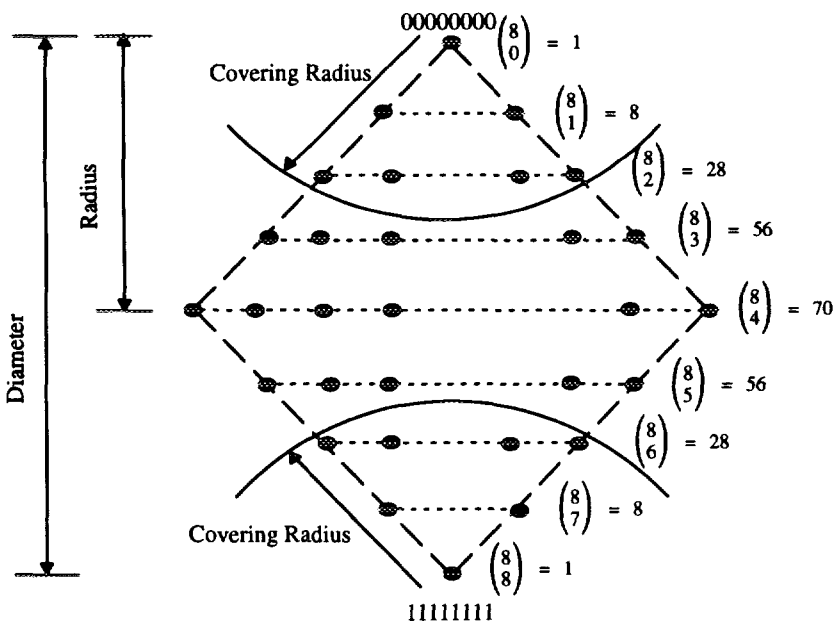


FIG. 3. The structure of $Q_8$ network.

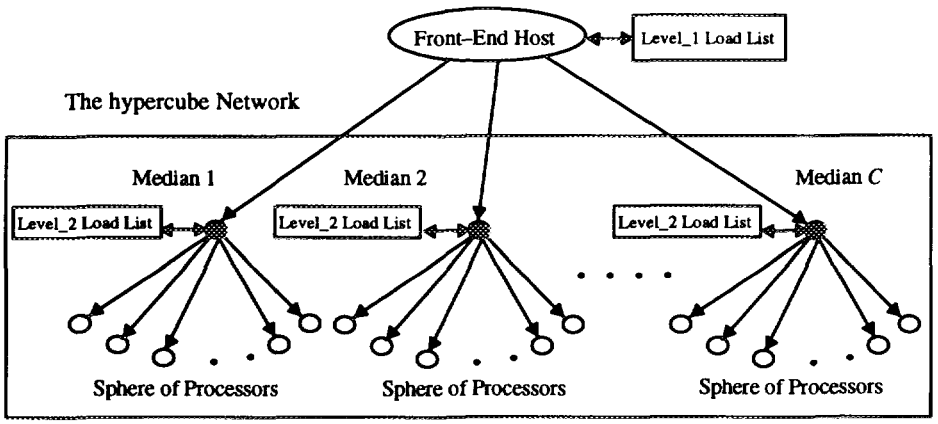


FIG. 4. The dual-level hierarchical scheduling scheme.

## 4. THE PROPOSED HIERARCHICAL SCHEDULING STRATEGY

Based on the aforementioned partitioning of hypercubes, we now present a hierarchical approach for scheduling and load balancing of dynamic task graphs in these systems. The motivation for this approach is that many contemporary parallel computers are usually equipped with a front-end host machine. The users interact with the parallel machine through the host which can be used to assign tasks to a set of processors using some "pseudoglobal" knowledge. Within the parallel system, further load balancing can be achieved by using some "local" knowledge. Figure 4 illustrates a conceptual model for this dual-level scheduling approach, the details of which are given below.

### *4.1. First Level Scheduling*

The first level scheduling is handled by the host computer that acts as a centralized scheduler. That is, when a user submits a new task to the host, the host assigns the root task to one of the median processors. The scheduling decision at this level is based on the load information, which is a linked list containing $C$ elements. We will call this linked list as *level_1 load list*. The elements of this list are the addresses (codewords) of medians nodes. The position of the address of a median node in the list is maintained according to the number of root tasks currently present in the sphere of that median. The first element of the linked list contains the median with the minimum number of root tasks assigned to it. This list is updated whenever either the host assigns a root task to a median node or a root task is completed. Since the host has no information about the complete structure of a task and it only interacts with the parallel system through the median nodes, this load metrics does not indicate the precise load status of the spheres. Rather, it provides an estimation of the load status of spheres in terms of root tasks only. This we refer to as "pseudoglobal" knowledge. At this level, it is not possible to maintain the load information in terms of the number of subtasks present at each individual processors (nodes) within the sphere, since that information changes rapidly and also it can result in a large amount of communication traffic.

For the proposed scheme, it can be noticed that the number of medians is only of the order of $O(\log n)$. Therefore, the complexity for maintaining the *level_1* list is far less than maintaining the list of all the nodes, as is required in a centralized scheduling scheme. It can also be noticed that there is no other communication between the host and the median processors since the host updates its list whenever it submits a new task to a median node or when a task gets completed and the results are returned back to the host.

