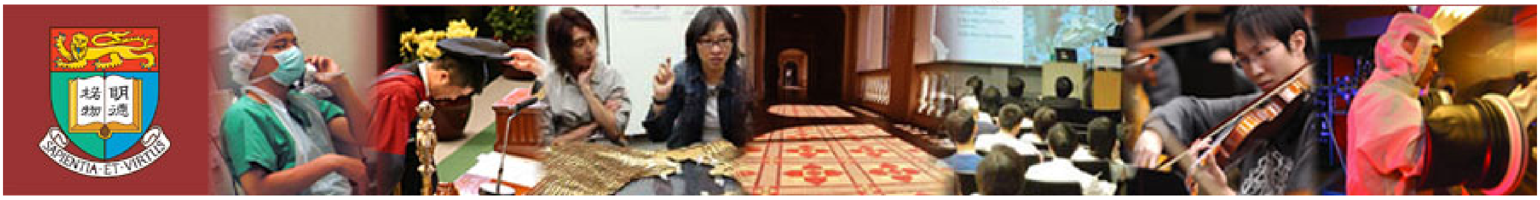


On Parallelizing the Multiprocessor Scheduling Problem

Ishfaq Ahmad, *Member*, *IEEE Computer Society*, and Yu-Kwong Kwok, *Member*, *IEEE*

Abstract—Existing heuristics for scheduling a node and edge weighted directed task graph to multiple processors can produce satisfactory solutions but incur high time complexities, which tend to exacerbate in more realistic environments with relaxed assumptions. Consequently, these heuristics do not scale well and cannot handle problems of moderate sizes. A natural approach to reducing complexity, while aiming for a similar or potentially better solution, is to parallelize the scheduling algorithm. This can be done by partitioning the task graphs and concurrently generating partial schedules for the partitioned parts, which are then concatenated to obtain the final schedule. The problem, however, is nontrivial as there exists dependencies among the nodes of a task graph which must be preserved for generating a valid schedule. Moreover, the time clock for scheduling is global for all the processors (that are executing the parallel scheduling algorithm), making the inherent parallelism invisible. In this paper, we introduce a parallel algorithm that is guided by a systematic partitioning of the task graph to perform scheduling using multiple processors. The algorithm schedules both the tasks and messages, and is suitable for graphs with arbitrary computation and communication costs, and is applicable to systems with arbitrary network topologies using homogeneous or heterogeneous processors. We have implemented the algorithm on the Intel Paragon and compared it with three closely related algorithms. The experimental results indicate that our algorithm yields higher quality solutions while using an order of magnitude smaller scheduling times. The algorithm also exhibits an interesting trade-off between the solution quality and speedup while scaling well with the problem size.

Index Terms—Parallel processing, resource management, processor allocation, scheduling, task graphs, parallel algorithms, message-passing, multiprocessors.

1 Introduction

Given a parallel program modeled by a directed acyclic graph (DAG), in which nodes represent the tasks and edges represent the communication costs as well as the dependencies among the tasks, the goal of a scheduling algorithm is to minimize the execution time by properly allocating and sequencing the tasks on the processors such that the precedence constraints are preserved. A DAG can be scheduled off-line if the structure of the program, in terms of its intertask dependencies and task execution and communications costs, is known before the program execution. The scheduling problem is intractable even when severe restrictions are imposed on the task graph and the machine model [11]. The following simplifying assumptions about the task graph are common:

- The structure of the task graph is restricted (e.g., tree, fork-join, etc.) rather than arbitrary;
- All nodes in the graph have the same computation costs;
- The communication costs on the edges are zero.

Likewise, the following assumptions about the machine model are common:

- The number of available processors is unlimited;
- The processors are fully connected;
- The network links are contention-free;
- The processors are homogeneous, that is, their processing speeds are the same.

The problem is complex even in the presence of such assumptions, and polynomial-time algorithms for optimal solutions are known only in three cases: 1) The task graph is a tree with unit node-weights and there exist multiple fully connected homogeneous processors [11]; 2) The task graph is arbitrarily structured with unit node-weights and the machine consists of exactly two homogeneous processors [11]; 3) The graph is interval-ordered [3]. The communication cost on the edges of the task graph is ignored in the first two cases and restricted to unit-cost in the third case. The problem is NP-complete when arbitrary communication costs are taken into account.

Due to the intractability of the problem, heuristics are devised for obtaining suboptimal solutions in an affordable amount of computation time. Even though most heuristics can produce high quality solutions, their time complexities are quite high [26]. Furthermore, heuristics designed with more relaxed assumptions tend to incur higher time complexities. Thus, many heuristics work well for small task graphs but do not scale well with the problem size. Therefore, the solution quality and applicability are usually in conflict with the goal of reducing the time complexity.

To reduce the time complexity without compromising the solution quality, a natural approach is to parallelize the scheduling algorithm. This has an extra advantage in that the parallel machine using the algorithm can schedule its own programs in a short time. Moreover, with the great

- *I. Ahmad is with the Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. E-mail: iahmad@cs.ust.hk.*
- *Y.-K. Kwok is with the Department of Electrical and Electronic Engineering, The University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: ykwok@eee.hku.hk.*

Manuscript received 18 Mar. 1997; revised 29 Apr. 1998.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 104681.

1045-9219/99/$10.00 © 1999 IEEE

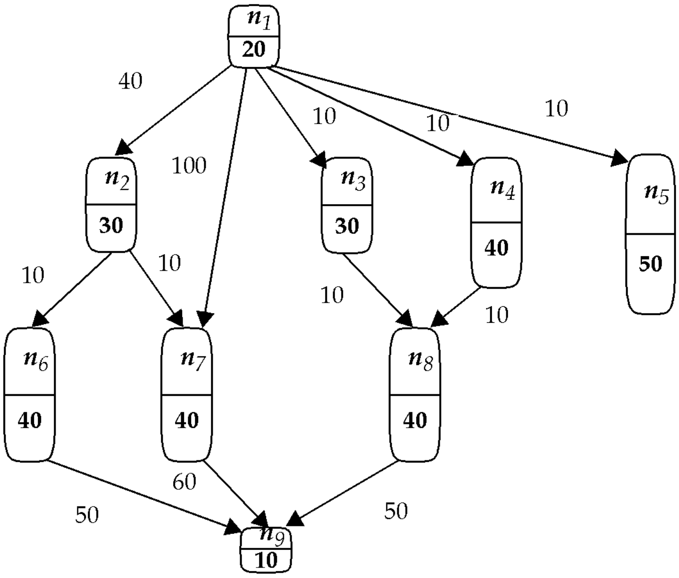


Fig. 1. A task graph.

advances in networking and communication tools, heterogeneous environments using both parallel machines and networks of workstations are becoming increasingly popular. In such an environment, a parallel machine can execute a fast algorithm to schedule tasks on multiple heterogeneous workstations. However, tackling the general DAG scheduling problem in parallel, for the most part, is an unexplored approach despite that there are a few recent attempts in devising parallel heuristics for a class of problems, such as the job-shop scheduling problem [9], [23] in which all the jobs are independent. Indeed, all of the well known DAG scheduling algorithms are sequential in nature. The reason being that the task ordering technique in scheduling is widely reckoned as inherently sequential as the time clock for ordering tasks is global. Thus, parallelism within a scheduling algorithm is invisible.

In this paper, we propose a parallel algorithm, called PBSA (Parallel Bubble Scheduling and Allocation), that produces high quality scheduling solutions without making any of the specific simplifying assumptions mentioned above. The novelty of the algorithm lies in a systematic partitioning of the task graph which guides the concurrent generation of subschedules. The algorithm works well for regular or irregular graph structures with arbitrary communication and computation costs, handles arbitrarily connected network of target processors, and is suitable for homogenous and heterogeneous processor systems.

The remaining paper is organized as follows: In the next section, we present a brief overview of various approaches that have been proposed for the DAG scheduling problem. In Section 3, we present the proposed algorithm, and discuss its design principles. Section 4 includes some scheduling examples illustrating the operation of the algorithm. We present the experimental results in Section 5, and conclude the paper with some final remarks in Section 6.

2 The Scheduling Problem and Related Work

In static scheduling, we represent a parallel program by a directed acyclic graph (DAG), also known as the task graph or macro-dataflow graph. In a DAG, $G = (V, E)$, V is a set of v nodes, representing the tasks, and E is a set of e directed edges, representing the communication message. Edges in a DAG are directed and, thus, capture the precedence constraints among the tasks. The cost of node n_i, denoted as $w(n_i)$, represents the computation costs of the task and the cost of an edge (n_i, n_j), denoted by $c(n_i, n_j)$, represents the communication cost of the message. The source node of an edge is called a parent node, while the destination node is called a child node. A node with no parent is called an entry node and a node with no child is called an exit node. A node can only start execution after it has gathered all of the messages from its parent nodes. Fig. 1 shows an example DAG which we will use in the subsequent discussion.

We assume the target platform to be a distributed memory multiprocessor system such that a processor and its local memory, constituting a *processing element* (PE), communicate with other PEs through message-passing. The communication cost between two nodes assigned to the same processor is assumed to be zero.

The objective of scheduling is to minimize the *schedule length*, which is defined as the maximum finish time of all nodes, by properly assigning tasks to processors such that the precedence constraints are preserved. We classify the existing algorithms proposed in this context into four categories:

- *Bounded Number of Processors (BNP) Scheduling:* A BNP algorithm schedules a DAG to a limited number of processors directly. The processors are assumed to be fully connected without any regard to link contention and scheduling of messages.

