

On Parallelization of Static Scheduling Algorithms

Min-You Wu, *Senior Member, IEEE* and Wei Shu, *Member, IEEE*

Abstract—Most static algorithms that schedule parallel programs represented by macro dataflow graphs are sequential. This paper discusses the essential issues pertaining to parallelization of static scheduling and presents two efficient parallel scheduling algorithms. The proposed algorithms have been implemented on an Intel Paragon machine and their performances have been evaluated. These algorithms produce high-quality scheduling and are much faster than existing sequential and parallel algorithms.

Index Terms—Static scheduling, parallel scheduling algorithm, macro dataflow graph, modified critical-path algorithm.



1 INTRODUCTION

STATIC scheduling utilizes the knowledge of problem characteristics to reach a global optimal or near optimal solution. Although many people have conducted their research in various manners, they all share a similar underlying idea: take a directed acyclic graph representing the parallel program as input and schedule it onto processors of a target machine to reduce the completion time. This is an NP-complete problem in its general form [1]. Due to the nature of the problem, many heuristic algorithms that produce satisfactory performance have been proposed [2], [3], [4], [5], [6], [7], [8].

Although these scheduling algorithms apply to parallel programs, the algorithms themselves are sequential and are executed on a single processor system. Scalability of static scheduling is restricted since a large memory space is required to store the macro dataflow graph. A natural solution to this problem is *to use multiprocessors to schedule tasks to multiprocessors*. It can be noted that without parallelizing the scheduling algorithm and running it on a parallel computer, a scalable scheduler is not feasible.

A parallel scheduling algorithm should have the following features:

- **High quality:** it is able to reduce the completion time of a parallel program.
- **Low complexity:** it is able to reduce the time for scheduling a parallel program.

These two requirements contradict each other in general. Many high-quality scheduling algorithms are of a high complexity. The Modified Critical-Path (MCP) algorithm was introduced to offer a good quality with relatively low complexity [3]. In this paper, we propose two parallelized versions of MCP. We will describe the MCP algorithm in the next section. We will also discuss different approaches

of parallel scheduling, as well as existing parallel algorithms in Section 3. In Sections 4 and 5, we will present the VPMCP and HPMCP algorithms, respectively. A performance study will be presented in Section 6.

2 BACKGROUND

2.1 The Macro Dataflow Graph and the MCP Algorithm

A macro dataflow graph consists of a set of nodes $\{n_1, n_2, \dots, n_n\}$ connected by a set of edges, each of which is denoted by e_{ij} . Each node represents a task, and the weight of node n_i , $w(n_i)$, is the execution time of the task. Each edge represents a message transferred from one node to another node and the weight of edge e_{ij} , $w(e_{ij})$, is equal to the transmission time of the message. The communication-to-computation ratio (CCR) of a parallel program is defined as its average communication cost divided by its average computation cost on a given system. In a macro dataflow graph, a node which does not have any parent is called an entry node whereas a node which does not have any child is called an exit node. A node cannot start execution before it gathers all of the messages from its parent nodes. In static scheduling the number of nodes, the number of edges, the node weight, and the edge weight are assumed to be known before program execution. The weight among two nodes assigned to the same processing element (PE) is assumed to be zero. Fig. 1 shows a macro dataflow graph for Gaussian elimination with partial pivoting. This graph is generated from a program which partitions a given matrix by columns [3].

The objective in static scheduling is to assign nodes of a macro dataflow graph to PEs such that the schedule length or makespan is minimized without violating the precedence constraints. There are many approaches to static scheduling. In the classical approach [9], which is also called list scheduling, the basic idea is to make a priority list of nodes, and then assign these nodes one by one to PEs. In the scheduling process, the node with the highest priority is chosen for scheduling. The PE that allows the earliest start time is selected to accommodate this node. Most of the reported scheduling algorithms are based on this concept employing variations in the priority assignment methods such as HLF

• M.-Y. Wu and W. Shu are with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816.
E-mail: {wu, shu}@ece.engr.ucf.edu.

Manuscript received 2 Nov. 1995; revised 4 Mar. 1997.

Recommended for acceptance by D. Eager.

For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 104085.

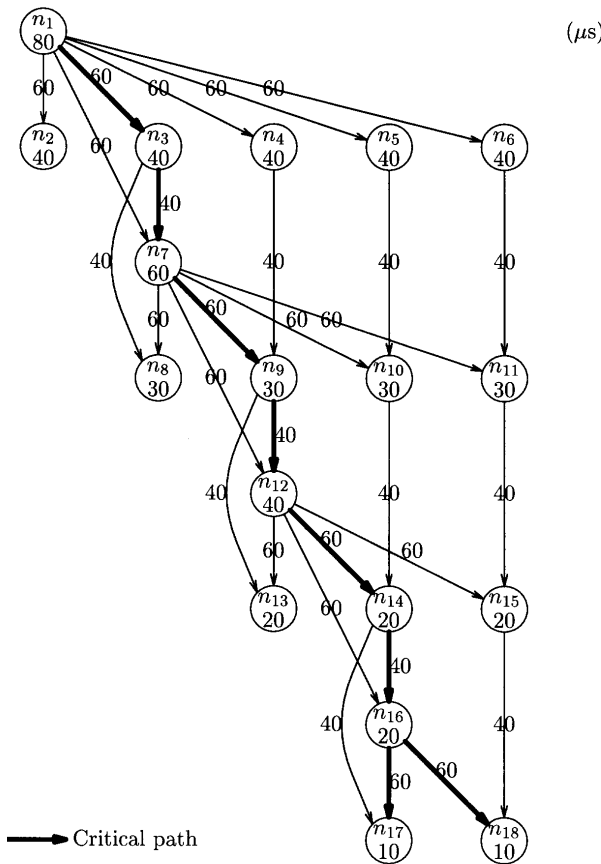


Fig. 1. A macro dataflow graph (Gaussian elimination).