### *4.2. Second Level Scheduling*

At the second level, task scheduling and load balancing is carried out by the individual processors in consultation with the median nodes of their respective spheres. For this purpose, another linked list, called *level_2 load* load list, is maintained by every median node. This list contains addresses (codewords) of all the processors in the sphere controlled by that median node. The position of a processor in the list is determined according to the number of subtasks currently assigned to that processor. When a root task is submitted to a median node by the host, that node consults the *level_2* load list and assigns this task to the most lightly loaded processor within its sphere. Execution of the root task may spawn new subtasks. The spawning processor then consults the median node to get the addresses of the most lightly loaded processor(s) within the sphere for scheduling the newly

spawned task(s). Again, the median uses the *level_2* list to satisfy this request. The messages for the invocation of subtasks are sent to the selected processors by the requesting processor, where the subtasks are subsequently executed. A subtask may in turn spawn more new subtasks for which the same procedure is repeated. Upon completion of a subtask, a processor sends a message to its median node to update the *level_2* load list. A task that spawns child-tasks gets suspended and waits for the results from its child-tasks. Accordingly, if a subtask has a parent, it sends back results to the processor where the suspended parent is waiting. The suspended parent is invoked after receiving the results. It then sends back its own results to its parent. If the parent task is the root task, it finishes its execution and a message from the median node is sent to the host which updates the *level_1* load list. Tasks arriving at any processor are executed using the First Come First Serve (FCFS) discipline.

### 4.3. Advantages of the Proposed Scheme

The proposed dual-level hierarchical scheduling strategy have many advantages. Some of these advantages are described below.

*4.3.1. Dual Level Load Balancing.* As mentioned above, for a massively parallel system, centralized or decentralized scheduling is not practical. A natural choice is a hierarchical strategy. One such strategy known as wave scheduling has been proposed in Ref. [16] that uses a multiple level hierarchy consisting of processors labelled as workers, managers, supermanagers, and so forth. The hierarchy is assumed to be based on a virtual machine which does not reflect the actual physical structure of the underlying network. Also, the dynamic creation of task has not been considered in that study. In addition, no definite methodology is given to describe the workload and performance of that approach which makes its merits. On the other hand, the strategy proposed in this paper assumes a realistic workload.

*4.3.2. Load Spreading.* For efficiency and good performance, it is required that the computations should be spread out in the network as far as possible in order to reduce resource contention. The proposed strategy achieves this objective by assigning initial computations to the medians of different spheres that are maximally spread set of processors in the hypercube topology. In that sense the proposed strategy spreads the creation of dynamic workload as much as possible.

*4.3.3. Increased Locality for Interprocessor Communication.* Since the task model requires efficient communication among parent and children, locality is an important consideration. In order to reduce the communication overhead, tasks must be assigned to processors that are closer to each other. In distributed strategies, the common practice is to allow a task to migrate from processor to processor until it finds a suitable processor. During this migration or drifting phase, a task may travel a number of hops with respect to the task to which it needs to communicate. The proposed strategy attempts to reduce this migration overhead by confirming the subtasks of a task graph within the same sphere.

*4.3.4. No Redundant Migrations.* Generally, in distributed scheduling strategies, allowing a task to migrate up to some specific number of hops distance may enable the task to find an idle or lightly loaded processor. However, multiple migrations proportionally increase the duration of drifting phase which can result in a substantial penalty. An additional disadvantage of multiple migrations is the problem of task thrashing [1]. The proposed strategy eliminates these problems by allowing transferring of a task only once from the processor where it is created to the most lightly loaded processor within its sphere.

*4.3.5. Efficient Parallel Scheduling.* Efficient load balancing can be achieved if every new task is assigned to the most lightly loaded processor of the system. A centralized control can be used for such purpose if the size of the system is quite small. The hierarchical scheduling which is intended for large-scale systems achieves this goal by dividing the larger system into multiple smaller regions and then by assigning a centralized control for each region.

Another important aspect of the scheduling is that when a number of new subtasks are created at a processor, it requests a list of processors to schedule the new tasks, from its median node. The median node provides this information using the sorted *level_2* load list. This whole interaction can be handled via a single request.

## 5. SIMULATION DETAILS

A discrete-event simulator has been developed to evaluate the performance of the proposed strategy for a hypercube multicomputer system. The processors in this system are assumed to operate in an asynchronous environment. The network is assumed to support message passing using point-to-point communication among the processors. The simulator simulates the partitioned hypercube topology based on the Hadamard Matrix.

For the workload, the root tasks are first submitted to the host which then assigns it to the median processors through first level scheduling as described in the previous section. As the computation proceeds, the root tasks are divided into subtasks. For the purpose of simulation, $L_{max}$ and $S_{max}$ of the task graph are taken as input parameters. The number of children for each node of the task

```
Level1_Schedule(main_task)
begin
        Get_level1_load_info()
        best_median = level1_load_list[0]
        Level2_Schedule(main_task -> root, best_median)
        Update_level1_load_list(best_median, add_task)
end
Level2_Schedule(sub_task, my_median)
begin
        Get_level2_load_info(my_median)
        best_processor = level2_load_list[my_median][0]
        Send_task(best_processor, sub_task)
        Update_level2_load_list(my_median, best_processor, add_task)
end
Complete_Sub_Task(sub_task)
    begin
        Update_level2_load_list(my_median, this_processor, delete_task)
        if (sub_task -> unscheduled_children > 0) then
            for i = 1 to sub_task -> unscheduled_children
                begin
                    Level2_Schedule(sub_task -> child[i], my_median)
                end
            Suspend_Task(sub_task)
        else if (sub_task == root) then
                if (all_results_received) then
                    Finish_task(main_task)
                    Update_level1_load_list(my_median, delete_task)
        else
        Send_Results(this_processor, sub_task -> parent -> processor, sub_task, sub_task -> parent)
end
```

FIG. 5. Pseudo code describing the events in the first and second-level scheduling and subtask completion procedures.

graph are generated using a uniform probability distribution between 0 and $S_{max}$. We assume that the execution time of a complete task graph is a random variable with an exponential distribution having an average of 1 time-unit. This time is equally divided among all the subtasks of the whole task graph. Figure 5 provides a pseudo code showing the possible events taking place during both levels of scheduling and when a subtask completes its execution. The task communication time is also assumed to be an exponentially distributed random variable. The communication time for sending results from child tasks to their parent tasks, is assumed to be negligible, since only results are passed back to the parents. For stability reason, we assume that the utilization ratio for every processor remains below 1; otherwise, the queue length of each processor approaches infinity, making it an unstable system. For this purpose, the task generation rate is accordingly adjusted.