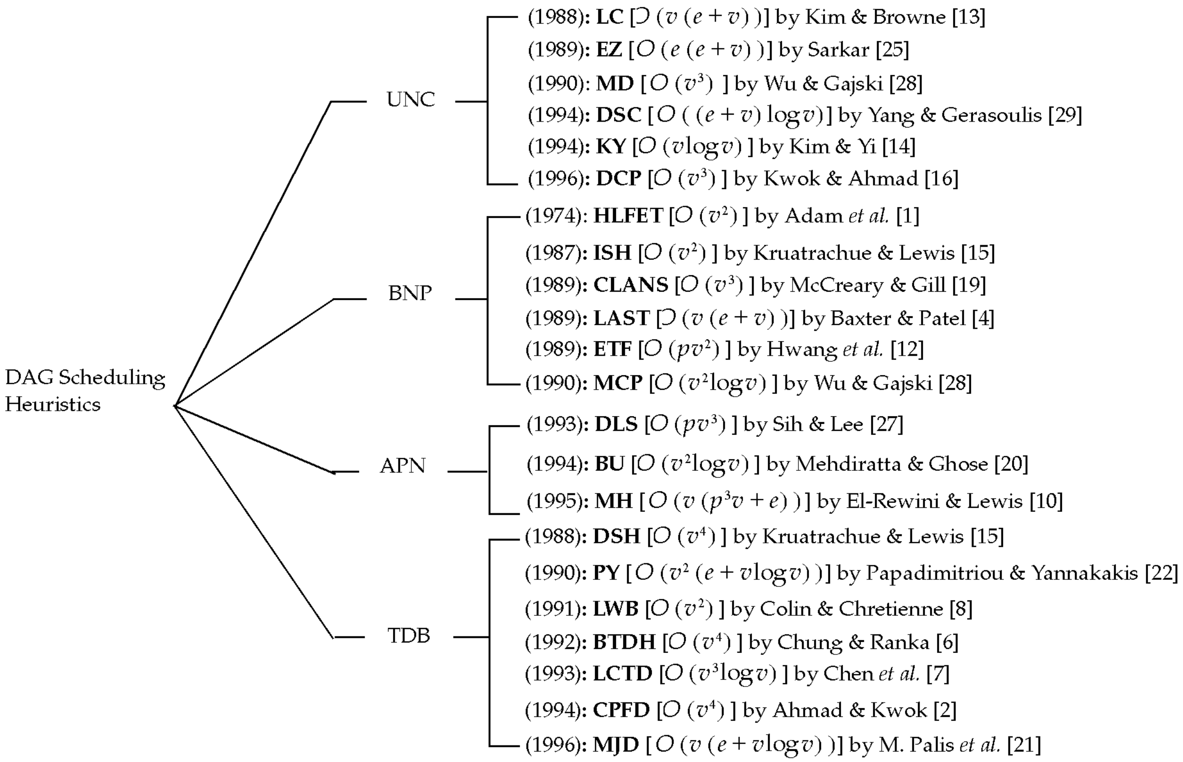

DAG Scheduling Heuristics
UNC
(1988): **LC** [$O(v(e+v))$] by Kim & Browne [13]
(1989): **EZ** [$O(e(e+v))$] by Sarkar [25]
(1990): **MD** [$O(v^3)$] by Wu & Gajski [28]
(1994): **DSC** [$O((e+v)\log v)$] by Yang & Gerasoulis [29]
(1994): **KY** [$O(v\log v)$] by Kim & Yi [14]
(1996): **DCP** [$O(v^3)$] by Kwok & Ahmad [16]
BNP
(1974): **HLFET** [$O(v^2)$] by Adam *et al.* [1]
(1987): **ISH** [$O(v^2)$] by Kruatrachue & Lewis [15]
(1989): **CLANS** [$O(v^3)$] by McCreary & Gill [19]
(1989): **LAST** [$O(v(e+v))$] by Baxter & Patel [4]
(1989): **ETF** [$O(pv^2)$] by Hwang *et al.* [12]
(1990): **MCP** [$O(v^2\log v)$] by Wu & Gajski [28]
APN
(1993): **DLS** [$O(pv^3)$] by Sih & Lee [27]
(1994): **BU** [$O(v^2\log v)$] by Mehdiratta & Ghose [20]
(1995): **MH** [$O(v(p^3v+e))$] by El-Rewini & Lewis [10]
TDB
(1988): **DSH** [$O(v^4)$] by Kruatrachue & Lewis [15]
(1990): **PY** [$O(v^2(e+v\log v))$] by Papadimitriou & Yannakakis [22]
(1991): **LWB** [$O(v^2)$] by Colin & Chretienne [8]
(1992): **BTDH** [$O(v^4)$] by Chung & Ranka [6]
(1993): **LCTD** [$O(v^3\log v)$] by Chen *et al.* [7]
(1994): **CPFD** [$O(v^4)$] by Ahmad & Kwok [2]
(1996): **MJD** [$O(v(e+v\log v))$] by M. Palis *et al.* [21]

Fig. 2. Some DAG scheduling heuristics.

- *Unbounded Number of Clusters (UNC) Scheduling:* A UNC algorithm schedules a DAG to an unbounded number of *clusters*. The clusters generated by these algorithms may be mapped onto the processors using a separate mapping algorithm. These algorithms assume the processors to be fully connected.
- *Arbitrary Processor Network (APN) Scheduling:* An APN algorithm performs scheduling and mapping on an architecture in which the processors are connected via a network topology. An APN algorithm also explicitly schedule communication messages on the network channels, taking care of the link contention factor. The proposed algorithm belongs to this class.
- *Task-Duplication-Based (TDB) Scheduling:* A TDB algorithm duplicates tasks in order to reduce the communication overhead. Duplication, however, can be used in any of the other three classes of algorithms.

Fig. 2 depicts a chronological summary of various algorithms reported in the recent literature, and their categorization according to our classification method. The summary also includes the complexities of these algorithms, in terms of the number of nodes (v) and edges (e) in the task graph, and the number of processors (p). This classification helps in making a comparison among the algorithms within the same class; clearly, algorithms belonging to different classes cannot be directly compared. The survey is by no means complete (an extensive taxonomy of the general scheduling problem is proposed in [5]) since a complete overview of the literature is beyond the scope of this paper. However, most of the reported DAG scheduling algorithms can be categorized according to our scheme. For our purpose, we will compare the proposed algorithm with the three other APN algorithms (BU, DLS, and MH).

Tackling the general scheduling problem in parallel is a relatively unexplored approach in that hitherto only a few techniques have been suggested for some restricted forms of the scheduling problem. Recently, Pramanick and Kuhl [23] have proposed a paradigm, called *Parallel Dynamic Interaction* (PDI), for developing parallel search algorithms for many NP-hard optimization problems. The have applied the PDI method to the job-shop scheduling problem in which a set of independent jobs are scheduled to homogeneous machines. De Falco et al. [9] have suggested the use of parallel simulated annealing and parallel tabu search algorithms for the task allocation problem, in which a *Task Interaction Graph* (TIG), representing communicating processes in a distributed systems, is to be mapped to homogeneous processors. A TIG is different from a DAG in that the former is an undirected graph with no precedence constraints among the tasks. In a recent study, we have proposed a parallel genetic algorithm for BNP scheduling [17].

3 The Proposed Algorithm

In the following, we first explain the sequential operation of the proposed PBSA algorithm, and then discuss the graph partitioning and parallelization issues. Table 1 summarizes the definitions of the notations used throughout the paper.

TABLE 1
Definitions of Notations

NOTATION	DEFINITION
n_i	A node in the parallel program task graph
$w(n_i)$	The computation cost of node n_i (also called node weight)
c_{ij}	The communication cost of the directed edge from node n_i to n_j (also called edge weight)
v	The total number of nodes in the task graph
e	The total number of edges in the task graph
TPE	A processing element (PE) in the target system to which the task graph is being scheduled
p	The number of processing elements in the target multiprocessor system
PPE	A physical processing element executing the PBSA algorithm
P	The number of physical processing elements executing the PBSA algorithm
CP	A critical path of the task graph
CPN	Critical path node
IBN	In-branch node
OBN	Out-branch node
$DAT(n_i)$	The possible data available time of node n_i
$ST(n_i)$	The start-time of node n_i
$FT(n_i)$	The finish-time of node n_i
$VIP(n_i)$	The parent node of n_i whose message arrives the last
Pivot_TPE	The processor (in the target system) from which nodes migrate to the adjacent processors
$Proc(n_i)$	The processor accommodating node n_i
CCR	Communication-to-computation Ratio (average edge weight divided by average node weight)
RPN	A remote parent node (not in the local graph partition of a slave PBSA program)
EPST	The earliest possible start-time of an RPN
LPST	The latest possible start-time of an RPN
EST	The estimated start-time of an RPN
ETPE	The estimated target processor to which a RPN may have been scheduled

3.1 Scheduling Serially

In the following, we call the processors which execute the PBSA algorithm the physical processing elements (PPEs) in order to distinguish them from the target processing elements (TPEs) to which the task graph is to be scheduled. The sequential version is a special case in which only a single PPE is used to execute the PBSA algorithm. The algorithms starts by scheduling all the nodes to one TPE. It then improves the schedule by migrating the nodes to other TPEs. As such we refer to the serialized algorithm as simply the BSA algorithm.

To determine an accurate scheduling order, we arrange the nodes in an order called the *CPN-Dominant sequence*. The sequence is determined as follows: In a task graph, there is a set of nodes and edges, forming a path from an entry node to an exit node, of which the sum of computation and communication costs is the maximum. This set of nodes is called the critical path (CP) of the task graph, and the sum of the computational costs of the nodes on the CP provides a loose lower bound on the schedule length of the graph. A CP can be easily found by using a depth-first search; there can be multiple CPs, from which we select the one with the maximum sum of computation costs.

The CP nodes (CPNs) are more important nodes (since their finish times determine the final schedule length) and should be considered for scheduling as early as possible in the scheduling process. However, a CPN can only start its execution if all of its parent nodes have finished their execution. Thus, before a CPN is scheduled, all of its parent nodes must be scheduled. The parent node of a CPN need not be a CPN itself. We call such a node an in-branch node (IBN), which is defined as a node that is not a CPN and from which there is a path reaching a CPN. In addition to CPNs and IBNs, there is another class of nodes called out-branch node (OBN). An OBN is a node which is neither a CPN nor an IBN. Based on this classification, the nodes of any connected graph can be divided into these three categories. Using this categorization, the relative importance of nodes are in the order: CPNs, IBNs, and OBNs. In the *CPN-Dominant sequence*, the precedence constraints among nodes are preserved in the following manner: the IBNs reaching a CPN are always inserted before the CPN; OBNs are appended to the sequence in a topological order so that a parent OBN is always inserted before its child OBN.

The CPN-Dominant sequence is constructed by a procedure called *Serial_Injection*, which assigns the entire

CPN-dominant sequence into a single processor called the *pivot* processor (the one with the largest number of incident links). The *Serial_Injection* procedure is outlined below.

SERIAL_INJECTION(PIVOT_PROCESSOR):

```
1. Initialize the CPN-Dominant Sequence as an empty list of
   tasks. Make the entry CPN to be the first node in the
   sequence. Set Position to 2. Let $n_x$ be the next CPN.
while not all CPNs are included in the CPN-Dominant
 Sequence do
2. If $n_x$ has all its parent nodes in the sequence then
3.    Put $n_x$ at Position in the sequence and increment
      Position.
4. else
5.    Let $n_y$ be the parent node of $n_x$ which is not in the
      sequence and has the largest b-level.[1] Ties are broken by
      choosing the parent with a smaller t-level. If $n_y$ has all
      its parent nodes in the sequence, put $n_y$ at Position in
      the sequence and increment Position. Otherwise,
      recursively include all the ancestor nodes of $n_y$ in the
      sequence so that the nodes with a larger value of b-level
      are considered first.
6.    Repeat the above step until all the parent nodes of $n_x$
      are in the sequence. Put $n_x$ in the sequence at Position.
7. endif
8. Make $n_x$ be the next CPN.
endwhile
9. Append all the OBNs to the sequence in a decreasing
   order of b-level.
10. Inject the CPN-Dominant Sequence to Pivot_Processor.
```

For scheduling tasks and messages, we employ an incremental adaptive approach. After the serial injection process, the nodes are incrementally "bubbled-up" by migrating to the adjacent processors in a breadth-first order. In the course of node migration, messages are also incrementally routed and scheduled to the most suitable time slots on an optimized route. This is because a node will not migrate if its start time cannot be reduced by the migration or if the destination processor for migration is not a valid choice as specified by the underlying routing scheme.

A candidate node for migration is a node that has its data arrival time (DAT) (defined as the time at which the last message from its parent nodes finishes delivery) earlier than its current start time on the pivot processor. The goal of a migration is to schedule the node to an earlier time slot on one of the adjacent processors that allows the largest reduction in the start time of the node. To determine the largest start time reduction, we need to compute the DAT and the start time of the node on each adjacent processor. A node can be scheduled to a processor if the processor has an idle time slot that starts later than the node's DAT and is large enough to accommodate the node. The following procedure outlines the computation of the start time of a node on a processor.

Computation of ST(n_i, Q):

Precondition: m nodes $\{n_{Q_1}, n_{Q_2}, \ldots, n_{Q_m}\}$ *have been scheduled on processor* Q ($m \geq 0$).

1. Check if there exists some k such that:

$$ST(n_{Q_{k+1}}, Q) - \max\{FT(n_{Q_k}, Q), DAT(n_i, Q)\} \geq w(n_i)$$

where $k = 0, \ldots, m$, $ST(n_{Q_{m+1}}, Q) = \infty$, *and* $FT(n_{Q_0}, Q) = 0$.

2. If such k exists, compute $\max\{FT(n_{Q_l}, Q), DAT(n_i, Q)\}$ with *l* being the smallest *k* satisfying the above inequality, and return this value as the start time of n_i on processor Q; otherwise, return ∞.

The above procedure states that to determine the start time of a node on a processor, we have to examine the first idle time slot, if any, before the first node on that processor. We check whether the overlapping portion, if any, of the idle time slot and the time period in which the node can start execution, is large enough to accommodate the node. If there is such an idle time slot, the start time for the node is the earliest one; if not, we proceed to try the next idle time slot.

The DAT of a node on a processor is constrained by the finish times of the parent nodes and the message arrival times. If the node under consideration and its parent node are scheduled to the same processor, the message arrival time of this parent node is simply its finish time on the processor (intraprocessor communication time is ignored). On the other hand, if the parent node is scheduled to another processor, the message arrival time depends on how the message is routed and scheduled on the links. To schedule a message on a link, we search for a suitable idle time slot on the link to accommodate the message. A message can be scheduled to a link if the link has an idle time slot that is later than the source node's finish time and is large enough to accommodate the message. The following procedure outlines the scheduling of a message $e_x = (n_i, n_j)$ to a link L:

Computation MST(e_x, L):

Precondition: m messages $\{e_1, \ldots, e_m\}$ *have been scheduled on the link L* ($m \geq 0$).

1. Check if *there exists some k such that*

$$MST(e_{k+1}, L) - \max\{MFT(e_k, L), FT(n_i, Proc(n_i))\} \geq c_{ij}$$

where $k = 0, \ldots, m$ *and* $MST(e_{m+1}, L) = \infty$, $MFT(e_o, L) = 0$.

