Task Allocation on a Network of Processors

Tsan-sheng Hsu, Member, IEEE Computer Society, Joseph C. Lee, Dian Rae Lopez, Member, IEEE Computer Society, and William A. Royce

Abstract—This paper studies the scheduling of tasks on a pool of identical workstations in a network where message passing is used for data transfer and communication between processors and where the precedence relations among tasks form a send-receive graph. Our parallel computation model differs from previous models by including all of the following practical considerations: 1) The sending and receiving of multiple messages from one processor to another is performed sequentially, 2) communication overhead is proportional to the message size, and 3) the starting and ending task must be performed on the same machine. These factors are crucial when performing parallel task execution using a pool of workstations whose communication primitives are provided by off-the-shelf packages, such as PVM, and whose message sizes are nontrivial. Although our model is new, using reduction from other well-known scheduling results shows that finding a scheduling with the optimal makespan is NP-hard. Our focus, therefore, is on developing and analyzing approximation algorithms for this problem. When the number of workstations in the network is abundant, a linear approximation algorithm is given with a proven performance bound of two times the optimal. When the number of available workstations is a fixed constant $k$ that is greater than 2, we show an $O(n\log n)$-time approximation algorithm which always performs better than $(3 + \frac{1}{2})$ times the optimal. Simulation results show that, on the average, both of our approximation algorithms perform much better than the worst case analysis and both generate schedulings whose makespans are very close to optimal.

Index Terms—Scheduling, parallel/distributed systems, approximation algorithms.

1 INTRODUCTION

This paper studies how to solve a problem that needs intensive computational time on an existing network of workstations. It is often the case that such a problem can be divided into a set of tasks with precedence constraints and communication overhead [11], [20], [26].

This paper uses a parallel computation model proposed in [19] which is based on Valiant’s asynchronous distributed memory architecture [32], [33]. It also takes into consideration the read/write contention of the QRQW model [15] and the latency/overhead time of the LogP model [9]. In addition to a fixed communication latency, a message sending cost is used which is proportional to the message size. This model is designed to represent an existing pool of networked workstations, each with its own memory and processor. In such an environment, interworkstation communications are performed using message passing and take significant time when nontrivial data movement is needed between tasks located on different nodes. We fix the communication latency to simplify the model and to reflect the situation that communication delay is predictable in a local area network with moderate traffic and also in a dedicated network.

The nontrivial data movement consideration is necessary to accommodate the fact that many applications, such as image processing, process large-sized input data which is stored in a node’s secondary storage and is needed by other nodes.

The growth of these networks and applications mandate more study into the efficient use of parallel computing power when solving such applications. More importantly, the model we use is general enough to be used for any algorithm which can be represented as a set of tasks which communicate with each other and whose execution and communication costs are known or can be estimated. An example where such an algorithm would be helpful is a network of computers using PVM parallel software [13]. Section 2.1 gives an example of using such a pool of workstations to perform parallel computation.

Previous parallel computation models for scheduling tasks have usually included a communication cost, representing the communication delay, associated with the sending of data between tasks which are located on different processors. Early work on this problem used graph theoretic techniques such as network flow and/or enumeration techniques [12], [25], [30]. Some later work concentrated on approximation algorithms [1], [18], [21], [27]. Research then evolved to more restricted models which allowed an infinite number of processors in the system. Polynomial algorithms were found for the case where the precedence constraints form a tree under certain constraints [11], [6], [8], [4], [22]. Another research avenue focuses on the practical side of aiding the generation of execution code on available dedicated parallel machines [17], [34], [35]. A good review of models and algorithms developed for this problem can be found in [3], [11], [5], [16], [20], [26]. Much of this work is very theoretical in nature, i.e., the models are too simplistic for practical

---

- T.-s. Hsu is with the Institute of Information Science, Academia Sinica, Nankang 11529, Taipei, Taiwan, ROC. E-mail: tshsu@iis.sinica.edu.tw.
- J.C. Lee is with the Computer Science Department, Brown University, Providence, RI 02912. E-mail: jcl@cs.brown.edu.
- D.R. Lopez is with the Division of Science and Mathematics, University of Minnesota-Morris, Morris, MN 56267. E-mail: lopezdr@mrs.umn.edu.
- W.A. Royce is with Measurement Technology Laboratories, Minneapolis, MN 55437. E-mail: broyce@mtlcorp.com.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 107003.
application to real machines. More recently, Valiant’s BSP Model [32], [33] provided a general framework with which to study more practical algorithms in an asynchronous distributed memory parallel architecture. The LogP model [9] and the QRQW model [15] attempted to further bridge the theoretical and practical models.

The major differences between our model and previous models are as follows: First, the time used in sending and receiving a message of nontrivial size should be proportional to the size of the message instead of being treated as insignificant. This factor is especially important when performing applications with large input size, such as image processing, on a network of workstations, especially where the entire input is larger than the main memory of a single node. Traditional models handle such data with either specially built parallel machines or coarse grain computations [26]. As a result, only communication delay, and not message size, between two nodes is considered. Second, our model sequentializes the sending and receiving of data in a node for the obvious reason of nontrivial data movements between different nodes. Finally, we require in our scheduling that the starting task and the ending task should be allocated on the same node because we assume the normal parallel computation paradigm used in a network of workstations, i.e., a job is initiated at a node and uses the help of idle machines in the network to execute some of the tasks. In such a paradigm, the initial node needs to obtain the final output, whose aggregate size may be nontrivial and receiving time for this data must be built into the model. Previous results for this model [19] prove that even finding an optimal algorithm for the simple one-level tree graph is NP-hard.

This paper gives algorithms and bounds for a one-level send-receive task graph, which is a directed graph with one source, one sink, and with every other vertex having an incoming edge from the source and an outgoing edge to the sink. A one-level send-receive task graph is frequently encountered when a job can be partitioned into disjoint unions of parallel tasks. Once the job and its input data are loaded onto one processor (workstation), parallel tasks (with their input data) are sent out to free processors (workstations) in the network and then the results of executing these tasks are collected by the original processor (workstation). Therefore, one-level send-receive graphs are often encountered in applications where one processor distributes tasks to helper processors and then receives back the results and computes and/or outputs the final solution. Examples of this type of application include matrix multiplication [28] and imaging, signal, or pattern matching algorithms using hierarchical tree data structures (e.g., quad-trees) [10], [23], [31].

We look at the complexity of the send-receive precedence task graph scheduling problem and at approximation algorithms for solving it. Scheduling a send-receive precedence task graph is considered both for the case when the number of available processors is unlimited and for when the number of available processors is fixed. Using reductions from well-known instances [6], [19], it is not surprising to discover that the problem of finding an optimal scheduling with a minimum makespan for the above cases is NP-hard. Therefore, approximation algorithms are developed and analyzed. Our algorithmic results are summarized in Table 1.