For the purpose of comparing the proposed scheme with distributed scheduling, we use a well known neighborhood averaging algorithm [7, 13]. In this algorithm, each processor makes autonomous scheduling decisions by collecting the load status information from its immediate neighbors. A task is either scheduled in a local execution queue or it is migrated to one of the neighbors over a link (communication channel). In this strategy, the initial tasks are submitted to all the processors with an equal average arrival rate.

The performance measure selected for the proposed strategy is the average response time of a whole task (task graph). This response time is the residence time of the main task (task graph) in the system and it consists of the processing times of all the subtasks belonging to the task graph, some of which may run in parallel, and the interprocessor communication times to transfer tasks.

A number of parameters can be selected as input for the simulator. These include the utilization ratio of a processor, the average communication and computation times, the total number of root tasks to be generated, the maximum number of hops a task can traverse (for the neighborhood averaging distributed scheduling algorithm), and the task graph parameters, $L_{max}$ and $S_{max}$. A limit of 10 on the maximum number of hops a task can traverse is selected for the distributed scheduling strategy. The rest of the parameters are varied for which the simulation results are presented in the next section. It is important to mention that the performance results obtained with a discrete-event simulation are valid only if the simulation is run for a time which is long enough to capture the average behavior of the system. As opposed

to some of the previous studies where the results represent the performance of a few sample task graphs, we have generated 100,000 task graphs and took the average of their response times for calculating the overall response time. Furthermore, as in every valid discrete event simulation, multiple runs of the same simulation have been conducted by using different sets of seeds for random number generation. The final results are the averages of those multiple runs with a 95% confidence interval.

## 6. PERFORMANCE RESULTS

In this section, we present the performance of the proposed strategy obtained through the simulation of a hypercube, $Q_8$, having 256 processors and compare it with the above mentioned distributed scheduling. $Q_8$ is partitioned into 16 identical spheres using 16 median processors, as described in Section 3. The impact of various parameters on the performance has been studied that include the loading conditions of processors, the degree of parallelism and depth of computation (by changing $S_{max}$ and $L_{max}$, respectively), and the cost of communication by varying value of the average task communication rate. We describe these results below.

### 6.1. *Impact of Amount of Load*

The load of a system is defined by the ratio of the average task generation rate to the average task processing rate at a processor. We analyze the impact of load, called utilization ratio, by controlling the task generation rate in the simulation. Since, the dynamic creation of subtasks also produces additional load, we have selected four different types of task graphs, representing binary, tertiary, and quaternary trees, and linear chains by choosing various values of $L_{max}$ and $S_{max}$. Figures 6, 7, 8, and 9, respectively, show the average response time of these task graphs for processor utilization ratio that varies from 0.4 to 0.9. The task communication rate for these figures has been selected to be 20 tasks/time-unit. We can note from Fig. 6, that for binary tree computations, a substantial improvement in the average response time is obtained by using the hierarchical scheduling strategy as compared to the distributed scheduling strategy. This figure also provides two other important pieces of information. First, the curve for the response time obtained for the hierarchical strategy is rather smooth for utilization ratios of 0.4 to 0.7. Second, the performance of both the strategies increases with an increase in load. This behavior is due to the fact that the median of a sphere allows a newly created task to be scheduled at the processor that has the least load within its sphere. This makes a task to go through minimum queuing delay. Additionally, the communication overhead for scheduling a task is very small due to a single migration. For utilization ratio up to 0.7, efficiency of hierarchical scheduling and load balancing does not permit building of large queues at processors. As a result, the average queueing delay of a subtask is reduced. At utilization ratios higher than 0.7, building of queues is inevitable since there are abundant tasks in the system. However, even at very high utilization ratios such as 0.9, the performance of the proposed strategy is substantially better than that of the distributed strategy.

Figure 7 shows results for tertiary trees for which both $L_{max}$ and $S_{max}$ are equal to 3. This task graph model generates more task as compared to the binary tree task graph. Although the computational load submitted to the system depends on the external task arrival rate, the presence of extra dynamically created tasks requires more scheduling decisions. Again, hierarchical scheduling exhibits a better performance. The response time results for quaternary trees, having $L_{max} = 3$ and $S_{max} = 4$, at various loading conditions are shown in Fig. 8. This task graph model generates a very large number of subtasks (this number can be as high as 85 subtasks for a single task graph). The increased response time for the distributed scheduling, as shown in Fig. 8, is due to the fact that a larger number of subtasks in the task graph increases the synchronization overhead of the graph. This overhead is less in hierarchical scheduling because the load information maintained by the median node allows each dynamically created task to get executed at the most lightly loaded processor of the sphere. This faster execution of individual subtasks in turn reduce the synchronization penalty for the parent task. On the other hand, in distributed scheduling, due to load balancing among neighboring processors, some subtasks may get executed quickly but some may migrate deep into the network and may traverse a number of hops. Some of them may even reach their maximum hop limit in which case they are forcibly executed without further migration. The delay in the execution of these subtasks and the communication overhead increase the waiting time of the suspended parent tasks. As a result, the response time of the whole task graph increases and thus results in the degradation of the performance of distributed scheduling. This effect becomes more dominant with an increase in the external arrivals. Comparing Fig. 7 with Fig. 6, we note that the response time curve for distributed scheduling strategy has shifted a little bit upward for low utilization ratio while the curve for the hierarchical strategy has shifted downward. We also observe that the response time curve of distributed scheduling has further shift upwards in Fig. 8, indicating an increase in the response times. The difference in the performance of the two strategies is also large at low loading condition for this graph model as compared to that of the previous graph model. This indicates that hierarchical scheduling achieves a better load balancing with reduced task granularity and larger num-