2. If such *k* exists, compute

$$\max\{MFT(e_{r+1}, L), FT(n_i, Proc(n_i))\}$$

with *r* being the smallest *k* satisfying the above inequality, and return such value as the start time of e_x on L (denoted by $MST(e_x, L)$); otherwise return ∞.

This procedure determines the start time of a message on a link, in a similar manner as the procedure used for determining the start time of a node on a processor. The DAT of the node on the processor is then simply the largest value of the message arrival times. The parent node that

[1] The *b-level* of a node is the length (sum of the computation and communication costs) of the longest path from this node to an exit node. The *t-level* of a node is the length of the longest path from an entry node to this node (excluding the cost of this node).

corresponds to this largest value of the message arrival time is called the *Very Important Parent* (VIP).

The sequential BSA algorithm is outlined below. The procedure *Build_Processor_List* constructs a list of processors in a breadth-first order from the first pivot processor. The procedure *Serial_Injection* constructs the *CPN-Dominant sequence* of the nodes and injects all the nodes to the first pivot processor.

THE BSA ALGORITHM:

```
1. Load processor topology and input task graph
2. Pivot_TPE ← the processor with the highest degree[2]
3. Processor_list ← Build_Processor_List(Pivot_TPE)
4. Serial_Injection(Pivot_TPE)
5. while Processor_list_not_empty do
6.   Pivot_TPE ← first processor of Processor_list
7.   for each $n_i$ on Pivot_TPE do
8.     if $ST(n_i, Pivot\_TPE) > DAT(n_i, Pivot\_TPE)$ or
         $Proc(VIP(n_i)) \neq Pivot\_TPE$ then
9.       Determine DAT and ST of $n_i$ on each adjacent
         processor $PE'$
10.      if there exists a $PE'$ such that
           $ST(n_i, PE') < ST(n_i, Pivot\_TPE)$ then
11.        Allow $n_i$ to migrate from $Pivot\_TPE$ to $PE'$ and
           update start times of nodes and messages
12.      else if $ST(n_i, PE') = ST(n_i, Pivot\_TPE)$ and
           $Proc(VIP(n_i)) = PE'$ then
13.        Allow $n_i$ to migrate from $Pivot\_TPE$ to $PE'$, and
           update start times of nodes and messages
14.      end if
15.    end if
16.  end for
17. end while
```

The procedure *Build_Processor_List* takes $O(p^2)$ time because it involves a breadth-first traversal of the processor graph. The procedure *Serial_Injection* takes $O(e+v)$ time because the CPN-Dominant sequence can be constructed in $O(e+v)$ time. Thus, the dominant step is the while-loop from step 5 to step 17. In this loop, it takes $O(e)$ time to compute the *ST* and *DAT* values of the node on each adjacent processor. If migration is done, it also takes $O(e)$ time. Since there are $O(v)$ nodes on the *Pivot_TPE* and $O(p)$ adjacent processors, each iteration of the while loop takes $O(p^2ev)$ time. Thus, the BSA algorithm takes $O(p^2ev)$ time.

If the target processors are heterogeneous, the decision as to whether a migration should be taken is determined by the finish times of the nodes rather than the start times. This is because, for heterogeneous processors, the execution time of a task varies for different processors; hence, even if a task can start at the same time for two distinct processors, its finish times can be different. Moreover, the first pivot processor will be the one on which the CP length is the shortest in order to further minimize the finish times of the CPNs by exploiting the heterogeneity of the processors.

3.2 Scheduling in Parallel

We parallelize the algorithm by partitioning the DAG into multiple parts. In a simple approach, one can partition the DAG horizontally or vertically [18], [24]. A horizontal partitioning means dividing the DAG into layers of nodes such that the paths in the DAG are levelized. In contrast, a vertical partitioning means dividing the DAG along the paths. Fig. 3 illustrates these two simple techniques. The partitioned pieces can then be scheduled independently and their resultant schedules can be combined. However, the efficiency of a horizontal or vertical partitioning depends on the graph structure. In the PBSA algorithm, we partition the CPN-Dominant sequence (which is neither vertical nor horizontal but is a combination of the two). The number of partitions is equal to the number of PPEs available. Each PPE independently schedules the nodes belonging to its own partition by using a sequential BSA algorithm. The subschedules for all the partitions are generated and then concatenated to form the final complete schedule.

Due to the dependencies between the nodes of two adjacent partitions, the PPEs have to share some global information in the scheduling process. Specifically, when a PPE attempts to schedule a node in its partition, it has to know the finish time of a parent node in another partition, called the *remote parent node* (RPN), so that the PPE can determine its earliest start time. A straight forward approach is to let the PPEs communicate and exchange information about the start times during the scheduling process. This approach, however, is not efficient because local schedules in the PPEs need to be revised, resulting in an excessive overhead due to communication among the PPEs which limits the speedup. In our approach, we use an estimated information in each PPE so that inter-PPE communication is minimized. These estimates are given in the following definitions:

Definition 1. *The earliest possible start time (EPST) of a node is the largest sum of computation costs from an entry node to the node but not including the node itself.*

Definition 2. *The latest possible start time (LPST) of a node is the sum of computation costs from the first node in the serial injection ordering to the node but not including the node itself.*

The start time of a node is bounded below by its EPST (due to the precedence constraints) and bounded from above by the node's LPST (because LPST is the start time of the node when the DAG is serialized by the serial injection process). Therefore, EPST and LPST represent the most optimistic estimate and the most pessimistic estimate for the start time of an RPN, respectively.

An RPN can be scheduled to start at any time between these two extremes. The crux is to pick an accurate estimate for a RPN's start time from all values between the two extremes. In our approach, if the RPN is a CPN, we take an optimistic estimate for its start time; otherwise, we take a conservative estimate. Specifically, the estimated start time of a RPN is taken to be its EPST if it is a CPN; otherwise, it is taken to be $\alpha\text{EPST} + (1-\alpha)\text{LPST}$. Here, the parameter α ($0 \leq \alpha \leq 1$), called the *importance factor*, indicates the timeliness of the scheduling of the RPN in that a larger value of implies that the RPN's estimated start time is closer to its EPST, while a smaller value implies its estimated start time is

[2] The *degree* of a PE is defined as its number of incident edges in the processor network graph.

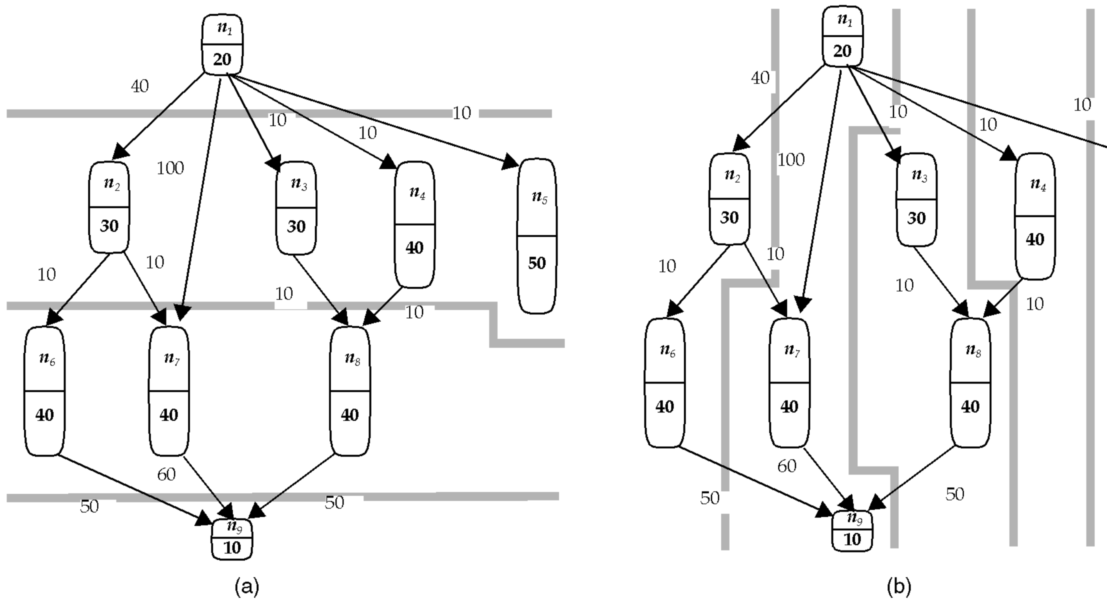


Fig. 3. Two simple strategies for parititioning a task graph. (a) Horizontal partitioning, (b) vertical partitioning.

closer to its LPST. In any case, α is a positive real number less than 1 and has to be determined heuristically. In Section 5, we present some experimental results about the effect of α on the schedule lengths.

In addition to the estimated start time of an RPN, we need to know the TPE to which the RPN is scheduled. This is essential in determining the DAT of a node to be scheduled in order to select the most suitable TPE for the node. We estimate this in the following manner: If the RPN is a CPN, then we assume that it will be scheduled to the same TPE as the highest level CPN in the local partition; otherwise, we randomly pick one TPE to be the one to which the RPN is scheduled. We call this TPE of an RPN the *estimated* TPE (ETPE).

In the PBSA algorithm, one PPE is designated as the master and the others PPEs as the slaves. The master PPE is responsible for all pre-scheduling and post-scheduling work. This includes the serial injection, the task graph partitioning, and the concatenation of subschedules including resolving any conflicts in subschedules. The slave procedure of PBSA is outlined below.

```
PROCEDURE PBSA_Slave:
1. Receive the target processor network from PBSA_Master.
2. Receive graph partition together with the RPNs
   information (i.e., ESTs and ETPEs) from PBSA_Master.
3. Apply the serial BSA algorithm to the graph partition. For
   every RPN, its EST and ETPE are used for determining
   the DAT of a node to be scheduled in the local partition.
4. Send the resulting subschedule to PBSA_Master.
```

To derive the time complexity, suppose there are m nodes in the local partition of *PBSA_Slave*. As step 3 in *PBSA_Slave* is the dominant step, the complexity of *PBSA_Slave* is $O(p^2e'm)$, where e' is the number of edges in the local partition.

3.3 Concatenating Subschedules

The master procedure concatenates the subschedules. The concatenation process involves a matching of TPEs between adjacent subschedules. Obviously, an exhaustive matching is not feasible. To reduce the time complexity of subschedules concatenation, we employ a two-phase method. The objective of the first phase is to minimize the start time of the most important node in every subschedule. Such a node is likely to be a CPN. The second phase is for rearranging the exit nodes (that is, the nodes without any successor in the same partition) of a subschedule so that they can start earlier. This rearrangement can potentially make the most important node of the next subschedule start earlier.

In the first phase, for every subschedule, the earliest node among all TPEs is determined. Call this node the *leader* node and the TPE to which the leader node is scheduled the *leader* TPE. The leader node, together with all its succeeding nodes on the leader TPE, are concatenated to a TPE of its preceding subschedule such that the start time of the leader node is scheduled as early as possible. Such a TPE is called the *image* of the leader TPE. The neighboring TPEs of the leader TPE are then concatenated to the corresponding neighboring TPEs of the leader TPE's image. This is done to all TPEs in a breadth-first manner. In the concatenation process, nodes may need to be moved to start later than their original scheduled times because of the accommodation of the interpartition communication messages. In addition, the corresponding schedule of communication messages may also need to be adjusted.

In the second phase, after a subschedule is concatenated with its preceding subschedule, all exit nodes in the subschedule are examined for rescheduling. Specifically, each exit node is rescheduled to the TPE that allows the minimum start time. The procedure for performing this two-phase concatenation process, called *Concat_Schedules*, is outlined below.

```
PROCEDURE CONCAT_SCHEDULES:
1. for every pair of adjacent subschedules do
2.     Determine the earliest node in the latter subschedule.
       Call this the leader node. Call its TPE the leader TPE.
```

```
    3.  Concatenate all nodes, which are scheduled on the TPE
        accommodating the leader node, to a TPE in the former
        subschedule so that the leader node can start as early as
        possible.
    4.  Concatenate the nodes on all other TPEs to the TPEs of
        the former subschedule in a breadth-first order
        beginning from the neighbors of the leader TPE.
    5.  Reschedule the exit nodes in the latter subschedule so
        that they can start as early as possible.
    6.  Walk through the whole concatenated schedule to
        resolve any conflict between the actual start times and
        the estimated start times.
7. end for
```

To derive the time complexity, suppose that there are at most m nodes in every subschedule. Since step 2 and step 5 take $O(m)$ time, while steps 3, 4, and 6 take $O(m^2)$ time, the time complexity of *Concat_Schedules* is then $O(Pm^2)$, where P is the number of PPEs (that is, the number of subschedules).