The organization of the rest of the paper is as follows: Section 2 shows an example and the network model that we are using and the fundamental properties of send-receive task graphs. Section 3 gives NP-completeness results. Section 4 presents approximation algorithms for both the cases of having abundant processors and having a limited number of processors. Section 5 shows simulation results. Section 6 gives some concluding remarks.

2 PRELIMINARIES

2.1 An Example

The tremendous increase in the usage of Pentium-type workstations in industry and government necessitates the development of better and faster algorithms for using a network of such machines in solving large practical problems. An example of such a problem is data compression. When working with large amounts of data, such as stored images, it is often desirable to be able to process this data and/or to send it to others over a computer network. If this task can be accomplished using a subset of the original data from which a reasonable copy of the original can be extracted, the cost and time of transporting and/or analyzing the data can be reduced.

It makes sense to use a networked group of workstations to perform such a job in parallel. Let us look at one simple example: image compression. Algorithms exist [24], [29] which divide an image into tasks which must be processed separately and then the results combined for the final compressed image. Our parallel scheduling model can enhance such algorithms by distributing tasks efficiently among a set of networked workstations in order to complete the data compression in a more timely manner (see Fig. 1).
The tasks $t_1, t_2, t_3$ are distributed by a main processor ($P_0$) to helper processors who execute the tasks assigned and then return their data to the original processor, $P_0$, which completes the final compressed image.

We note that even this small problem has obvious processing bottlenecks in the sending of data to the helper processors and in the receiving of the processed data by the main processor. We also note that if the processors are at a distance from each other, even if over high speed communication lines, there will be some latency incurred before the first bit of information arrives at its destination. Also, a large amount of data will take longer to complete its arrival than a smaller amount. It is the addition of this set of parameters and issues into the scheduling model which makes our model a practical, more usable scheduling tool.

System latency ($L$) can be measured in advance and communication times ($c_{i,j}$) can be estimated according to the size of the data being sent. By processing all sending and all receiving of data sequentially, we can incorporate all extra time caused by the send/receive bottlenecks into our scheduling algorithm. We will return later to this example with actual costs displayed to illustrate our algorithms.

First, we give an explanation of the model used for such scheduling. The following sections explain how our scheduling algorithms work and prove performance bounds for both $n$ processors and a fixed number $k$ of available processors.

### 2.2 The Network Model

Let $J = \{t_0, t_1, \ldots, t_{n+1}\}$ be a set of tasks whose precedence constraints form a directed acyclic graph $PC(J)$. In a precedence graph for a set of tasks, the weight on a directed edge $(u \rightarrow v)$ which points from $u$ to $v$ represents the communication time needed for $u$ to send data to $v$ if $u$ and $v$ are located on different processors. The weight on each node represents its execution time. In this model, we...
consider the case when all processors in the system are identical. Thus, the execution time of a task is the same, independent of which processor the task is placed on.

We schedule \( J \) on uniform processors \( P_0, P_1, \ldots, P_n \) with a system I/O latency, \( L \). Note that \( n \leq r \). In this model, task \( t_i \) takes \( e_i \) time to finish its computation and, after its completion, transmits data to the processor on which task \( t_j \) is allocated if there is a precedence relation from \( t_i \) to \( t_j \). Task \( t_j \) cannot start executing unless it has received all data from \( t_i \). We assume that the communication time is zero between two tasks allocated to the same processor. If \( t_i \) and \( t_j \) are allocated on different processors, then the sending time for \( t_i \) is \( c_{i,j} \) and the receiving time for \( t_j \) is also \( c_{i,j} \). All data streams are transmitted in a pipelined fashion, i.e., after \( t_i \) starts sending, all data arrive at \( t_j \) in \( c_{i,j} + L \) units of time. If a task needs to send or receive two data elements at the same time, the two I/O operations must take place in sequence. An example of a timing diagram for executing tasks in this model is shown in Fig. 2.

A scheduling \( S \), for \( J \), is an assignment of tasks to processors. A legal realization for \( S \) is the assignment of starting times for all tasks allocated to each processor such that it satisfies the precedence constraints and the I/O latency requirement. The makespan of a processor \( P_i \) for a realization is the time at which the processor \( P_i \) finishes all tasks allocated to it. The makespan of a legal realization is the largest makespan among all processors. A legal realization with the smallest makespan is a best realization. The makespan of a scheduling \( S \) is the makespan of its best realization and is denoted as \( C(S) \). An optimal scheduling \( J \) is a scheduling with the smallest possible makespan. For convenience, we assume that \( t_0 \) is allocated to \( P_0 \).

### 2.3 Send-Receive Task Graphs

A directed acyclic graph \( G = (V, E) \) is a send-receive graph with source \( v_0 \) and sink \( v_{n+1} \) if \( V = \{v_0, v_1, \ldots, v_n, v_{n+1}\} \) and \( E = \{(v_0 \rightarrow v_i) \mid 1 \leq i \leq n\} \cup \{(v_i \rightarrow v_{n+1}) \mid 1 \leq i \leq n\} \). The vertices \( v_1, \ldots, v_n \) are intermediate vertices.

For the rest of this paper, let \( J = \{t_0, t_1, \ldots, t_n, t_{n+1}\} \) be a set of \( n+2 \) tasks whose \( PC(J) \) forms a send-receive graph with source \( t_0 \) and sink \( t_{n+1} \). Let \( e_i, 0 \leq i \leq n+1 \), be the execution time of task \( t_i \) and let \( c_{0,j}, 1 \leq j \leq n \), be the sending cost of \( (t_0 \rightarrow t_j) \) and \( (t_i \rightarrow t_{n+1}) \), respectively. We say that \( t_{n+1} \) is the total communication time for the task \( t_i \). An intermediate task \( t_i \) is light if \( e_i \leq c_{0,i} + c_{i,n+1} \). An intermediate task that is not light is heavy. An example is illustrated in Fig. 3. We also use the following notation throughout the paper:

- \( J' = \{t_i \mid 1 \leq i \leq n \text{ and } e_i \leq c_{0,i} + c_{i,n+1}\} \), i.e., the set of light tasks.
- \( J'' = J \setminus (J' \cup \{t_0, t_{n+1}\}) \), i.e., the set of heavy tasks.
- \( E' = \sum_{t_i \in J'} e_i \).
- \( C_0' = \sum_{t_i \in J'} c_{0,i} \).
- \( C_{n+1}' = \sum_{t_i \in J'} c_{i,n+1} \).

We want to find a scheduling of the above tasks with the constraint that \( t_0 \) and \( t_{n+1} \) must be allocated on the same processor.

### 2.4 Properties

We assume that we have a pool of identical processors with the notion that \( P_i \) denotes the \( i \)-th processor and \( L \) is the interprocessor communication latency. Without loss of generality, we assume that \( t_0 \) and \( t_{n+1} \) are both scheduled on \( P_0 \). Given an assignment of tasks to processors, we first state a lemma characterizing one of its best realizations.