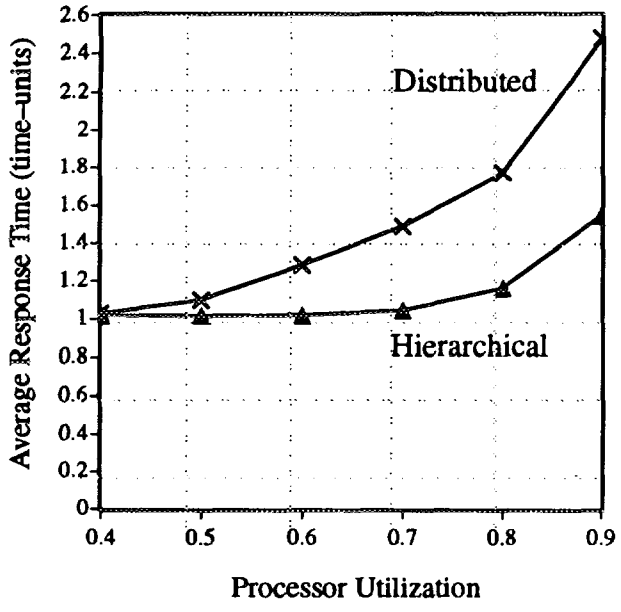


FIG. 6. Results for the binary tree task graph ($L_{max} = 3$, $S_{max} = 2$).

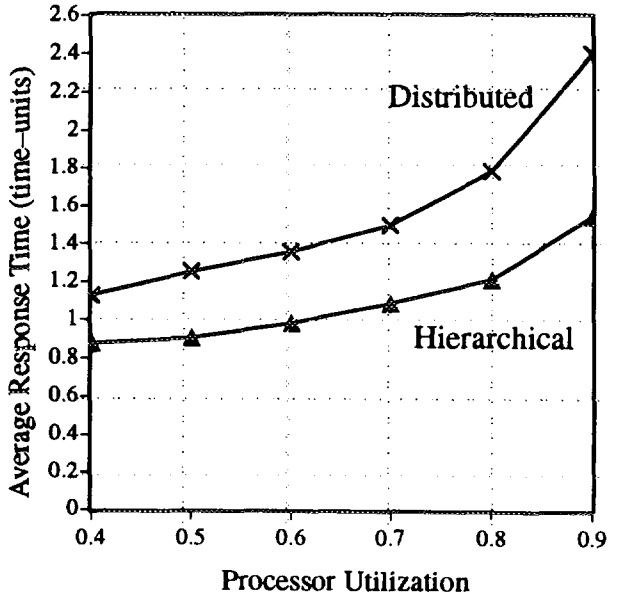


FIG. 7. Results for the tertiary tree task graph ($L_{max} = 3$, $S_{max} = 3$).

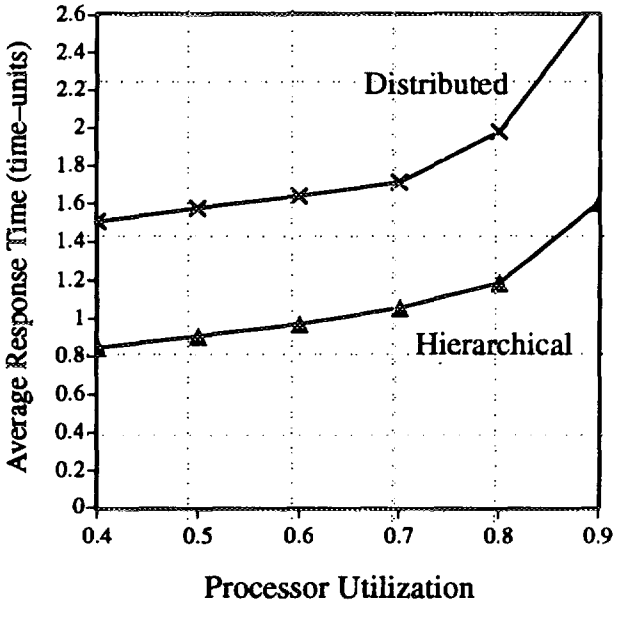


FIG. 8. Results for the quaternary task graph ($L_{max} = 3$, $S_{max} = 4$).

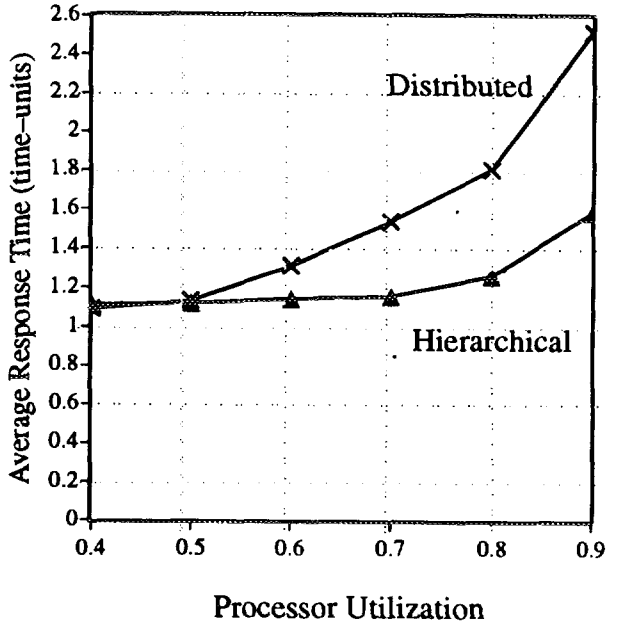


FIG. 9. Results for the linear task graph ($L_{max} = 7$, $S_{max} = 1$).