The final schedule generated from concatenating the subschedules by the procedure *Concat_Schedules* is a valid schedule. This is formalized in the following theorem:

Lemma. *The final schedule produced by Concat_Schedules preserves the precedence constraints.*

Proof. Clearly, the precedence constraints within a subschedule are preserved by the slave program. On the other hand, interpartition precedence constraints are preserved because the CPN-Dominant sequence maintains the precedence constraints. Finally, step 6 of the procedure *Concat_Schedules* resolves any potential conflict between pairs of adjacent subschedules by pushing down nodes in the succeeding subschedules. The theorem is proven. □

With the procedure *Concat_Schedules*, the master procedure of the PBSA algorithm can be outlined below.

PROCEDURE *PBSA_Master*:

```
1. Load processor network topology and input task graph.
2. Serial_Injection
3. Partition the task graph into equal sized sets according to
   the number of node PPEs available. Determine the ESTs
   and ETPEs for every RPNs in all partitions.
4. Broadcast the processor network topology to all
   PBSA_Slave.
5. Send the particular graph partition together with the
   corresponding ESTs and ETPEs to each PBSA_Slave.
6. Wait until all PBSA_Slave finish.
7. Concat_Schedules
```

If there are P PPEs, the maximum size m of each partition will then be $\lceil\frac{v}{P}\rceil$. The dominant steps in *PBSA_Master* are steps 6 and 7. As describe above, step 6 takes $O(p^2e'm)$ and step 7 takes $O(Pm^2)$. The time complexity of *PBSA_Master* is then $O\left(p^2e'\lceil\frac{v}{P}\rceil + P\lceil\frac{v}{P}\rceil^2\right)$. Taking e' to be $\lceil\frac{e}{P}\rceil$, the time complexity is $O\left(p^2\lceil\frac{e}{P}\rceil\lceil\frac{v}{P}\rceil + P\lceil\frac{v}{P}\rceil^2\right)$.

To analyze the theoretical speedup, denoted by S_{BSA}^{PBSA}, of the PBSA algorithm with respect to the serial BSA algorithm, we start with the following expression:

node	*t-level*	*b-level*	*CPN-Dominant order*
*n_1	0	230	1
n_2	60	150	2
n_3	30	140	5
n_4	30	150	4
n_5	30	50	9
n_6	100	100	7
*n_7	120	110	3
n_8	80	100	6
*n_9	220	10	8

Fig. 4. The *t*-levels, *b*-levels, and the CPN-Dominant sequence of the nodes.

$$S_{BSA}^{PBSA} = O\left(\frac{p^2ev}{p^2\lceil\frac{e}{P}\rceil\lceil\frac{v}{P}\rceil + P\lceil\frac{v}{P}\rceil^2}\right).$$

Dropping the ceiling operators, we have:

$$S_{BSA}^{PBSA} = O\left(\frac{p^2ev}{\frac{p^2ev}{P^2} + \frac{v^2}{P}}\right) = O\left(\frac{1}{\frac{1}{P^2} + \frac{v}{p^2eP}}\right).$$

Since the second term in the denominator is much smaller than the first term, we ignore it and get the following approximate theoretical speedup of PBSA over serial BSA:

$$S_{BSA}^{PBSA} = O(P^2).$$

That is, the speedup grows as the square of the number of PPEs is used, which is superlinear. This superlinear speedup is mainly due to the fact that the PBSA algorithm estimates the start times of RPNs, allowing it to spend less time in scheduling interpartition edges.

4 Scheduling Examples

In this section, we present some examples to demonstrate the operation of the proposed algorithm using the task graph shown in Fig. 1. We describe the schedules generated by the serial BSA algorithm and the PBSA algorithm using three PPEs for a ring of four homogeneous target processors. The CPN-Dominant sequence is constructed as follows. The critical path of the task graph is $\{n_1, n_7, n_9\}$. The CPN-Dominant sequence is as follows:

$$n_1, n_2, n_7, n_4, n_3, n_8, n_6, n_9, n_5$$

(see Fig. 4; the CPNs are marked by an asterisk).

The serial BSA algorithm allocates the entire CPN-Dominant sequence to the first pivot processor PE 0 (Fig. 5a). In the first phase, n_1, n_2, and n_7 do not migrate because they are already scheduled to finish at the earliest possible times. But n_4 migrates to PE 1 because its start time will improve. Similarly, n_3 migrates to a neighboring processor PE 3, and n_8 migrates to PE 1. Fig. 5b shows the intermediate schedule after these migrations. Next, n_6 migrates to PE 3 (see Fig. 5c). The last CPN, n_9, migrates to PE 1, where its VIP n_8 is scheduled. This migration allows the only OBN n_5 to move up. The resulting schedule is shown in Fig. 5d, which is the final schedule as no more nodes can migrate to improve their start times. The final schedule length is 160. For this example, we also executed the MH,

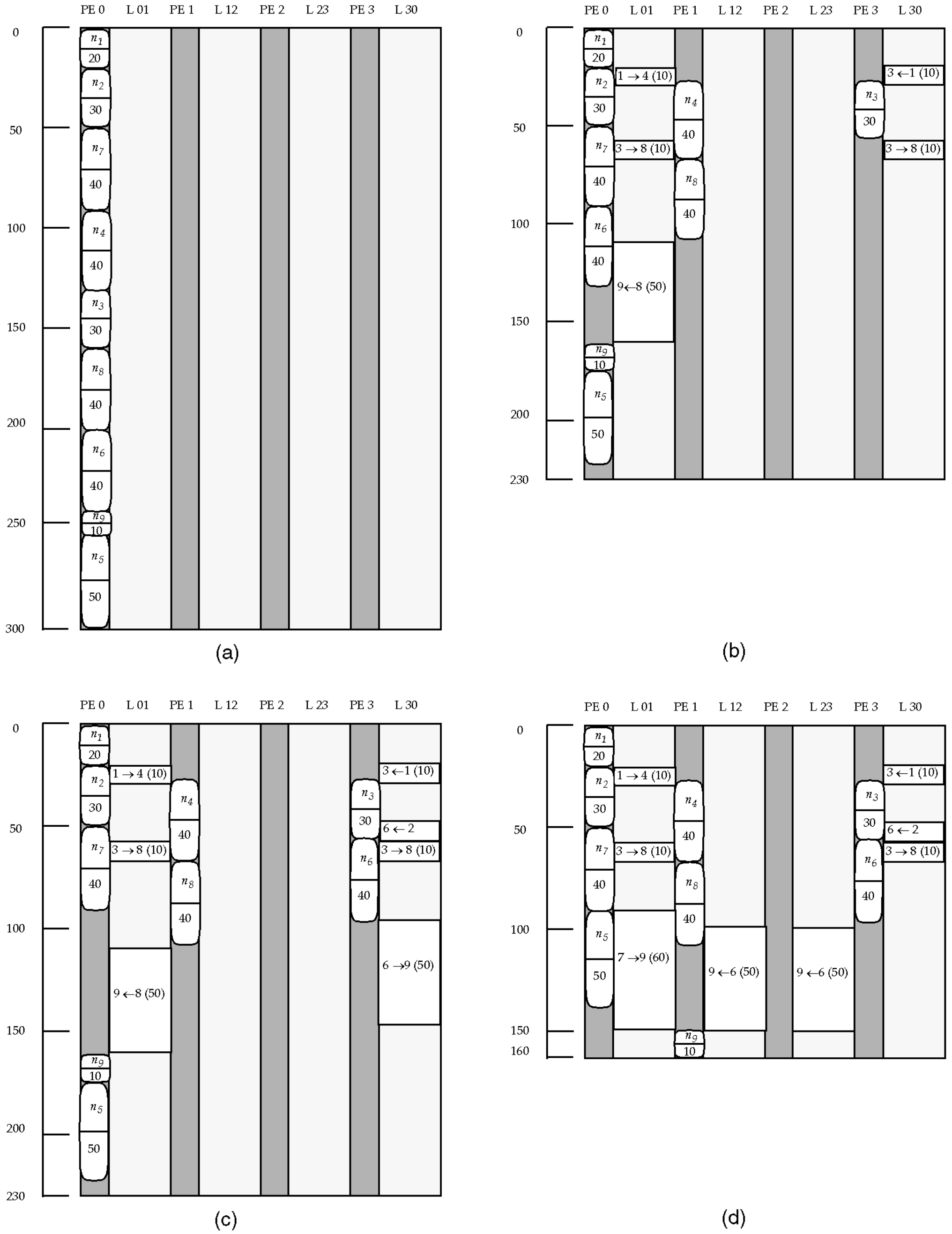
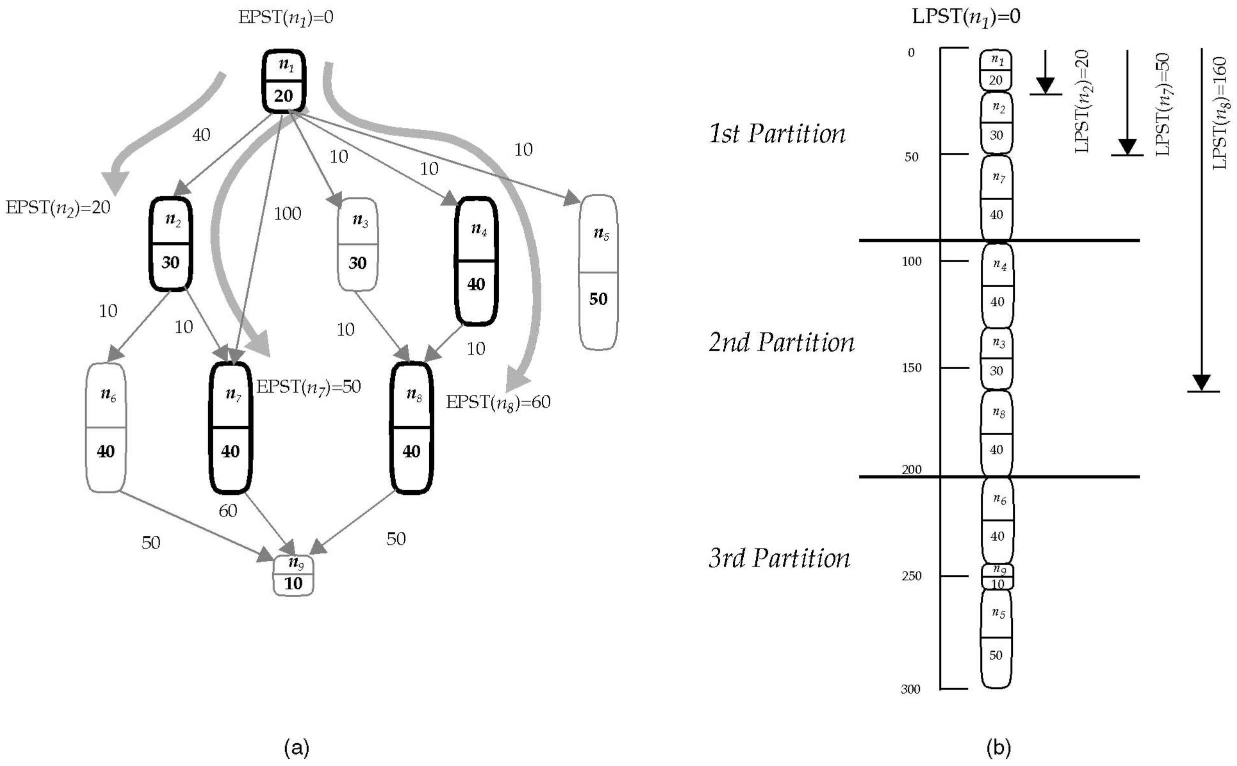


Fig. 5. (a) Intermediate schedule produced by BSA after serial injection (schedule length = 300, total comm. cost = 0), (b) intermediate schedule after n_4, n_3, n_8 migrate to neighboring processors (schedule length = 220, total comm. cost = 90), (c) intermediate schedule after n_6 migrates to PE 3 (schedule length = 220, total comm. cost = 150), (d) final schedule after n_9 migrates to PE 1 (schedule length = 160, total comm. cost = 220).

DLS, and BU algorithms. The schedule length is 200 for the MH and DLS algorithms and 270 for the BU algorithm.