(Highest level First), LP (Longest Path), LPT (Longest Processing Time), and CP (Critical Path) [10], [3], [11].

The Modified Critical-Path (MCP) algorithm that we are going to parallelize is a simple algorithm. This algorithm modifies the original critical path algorithm by incorporating the edge weights. It schedules a macro dataflow graph on a bounded number of PEs. To define this scheduling algorithm succinctly, we will first define the *As-Late-As-Possible (ALAP)* time of a node. The ALAP time is defined as $T_L(n_i) = T_{critical} - level(n_i)$, where $T_{critical}$ is the length of the critical path, and $level(n_i)$ is the length of the longest path from node n_i to an exit node, including node n_i [12]. The path length counts both computation and communication times. This means that the path length is the summation of the node weights and edge weights along the path. The ALAP time can be obtained by the ALAP binding which is created by moving downward through the macro dataflow graph. In an ALAP binding, each node is assigned the latest possible execution time that does not delay the execution of any node on the critical path. The ALAP binding of Fig. 1 is shown in Fig. 2. Shown in Table 1 are ALAP times of nodes which are sorted by increasing order of ALAP times.

The MCP algorithm is shown in Fig. 3. In step 3, when determining the *start_time*, idle time slots created by communication delays are also considered. A node can be inserted to the first feasible idle time slot. This method is called an *insertion algorithm*.

The MCP algorithm has been compared to four other well-known scheduling algorithms under the same as-

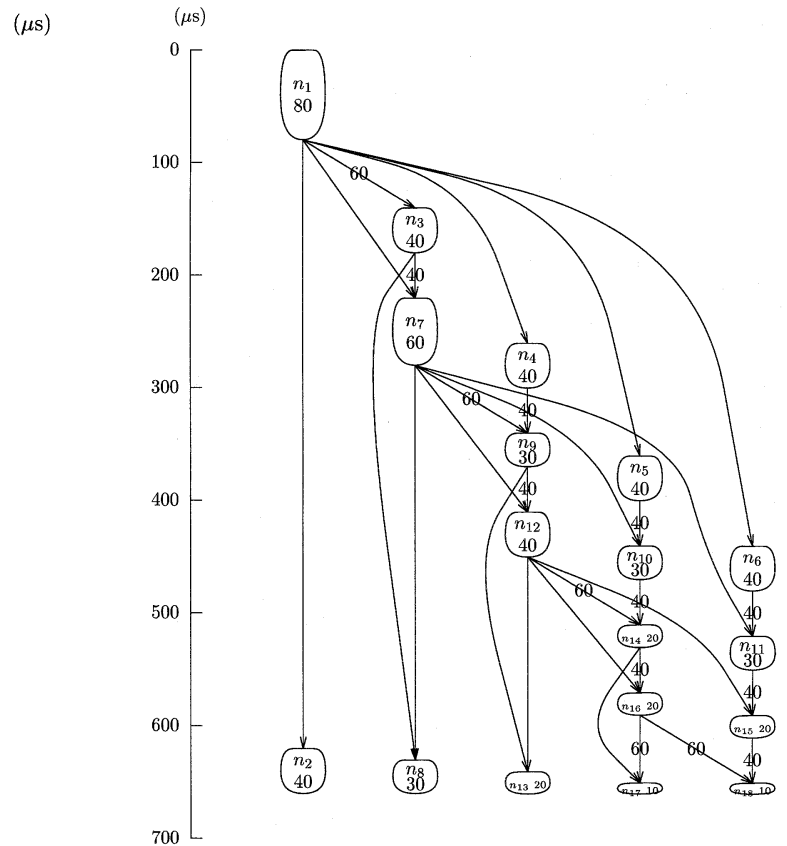


Fig. 2. The ALAP binding of Fig. 1.

sumption: ISH [13], ETF [14], DLS [6], and LAST [15]. It has been shown that MCP performed the best [16]. The complexity of the MCP algorithm is $O(n^2 \log n)$, where n is the number of nodes in a graph. In the second step, the ties can be broken randomly to have a simplified version of MCP. The scheduling quality only varies a little, but the complexity is reduced to $O(n^2)$. In the following, we will use this simplified version of MCP. Fig. 4 shows the schedule of Fig. 1 on three PEs by applying the MCP algorithm.

TABLE 1
THE ALAP TIMES OF FIG. 1 (μsec)

Node	ALAP time
n_1	0
n_3	140
n_7	220
n_4	260
n_9	340
n_5	360
n_{12}	410
n_6	440
n_{10}	440
n_{14}	510
n_{11}	520
n_{16}	570
n_{15}	590
n_2	620
n_8	630
n_{13}	640
n_{17}	650
n_{18}	650

The MCP Algorithm

- 1) Calculate the ALAP time of each node.
- 2) Sort the node list in an increasing ALAP order. Ties are broken by using the smallest ALAP time of the successor nodes, the successors of the successor nodes, etc.
- 3) Schedule the first node in the list to the PE that allows the earliest *start_time*, considering idle time slots. Delete the node from the list and repeat Step 3 until the list is empty.