FIGS. 6–9. The average response time yielded by the hierarchical and the distributed scheduling strategies at varying processor utilization, for various task graphs.

ber of sub-tasks. This observation is explained in more detail in the next section.

Figure 9 shows the results for task graphs with $L_{max} = 7$ and $S_{max} = 1$. The task graph in this case consists of a series of computational modules, forming a linear structure. Note that in this case, a task graph with the number of level less than 7 is also possible since the number of sub-tasks at a level can also be 0. In that case, the task graph does not expand. For such tasks, the computation is *more sequential than parallel*. For this type of task model, the distributed scheduling strategy performs marginally better than the hierarchical strategy at low utilization ratios. This is because at low loading, if a processor does not have enough tasks waiting in the queue, it tends

to keep the newly created single task locally. In the distributed scheduling, this behavior eliminates the communication overhead. On the other hand, in the hierarchical strategy, the root task may have to make one migration from the median processor to some other processor within the sphere. However, this better performance of the distributed scheduling strategy is sustained only for a small range of load. At high load, more processors attempt to transfer their load to other processors and, therefore, fewer newly created tasks are kept locally, which results in a degradation of performance. The hierarchical strategy transfers tasks to better destinations as compared to the distributed strategy. Consequently, for utilization ratios higher than 0.5, the response time for the proposed strategy improves.

### 6.2. *Effect of Parallelism and Computational Depth*

In this section, we investigate the effect of parallelism and depth of computation in more detail. For this purpose, three sets of experiments for the task graph have been selected for obtaining performance results.

For the first experiment, we study the effect of parallelism by setting $L_{max}$ equal to 1 and varying $S_{max}$ from 2 to 12. This type of task graphs represent parallel computations where a single problem is decomposed into multiple independent tasks. No further tasks are created and the whole problem completes when the last task sends its results back to the root task. The performance results for this case are presented in Fig. 10, which shows the response times for both the scheduling strategies at two utilization ratios, 0.5 and 0.8. The communication rate has been again fixed as 20 tasks per time-unit. At both low and high loading conditions, the hierarchical strategy is shown to outperform the distributed strategy. The difference in performance increases with an increase in $S_{max}$. As expected, the difference in the performance of both strategies is higher when utilization ratio is 0.8. It can be noted that the response time first decreases with an increase in $S_{max}$ and then it starts increasing. This behavior indicates that the optimal performance is achieved at a certain degree of parallelism ($S_{max}$) for both the strategies. The reason is that for smaller values of $S_{max}$, there is not enough parallelism in the task graph to yield a good response time. On the other hand, for a large value of $S_{max}$, the penalty for synchronizing a large number of subtasks at any level can be severe. Hence, there is a range of $S_{max}$ at which the response time is the minimum, as is clear from Fig. 10. It is interesting to note that for the same set of parameters, this range is different for both the strategies. For distributed scheduling, this range consists of some smaller values of $S_{max}$ because of higher synchronization overhead, as explained above, and due to the excessive subtask migrations. This range, however, is also dependent on the utilization ratio. The task response time for the values of $S_{max}$ greater than 12 have not been determined due to the high simulation cost, but we conjecture that the response time will increase for those values.

For the second experiment, we have used task graph models for which $S_{max}$ was kept fixed at value 1 and while $L_{max}$ was varied from 1 to 9. The results for this task graph are presented in Fig. 11, which shows a slight increase in the response time for both the strategies as $L_{max}$ is increased, at high load. There is no significant change in response time at low loading conditions. Also at low load, the performance of distributed scheduling strategy is slightly better than that of hierarchical strategy. The reason for this behavior is already explained in the previous section.

In another experiment, we have examined how binary trees of various heights perform under both strategies, by keeping $S_{max}$ fixed as 2 and varying $L_{max}$ from 1 to 4. The simulation results for this experiment are illustrated in Fig. 12. We note that at both high and low loading conditions, the response time increases as the depth of computation, $L_{max}$, is increased. This is due to the fact that with an increase in the number of levels, the number of subtasks in a tree increase substantially. This can lead to scattering of subtasks to different processors and synchronization can cause large delays in completing the execution of entire task graph. However, the performance of the hierarchical strategy remains superior to the distributed strategy.

### 6.3. *Effect of Communication Cost*

Finally, to evaluate the impact of communication on the performance of both scheduling strategies, we present simulation results that compare the two strategies for different task communication rates. These results are presented in Figs. 13, 14, 15, and 16 for four different utilization ratios: 0.5, 0.6, 0.7, and 0.8, respectively. The task graph selected for these results has $L_{max} = 3$ and $S_{max} = 2$. These results serve two purposes. First, the variations in response times for different communication rates can be examined, and second, the combined effect of load and communication cost can be analyzed. As expected, the task response time is directly affected by the communication cost incurred for sending a task generation message. For low and medium loading conditions, corresponding to utilization ratios of 0.5 and 0.6, respectively, the response time improves rapidly when the task transfer rate varies from 4 to 12 tasks per time-unit. Beyond this range, it starts saturating at a fixed value. On the other hand, at high loads such as utilization ratios of 0.7 and 0.8, the response time decreases more sharply and saturates at higher values of communication rate.