Next, we execute the PBSA algorithm using three PPEs for the same task graph. Given the CPN-Dominant sequence, the task graph is now partitioned into three sets: $\{n_1, n_2, n_7\}$, $\{n_4, n_3, n_8\}$, and $\{n_6, n_9, n_5\}$ (see Fig. 6a and Fig. 6b. Fig. 6c shows the estimated parameters of the remote parent nodes (RPNs). According to the partitioning, there are only four RPNs, namely, n_1, n_2, n_7, and n_8. The value of the importance factor α is taken to be 0.7. For the nodes n_1, n_2, and n_7, their EPSTs are equal to their respective LPSTs. This is because they occupy the earliest possible positions in the CPN-Dominant sequence. The ESTs of these three nodes, therefore, are equal to the EPSTs (and LPSTs as well) independent of α. Indeed, their ESTs are correct with reference to the schedules shown in Fig. 5. Also, their ETPEs are all chosen to be PE 0. The EPST of n_8 is equal to the sum of computation costs of n_1 and n_4. The LPST, on the other hand, is equal to the sum of computation costs of the list of nodes $\{n_1, n_2, n_7, n_4, n_3\}$. Thus, the EST of n_8 is given by $60 \times 0.7 + 160 \times 0.3 = 90$. Since n_8 is not a CPN, its ETPE is randomly chosen to be PE 1.

A parallel execution of *PBSA_Slave* results in three subschedules, which are shown in Fig. 7a, Fig. 7b, and



(a)

(b)

RPN	EPST	LPST	EST	ETPE
n_1	0	0	0	PE 0
n_2	20	20	20	PE 0
n_7	50	50	50	PE 0
n_8	60	160	90	PE 1

(c)

Fig. 6. (a) Calculation of the EPSTs of the RPNs n_1, n_2, n_7, and n_8, (b) calculation of the LPSTs of the RPNs, (c) the estimated values of the RPNs.

Fig. 7c. For clarity, the RPNs of each partition are not shown in the subschedules. In the concatenation of the subschedules (see Fig. 8), the first two partitions are concatenated directly without any need of resolving conflicts. This is because of the accurate estimations of start times of the RPNs. The concatenation of the second and third partitions needs some more explanation. Since n_6 is the leader task in this partition, the *Concat_Schedules* procedure has to minimize its start time. To accomplish this, n_6 is appended to TPE 2 instead of TPE 0. This, in turn, makes n_9 to be scheduled to TPE 2. The OBN n_6 is appended to TPE 3 after the final walk-through of the schedule, which is done for minimization of start times. The generated schedule is almost similar to that of the BSA algorithm and the schedule length is the same. The only difference is in the scheduling of n_9. However, such scheduling of n_9 does not affect the final schedule length. Even though the value of EST of the RPN n_8 is over-estimated, scheduling n_9 to the same processor as n_6 turns out to be a good decision. This is because n_9 does not need to wait for the data from n_6 and, thus, can start earlier.

Finally, we note that the total communication costs incurred in the combined schedule is larger than that of the BSA algorithm. The reason is that the BSA algorithm does not allow nodes to migrate to other target processors if their start times do not improve. However, the slave program of the PBSA algorithm does not have this global knowledge about the intermediate state of the schedule, and thus it attempts to locally schedule every node to start at the earliest possible time, resulting in a higher utilization of the communication links.

5 Results

In this section, we present the performance results of the PBSA algorithm and compare it with the MH, DLS, and BU algorithms. Since the MH, DLS, and BU algorithms are sequential, we also compare their performance with our serialized BSA algorithm. Furthermore, we compare the solution quality and efficiency of the PBSA and BSA algorithms and observe the trade-off between the solution quality and running time.

5.1 Workload

For testing the algorithms, we generated two suites of task graphs: regular graphs and irregular graphs. The regular graphs represent three parallel applications including Gaussian elimination [28], Cholesky factorization [16], and

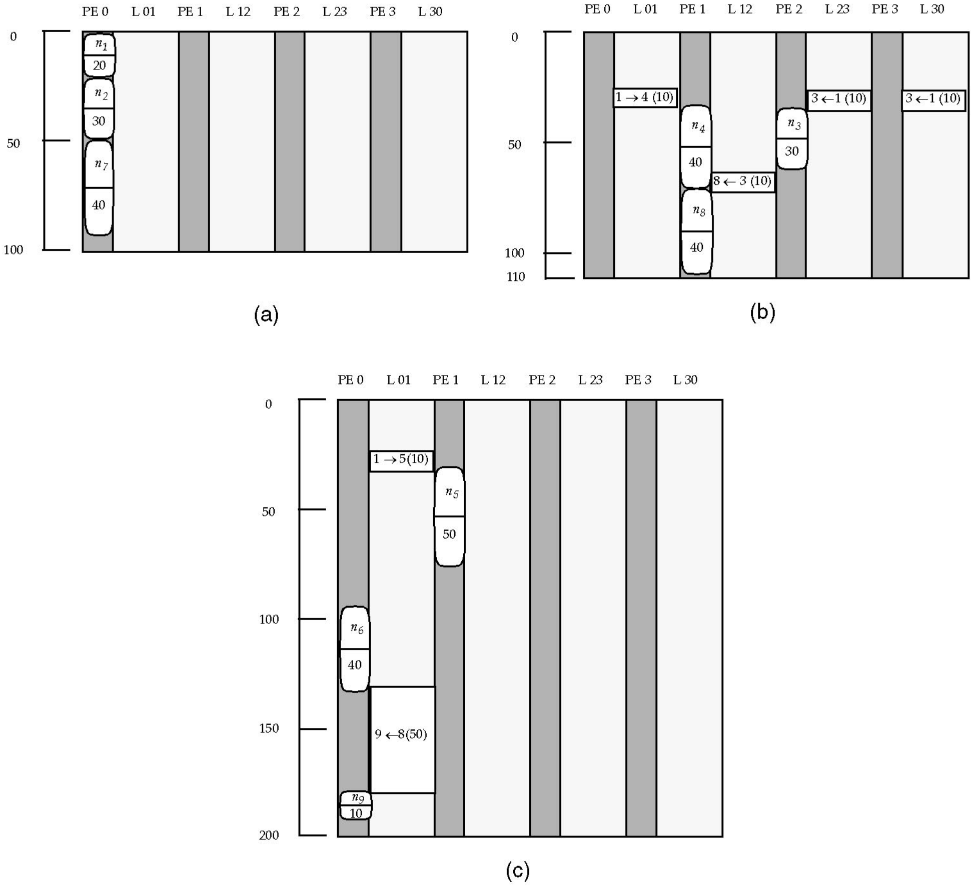


Fig. 7. (a) The schedule of the first partition, (b) the schedule of the second partition, (c) the schedule of the third partition.

FFT [16]. There exists a unique critical path in each Gaussian elimination and Cholesky factorization graphs. However, the lengths of some other paths in a Gaussian elimination graph are the same as that of the critical path. The FFT graphs are "dense" graphs in that they contain more edges than nodes. Furthermore, all the paths in an FFT graph are of equal length and, therefore, are all critical paths. Fig. 9 includes some miniature examples of these regular graphs. Since these applications operate on matrices, the number of nodes (and edges) in their task graphs depends on the size of the data matrix. The number of nodes in the task graph for each application is roughly $O(N^2)$, where N is the dimension of the matrix. For a given N, the number of nodes is about the same for all applications. In our experiments, we varied N from 19 to 64 with increments of 5 so that the numbers of nodes in the graphs range approximately from 200 to 2,000.

The suite of irregular task graphs consists of graphs with randomly generated structures. Within each type of graph, we chose three values of CCR (0.1, 1.0, and 10.0). A value of CCR equal to 0.1 represents a computation-intensive task graph (or coarse granularity), a value of 10.0 represents a communication-intensive task graph (or fine granularity), and a value of 1.0 represents a graph in which computation and communication are just about balanced. For the regular graphs, we generated the weights on the nodes and edges such that the average value of CCR corresponded to 0.1, 1.0, or 10.0. We generated the irregular graphs as follows: First, we randomly selected the computation cost of each node in the graph from a uniform distribution with mean equal to 40 (minimum = 2 and maximum = 78). Beginning with the first node, we chose a random number indicating the number of children from a uniform distribution with mean equal to $\frac{v}{10}$, thus, the connectivity of the graph increases with the size of the graph. We also randomly selected the communication cost of each edge from a uniform distribution with a mean equal to 40 times the specified value of CCR. The sizes of random graphs range from 200 to 2,000 nodes with increments of 200.

We used four target system topologies: an 8-node hypercube, a 4×2 mesh, an 8-node fully connected network, and an 8-node random topology. We assume these target systems to be composed of homogeneous processors. Unless otherwise stated, all results of the PBSA algorithm were generated with α (the *importance factor* used in estimating the start times of RPNs) equal to 0.5.

We implemented the scheduling algorithms on an Intel Paragon/XP-S; the sequential algorithms (MH, DLS, BU, and BSA) were executed on a single processor (an i860/XP) of the Paragon. For the PBSA algorithm, we used 2, 4, 8, and 16 processors.

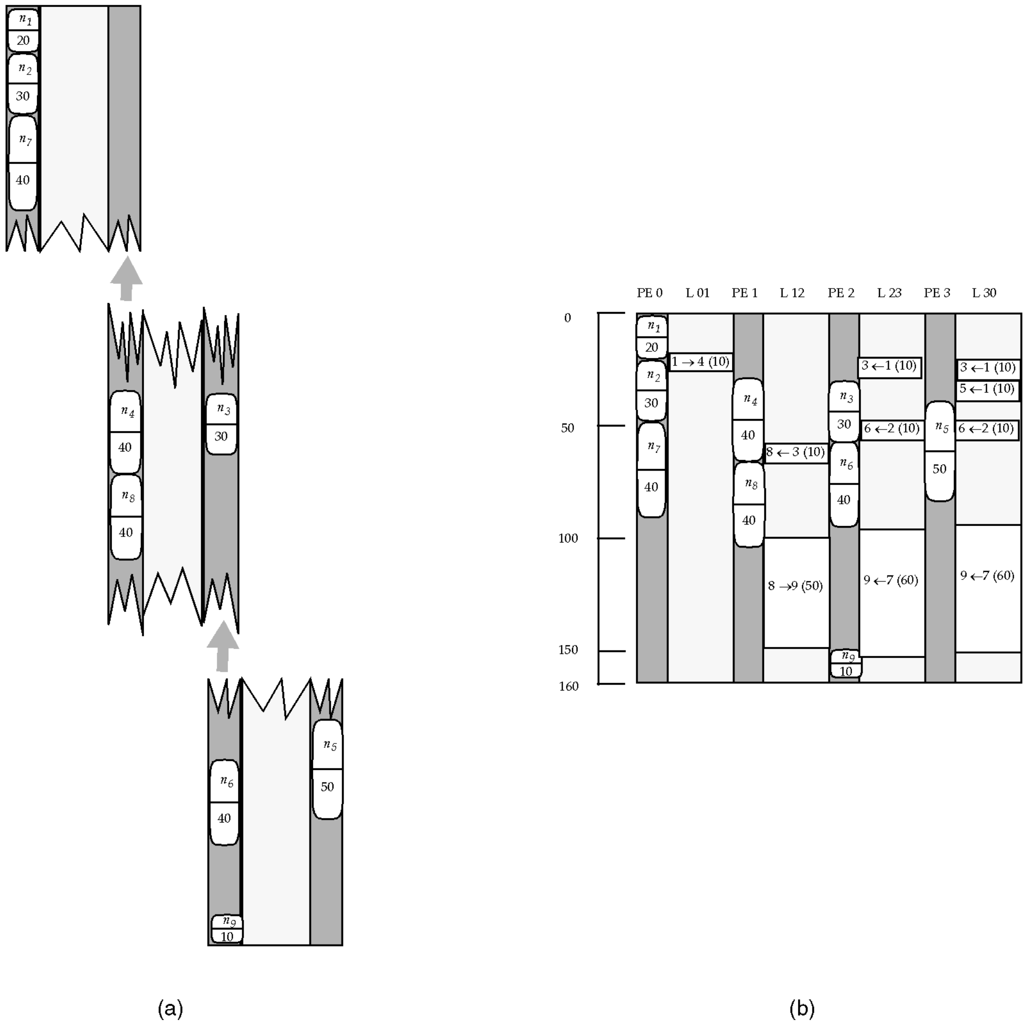


Fig. 8. (a) The subschedules concatenation process, (b) the combined final schedule generated by the PBSA algorithm (schedule length = 160, total comm. costs incured = 240).