Fig. 3. The modified critical-path algorithm.

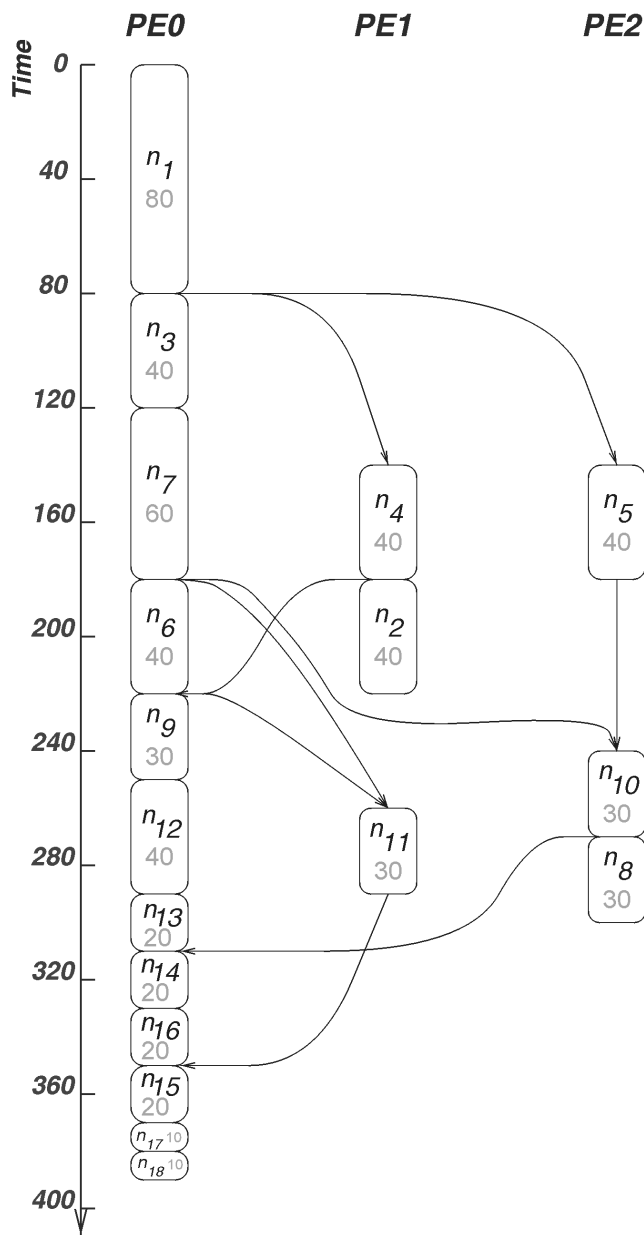


Fig. 4. The schedule of Fig. 1 by the MCP algorithm.

2.2 Test Environment

We implement our algorithms on Intel Paragon. Paragon is a homogeneous distributed memory machine, consisting of an ensemble of computing nodes that communicate across a two-dimensional mesh interconnection network.

The computing node is Intel i860 XP with 32 Mbytes memory and has peak speed of 75 MFlops at 50 MHz. We implemented our parallel scheduling algorithms with C programming language plus message passing. Paragon uses wormhole routing so that the message passing time is almost independent of the distances between communicating nodes. Therefore, the topological mapping is not crucial.

We use the same random graph generator in [17]. The synthetic macro dataflow graphs are randomly generated and used to test the scalability and robustness of the scheduling algorithms. These macro dataflow graphs were synthetically generated in the following manner. Given n , the number of nodes in the macro dataflow graph, the height of the macro dataflow graph was randomly generated from a uniform distribution with mean roughly equal to \sqrt{n} . For each level, a random number of nodes were generated from a uniform distribution with mean roughly equal to \sqrt{n} . Then, the nodes from the higher level to the lower level were connected randomly. The edge weights were also randomly generated. The average node weight is set to be 10. Three values of the communication-computation-ratio (CCR) were selected to be 0.1, 1, and 10. The weights on the nodes and edges were generated randomly so that the average value of CCR corresponded to 0.1, 1, or 10. The sizes of the random graphs were varied from 500 to 4,000. For each size and CCR, 10 different random graphs were generated. Performance data presented in this paper are the average over 10 graphs.

3 APPROACHES TO PARALLELIZATION OF SCHEDULING ALGORITHMS

We further define that the PEs that execute a parallel scheduling algorithm are called the *physical PEs (PPEs)* in order to distinguish them from the *target PEs (TPEs)* to which the macro dataflow graph is to be scheduled. The basic idea behind parallel scheduling algorithms is that instead of identifying one node to be scheduled each time, we identify a set of nodes that can be scheduled in parallel. The quality and speed of a parallel scheduler depend on data partitioning. There are two major data domains in a scheduling algorithm, the source domain and the target domain. The source domain is the macro dataflow graph and the target domain is the schedule for target processors. We consider two parallel scheduling approaches: the vertical and horizontal schemes. They are illustrated in Fig. 5, where the macro dataflow graphs are mapped into the time-space domain. In the *vertical* scheme, each PPE is assigned a set of graph nodes using space domain partitioning. Also, each PPE maintains schedules for one or more TPEs. In the *horizontal* scheme, each PPE is assigned a set of graph nodes using time domain partitioning. The resultant schedule is also partitioned so that each PPE maintains a portion of the schedule of every TPE. Each PPE schedules its own portion of the graph before all PPEs exchange information with each other to determine the final schedule. In Fig. 5, we assume that three PPEs schedule the graph to six TPEs. Thus, in the vertical scheme, each PPE holds schedules of two TPEs. In the horizontal scheme, each PPE holds a portion of schedules of six TPEs.