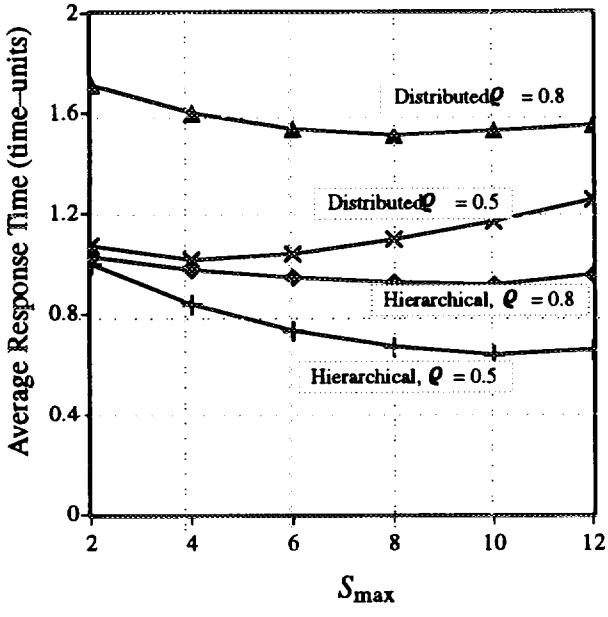


FIG. 10. The impact of the degree of subtask generation on the average response time with $L_{max} = 1$. The symbol $r$ indicates utilization ratio per processor.

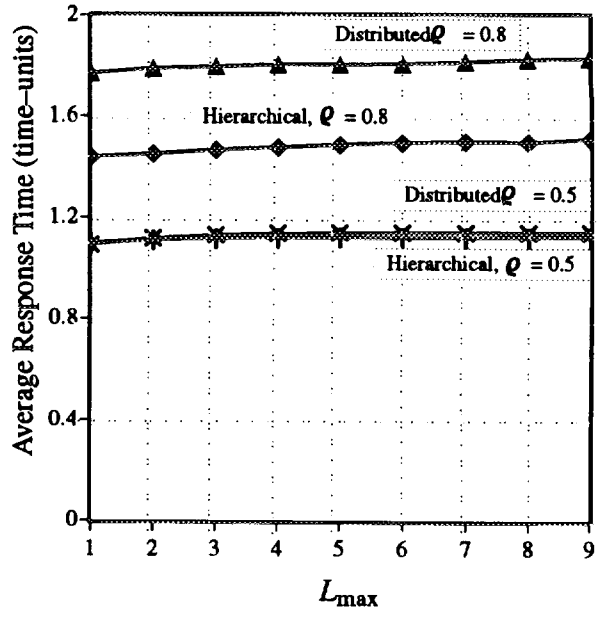


FIG. 11. The impact of number of levels in the task graph on the average response time with $S_{max} = 1$.

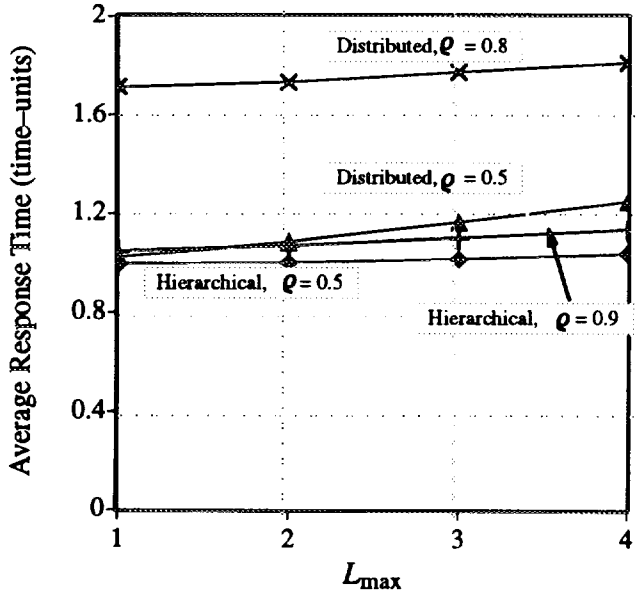


FIG. 12. The comparison of average response time for both strategies for binary tree of various depths. $S_{max} = 2$.

At a higher communication rate, the network latency is negligible and only the queuing delays affect the quality of load balancing algorithms for both the strategies. From these figures, we note that at higher communication rates, the difference between the performance of the two schemes increases with an increase in the processor utilization ratio. This suggests that the distributed load balancing algorithm is outperformed by the hierarchical strategy. This is due to the fact that load balancing within a sphere results in a much better utilization of resources since the scope of load information at the sphere level is greater as compared to the distributed scheme where such a scope is limited only to immediate neighbors.

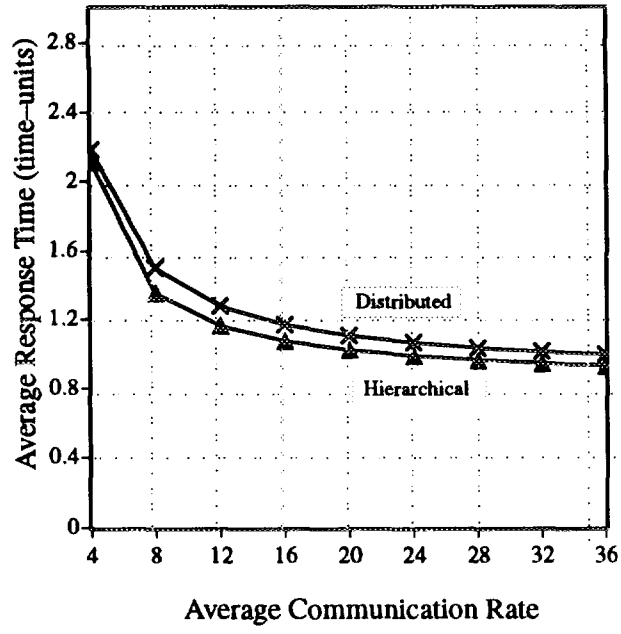


FIG. 13. Results for processor utilization = 0.5.