**Lemma 2.1.** A best realization for a scheduling of tasks \( J \) whose \( PC(J) \) is a send-receive graph with source \( t_0 \) and sink \( t_{n+1} \) on an unlimited number of processors can be characterized as follows:

1. \( P_0 \) executes \( t_0 \) first.
2. \( P_0 \) then sends out the needed data to the intermediate tasks that are not allocated on \( P_0 \).
Fig. 4. A realization for a scheduling on the right for the send-receive task graph on the left. Assume that tasks $t_0$, $t_1$, and $t_3$ are allocated on $P_0$. Tasks $t_1$, $t_2$, and $t_4$ are allocated on processors other than $P_0$.

\begin{enumerate}
\item $P_0$ executes the intermediate tasks that are allocated to itself.
\item $P_0$ performs data collecting.
\item Finally, $P_0$ executes task $t_{n+1}$.
\end{enumerate}

**Proof.** Let $R$ be the realization defined in the lemma. We divide the activities on $P_0$ into the following five phases:

1. executing $t_0$
2. sending data,
3. executing intermediate tasks,
4. receiving data, and
5. executing $t_{n+1}$.

It is obvious that executing $t_0$ and sending data should be done first and executing $t_{n+1}$ should be done last. Assume that a modified realization $R'$ of $R$ by inserting the activity of executing an intermediate task into the fourth phase gives us a better makespan. In order for the makespan of $R'$ to be less than the makespan of $R$, there must be some idle time on $P_0$ during the fourth phase in $R$ and there is no idle time on $R'$. But, idle time during the fourth phase in $R$ means that $P_0$ is waiting to receive data. In $R'$, $P_0$ would have to wait the same amount of time, then receive its data, and then execute its intermediate tasks, which means the makespan of $R'$ would be increased by the time it takes to execute the intermediate tasks making it at least that much greater than the makespan of $R$, a contradiction. $\square$

We remark that the order of sending out data for tasks not allocated on $P_0$ is not specified in Lemma 2.1. It may have to check all possible orders to find one with the best realization.

An example of a realization for a scheduling on a set of tasks $J$ whose PC($J$) is a send-receive graph is shown in Fig. 4. In a directed graph, an edge $(u \to v_i)$ points from the vertex $u$ to the vertex $v_i$. The next lemma shows that a task $t_i$ whose difference (i.e., the execution time minus its total communication time $c_i = (c_{0,i} + c_{i,n+1})$) is nonpositive can be allocated to a processor with its parent to have an optimal scheduling.

**Lemma 2.2.** Let $S$ be a scheduling for a set of tasks whose PC($J$) is a send-receive graph. By reallocating all tasks with nonpositive differences to $P_0$, the resulting scheduling has a makespan equal to or better than that of $S$.

**Proof.** Let $t_w$ be a task with a nonpositive difference $d_w$, which is allocated to a processor other than $P_0$ in an optimal scheduling. By reallocating $t_w$ to $P_0$, the makespan for $P_0$ is increased by $d_w$. Since $d_w \leq 0$, the makespan on $P_0$ does not increase. On the other hand, the makespan for $P_i$, $i > 0$, is also not increased. Thus, the new scheduling is also optimal. We can continue this process until all tasks with nonpositive differences satisfy the property specified in the lemma. $\square$

Let OPT($J$) be the makespan of an optimal scheduling for $J$ on an unlimited number of identical processors. We first give a lemma to bound from below the value of OPT($J$).

**Lemma 2.3.**

\begin{enumerate}
\item An optimal scheduling for $J$ is to schedule all tasks on $P_0$ if and only if, for all tasks $t_i$, $\sum_{j=1}^{m} e_j \leq c_{i,n+1} + 2L + c_{0,i}$.
\item If scheduling all tasks on $P_0$ is not an optimal scheduling, then OPT($J$) $> e_0 + e_{n+1} + 2L$.
\item OPT($J$) $\geq e_0 + e^{'} + c_{0,n+1} + e_{n+1}$.
\item OPT($J$) $\geq e_i, 0 \leq i \leq n + 1$.
\end{enumerate}

**Proof.**

The “only if” part of the proof is trivial since putting a task on another processor in this case only increases the makespan. We now prove the “if” part.

Let $S$ be an optimal scheduling with all tasks allocated to $P_0$. Thus, the makespan of $S$ is $e_0 + \sum_{i=1}^{n} e_i$. Assume that there is a scheduling $S'$ with at least one task $t_w$ with $1 \leq w \leq n$ and $e_w > c_{0,w} + c_{w,n+1}$ such that

$$\sum_{i=1}^{n} e_i > c_{0,w} + 2L + e_w + c_{w,n+1}.$$  

Because $e_w > c_{0,w} + c_{w,n+1}$ and

$$e_0 + c_{0,w} + 2L + e_w + c_{w,n+1} < e_0 + \sum_{i=1}^{n} e_i,$$

$$e_0 + \sum_{i=1}^{n} e_i - e_w + c_{0,w} + c_{w,n+1} < e_0 + \sum_{i=1}^{n} e_i.$$  

This implies that $M(S') < M(S)$, which is a contradiction since $S$ is an optimal scheduling. The conclusion follows.

2. If scheduling all tasks on $P_0$ is not an optimal scheduling, then we must at least schedule one task $t_s$, $s > 0$, on processor $P_s$. The makespan of $P_s$ is at least $e_0 + c_{0,s} + 2L + e_s + c_{s,n+1}$ and the
3 Bounds for Finding an Optimal Scheduling

Given a set of tasks whose precedence constraints form a send-receive graph, the send-receive task allocation problem is the problem of finding a scheduling whose makespan is the minimum. Given a constant \( q \), the decision version of the send-receive task allocation problem is the problem of finding a scheduling whose best realization is less than \( q \). In this section, we show, by an easy reduction (also easy to prove by reducing from Crétienne et al.’s [5] proof of the send-receive graph using a different model), that the decision version of the send-receive task allocation problem on an unlimited number of processors is NP-complete. We then prove a more difficult assertion. When the number of available processors is a fixed constant greater than 2, the decision version of the send-receive task allocation problem remains NP-complete even when the execution times of tasks are equal.

Lemma 3.1. The decision version of the send-receive task allocation problem on an unlimited number of identical processors is NP-complete.

Proof. Let \( J \) be a set of tasks whose PC(J) is a send-receive graph. Assume that \( e_{i+1} = 0 \) and \( e_{n+1} = 0 \) for tasks in \( J \). Hence, the PC(J) can be expressed as a directed out-tree. From [19], we know that scheduling a one-level directed out-tree is NP-complete. Hence, this lemma holds. \( \square \)

Given a finite set \( F \) with \( |F| = m \), a positive integer weight \( w(f) \) for each element \( f \in F \) and positive integers \( b_i \) and \( n_i \), the decision version of the bin packing problem asks whether \( F \) can be partitioned into \( n_i \) subsets \( F_1, \ldots, F_{n_i} \) such that \( \sum_{f \in F_i} w(f) \leq b_i \) for all \( i \). We say that \( b_i \) is the bin size and \( n_i \) is the number of bins. The bin packing problem is NP-complete [14].