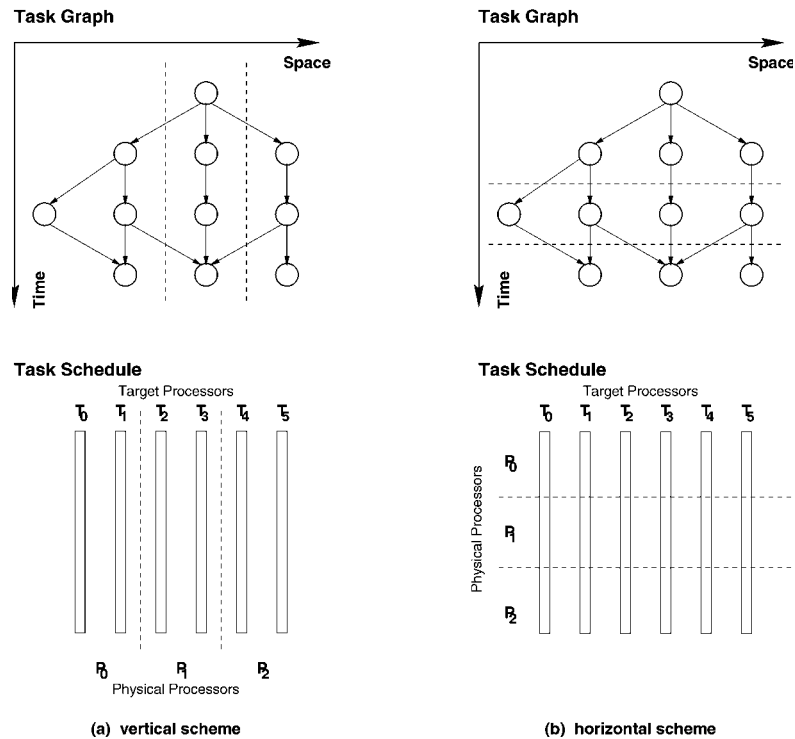


Fig. 5. Vertical and horizontal schemes.

The vertical scheme and the horizontal scheme are outlined in Figs. 6 and 7, respectively. In the vertical scheme, PPEs exchange information and schedule graph nodes to TPEs. Frequent information exchange may result in a large communication overhead. With horizontal partitioning, each PPE can schedule its graph partition without exchanging information with other PPEs. In the last step, PPEs exchange information of their subschedules and concatenate them to obtain a final schedule. The problem with this method is that the start times of all partitions other than the first one are unknown. The time needs to be estimated and scheduling quality depends on the estimation.

-
- 1) Partition the graph into P equal sized sets using space domain partitioning.
 - 2) Every PPE cooperates together to generate a schedule and each PPE maintains schedules for one or more TPEs.
-

Fig. 6. The vertical scheme for parallel scheduling.

-
- 1) Partition the graph into P equal sized sets using time domain partitioning.
 - 2) Each PPE schedules its graph partition to generate a subschedule.
 - 3) PPEs exchange information to concatenate subschedules.
-

Fig. 7. The horizontal scheme for parallel scheduling.

Little work has been done in designing a parallel algorithm for scheduling. The only algorithm in this area is in the horizontal scheme, which is the parallel BSA (PBSA) algorithm [17]. The BSA algorithm takes into account link con-

tention and communication routing strategy. A PE list is constructed in a breadth-first order from the PE having with the highest degree (pivot PE). This algorithm constructs a schedule incrementally by first injecting all the nodes to the pivot PE. At this point, it tries to improve the start time of each node by migrating it to the adjacent PEs of the pivot PE only if the migration will improve the start time of the node. After a node is migrated to another PE, its successors are also moved with it. Then, the next PE in the PE list is selected to be the new pivot PE. This process is repeated until all the PEs in the PE list have been considered. The complexity of the BSA algorithm is $O(p^2en)$, where p is the number of TPEs, n the number of nodes, and e the number of edges in a graph.

The PBSA algorithm parallelizes the BSA algorithm in the horizontal scheme. The nodes in the graph are sorted in a topological order and partitioned into P equal sized blocks. Each partition of the graph is then scheduled to the target system independently. The PBSA algorithm resolves the dependencies between the nodes of partitions by calculating an *estimated start time* of each parent node belonging to another partition, called the *remote parent node (RPN)*. This time is estimated to be between the earliest possible start time and the latest possible start time. After all the partitions are scheduled, the independently developed schedules are concatenated. The complexity of the PBSA algorithm is $O(p^2en/P^2)$, where P is the number of PPEs.

4 THE VPMCP ALGORITHM

MCP is a list scheduling algorithm. In a list scheduling, nodes are ordered in a list according to priority. A node at the front of the list is always scheduled first. Scheduling a node depends on the nodes that were scheduled before this

- 1) (a) Compute the ALAP time of each node and sort the node list in an increasing ALAP order. Ties are broken randomly.
- (b) Divide the node list with cyclic partitioning into equal sized partitions, and each partition is assigned to a PPE.
- 2) (a) The PPE that has the first node in the list broadcasts the node, along with its parent information, to all PPEs.
- (b) Each PPE calculates a *start_time* for the node on each of its TPEs. The earliest *start_time* is obtained by parallel reduction of minimum.
- (c) The node is scheduled to the TPE that allows the earliest start time.
- (d) The parent information of children of the scheduled node is updated. Delete the node from the list and repeat this step until the list is empty.