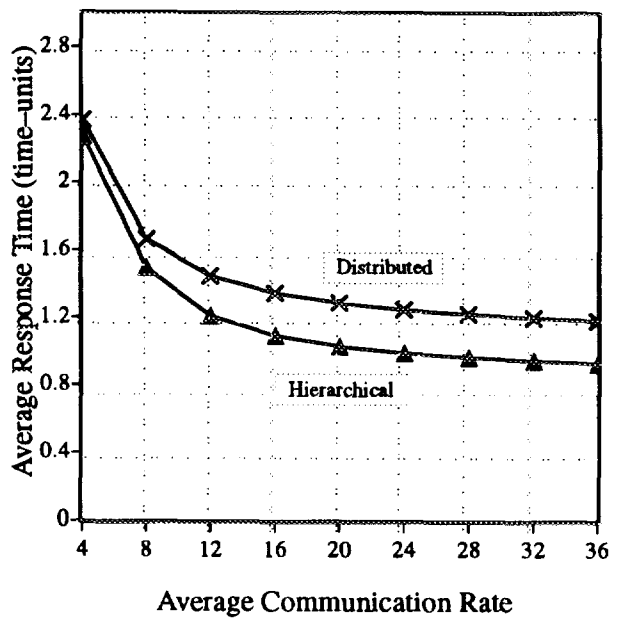


FIG. 14. Results for processor utilization = 0.6.

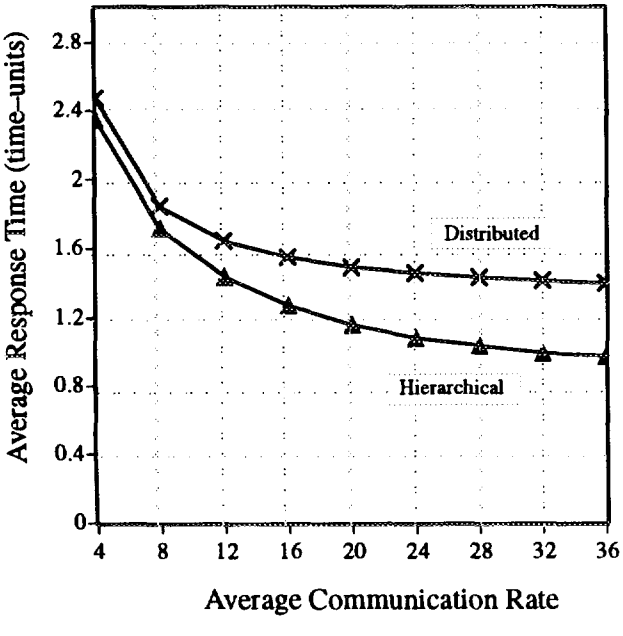


FIG. 15. Results for processor utilization = 0.7.

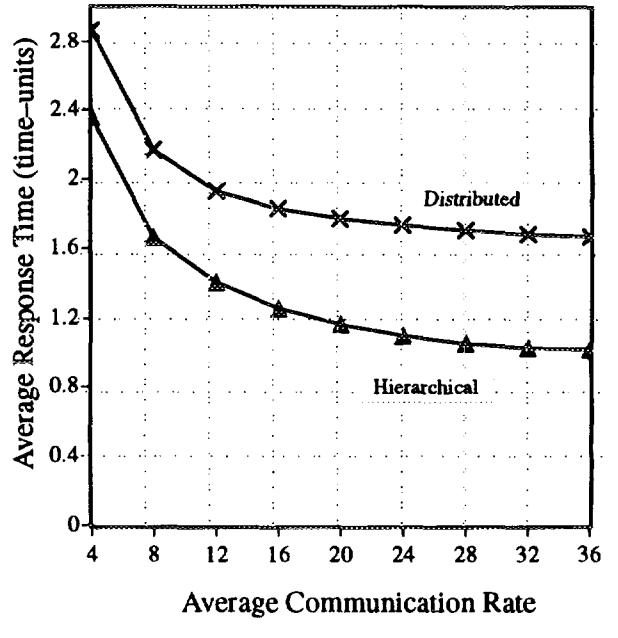


FIG. 16. Results for processor utilization = 0.8.

FIGS. 13–16. The impact of communication rate on performance.

## 7. CONCLUSIONS

In this paper, we have proposed a dual-level hierarchical scheduling scheme for dynamically evolving computations on the hypercube system using its partitioning capability into identical spheres. The motivation for this approach is that many contemporary parallel computers are usually equipped with a front-end host machine which controls the multicomputer system and can be used to assign initial tasks to the median processors of the partitioned system using a *pseudoglobal* knowledge. Within each partitioned region, further load balancing is achieved by using the load information maintained by the median processor for that region. The management of

load information at both levels is handled using linked lists that are easy to manipulate. An extensive simulation study has revealed that the proposed hierarchical scheduling strategy yields substantial improvement over a well known distributed scheduling algorithm. In particular, its performs very well at high loading conditions and for task graphs having a high degree of parallelism.

## ACKNOWLEDGMENT

The work presented in this paper was partly supported by National Science Foundation Grants CDA-9121771 and CCR-8809165.