We will show that the decision version of the bin packing problem can be reduced, in polynomial time, to the following special instance of the decision version of the send-receive task allocation problem: Let \( J^* \) be a set of \( k \cdot (m+1) + 1 \) tasks \( t_{0}, t_{1}, \ldots, t_{k \cdot (m+1)} \) whose PC\((J^*)\) is a send-receive graph with the source task \( t_{0} \) and the sink task \( t_{k \cdot (m+1)} \). Let \( e_i \) be the execution time of task \( t_i \) and \( e_i = e \), \( 1 \leq i \leq k \cdot (m+1) \). Let \( c_{0,i} \) and \( c_{i,k \cdot (m+1)} \) be the communication times from task \( t_{0} \) to \( t_i \) and from \( t_i \) to \( t_{k \cdot (m+1)} \), respectively. Let \( c_{k \cdot (m+1)} = 0 \) and let \( c_{k \cdot (m+1)} = 0 \). The values of \( e \), \( c_{0,i} \), and the latency \( L \) satisfy the following constraints:

1. \( c_{0,i} = 0 \), \( 1 \leq i \leq (k-2) \cdot (m+1) + 1 \),
2. \( L + \sum_{i=1}^{(k-1) \cdot (m+1)} c_{0,i} < L \),
3. \( c_{0,i} > e \cdot (k-1) \cdot (m+1) + 1 \leq i \leq k \cdot (m+1) \),
4. \( L > \sum_{i=1}^{(k-1) \cdot (m+1)} c_{0,i} \),
5. \( c_{0,i} \leq c_{0,i+1}, 1 \leq i < k \cdot (m+1) \).

Lemma 3.2. There is a scheduling for \( J^* \) on \( k \) processors whose makespan is less than \( e_0 + L + (m+2) \cdot e \) and satisfies the following properties:

- \( t_{0}, t_{(k-1) \cdot (m+1)+1}, \ldots, t_{k \cdot (m+1)} \) are scheduled on \( P_0 \).
- Exactly \( m+1 \) tasks in \( \{t_1, \ldots, t_{(k-1) \cdot m+2}\} \) are scheduled on processor \( P_i \), \( 1 \leq i < k \).
- The makespan of \( P_i \), \( 1 \leq i \leq k \), is greater than the makespan of \( P_0 \).

Proof. The makespan of any scheduling which satisfies the above constraints is less than \( e_0 + L + (m+2) \cdot e \). Hence, such a scheduling exists.

By Lemma 2.2, constraint 3 implies that tasks \( t_{(k-1) \cdot (m+1)+1}, \ldots, t_{k \cdot (m+1)} \) are scheduled on \( P_i \) and \( P_j \) where \( t_i \) is scheduled. Let \( S \) be any scheduling which satisfies the properties above. The makespan of \( P_0 \) for \( S \) is equal to \( e_0 + \sum_{i=1}^{(k-1) \cdot (m+1)} c_{0,i} + (m+1) \cdot e \). The makespan of \( P_i \), \( 1 \leq i < k \), for \( S \) is less than or equal to

\[
e_0 + L + \sum_{i=1}^{(k-1) \cdot (m+1)} c_{0,i} + (m+1) \cdot e
\]

and is greater than or equal to \( e_0 + L + (m+1) \cdot e \). By constraint 4,

\[
e_0 + L + (m+1) \cdot e > e_0 + \sum_{i=1}^{(k-1) \cdot (m+1)} c_{0,i} + (m+1) \cdot e.
\]

Assume that \( S^* \) is an optimal scheduling not satisfying the statement of the lemma. If, in \( S^* \), an extra task \( t_{0,w} \), \( 1 \leq w \leq (k-1) \cdot (m+1), \) is scheduled on \( P_i \), then the makespan of \( P_0 \) for \( S^* \) is larger than the makespan of \( P_0 \) for \( S \) because \( e > c_{0,w} \). The makespan of \( P_0 \) for \( S^* \), which is at least \( e_0 + (m+2) \cdot e \), is also larger than the makespan of \( P_i, 1 \leq i < k \), because constraint 2 implies

\[
e_0 + (m+2) \cdot e > e_0 + L + \sum_{i=1}^{(k-1) \cdot (m+1)} c_{0,i} + (m+1) \cdot e.
\]

Thus, no task other than \( t_{0}, t_{(k-1) \cdot (m+1)+1}, \ldots, t_{k \cdot (m+1)} \) is scheduled on \( P_i \).

If, in \( S^* \), one processor in \( P_i \), \( 1 \leq i < k \), is scheduled with more than \( m+1 \) tasks, then, without loss of generality, assume this processor is \( P_i \). The makespan of \( P_i \) for \( S^* \) is larger than the makespan of \( P_0 \), and for all \( i, S \). Thus, the lemma holds. \( \square \)

Using the above lemmas, we can prove the NP-hardness of our scheduling problem even for equal task execution times. We note that other models and/or graph structures have polynomial algorithms for the equal task case. Under a more theoretical model used in [6], for example, it is found...
that a precedence graph with equal task execution times which forms a tree can be scheduled in polynomial time. In [19], which uses the same model as this paper, equal tasks can be scheduled polynomially for a one-level tree. Other polynomial algorithms for scheduling similar structures can be found in [7], [5], [35]. Theorem 3.3 shows, however, that the send-receive graph with the addition of the parameters in our model is more difficult to schedule.

**Theorem 3.3.** The decision version of the send-receive task allocation problem on a fixed number \( k \) of processors, \( k > 2 \), is NP-hard even when intermediate tasks have equal execution time.

**Proof.** We will show that the bin packing problem is a special case of the decision version of the send-receive task allocation problem. Given any instance of the bin packing problem on a set of \( m \) elements \( \mathcal{F} = \{f_1, f_2, \ldots, f_m\} \), let the weight of \( f_i \) be \( w(f_i) \), \( 1 \leq i \leq m \), and the number of bins be \( m_0 = k - 1 \). Since \( k > 2 \), \( m_0 > 1 \). We may assume that \( w(f_i) \leq w(f_{i+1}) \), \( 1 \leq i < m \). We construct the instance of the scheduling problem described in Lemma 3.2 as follows: The communication time for \( t'_{(k-2)-(m+1)+1} \) is \( w(f_i) \).

We select the values of \( e \), \( L \), and \( c_{0,i} \), \( 1 \leq i \leq (k-2)-(m+1)+1 \) and

\[
(k-1)-(m+1)+1 \leq i \leq k-(m+1),
\]

as described before the statement of Lemma 3.2. If a scheduling \( S \) satisfies the statement of Lemma 3.2, some task with zero communication time is scheduled on \( P_i \), \( 1 \leq i < k \), because exactly \( m + 1 \) tasks are allocated on each processor and there are \( (k-2)-(m+1)+1 \) tasks with zero communication time.