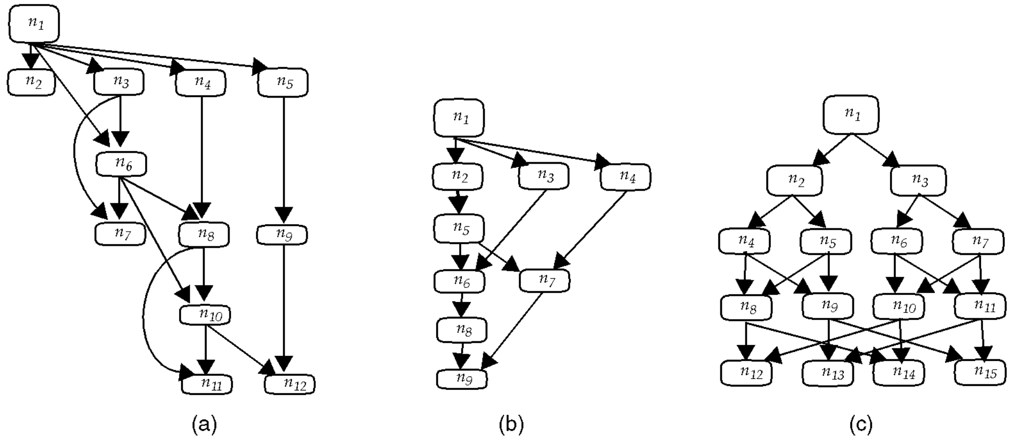
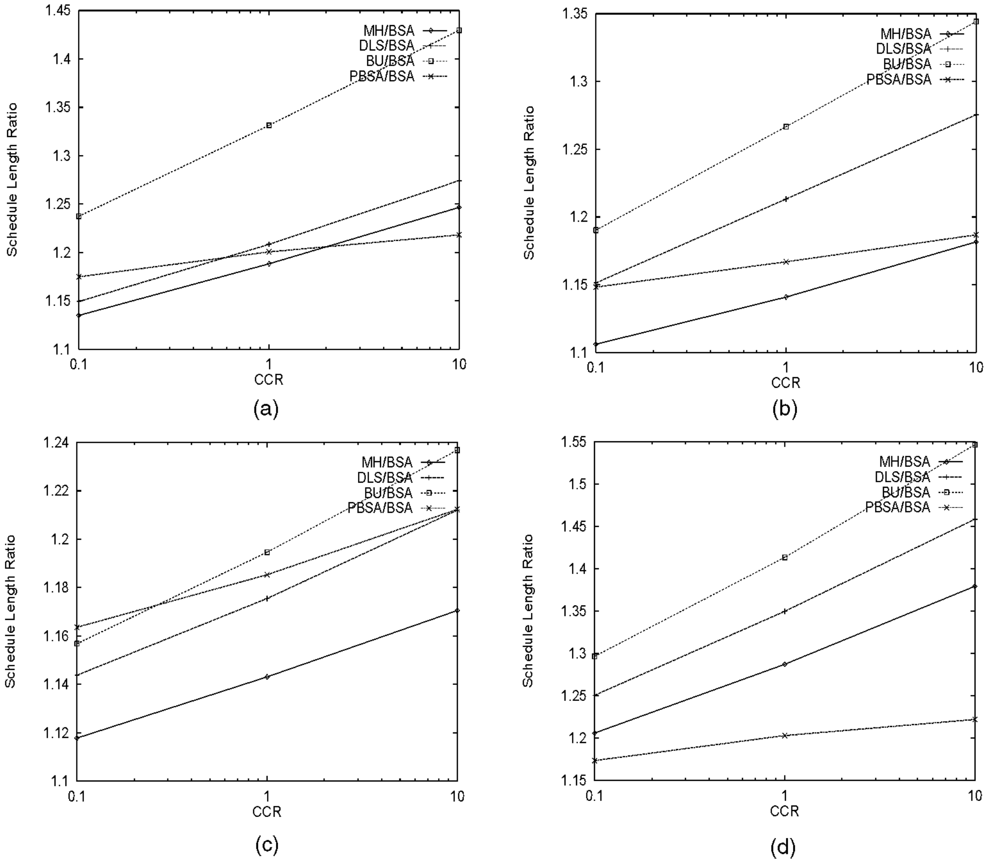


Fig. 9. Miniature examples for (a) Gaussian elimination task graph, (b) Cholesky factorization task graph, and (c) FFT task graph.

5.2 Schedule Lengths and Speedups

In our first experiment, we compared the schedules produced by the BSA algorithm and the PBSA algorithm (16 Paragon processors) with those of the MH, DLS, and BU algorithms, using the four types of task graphs of various sizes and four target topologies. We found that the graph size and processor network topology did not have a significant impact on the relative schedule lengths, but the value of CCR did affect the schedule length. Fig. 10 shows the impact of CCR on the ratios of the schedule length generated by the MH, DLS, BU, and PBSA algorithms, to



(a)

(b)

(c)

(d)

Fig. 10. Average ratios of schedule lengths generated by the MH, DLS, BU, and PBSA (using 16 PPEs) algorithms to those of the BSA algorithm for the four types of graphs against CCR. (a) Gaussian elimination graphs, (b) Cholesky factorization graphs, (c) FFT graphs, (d) random graphs.

those of the BSA algorithm. Each point on the curve is an average of 40 schedule length ratios (10 graphs and four topologies). From these results, we make the following observations:

- The BSA algorithm outperforms the MH, DLS, and BU algorithms since all ratios are greater than 1;
- The margin of improvement is different for each algorithm: the largest improvement is over the BU algorithm (up to about 55 percent for the random graphs); the minimum improvement is over the MH algorithm;
- The margin of improvement is larger when the value of CCR is large;
- The PBSA algorithm (using 16 PPEs) outperforms the MH, DLS, and BU algorithms in most cases;
- For the Gaussian elimination and Cholesky factorization graphs, the PBSA algorithm performed comparably with the DLS algorithm for smaller values of CCR. For the FFT graphs, the PBSA algorithm performed worse than the DLS and MH algorithm.
- The PBSA algorithm outperformed the BU algorithm for all the regular graphs, except for the FFT graphs with CCR equal to 0.1.

These observations are explained through the following reasons: The CPN-Dominant sequence captures a relatively better tasks ordering for scheduling in that a more important node can be scheduled at an earlier time. Indeed, a closer look at the scheduling traces of the BU algorithm reveals that its inferior performance is primarily due to its strategy of evenly distributing the tasks to the processors. On the other hand, the MH and DLS algorithms perform relatively better than BU because they minimize the start times of nodes. The incremental message-scheduling strategy in the BSA algorithm is another major reason for its better performance. In the BU, DLS, and MH algorithms, scheduling of messages relies on the information stored in the infrequently updated routing tables. The inaccuracy of the tables inevitably leads to inefficient utilization of communication links in that some links are contended by several messages while some links are idle. This, in turn, delays the start times of nodes.

A longer schedule length produced by PBSA (using 16 PPEs) compared to its sequential counterpart, BSA, is for two reasons: 1) some inaccuracies are incurred due to the estimation of the start times of RPNs; and 2) the procedure for merging the partial schedules can cause some additional performance degradation. After observing the scheduling traces, we found that the adverse effect of inaccurate estimation is more profound when the task graph contains multiple critical paths. However, for the random graphs with more general structures, the PBSA algorithm yields a better performance.

Table 2 provides the scheduling times (in seconds) for these serial algorithms on a single node of the Paragon (the values were taken as the average across four target

TABLE 2
Average Running Times of the MH, DLS, BU, and PBSA (Using 16 PPEs) Algorithms for Various Task Graphs Across All CCRs and Target Topologies

Graph type	Graph sizes	Algorithm				
		MH	DLS	BU	BSA	PBSA (16PPEs)
Cholesky factorization	19x19 matrix(209 nodes)	6.21	10.40	3.80	8.13	0.26
	24x24 matrix(324 nodes)	81.55	138.88	55.95	110.21	3.49
	29x29 matrix(464 nodes)	109.25	194.55	66.65	149.63	4.64
	34x34 matrix(629 nodes)	295.66	494.07	151.76	374.25	11.37
	39x39 matrix(819 nodes)	793.94	1413.87	570.16	1087.61	32.95
	44x44 matrix(1034 nodes)	939.02	1556.77	660.11	1235.58	37.21
	49x49 matrix(1274 nodes)	1650.72	2744.10	1012.97	2143.84	63.80
	54x54 matrix(1539 nodes)	3853.99	5973.70	2777.03	4817.49	141.69
	59x59 matrix(1829 nodes)	4141.27	7122.93	2951.03	5521.67	157.76
	64x64 matrix(2144 nodes)	5036.51	8224.08	3096.91	6375.31	177.09
Gaussian elimination	19x19 matrix(228 nodes)	7.69	12.88	4.71	10.07	0.32
	24x24 matrix(348 nodes)	101.00	172.01	69.29	136.50	4.37
	29x29 matrix(493 nodes)	135.31	240.97	82.55	185.32	5.79
	34x34 matrix(663 nodes)	366.20	611.93	187.96	463.53	14.26
	39x39 matrix(858 nodes)	983.34	1751.16	706.18	1347.06	40.82
	44x44 matrix(1078 nodes)	1163.04	1928.15	817.59	1530.33	45.01
	49x49 matrix(1323 nodes)	2044.52	3398.73	1254.62	2655.27	76.96
	54x54 matrix(1593 nodes)	4773.39	7398.78	3439.53	5966.74	170.96
	59x59 matrix(1888 nodes)	5129.21	8822.16	3655.02	6838.91	195.39
	64x64 matrix(2208 nodes)	6238.02	10186.00	3835.71	7896.19	219.33
FFT	32x32 matrix(95 nodes)	7.67	12.85	4.70	10.05	0.33
	64x64 matrix(223 nodes)	365.35	610.52	187.53	462.46	14.45
	128x128 matrix(511 nodes)	1160.35	1923.70	815.70	1526.80	43.62
	256x256 matrix(1151 nodes)	4762.37	7381.69	3431.58	5952.96	165.36
	512x512 matrix(2559 nodes)	6223.61	10162.48	3826.85	7877.95	212.91
Random	200 nodes	7.45	12.49	4.56	9.77	0.23
	400 nodes	97.94	166.81	67.19	132.37	3.15
	600 nodes	131.22	233.67	80.06	179.71	4.18
	800 nodes	355.12	593.42	182.28	449.50	10.70
	1000 nodes	953.59	1698.19	684.81	1306.31	29.02
	1200 nodes	1127.85	1869.82	792.86	1484.03	32.98
	1400 nodes	1982.66	3295.89	1216.66	2574.94	57.22
	1600 nodes	4628.98	7174.93	3335.46	5786.22	123.11
	1800 nodes	4974.03	8555.25	3544.44	6632.00	144.17
	2000 nodes	6049.29	9877.84	3719.66	7657.29	159.53

topologies and three CCRs). The running times of the PBSA algorithm using 16 PPEs on the Paragon are also included for comparison. We observe that the running times of the sequential algorithms approach thousands of seconds when the number of nodes is more than 800. The DLS algorithm takes significantly longer time than the other algorithms. For instance, to schedule a 2,000-node random graph, the PBSA algorithm takes only about three minutes but the DLS algorithm takes more than two hours. These results also indicate that MH is about 30 percent faster than DLS, BSA is about 20 percent faster than DLS. The BU algorithm is the fastest among the sequential algorithms. In contrast, the running times of the PBSA algorithm are nearly two orders of magnitude less than those of the sequential algorithms, thereby demonstrating a superlinear speedup. This makes the PBSA algorithm useful for generating schedules for very large task graphs.