## REFERENCES

1. Ahmad, I., and Ghafoor, A. Semi-distributed load balancing for massively parallel multicomputer systems. *IEEE Trans. on Software Engrg.* **SE-17** (Oct. 1991), 987–1004.
2. Ahmad, I., Ghafoor, A., and Mehrotra, K. Performance prediction for distributed load balancing on multicomputer systems. *Proc. of Supercomputing '91*, New York, Nov. 1991, pp. 830–839.
3. Baumgartner, K. M., and Wah, B. W. Gammon: A load balancing strategy for local computer systems with multiaccess networks. *IEEE Trans. Comput.* **C-38** (Aug. 1989), 1098–1109.
4. Eager, D. L., Lazowska, E. D., and Zahorjan, J. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Engrg.* **SE-12** (May 1986), 662–675.
5. Fox, G. C. A review of automatic load balancing and decomposition methods for the hypercube. In Shultz, M. (Ed.). *Numerical Methods for Modern Parallel Computer Architectures*. Springer-Verlag, Berlin, 1988, pp. 63–76.
6. Hall, M., Jr. *Combinatorial Theory*, 2nd ed. Wiley, New York, 1986.
7. Kalé, L. V. Comparing the performance of two dynamic load distribution methods. *Proceedings of International Conference on Parallel Processing*, 1988, pp. 8–12.
8. Lin, F. C. H., and Keller, R. M. Gradient Model: A demand driven load balancing scheme. *Proceedings of 6th International Conference on Distributed Computing Systems*. 1986, pp. 329–336.
9. Madala, S., and Sinclair, J. B. Performance of synchronous parallel algorithms with regular structures. *IEEE Trans. Parallel Distribut. Systems* 2 (Oct. 1991), 495–502.
10. Ni, I. M., and Hwang, K. Optimal load balancing in multiple processor system with many job classes. *IEEE Trans. Software Engrg.* **SE-11** (May 1985), 491–496.
11. Qian, X., and Yang, Q. Load balancing on generalized hypercube and mesh multiprocessors with LAL, *Proceedings of the 11th International Conference on Distributed Computing Systems*. May 1991, pp. 402–409.
12. Reed, D. A. The performance of multimicrocomputer networks supporting dynamic workloads. *IEEE Trans. Comput.* **C-33** (Nov. 1984), 1045–1048.
13. Saltore, V. A. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks. *Proceedings of the 11th International Conference on Distributed Computing Systems*. May 1991, pp. 328–335.
14. Shu, W. Chare kernal and its implementation on multicomputers. Ph.D. Dissertation, Univ. of Illinois, Urbana, Illinois, 1990.
15. Singh, A., Shaeffer, J., and Green, M. A template-based approach to the generation of distributed applications using a network of workstations. *IEEE Trans. Parallel Distribut. Systems* **2** (Jan. 1991), 52–67.
16. Tilborg, A. M., and Wittie, L. D. Wave scheduling—Decentralized scheduling of task forces in multicomputers. *IEEE Trans. Comput.* **C-33** (Sept. 1984), 835–844.
17. Towsley, D., Rommel, C. G., and Stankovic, J. A. Analysis of fork-join program response times on multiprocessors. *IEEE Trans. Parallel and Distribut. Systems* **1** (July 1990), 286–303.
18. Williams, R. D. Performance of dynamic load balancing algorithms for unstructured mesh calculation. *Concurrency: Practice and Experience* **3** (Oct. 1991), 457–481.
19. Xu, J., and Hwang, K. Heuristic methods for dynamic load balancing in a message-passing supercomputer. *Proceedings of Supercomputing '90*. Nov. 1990, pp. 888–897.

Received July 1992; revised November 24, 1992; accepted December 23, 1992

---

ISHFAQ AHMAD received his B.Sc. degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan, and his M.S. degree in computer engineering and Ph.D. degree in computer science from Syracuse University. His research interests include various aspects of parallel and distributed computing, high performance computer architectures and their assessment, and performance evaluation. Currently, he is a faculty member in the Department of Computer Science at Hong Kong University of Science and Technology. He received the Best Student Paper Awards at Supercomputing '90 and Supercomputing '91. Dr. Ahmad is a member of the IEEE Computer Society and ACM.

ARIF GHAFOOR received his B.Sc. degree in Electrical Engineering from the University of Engineering and Technology, Lahore, Pakistan in 1976, and the M.S., M.Phil., and Ph.D. from Columbia University, in 1977, 1980, and 1985, respectively. Currently, he is an associate professor in the School of Electrical Engineering, at Purdue University. His research interests include design and analysis of parallel and distributed systems and multimedia systems. His research in these areas has been funded by the DARPA, the National Science Foundation, NYNEX, AT&T, Fuji Electric Corp., General Electric, and the New York State Center for Computer Application and Software Engineering (CASE) at Syracuse University. He has served on the program committees of various IEEE conferences. Currently, he is serving on the editorial board of Multimedia Systems. He is consultant to many companies, including Bell Labs and General Electric, in the area of telecommunications and distributed systems. He is a senior member of the IEEE and a member of Eta Kappa Nu.

GEOFFREY FOX is an internationally recognized expert in the use of parallel architectures and the development of concurrent algorithms. He is also a leading proponent for the development of computational science as an academic discipline and a scientific method. His research on parallel computing has focused on development and use of this technology to solve large scale computational problems. Fox directs ACTION-NYS, which is focused on accelerating the introduction of parallel computing into New York State industry. Fox earned his Ph.D. in theoretical physics from Cambridge University in 1967. His research experience includes work at the Institute for Advanced Study at Princeton, Lawrence Berkeley Laboratory, Cavendish Laboratory at Cambridge, Brookhaven National Laboratory, and Argonne National Laboratory. He has served as Dean for Educational Computing and Assistance Provost for Computing at Caltech. Fox is currently a professor of computer science and physics at Syracuse University and Director of the Northeast Parallel Architectures Center. He coauthored *Solving Problems on Concurrent Processors* and edits *Concurrency: Practice and Experience* and the *International Journal of Modern Physics: C*.