In a best realization for \( S \), (implied by Lemma 2.1), the sending of data from \( P_i \) for tasks not allocated on \( P_i \) with zero communication time should be executed first. Assume that \( F_1, 1 \leq i < k \), is the set of task indexes that are allocated on \( P_i \). Thus, the makespan of \( P_i \) is \( c_0 + L + \sum_{j \in F_i} c_{0,j} + (m+1) e \). Hence, a makespan of \( S \) less than \( c_0 + L + b_n + (m+1) e \) implies a solution to the bin packing problem with the bin size \( b_n \) and vice versa. This proves our theorem. \( \square \)

**4 SCHEDULING ALGORITHMS**

In this section, we give approximation algorithms for the send-receive task allocation problem. The approximation ratio of our algorithm is 2 for finding a scheduling on an unlimited number of processors and is \( 3 + \frac{k-2}{k} \) for the algorithm on a fixed number of \( k \) processors.

**4.1 Using an Unlimited Number of Processors**

We show an approximation algorithm in Fig. 5 for scheduling send-receive task graphs on a network with an unlimited number of processors and then prove the makespan of the resulting scheduling is less than two times the optimal makespan. Task execution times can be of any value.

**Theorem 4.1.**

1. Algorithm SR runs in \( O(n) \) time.
2. The makespan of the scheduling produced by Algorithm SR is at most twice the optimal makespan.
Algorithm SR has at least an upper bound which is less than or equal to $5 \times$ the optimal value. Hence, the computation must begin and end on the same processor. Yang and Gerasoulis [34, 35] have studied algorithms for scheduling DAGs on unlimited processors with special attention given to large fork/join graphs. Since the send/receive graph is one large fork followed by one large join, it is possible to use their algorithm if the first and last task happen to be placed in the same cluster.

For instance, in our example above, the DSC algorithm would then place tasks $t_0$ and $t_3$ together in a cluster (i.e., on the same processor) in order to reduce the parallel time along the critical path in the graph. Algorithm DSC would then place tasks $t_1$ and $t_2$ in unit clusters since adding either of them to the given cluster would increase the parallel time of the graph. Now, executing the join part of Algorithm DSC, $t_4$ would be added to the cluster with $t_0$ and $t_3$ (reducing parallel time to a minimum), but no other tasks would be added since doing so would increase the parallel time of the algorithm. The makespan of the DSC algorithm would then be 18 using our model (see Fig. 7), which is the same as placing all the tasks on $P_0$. For the model used by the DSC algorithm, however, the algorithm performs much better for the above graph since their model does not deal with sequential send/receive, latency, or master/slave configurations.

**Fig. 6.** An example illustrating a task graph with five tasks.

**Proof.** Part 1 is trivial. We now prove part 2. If the condition in Step 1 succeeds, then we find an optimal scheduling. Hence, the lemma holds. Assume that the condition in Step 1 fails. Let $M^*$ be the optimal makespan and let $M$ be the makespan of the scheduling returned by Algorithm SR. Since we reached Step 2, we know that either $M = M^*$ or $M^* \geq e_0 + c_0 + 2L + e_1 + c_{n+1} + e_{n+1}$ for all $1 \leq i \leq n$. Then, the makespan $M$ on $P_0$, which is based on computations done on $P_i$, is less than or equal to

$$e_0 + C_0' + E' + 2L + e_1 + C_{n+1}' + e_{n+1},$$

which is less than $2 \times M^*$ by the above and by Lemma 2.3, part 3.

We now prove, by showing an example, that Algorithm SR has at least an upper bound which approaches 1.5 times the optimal value.

**Lemma 4.2.** The asymptotic worst case makespan for Algorithm SR can approach 1.5 times the optimal value.

**Proof.** Consider the following worst case example: $J = \{t_0, t_1, \ldots, t_{n+1}\}$ is the set of tasks to be allocated to an unlimited number of processors, where $n = 2m$. Let $e_0 = e_{n+1} = 0$ and let $e$ be a positive number that is less than 1. Let $c_i = 1 - e$, $c_{n+1} = 0$, and $e_i = 1 + e$, for all $1 \leq i \leq n$. Let $L = m \times (1 + e)/2 - e$. Hence, the checking of the condition in Step 1 of Algorithm SR fails. The optimal scheduling is for $P_0$ to execute $t_0, \ldots, t_m, t_{n+1}$ and for $P_{m+i}$, $1 \leq i \leq m$, to execute $t_{m+i}$, since placing any task back on $P_0$ increases the makespan of $P_0$ by $2e$ and taking any task from $P_0$ to place on another processor increases the makespan of $P_0$ by $1 - e$. Hence, the optimal makespan is $m \times (1 - e) + m \times (1 + e) = 2m$. The makespan of the scheduling returned by Algorithm SR is to not place all intermediate tasks on $P_0$ with the makespan $(2m - 1)(1 - e) + 2L + 1 + e = 3m - m \times e$, which has an asymptotic limit of 1.5.

**4.1.1 An Example Using an Unlimited Number of Processors**

Fig. 6 gives a simple example of a send-receive graph. We will use this example to explain how Algorithm SR works. Assume the latency, $L$, is 2 units of time. The communication time $c_{ij}$ and the estimated execution time $e_i$ is given for each task. Task $t_0$ has no communication since it prepares the data for distribution. Likewise, task $t_0$ is used by the main processor $P_0$ to process the final result and has no communication in our algorithm. All other communication between tasks $t_i$ and $t_j$ is represented on the edges of the graph between nodes (tasks) $t_i$ and $t_j$. We note that, practically speaking, the communication to and from the intermediate tasks can often be very different. For instance, a large amount of data could be sent to a task to be processed with the result being very small and vice versa.

Using Algorithm SR, we reach an optimal schedule as follows:

**Step 1:** Putting all tasks on processor $P_0$ gives a makespan of 18 calculated by summing all execution costs $e_i$ of tasks. There is no intertask communication if the tasks are all on the same processor.

**Step 2:** By removing each task independently from $P_0$, we find that the best makespan is 16 since when removing $t_1$, the finish time on $P_0$ is calculated as

$$\max\{e_0 + c_0 + e_2 + e_3 + c_{14} + e_4, e_0 + L + c_{14} + e_1 + e_3 + L + c_{14} + e_4\} = \max\{16, 16\} = 16.$$

**Step 3:** Since $t_1$ is a heavy task, i.e., $e_1 > c_{14}$, we schedule $t_1$ on a separate processor and again reach a makespan of 16.

**Step 4:** Our best makespan is 16, found in both Steps 2 and 3 above.

Note that this simple linear algorithm finds an optimal solution for one, two, or three tasks. It is difficult to compare Algorithm SR with other algorithms in the literature since they are written for a different model which does not contain as many parameters or the constraint that the computation must begin and end on the same processor.
Using Algorithm SR to program the main processor for scheduling the tasks gives us a total time of 16 (see Fig. 8), a considerable time savings.

4.2 Using a Fixed Number of Processors

This section gives an approximation algorithm to find a scheduling for a fixed number $k$ of processors. The makespan of the scheduling generated by our algorithm is within $3 + \frac{k-2}{k}$ times the optimal makespan.