Fig. 8. The VPMCP1 algorithm.

node. Therefore, it is basically a sequential algorithm. Its heavy dependences make parallelization of MCP very difficult. In MCP, nodes must be scheduled one by one. However, when scheduling a node, the start times of the node on different TPEs can be calculated simultaneously. This parallelism can be exploited in the vertical scheme.

If many nodes are scheduled simultaneously, the resultant schedule may not be the same as the one produced by MCP. In the vertical scheme, we may schedule many nodes at the same time. This way, parallelism can be increased and overhead reduced. In the horizontal scheme, different partitions must be scheduled simultaneously for a parallel execution. The schedule length will vary and, in general, will be longer than that produced by the sequential MCP. Exploiting more parallelism may lower scheduling quality. We will study the degree of quality degradation when increasing parallelism.

We call the vertical version of parallel MCP the VPMCP algorithm and the horizontal version the HPMCP algorithm. The VPMCP algorithm is described in this section and the HPMCP algorithm will be presented in the next section.

Before describing the VPMCP algorithm, we present a simple parallel version of the MCP algorithm. This version schedules one node each time so that it produces the same schedule as the sequential MCP algorithm. Each PPE maintains schedules for one or more TPEs. Therefore, it is a vertical scheme. We call this algorithm VPMCP1, which is shown in Fig. 8. The nodes are first sorted by the ALAP time and cyclicly divided into P partitions. The nodes in places $i, i + P, i + 2P, \dots$ of the sorted list are assigned to PPE i . The nodes are scheduled one by one. Each node is broadcast to all PPEs along with its parent information including the scheduled TPE number and time. Then, the start times of each node on different TPEs can be calculated in parallel. The node is scheduled to the TPE that allows the earliest start time. Consequently, if a PPE has any node that is a child of the newly scheduled node, the corresponding parent information of the node is updated.

The VPMCP1 algorithm parallelizes the MCP algorithm directly. It produces exactly the same schedules as MCP. However, since each time only one node is scheduled, the degree of parallelism is limited and granularity is too fine. To solve this problem, a number of nodes could be scheduled simultaneously to increase granularity and to reduce communication. When some nodes are scheduled simultaneously, they may

- 1) (a) Compute the ALAP time of each node and sort the node list in an increasing ALAP order. Ties are broken randomly.
- (b) Divide the node list with cyclic partitioning into equal sized partitions and each partition is assigned to a PPE. Initialize an available list. A node is available if all of its parent nodes, if any, have been scheduled, and all of the nodes that have a higher priority have been scheduled or are in the available list. Sort the list in an increasing ALAP order.
- 2) (a) The first p nodes in the available list are broadcast to all PPEs by using the parallel concatenation operation, along with their parent information. If there are less than p nodes in the available list, broadcast the entire available list.
- (b) Each PPE calculates a *start_time* for the node on each of its TPEs. These *start_times* are made available to every PPE by parallel concatenation.
- (c) A node is scheduled to the TPE that allows the earliest start time. If more than one node competes for the same time slot in a TPE, the node with smaller ALAP time gets the time slot. The node that does not get the time slot is then scheduled to the time slot that allows the second earliest *start_time*, and so on.
- (d) The parent information is updated for the children of the scheduled node. Delete these nodes from the available list and update the available list. Repeat this step until the available list is empty.

Fig. 9. The VPMCP algorithm.

conflict with each other. Conflict may result in degradation of scheduling quality. The condition that allows some nodes to be scheduled in parallel without changing scheduling quality can be stated as follows. In the MCP scheduling sequence, a node obtains its earliest *start_time* after all of its former nodes in the list have been scheduled. When a set of nodes are scheduled in parallel, each node obtains its earliest *start_time* independently. A node may obtain its earliest *start_time* which is earlier than the one in MCP scheduling when some of its former nodes in the set have not been scheduled. In this case, it must conflict with one of its former nodes. Therefore, if a node is scheduled a place that does not conflict with its former nodes, it obtains the same earliest *start_time* and is scheduled to the same place as it would be in the MCP scheduling sequence. A set of nodes can obtain their earliest *start_time* simultaneously and be scheduled accordingly. When a node conflicts with its former nodes, then this node and the rest of the nodes will not be scheduled before they obtain their new earliest *start_times*. In this way, more than one node can be scheduled each time. However, many nodes may have the same earliest *start_time* in the same TPE. It can be noted that there could be many conflicts. In most cases, only one or two nodes can be scheduled each time.

To increase the number of nodes to be scheduled in parallel, we may allow a conflict node to be scheduled to its suboptimal place. Therefore, when a node is found to be in conflict with its former nodes, it will be scheduled to the next nonconflict place. With this strategy, p nodes can be scheduled each time, where p is the number of TPEs. We use this strategy in our vertical version of parallel MCP, which is called the VPMCP algorithm. The details of this algorithm are shown in Fig. 9. An available list is constructed and sorted by ALAP times. A set of nodes selected

from the available list are broadcast to all PPEs. The *start_time* of each node is calculated independently. Then, the *start_times* are made available to every PPE by another parallel concatenation. Some nodes may compete for the same time slot in a TPE. This conflict is resolved by the smallest-ALAP-time-first rule. A node that does not get the best time slot will try its second best place, and so on, until it is scheduled to a nonconflict place.