To further investigate the effects of parallelization, we applied the PBSA algorithm to the four types of graphs using 2, 4, 8, and 16 processors on the Paragon. As before, we computed the schedule length ratios with respect to the serial BSA algorithm. The results are summarized in Fig. 11. Since the effect of task graph size, graph type, and target topology are found to be insignificant, we show in Fig. 11 the average schedule length ratios across all graph types, graph sizes, and topologies, with each point on the curves representing the average of 140 ratios (four topologies, 35 graphs). The performance degradation percentage of PBSA with respect to BSA ranges from 3 to 22 percent. In most cases, however, the performance degradation is less than 10 percent. As noted earlier, the degradation in PBSA's performance is due to the inaccuracy in start times estimation and subschedules concatenation. Both of these effects aggravate with an increase in the number of processors. However, the amount of degradation is smaller compared to the improvements BSA yielded over MH, DLS, and BU. Indeed, we observe that the average schedule length ratios in Figs. 11a, 11b, and 11c are all less than 1, implying that the overall performance of PBSA is better

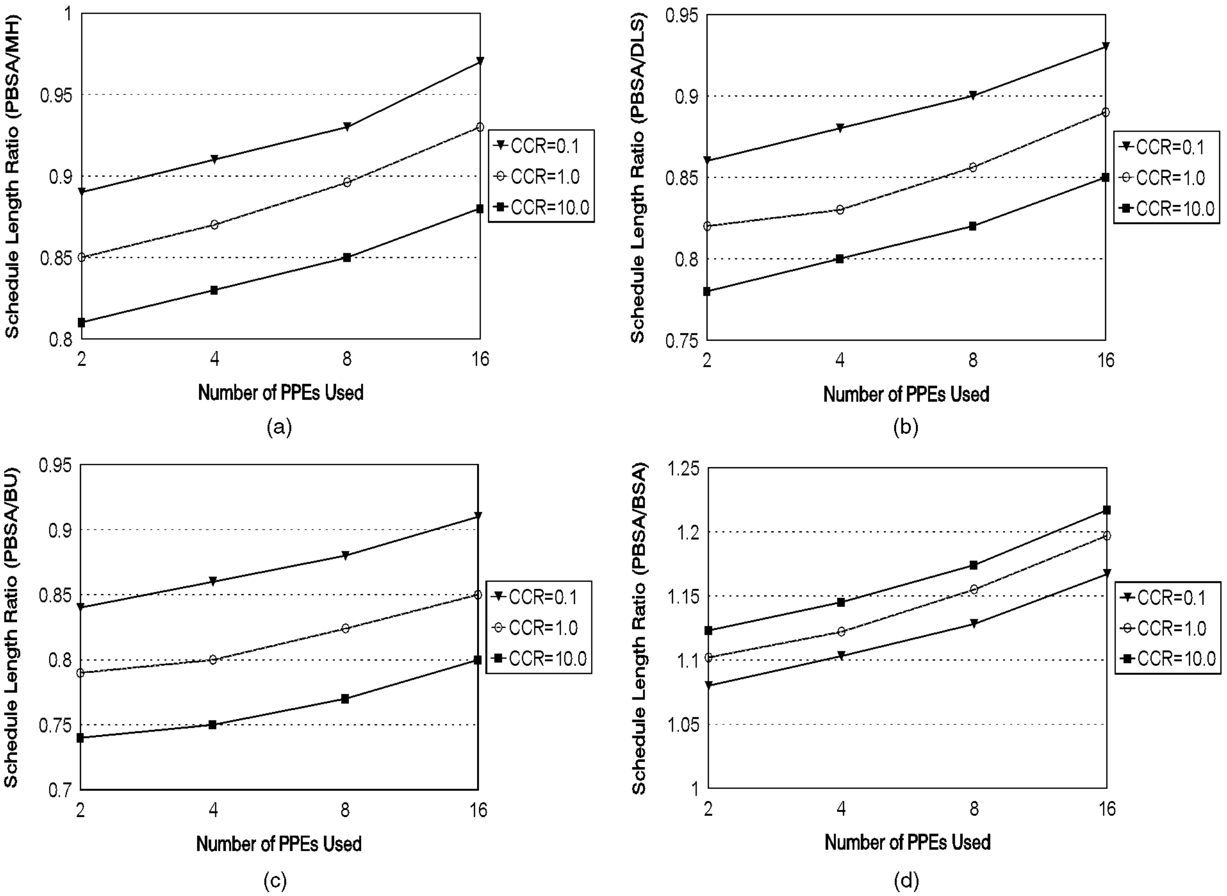


Fig. 11. Average schedule length ratios of the PBSA algorithm to the (a) MH algorithm, (b) DLS algorithm, (c) BU algorithm, (d) BSA algorithm, for all graphs and topologies.

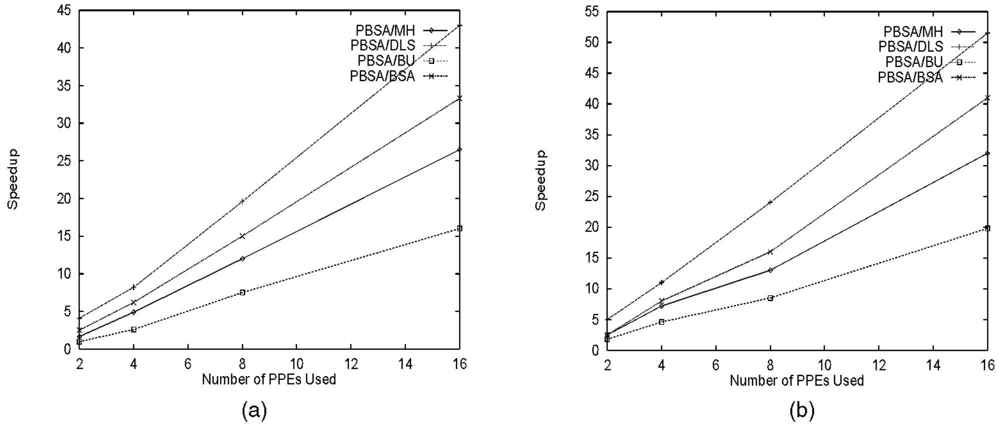
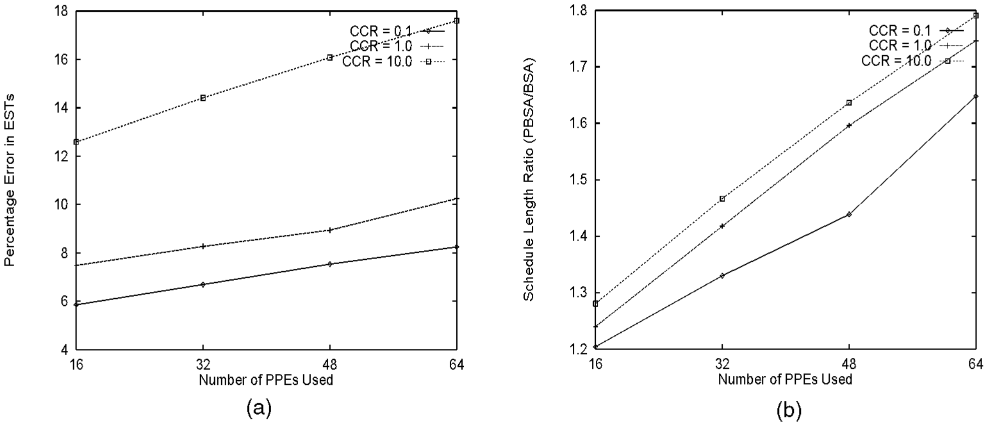


Fig. 12. Average speedups of PBSA with respect to the sequential algorithms for (a) regular graphs (Gaussian elimination, Cholesky factorization, and FFT), and (b) random graphs.

than the other algorithms. An interesting observation is that the average schedule length ratios of PBSA over MH, DLS, and BU are smaller when CCR is larger; in contrast, in Fig. 11d, we observe that the ratios increase with CCRs. This observation implies that compared with the PBSA algorithm, the performance of the MH, DLS, and BU algorithms is more sensitive to the value of CCR.

We also calculated the speedups by dividing the running times of the MH, DLS, BU, and serial BSA algorithms by those of the PBSA algorithm. Fig. 12 shows the speedup of PBSA using 2, 4, 8, and 16 processors with various sizes of task graphs, for the regular task graphs (Cholesky factorization, Gaussian elimination, and FFT) and random graphs. These results indicate that the parallelization strategy used in PBSA has both positive and negative effects. By negative effects we mean potential inaccurate decisions in scheduling which can cause a longer schedule length. On the positive side, the algorithm becomes faster because its workload is reduced. As we have shown in Section 3, the theoretical speedup of PBSA



(a)

(b)

Fig. 13. (a) Average absolute percentage error in the estimated start time of RPNs and (b) average schedule length ratios (PBSA to BSA) for ten 1,000-node random graphs.

with respect to serial BSA is $O(P^2)$, which is superlinear. The plots indicate that the parallel PBSA when run on two processors was about 6 to 10 times faster than the serial BSA. By using more PPEs, the speedup appears to increase almost linearly. With 16 PPEs, the speedup is close to 50. The observed speedup, therefore, agrees with the predicted superlinear speedup.

5.3 Scalability of the PBSA Algorithm

To investigate the scalability of the PBSA algorithm, we used a larger number of PPEs. As the results in Section 5.2 indicated, the accuracy in start times estimation critically affects the quality of the final schedule. And it appears that the accuracy deteriorates with an increasing number of PPEs (since the number of partitions also increases). Thus, we applied the PBSA algorithm to ten 1,000-node random graphs for each CCR using 16, 32, 48, and 64 processors on the Paragon and noted the average absolute percentage error in the ESTs (estimated start times of the RPNs), as well as the schedule length ratios (PBSA to BSA). The absolute percentage error of EST is defined as follows:

$$\frac{|EST - ST|}{ST} \times 100\%.$$

This percentage was measured for each RPN and an average was computed. Fig. 13a indicates that the percentage error in EST is less than 10 percent for small values of CCR even when 64 PPEs are used. However, the percentage error is larger when CCR is equal to 10. The reason being that the range of probable ST for each RPN is larger when the communication costs are larger. Although the percentage error is well below 20 percent in most cases, the schedule lengths degrade quite rapidly when more PPEs are used, as can be seen from Fig. 13b. The degradation occurs during the concatenation process, which in some cases enhances the adverse effect of inaccuracy in start times estimation. However, the schedule lengths are still within a factor of 2 from those generated by the BSA algorithm.

The above results imply that the PBSA algorithm is reasonably scalable in both solution quality and speedup. Furthermore, it exhibits a trade-off between performance and scheduling time, providing the user with a choice between the faster version with some loss in performance and the slower version with better performance.

5.4 Comparison of Different Partitioning Strategies

In this section, we present some experimental results to illustrate the efficacy of the CPN-Dominant sequence partitioning strategy used in the parallelization of the PBSA algorithm. We compared our scheme against two other simple graph-theoretic methods:

- **Level-Based Sequence:** In this method, the task graph is partitioned into a number of horizontal layers (see Fig. 13a) in the following manner. Using the depth-first search, the graph is traversed from entry nodes to exit nodes such that the level number of every entry node is set to be 0. Then, for any node having at least a parent node at level i, its level number is assigned as $i + 1$. Nodes within the same layer are then sorted in a descending order of the number of children. Thus, a sequence of nodes is constructed which can be partitioned according to the number of PPEs available.
- **Random Topological Sequence:** In this method, the nodes in the task graph are first topologically sorted

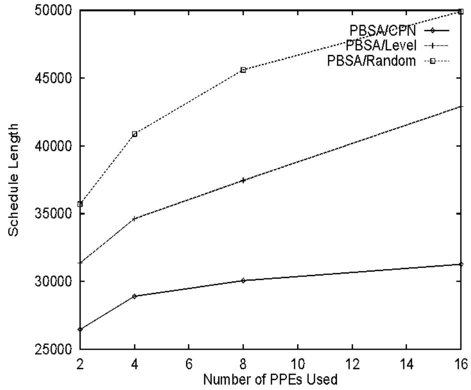


Fig. 14. Peformance of three different task graph partitioning strategies.