The following lemma is crucial in developing our approximation algorithm. It can be applied to one-level precedence tree graphs, as well as to send-receive graphs, for partitioning a set of tasks to be scheduled on a fixed number of processors. Given a set $A$ of $n$ items with a binary flag for each element, a biased $k$-partition is a disjoint partition of $A$ into $k$ sets such that all items with the flag value 0 are in one set. The following lemma shows that the ratios of elements placed in each bin compared to an optimal placement is bounded.

**Lemma 4.3.** Let $A = \{a_1, \ldots, a_n\}$ be a set of $n$ items with a weight $w_i$, and a binary flag $f_i$, for each item $a_i$. Let $w_i \geq w_{i+1}$, $1 \leq i < n$, and let $W = \sum_{i=1}^{n} w_i$. Let $W^*$ be the sum of weights for elements with the flag value 1. Let $k$ be a given integer that is at most $n$ and let $U = \max\{w_i, \frac{W}{k}\}$. We can find a biased $k$-partition $B_1, \ldots, B_k$ for $A$ such that:

1. $B_1$ contains all elements with the flag value 0 and $(\sum_{a_i \in B_1} w_i) - (W - W^*)/U \leq 3 - 2/k$ and
2. For all $2 \leq j \leq k$, $\sum_{a_i \in B_j} w_i/U \leq 2 - 2/k$.

**Proof.** Our algorithm uses the so-called best fit descending order heuristic and runs as shown in Fig. 9. Let $M$ be the largest sum of weights in a subset produced by Algorithm PAR and $B_i$ is a subset with the sum of weights $M$. Note that $\frac{W^*}{k} \leq U$ and $M \geq w_1$. We have the following three cases:

**Case 1:** $B_i$ contains one item. Hence, $M/U \leq 1$.

**Case 2:** $B_i$ contains more than one item and, for all $2 \leq i \leq k$, $B_i$ contains at most one item. Thus, $t = 1$ and $M \leq w_1 + U$. Since $U \geq w_1$, $M/U \leq 2$.

**Case 3:** $B_i$ contains more than one item and there exists some $i, 2 \leq i \leq k$, such that $B_i$ contains at least two items. Note that, during Step 5, Algorithm PAR never places two items in some $B_i$, $2 \leq i \leq k$, before having placed at least one item in each $B_j, 1 \leq j \leq k$. We let $W^*_{i,j}$, $j \neq 1$, be the sum of the weights in subset $B_i$ right after Step 5 of Algorithm PAR has placed $a_1, \ldots, a_i$ and before it places $a_{i+1}$. The value $W^*_{i,1}$ is the sum of weights in subset $B_1$ minus $\frac{W}{k}$ right after Step 5 of Algorithm PAR has placed $a_1, \ldots, a_i$ and before it places $a_{i+1}$. We are interested only in those allocations which increase $M$. In a worst case scenario, this means placing the largest item possible in the minimum-weight subset and producing a new $M$. Assume that Algorithm PAR places item $a_i$ into subset $B_j$ and $M = \max W^*_{i,j}$. Let $x$ be the smallest integer such that 1) $W^*_{x-1,j} > 0$ for all $j$, 2) placing $a_x$ creates the first subset with more than one item with flag value 1, 3) $W^*_{x,x} = M$. By our assumption, such an $x$ exists and $x > k$.

Before proving our main result, we need the following claim:

**Claim 4.4.** $M - W^*_{x,i} \leq w_{i+1}$ for all $z \geq x$ and for all $i$.

**Proof.** We will prove this claim by induction on the value of $z$. From the way Algorithm PAR works (on the worst case assumed above), we must place an item with the weight $w_{i}$, changing the subset $B_{i}$, from one with the smallest sum of weights to one with the largest sum of weights. Thus, $W^*_{x,x} - W^*_{x,y} \leq w_x \leq w_{i+1}$ if $y \neq r_x$. Hence, our claim is true for $z = x$. Assume that our claim is true for $z = y$ and $y$ is some integer at least $x$.

Then, $W^*_{y+1,x} = \min_{i=1}^{y} W^*_{y,i}$. If $W^*_{y+1,x} < M_{y+1}$, then the claim is obviously true from the induction hypothesis. Assume the case when $W^*_{y+1,x} = M_{y+1}$. Since
Algorithm PAR /* Partition $n$ items into $k$ subsets, $B_1, B_2, \ldots, B_k$. */

1. Assume, without loss of generality, that $w_i \geq w_{i+1}$, $1 \leq i < n$;
2. $B_1 = B_2 = \cdots = B_k = \emptyset$;
3. $r_1 = -W^*/k$; $r_2 = \cdots = r_k = 0$;
4. for $i = 1$ to $n$
   if $f_i = 0$
      place $a_i$ into $B_1$;
      $r_1 = r_1 + w_i$;
   endif;
5. for $i = 1$ to $n$
   if $f_i = 1$
      Place $a_i$ into $B_y$ such that $r_y = \min_{j=1}^k r_j$;
      If there are several $y$’s satisfying the above, pick the smallest one;
      $r_y = r_y + w_i$;
   endif;
6. end.

Fig. 9. Algorithm PAR.

$W^*_{y+1,x+1} - W^*_{y,x+1} \leq w_{y+1}$ and $W^*_{y,x+1} = \min_{i=1}^k W^*_{y,z}$,
$W^*_{y+1,x+1} - W^*_{y+1,x} \leq w_{y+1} - w_{k+1}$ for all $i \neq y+1$. Hence, the difference between $M_k$ and $W^*_{z,i}$, for all $z \geq x$ and for all $i$, is less than or equal to $w_{k+1}$, completing the induction. □

There are two cases.

Case 1. The sum of weights of elements in $B_1$ is less than $M$. Then, $W^* \geq M + (k-1)(M-w_{k+1}) + U$ because of Claim 4.4. This implies $M \leq \frac{(k-1)U}{k-1} + \frac{k-1}{k} w_{k+1}$. Note that $W^* \geq k w_{k+1}$ and $U \geq \frac{W}{k}$. Thus,
\[
\frac{M}{U} \leq \frac{k-1}{k} + \frac{w_{k+1}}{W^*} < \frac{k-1}{k} + \frac{(k-1)w_{k+1}}{k w_{k+1}} = 2 - \frac{2}{k}.
\]

Case 2. The sum of weights of elements in $B_1$ is $M$. Then, $M \geq \frac{W}{k}$. Let $M' = M - \frac{W}{k}$. Then, $W^* \geq M' + (k-1)(M-w_{k+1}) + \frac{W}{k}$. Using a similar approach, we can prove that $\frac{M'}{U} \leq 2 - \frac{2}{k}$. Thus, $\frac{M'}{U} \leq 3 - \frac{2}{k}$.

Corollary 4.5. If we replace $-W^*/k$ with $-\max\{W^*/k; 2L\}$ in Step 3 of Algorithm PAR, then Lemma 4.3 still holds.