The time for calculation of the ALAP time and sorting is $O(e + n \log n)$. The parallel scheduling step is of $O(n^2/P)$. It can be seen that the complexity of the VPMCP algorithm is $O(e + n \log n + n^2/P)$, where n is the number of nodes, e the number of edges, and P the number of PPEs. The number of communications is $O(2n)$ for VPMCP1 and $O(2n/p)$ for VPMCP, where p is the number of TPES.

Fig. 10 shows a running example of the VPMCP algorithm which schedules Fig. 1 to three TPES. In the beginning, only node n_1 is available. After it is scheduled to PE 0, nodes $n_2, n_3, n_4, n_5,$ and n_6 are ready. However, nodes $n_2, n_4, n_5,$ and n_6 have lower priorities than nonready node n_7 . Thus, only node n_3 is in the available list and scheduled. At this time, nodes $n_2, n_4, n_5, n_6,$ and n_7 are ready, but only nodes n_7 and n_4 are in the available list because n_2, n_5 and n_6 have lower priorities than nonready node n_9 . Nodes n_7 and n_4 have the earliest start time 120 at PE 0 and conflict with each other. Node n_7 has a higher priority than node n_4 , so that it takes the place and node n_4 must be scheduled to PE 1. The partial schedule is shown in Fig. 10a. Now nodes n_9 and n_5 can be scheduled. Node n_9 is scheduled to PE 0 and node n_5 to PE 2 that allows

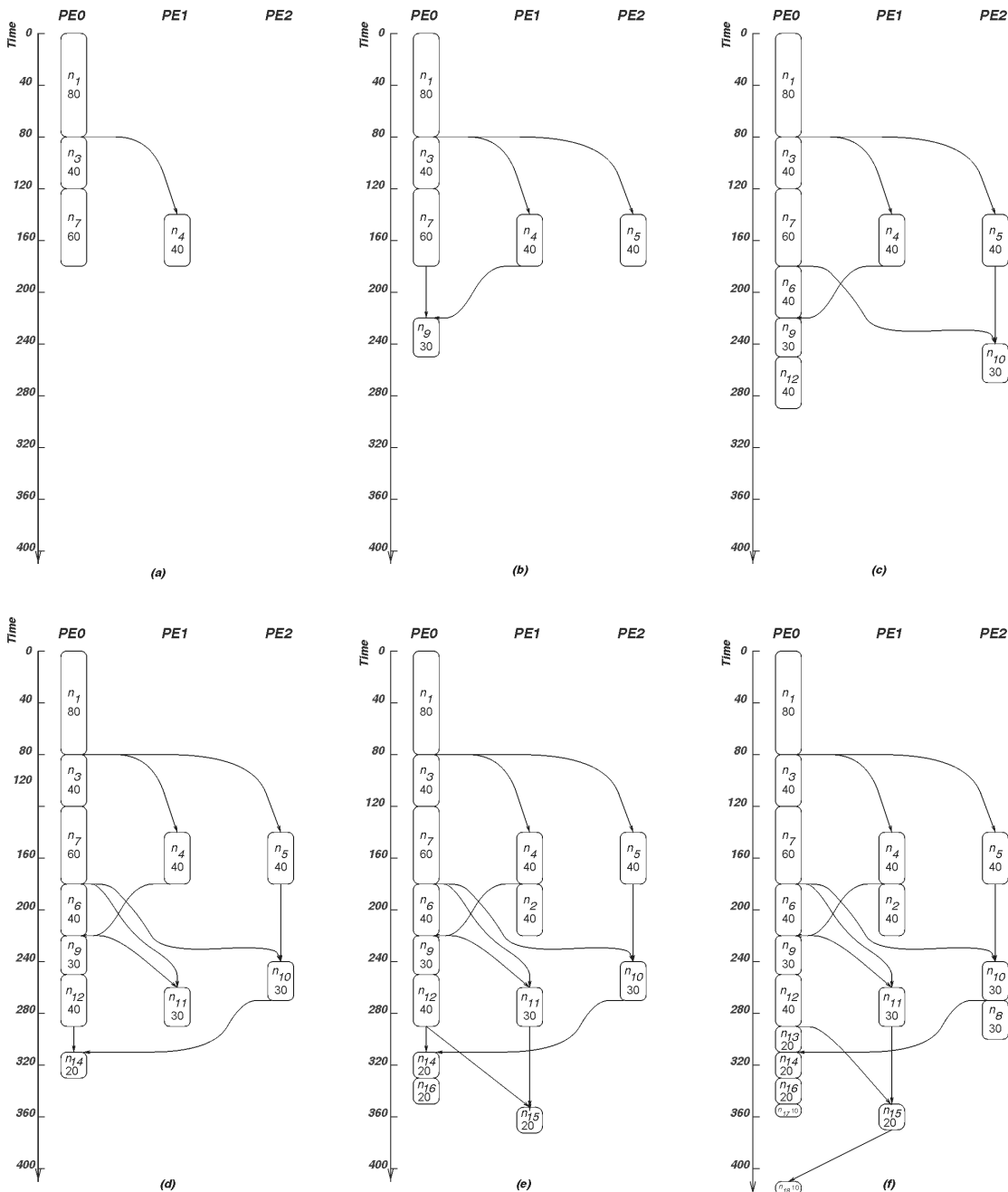


Fig. 10. Running example for the VPMCP algorithm.