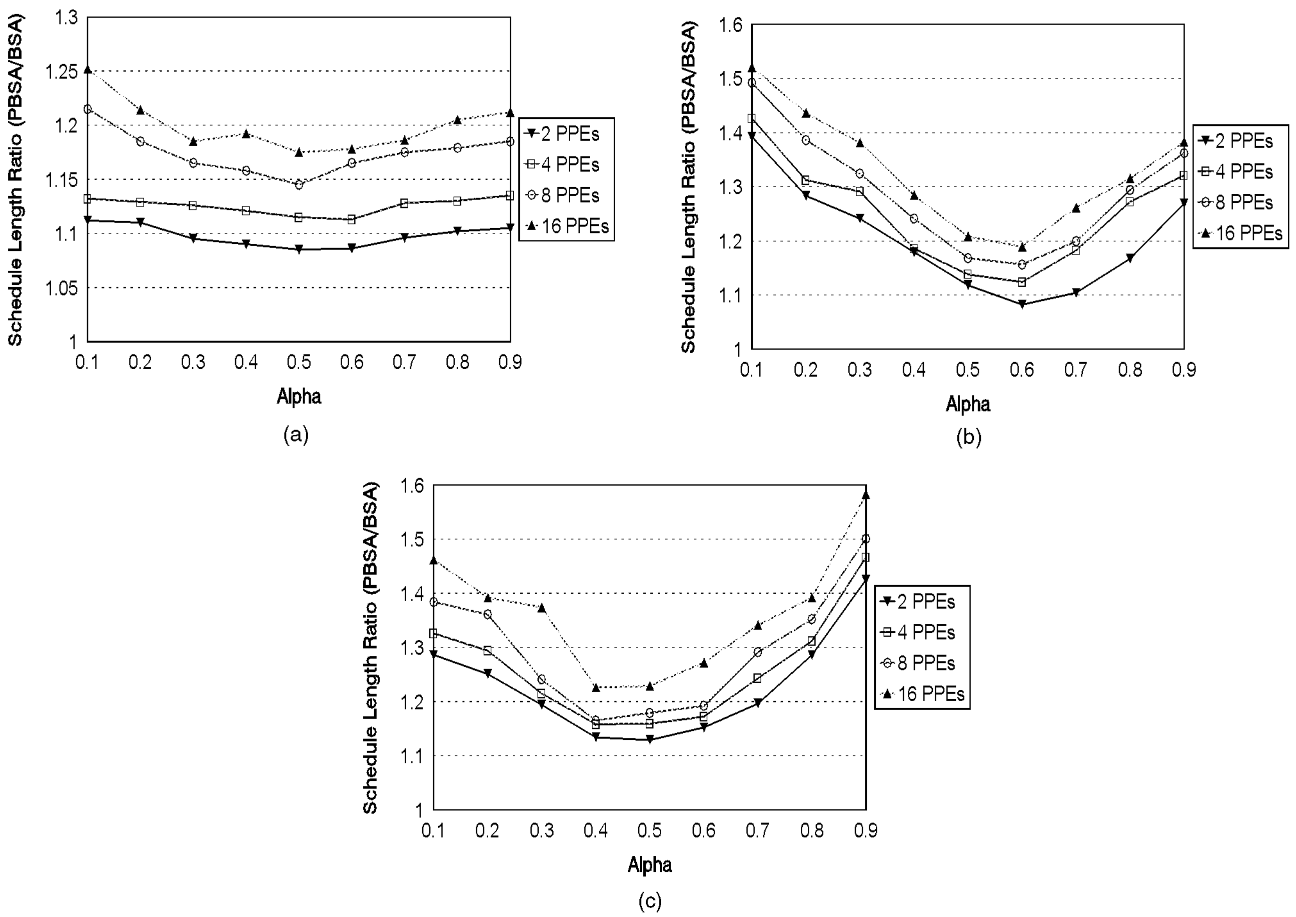


Fig. 15. The effect of parameter α on schedule length. (a) CCR = 0.1, (b) CCR = 1.0, (c) CCR = 10.0.

using a simple depth-first search without regard to the node and edge weights. The resulting topological list of nodes is then randomly perturbed by swapping some randomly selected pairs of independent nodes such that the precedence constraints are still preserved. For example, for the task graph shown in Fig. 11, suppose the initial topological list is $\{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9\}$, then n_2 and n_3 can be swapped. Again, a list of nodes is obtained which can be partitioned according to the number of PPEs given.

The level-based strategy is a variant of horizontal partitioning of a DAG. The random topological strategy is, however, an arbitrary combination of horizontal and vertical partitioning. Note that a purely vertical partitioning strategy cannot be used with the subschedules concatenation scheme.

To compare the three different partitioning strategies, we modified the *Serial_Injection* process in the *PBSA_Master* procedure to implement three versions of the PBSA algorithm: 1) the original PBSA, 2) PBSA/Level (PBSA with level-based partitioning), and 3) PBSA/Random (PBSA with random topological partitioning). We then applied these versions to ten 1,000-node random graphs and compared their average schedule lengths. Fig. 14 includes the results of these experiments. It is apparent from the results that the CPN-Dominant sequence partitioning is indeed considerably more effective than the other two graph-theoretic methods.

5.5 Effects of the Start-times Estimation Technique

To study the effect of the *importance factor* α on the performance of the PBSA algorithm (α was set to be 0.5 in all previous results), we varied α from 0.1 to 0.9 with increments of 0.1, and for each value of α, we ran the PBSA algorithm on ten 1,000-node random graphs with three values of CCR: 0.1, 1.0, and 10.0. Fig. 15 shows three plots of the ratios of the average schedule lengths (for ten graphs) generated by PBSA to those of BSA against α. We observe that when CCR is small (0.1), α does not have a noticeable effect. This is because a smaller value of CCR incurs less communication and, thus, the values of EPST and LPST do not differ too much. When CCR is moderate (1.0), the curves tend to be convex and the smaller the value of α, the worse the performance of PBSA. Optimal values lie somewhere between 0.4 and 0.6. The main reason is that small makes the estimations bias towards LPST, which is not accurate for moderate CCR. However, when the value of CCR is large (10.0), the curves become more convex. That is, the performance is highly sensitive to extreme values of α. In this case, a small value of α is better as it makes the estimations bias towards LPST, which is more accurate for large CCR (since communication weights are large, nodes tend to be scheduled late). Based on these results, setting α to be 0.5 is a reasonable compromise for handling general task graphs.

6 Conclusions and Future Work

Parallelization of a multiprocessor scheduling algorithm is a natural approach to reducing the time complexity of the algorithm. In this paper, we have presented a novel parallel algorithm which can provide a scalable schedule and can be useful for scheduling large task graphs which are virtually impossible to schedule using sequential algorithms. The proposed algorithm outperforms three well-known algorithms reported in the literature, while requiring significantly shorter running times. There are a number of avenues for extending this research, though. While the PBSA algorithm yields a considerable speedup over the sequential BSA algorithm, some performance degradation is observed with the former. This is primarily due to the inaccuracy of estimating the start times of the parent nodes which do not belong to the current partition. New heuristics are needed for improving the accuracy of estimation.

Acknowledgments

We would like to thank the referees for their constructive comments which have greatly improved the overall quality of this paper. Thanks are particularly extended to referee B, whose insightful suggestions helped in a better presentation of the experimental results. This research was supported by the Hong Kong Research Grants Council under contract numbers HKUST734/96E and HKUST6076/97E. A preliminary version of portions of this paper was presented at the 1995 International Parallel Processing Symposium.

References

[1] T.L. Adam, K.M. Chandy, and J. Dickson, "A Comparison of List Scheduling for Parallel Processing Systems," *Comm. ACM,* vol. 17, pp. 685-690, Dec. 1974.

[2] I. Ahmad and Y.-K. Kwok, "On Exploiting Task Duplication in Parallel Program Scheduling," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 9, pp. 872-892, Sept. 1998.

[3] H.H. Ali and H. El-Rewini, "The Time Complexity of Scheduling Interval Orders with Communication is Polynomial," *Parallel Processing Letters,* vol. 3, no. 1, pp. 53-58, 1993.

[4] J. Baxter and J.H. Patel, "The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm," *Proc. 1989 Int'l Conf. Parallel Processing,* vol. II, pp. 217-222, Aug. 1989.

[5] T.L. Casavant and J.G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. Software Eng.,* vol. 14, no. 2, pp. 141-154, Feb. 1988.

[6] Y.C. Chung and S. Ranka, "Application and Performance Analysis of A Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed-Memory Multiprocessors," *Proc. Supercomputing '92,* pp. 512-521, Nov. 1992.

[7] H. Chen, B. Shirazi, and J. Marquis, "Performance Evaluation of A Novel Scheduling Method: Linear Clustering with Task Duplication," *Proc. Int'l Conf. Parallel and Distributed Systems,* pp. 270-275, Dec. 1993.

[8] J.Y. Colin and P. Chretienne, "C.P.M. Scheduling with Small Computation Delays and Task Duplication," *Operations Research,* pp. 680-684, 1991.

[9] I. De Falco, R. Del Balio, and E. Tarantino, "An Analysis of Parallel Heuristics for Task Allocation in Multicomputers," *Computing: Archiv für Informatik und Numerik,* vol. 59, no. 3 pp. 259-275, 1997.

[10] H. El-Rewini and T.G. Lewis, "Scheduling Parallel Programs onto Arbitrary Target Machines," *J. Parallel and Distributed Computing,* vol. 9, no. 2, pp. 138-153, June 1990.

[11] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company 1979.

[12] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM J. Computing,* vol. 18, no. 2, pp. 244-257, Apr. 1989.

[13] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *Proceedings of the 1988 International Conference on Parallel Processing,* vol. II, pp. 1-8, Aug. 1988.

[14] D. Kim and B.G. Yi, "A Two-Pass Scheduling Algorithm for Parallel Programs," *Parallel Computing,* vol. 20, pp. 869-885, 1994.

[15] B. Kruatrachue and T.G. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software,* pp. 23-32, Jan. 1988.

[16] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs onto Multiprocessors" *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 5, pp. 506-521, May 1996.

[17] Y.-K. Kwok and I. Ahmad, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using A Parallel Genetic Algorithm," *J. Parallel and Distributed Computing,* vol. 47, no. 1, pp. 58-77, Nov. 1997.

[18] B. Lee, A.R. Hurson, and T.Y. Feng, "A Vertically Layered Allocation Scheme for Data Flow Systems," *J. Parallel and Distributed Computing,* vol. 11, pp. 175-187, 1991.

[19] C. McCreary and H. Gill, "Automatic Determination of Grain Size of Efficient Parallel Processing," *Comm. ACM,* vol. 32, no. 9 pp. 1,073-1,078, Sept. 1989.

[20] N. Mehdiratta and K. Ghose, "A Bottom-Up Approach to Task Scheduling on Distributed Memory Multiprocessor," *Proc. 1994 Int'l Conf. Parallel Processing,* vol. II, pp. 151-154, Aug. 1994.

[21] M.A. Palis, J.-C. Lieu, and D.S.L. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 1, pp. 46-55, Jan. 1996.

[22] C.H. Papadimitriou and M. Yannakakis, "Towards An Architecture-Independent Analysis of Parallel Algorithms," *SIAM J. Computing,* vol. 19, no. 2, pp. 322-328, Apr. 1990.

[23] I. Pramanick and J.G. Kuhl, "An Inherently Parallel Method for Heuristic Problem-Solving: Part I—General Framework," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 10, pp. 1,006-1,015, Oct. 1995.

[24] C.V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System," "IEEE Trans. Computers," vol. 21, pp. 137-146, Feb. 1972.

[25] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Cambridge, Mass.: MIT Press, 1989.

[26] B. Shirazi, M. Wang, and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Scheduling," *J. Parallel and Distributed Computing,* no. 10, pp. 222-232, 1990.

[27] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 2, pp. 75-87, Feb. 1993.

[28] M.-Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Transactions on Parallel and Distributed Systems,* vol. 1, no. 3, pp. 330-343, July 1990.

[29] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on An Unbounded Number of Processors," *IEEE Transactions on Parallel and Distributed Systems,* vol. 5, no. 9, pp. 951-967, Sept. 1994.

Ishfaq Ahmad received a BSc degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan, in 1985. He received his MS degree in computer engineering and PhD degree in computer science, both from Syracuse University, in 1987 and 1992, respectively. Currently, he is an associate professor in the Department of Computer Science at the Hong Kong University of Science and Technology, where he also directs the Video Technology Centre. His research interests are in the areas of parallel programming tools, scheduling and mapping algorithms for scalable architectures, video technology, MPEG-3 and MPEG-4 compression algorithms, and interactive multimedia systems. He has authored or co-authored more than 80 papers in the above areas. He has received numerous research and teaching awards, including the Best Student Paper Award at Supercomputing '90 and Supercomputing '91, and the Teaching Excellence Award by the School of Engineering at HKUST. Dr. Ahmad has served on the committees of various international conferences, has been a guest editor for two special issues of *Concurency*, "Practice and Experience" related to resource management, and is co-guest-editing a forthcoming special issue of the *Journal of Parallel and Distributed Computing* on the topic of software support for distributed computing. He also serves on the editorial board of *Cluster Computing*. He is a member of the IEEE Computer Society.

Yu-Kwong Kwok received his BSc degree in computer engineering from the University of Hong Kong in 1991, the MPhil and PhD degrees in computer science from the Hong Kong University of Science and Technology in 1994 and 1997, respectively. Currently, he is an assistant professor in the Department of Electrical and Electronic Engineering at the Univeristy of Hong Kong. Before joining the University of Hong Kong, he was a visiting scholar for one year in the parallel processing laboratory at the School of Electrical and Computer Engineering at Purdue University. His research interests include software support for parallel and distributed processing, heterogeneous computing, and multimedia systems. He is a member of the IEEE, the IEEE Computer Society, and the ACM.