Proof. The proof of Lemma 4.3 is valid for any value of $r_1$ that is at most $-W^*/k$. □

In running Algorithm PAR, we can use a priority queue to retrieve a subset with the minimum sum of weights and maintain the priority queue in $O(\log n)$ time [2]. Thus, the running time of Algorithm PAR, including sorting items according to their weights, is $O(n \log n)$. We now use Algorithm PAR to find a scheduling for a send-receive task graph using $k$ processors.

Theorem 4.6.

1. Algorithm KSR (shown in Fig. 10) runs in $O(n \log n)$ time.
2. The makespan of the scheduling produced by Algorithm KSR is less than $3 + \frac{\log n}{k}$ times the optimal makespan.

Proof. If Step 1 of Algorithm KSR succeeds, then we have found an optimal scheduling. Hence, we assume Step 1 fails. Recall that $J''_u$ is the set of heavy tasks in $t_1, \ldots, t_n$. Note that $W = \sum_{i=1}^n w_i$ and $W^*$ are the sum of execution time for tasks in $J''_u$. $U = \max\{\max_{i=1}^n w_i, \frac{W}{k}\}$ and $\text{OPT}(J) \geq U$. Since Step 1 fails, $\text{OPT}(J) \geq 2L$. Let $Q_i$, $0 \leq i < k$, be the sum of execution times for the tasks in $J''_u$ that are allocated on $P_i$. By part 1 in Lemma 4.3 and the facts that $\text{OPT}(J) \geq U$ and $Q_0 \geq E' + e_0$, $Q_0 \leq (3 - \frac{2}{k})\text{OPT}(J)$. By part 2 in Lemma 4.3 and the fact that $\text{OPT}(J) \geq U$, $Q_i \leq (1 + \frac{k-2}{k})\text{OPT}(J)$, for $1 < i < k$. Since we must schedule at least one task in $J''_u$ on $P_i$, $i > 0$, we also know that:
\[
\text{OPT}(J) \geq e_0 + \epsilon_{n+1} + E' + C''_0 + C''_{n+1} + 2L.
\]

The makespan of $P_0$, whose makespan is always greater than the makespan of other processors, is at most $e_0 + C''_0 + \max\{Q_0 + E', Q_i + 2L\} + C''_{n+1} + e_{n+1}$, which
Algorithm KSR /* Scheduling send-receive task graphs using $k$ PE’s. */
1. /* Check whether scheduling all tasks on $P_0$ is optimal. */
   \[\sum_{i=1}^{n} e_i \leq c_{0,i} + 2L + e_i + c_{i,n+1} + e_{n+1}\]
   for all tasks $t_i$ with $e_i > c_{0,i} + c_{i,n+1}$, 
   then Allocate all tasks on $P_0$;
2. else /* Allocating all tasks on $P_0$ is not optimal. */
   Use Algorithm PAR (with modifications from Corollary 4.5 to partition tasks $t_0, \ldots, t_n$ into a biased $k$-partition by using $e_i$ as their weights, the flag value 0 for the light tasks and $t_0$, and the flag value 1 for the remaining tasks;
   Allocate tasks in the subset containing tasks with flag values 0 to $P_0$;
   Allocate tasks in the $(i + 1)$th subset to $P_i$, $0 \leq i < k$;
3. return the resulting scheduling;
4. end.

We place $t_4$ in $B_2$, giving us $r_2 = 8$. We place $t_2$ in $B_1$, giving us $r_3 = 7$. We now place $t_5$ in $B_1$ and $t_5$ in $B_1$ since $B_1$ has the lowest $r$-value both times. Now, we can complete the allocation to processors by placing all tasks in $B_1$ on $P_0$, tasks in $B_2$ on $P_1$, and tasks in $B_3$ on $P_2$.

Step 3: We return the schedule with a makespan of 25, as shown in Fig. 12.

In order to check Algorithm KSR in a more real-life situation, we used the data compression algorithm found in [24] on a network of Pentium workstations. We chose a large image and did many iterations of the algorithm in order to provide a test of both execution times and the communication times needed to send the image to the processors and receive the compressed result. Using the parallel language PVM, we compared the time it took to compress an image in parallel using PVM’s built in round robin assignment of tasks to Algorithm KSR’s time for assignment of tasks to two processors. We randomly divided the image into 4,096 squares and the squares into 16 tasks which were then allocated to processors which processed the image. After 77 runs, on average, the makespan using PVM’s round robin assignment took 351.7 seconds as compared to Algorithm KSR’s makespan of 238.4 seconds.

As the next section’s simulation results show, Algorithm KSR has performance ratios which are very close to optimal.

5 Simulation Results

The simulation of algorithm KSR was performed by randomly producing 1,000 task sets with parameters selected from specified ranges. The results are shown in Figs. 13, 14, and 15.

These task sets were run through a brute force algorithm which determined the makespan of the optimal schedule. Then, the same task sets were run using Algorithm KSR. The recorded performance value for each set was the
average ratio of KSR’s makespan to the optimal makespan over all 1,000 sets. In the graphs, we show that the performance curves follow the same pattern when we vary important parameters such as the number of tasks, size of latency, and size of communication time.

As shown in the graphs, latency ranges varied from 0.01 to 0.05, 0.01 to 0.5, and 0.01 to 1.0. Execution times varied between 0.01 and 1.0 at all times. Low communication times varied between 0.01 and 0.5 for both $c_{ij}$ and $c_{ij+1}$. High communication times varied between 0.01 and 0.5 for $c_{ij}$ and between 0.01 and 1.0 for $c_{ij+1}$. Therefore, a parameter could be as much as 100 times the size of another. We also varied one parameter against another to try to represent as many different cases as possible. Note that it is the relative size of one parameter to another, not the actual sizes used, that allow for a large variety of parameter sizes.

The last data point of each performance curve does not represent a full 1,000 sets because of the computation time taken by the brute force algorithm. This is also the reason that some of the curves do not have points for the last set of tasks—it just took too long to find an optimal schedule. Performance still indicates the same curve, however. All the performance graphs show that our algorithm performs very well in simulation even when the values for latency and communication delay vary. When the simulated network consists of four processors, the average makespan is no more than 2 percent above the optimal value. When the simulated network consists of seven processors, the average makespan is no more than 6 percent above the optimal value.

The algorithm performance graphs each start at a performance ratio of 1, using either two or three tasks. These schedules consist of only the sending and receiving tasks plus possibly one more task. Tasks $t_0$ and $t_{n+1}$ are always placed on processor zero and Algorithm KSR also forces the third task to be placed on $P_0$, giving an optimal schedule. As the number of tasks increases, the graphs peak, then drop toward optimal again. We conjecture that this behavior might be because Algorithm KSR schedules some tasks on the wrong processors and the cost of a misscheduled task is magnified given only a few total tasks, but later offset by the correctly scheduled tasks as the number of tasks increases.