the earliest start time for the two nodes, respectively. The partial schedule is shown in Fig. 10b. Next, nodes n_{12} , n_6 , and n_{10} can be scheduled. Nodes n_{12} and n_6 are scheduled to PE 0 but in different time slots. Node n_{10} is scheduled to PE 2. The partial schedule is shown in Fig. 10c. Then, nodes n_{14} and n_{11} are scheduled to PEs 0 and 1, respectively. The partial schedule is shown in Fig. 10d. At this point, nodes n_{16} , n_{15} , n_2 , n_8 , and n_{13} can be scheduled but only the first three nodes are scheduled since we only have three PEs. Node n_{15} conflicts with node n_{16} on PE 0 and is scheduled to PE 1. Fig. 10e shows the partial schedule. Then, nodes n_8 , n_{13} , and n_{17} are scheduled. Finally, node n_{18} is scheduled and the final schedule is shown in Fig. 10f.

The VPMCP algorithm is compared to VPMCP1 in Table 2. Here, the size of graphs is 4,000 nodes. Both the number of TPEs and the number of PPEs are 4. It can be seen from the table that VPMCP1 produces a better scheduling quality. However, its heavy communication results in longer running time. VPMCP reduces running times and still provides an acceptable scheduling quality. The schedule lengths are between 0.3 percent and 1.1 percent longer than that produced by VPMCP1.

TABLE 2
COMPARISON OF VERTICAL STRATEGIES

CCR	schedule length		running time (sec)	
	VPMCP1	VPMCP	VPMCP1	VPMCP
0.1	9970	10011	8.1	4.95
1	10196	10224	9.76	5.15
10	16272	16455	10.2	5.02

5 THE HPMCP ALGORITHM

In a horizontal scheme, different partitions of a macro dataflow graph are scheduled simultaneously. We call a horizontal version of parallel MCP the HPMCP algorithm, which is outlined in Fig. 11.

- 1) Build a priority node list in an increasing ALAP order and partition the node list to PPEs.
- 2) Each PPE applies the MCP algorithm to its partition to produce a subschedule.
- 3) Concatenate each pair of adjacent subschedules.

Fig. 11. The HPMCP algorithm sketch.

In the HPMCP algorithm, the nodes are first sorted by the ALAP time. Therefore, the node list is in a topological order and then partitioned into P equal sized blocks to be assigned to P PPEs. In this way, the graph is partitioned horizontally.

When the graph is partitioned, each PPE will schedule its partition to produce its subschedule. Then, these subschedules will be concatenated to form the final schedule. Three problems are to be addressed for scheduling and concatenation: information estimation, concatenation permutation, and postinsertion.

5.1 Information Estimation

The major problem in the horizontal scheme is how to resolve the dependences between partitions. In general, a

latter partition depends on its former partitions. To schedule a latter partition, the PPE needs information of its former subschedules. Since such information does not exist before the subschedules of former partitions have been produced, an estimation is used instead. We need to estimate the schedule time of remote parent nodes (RPNs). This information can help a node to determine its earliest start time in the latter partition.

In the PBSA algorithm [17], the start time of each RPN is estimated. The estimated start time of an RPN is defined as $\alpha EPST + (1 - \alpha) LPST$, where EPST is the earliest possible start time and LPST the latest possible start time [17]. The parameter α is equal to 1 if the RPN is on the critical path. Otherwise, it is equal to the length of the longest path from the start point through the RPN to the end point, divided by the length of the critical path. We call this estimation the PBSA estimation.

In HPMCP, we simply ignore all dependences between partitions. If all parents of a node are remote, the node becomes an entry node in its partition. These nodes can start at the same time in its partition, that is the local time 0. This is called the HPMCP estimation.

Now, we compare the two approaches of estimation as shown in Table 3. In this and next two tables, the size of graphs is 4,000 nodes. Both the number of TPEs and the number of PPEs are 4. The column "HPMCP est." shows the performance of HPMCP. The column "PBSA est." shows the performance of HPMCP with PBSA estimation. The schedule lengths and running times are compared. The running time of PBSA estimation is longer.

TABLE 3
COMPARISON OF ESTIMATION ALGORITHMS

CCR	schedule length		running time (sec)	
	HPMCP est.	PBSA est.	HPMCP est.	PBSA est.
0.1	9992	10158	2.67	4.01
1	10223	10549	2.97	4.40
10	16518	16924	2.72	4.33

The schedule length produced by PBSA estimation is always longer than that produced by HPMCP estimation. It implies that a more complex estimation algorithm cannot promise good scheduling and a simpler algorithm may be better. However, that is not to say that we should use the simplest one in the future. It is still possible to find a good estimation to improve performance. The simple estimation used in HPMCP sets a baseline for future estimation algorithms.

5.2 Concatenation Permutation

After each PPE produces its subschedule, the final schedule is constructed by concatenating these subschedules. Because there is no accurate information of the former subschedules, it is not easy to determine the optimal permutation of TPEs between adjacent subschedules. This means we have to determine which latter TPE should be concatenated to which former TPE. A heuristic is necessary. In the PBSA algorithm, a TPE with the earliest node is concatenated to the TPE of the former subschedule that allows the earliest execution. Then, other TPEs are concatenated to the TPEs in the former subschedule in a breadth-first order [17]. In the HPMCP algorithm, we assume that the start

time of each node within its partition is the same. Therefore, the above algorithm cannot be applied. We simply do not perform permutation of TPEs in HPMCP. That is, a TPE in the latter subschedule is concatenated to the same TPE in the former subschedule. An alternative heuristic can be described as follows: each PPE finds out within its subschedule which TPE has most critical-path nodes and permutes this TPE with TPE 0. With this permutation, as many critical path nodes as possible are scheduled to the same TPE, and the critical path length could be reduced. This permutation algorithm is compared to the nonpermutation algorithm in Table 4. The time spent on the permutation step causes this algorithm to be slower since extra time is spent on determining whether a node is on the critical path. In terms of the schedule length, the permutation algorithm makes some test cases better than the nonpermutation algorithm. It does not improve performance much. Therefore, no permutation is performed in the HPMCP algorithm.