Even when omitting the schedules using two or three tasks which are always allocated optimally by Algorithm KSR, the optimal makespan was found in 84.9 percent of all the other KSR schedules generated. This percentage increases as the number of tasks increases. For instance, in Fig. 13, the percentage of KSR solutions which were optimal using 16 tasks is 99 percent, 96.5 percent, and
97.3 percent for $L$ varying from 0.01 to 0.05, 0.5, and 1.0, respectively.

Increases in latency, as the graphs show, tend to make the algorithm’s performance worse since latency magnifies the effect of mischeduling a task. Increased latency also has the effect of causing the peaks of the performance curves to occur with a higher number of tasks. For a possible explanation, consider the situation where task numbers are low. With a high latency, Algorithm KSR’s makespan will be higher than with a low latency, but the optimal makespan will also be higher. Thus, the performance ratio will actually be lower than with a low latency since this increased overhead in the ratio will outweigh KSR’s approximation error. As the number of tasks increases, the approximation error increases beyond the overhead incurred by a high latency and the graph rises and peaks. After the peak, the cost of a misscheduled task is offset by the correctly scheduled tasks, so the graph goes down, albeit slower than with a low-latency graph, again because of the overhead. Similarity between the curve shapes leads us to conclude that, for a large number of tasks, changes in latency have little effect on the algorithm’s performance.

6 CONCLUDING REMARKS
We have studied the task-graph scheduling problem targeting the popular parallel computation paradigm of a pool of workstations in a network. We assume that the network was not built especially for parallel scientific computation and that its interprocessor communication is achieved by using off-the-shelf packages, e.g., PVM. Full usage of idle machine cycles in such an environment has became increasingly important as hardware technology used in CPU and communication linkage designs advances.
Our algorithms take advantage of such idle cycles by scheduling all processors as full as possible to reduce makespans.

It is often the case that a job initiated from a workstation can be split into tasks which are sent to idle workstations on the network. Results are then sent back to the starting machine and merged to form the final solution. Our model differs from previous models by taking into consideration the nontrivial data movements between tasks. Hence, in this model, the communication time is proportional to the size of the transferred data. We also require multiple messages arriving and/or leaving a workstation to take place in sequence. We charge a fixed network latency in the model. We require that the first and last tasks be located on the main processor. Thus, our model is more realistic and accurate than previous models, especially when the job needs intensive computation time and large data storage.

The complexity of finding an optimal schedule for our model was studied. We show by reducing from well-known instances that finding such a schedule is NP-hard when the number of workstations is unlimited. It remains an NP-hard problem even when the number of workstations is a fixed constant greater than 2 and when the computation time for the intermediate tasks are equal.

We have shown approximation algorithms for finding an efficient scheduling when the number of available workstations is abundant and when the number of available workstations is a fixed constant, regardless of the input size. For the former, we give a linear-time approximation algorithm which returns a scheduling whose makespan is within twice the optimal makespan. When the number of available workstations is a fixed constant $k$ that is greater than 2, we show an $O(n \log n)$ approximation algorithm which returns a scheduling whose makespan is within $(3 + \frac{1}{k})$ time of the optimal makespan.

Our simulation results show that the worst cases in our algorithms rarely happen. Furthermore, on the average, all the approximation algorithms perform much better than our worst case analysis and generate schedulings whose makespans are within 1.1 times the optimal makespan when the number of tasks in the system is small enough for the finding of an optimal makespan in reasonable time. We also compared our scheduling algorithm to PVM’s round robin task scheduling algorithm with a real problem, data compression.

By doing the above, we have achieved a balance of theory and practice in carrying out our research. We first developed approximation algorithms for our problem to establish worst-case lower bounds. Our algorithms are based on heuristics, yet we show that their performance ratios can be mathematically analyzed. By actually coding and intensively testing our algorithms, we showed that they also performed very well in practice.

The algorithms presented may be used to solve large problems which can be broken down into subproblems and processed in parallel. Future avenues for research include the following:

1. Further refinements and better analyses of our algorithms to achieve a better theoretical worst-case performance bound.

2. Adding more constraints, such as the effect of a dedicated fast communication coprocessor in each workstation and the case when the size of each message is trivial, into our model to more accurately relate to the real world.

3. Further verifying our results by actually implementing the algorithms to solve practical problems and making real-time measurements of the parameters involved.

4. The algorithms and bounds herein form a basis for further study of more complicated task precedence graphs.

ACKNOWLEDGMENTS


The research reported in this paper was supported in part by National Science Council of Taiwan Grants 87-2213-E-001-022, 88-2213-E-001-026, 89-2213-E-001-015, and 89-2218-E-001-001, a single quarter leave, two MAP Undergraduate Research Awards, and a UROP Undergraduate Research Opportunities Grant from the University of Minnesota, Morris, Academia Sinica, Taipei, and the University of Minnesota China Center, Minneapolis.

The authors appreciate the comments from anonymous referees, which greatly improved the overall quality of this paper.

REFERENCES


Tsan-sheng Hsu graduated from National Taiwan University with a Bachelor of Science degree in computer science. He entered the PhD program of the University of Texas at Austin in 1988. He received the MS degree in computer science in 1990 and a PhD degree in computer science in 1993. He then joined the Institute of Information Science, Academia Sinica, Taiwan, as an assistant research fellow. He has been an associate research fellow since 1996. His research interests focus on the design, analysis, implementation, and performance evaluation of algorithms in graph theory, and in parallel and distributed computation. He is a member of the ACM, ACM SIGACT, IEEE Computer Society, SIAM, SIAM SIAG/DM, and is a lifetime member of IIIM.

Joseph C. Lee received the BS degree in computer science with a minor in studio art from the University of Minnesota, Morris Campus (UMM), in 1999. He is now a PhD student in computer science at Brown University, where his research areas include computer graphics and scientific visualization. As an undergraduate at UMM, Joe took part in the ACM Programming Contest World Finals and his team also placed first in the IBM-sponsored JAVA programming contest held at SIGCSE ’98 in Atlanta. Lee also presented his parallel fractal image compression research at the 12th National Conference on Undergraduate Research, Salisbury, Maryland, April, 1998.

Dian Rae Lopez received the BS degree from Moorhead State University in 1966 and the MS degree in mathematics from Michigan Technological University in 1969. She received the PhD degree in computer science in 1992 from Texas A&M University for her work on scheduling in parallel systems. Dr. Lopez is now an associate professor of computer science at the University of Minnesota, Morris campus. Her interests are in the area of parallel and distributed scheduling, graph theory, and algorithm design and analysis. She served three years as a project leader for the Computing Research Association’s Committee on the Status of Women in Computer Science and Engineering. She is a member of the IEEE Computer Society, ACM, ACM SIGACT, and ACM SIGCSE.

William A. Royce received the BA degree in computer science from the University of Minnesota, Morris campus, in 1997. Royce is a software engineer at Measurement Technology Laboratories in Bloomington, Minnesota, where he develops automated measurement systems for use in government and private laboratories. He is currently developing software to aid environmental agencies in collecting data for Environmental Protection Agency-mandated air monitoring. When not in the midst of creating new software components, you may find him venturing through the back country of Minnesota.