TABLE 4
COMPARISON OF PERMUTATION ALGORITHMS

CCR	schedule length		running time (sec)	
	No permut.	Permut	No permut.	Permut
0.1	9992	9990	2.67	3.11
1	10223	10217	2.97	3.30
10	16518	16518	2.72	3.19

5.3 Postinsertion

Finally, we walk through the entire concatenated schedule to determine the actual start time of each node. Some refinement can be performed in this step. In a horizontal scheme, the latter PPE is not able to insert nodes to former subschedules due to lack of their information. This leads to some performance loss. It can be partially corrected at the concatenation time by inserting the nodes of a latter subschedule into its former subschedules. Improvement of this postinsertion algorithm is shown in Table 5. Compared to noninsertion, the postinsertion algorithm can reduce the scheduling length. However, it spends much more time for postinsertion. Also, it is a sequential operation and can become a bottleneck of execution. In the following, we do not perform this postinsertion.

TABLE 5
COMPARISON OF POSTINSERTION ALGORITHMS

CCR	schedule length		running time (sec)	
	No insert.	Insert.	No insert.	Insert.
0.1	9992	9989	2.67	3.98
1	10223	10221	2.97	4.80
10	16518	16433	2.72	4.44

Now we present a detailed HPMCP algorithm in Fig. 12. After the graph is partitioned and assigned to PPEs, each PPE schedules its partition using MCP. When applying MCP, we ignore the dependences between partitions so that each partition can be scheduled independently. A node is treated as an entry node in its partition if all of its parent nodes are remote. Nodes are scheduled in the order of their ALAP priorities. Each PPE schedule its partition starting from its local time 0. Then adjacent subschedules are concatenated to form the final schedule without permutation of TPEs in different subschedule and without post-insertion performed. The final schedule is then walked through to

determine the actual start time of each node and the final schedule length. The time for calculation of the ALAP time and sorting is $O(e + n \log n)$. The second step of parallel scheduling is $O(n^2/P^2)$, and the third step spends $O(e)$ time. Therefore, the complexity of the HPMCP algorithm is $O(e + n \log n + n^2/P^2)$, where n is the number of nodes and e the number of edges in a graph, p is the number of TPEs and P the number of PPEs.

Fig. 13 shows a running example of the HPMCP algorithm which schedules Fig. 1 to three TPEs. The graph is partitioned into three partitions, each has six nodes. Nodes $n_1, n_3, n_7, n_4, n_9,$ and n_5 are assigned to PPE0; nodes $n_{12}, n_6, n_{10}, n_{14}, n_{11},$ and n_{16} to PPE1; and nodes $n_{15}, n_2, n_8, n_{13}, n_{17}$ and n_{18} to PPE2. Note that nodes $n_6, n_{10},$ and n_{12} in PPE1 and nodes $n_2, n_8, n_{13}, n_{15},$ and n_{17} in PPE2 become ready nodes when ignoring the dependences between partitions. Each partition of nodes are scheduled independently. The partial schedules are shown in Fig. 13a. They are concatenated to form the final schedule. Corresponding TPEs are concatenated without permutation. Furthermore, no post-insertion performed. Finally, we walk through the entire schedule to determine the actual start time of each node as shown in Fig. 13b.

- 1) Compute the ALAP time of each node and sort the node list in an increasing ALAP order. Ties are broken randomly.
 - Partition the node list into equal sized blocks and each partition is assigned to a PPE.
- 2) Each PPE applies the MCP algorithm to its partition to produce a subschedule, ignoring the edges between a node and its remote parent nodes. Schedule the first node n_i in the list to the TPE that allows the earliest *start_time* $t(n_i)$. Delete n_i from the list and repeat this scheduling step until the list is empty.
- 3) Concatenate each pair of adjacent subschedules. A TPE in the latter subschedule is concatenated to the counterpart TPE in the former subschedule with no postinsertion performed. Walk through the schedule to determine the actual start time of each node.

Fig. 12. The HPMCP algorithm.

6 PERFORMANCE STUDY

We present the performance of our parallel scheduling algorithm with three measures: schedule length, running time, and speedup. First, performance of the VPMCP algorithm is presented. In Tables 6 and 7, graphs of 500, 1,000, 2,000, and 4,000 nodes are scheduled to four TPEs. Note that in the vertical scheme, the number of PPEs cannot be larger than the number of TPEs. The schedule length provides a measure of scheduling quality. The results shown in Table 6 are the schedule lengths and the ratios of the schedule lengths produced by the VPMCP algorithm to that produced by the sequential MCP algorithm. The ratio was obtained by running the VPMCP algorithm on 2 and 4 PPEs on Paragon and taking the ratios of the schedule lengths produced by it to those of sequential MCP running on one PPE. In most cases, the schedule lengths of VPMCP are not more than 1 percent longer than that produced by MCP.

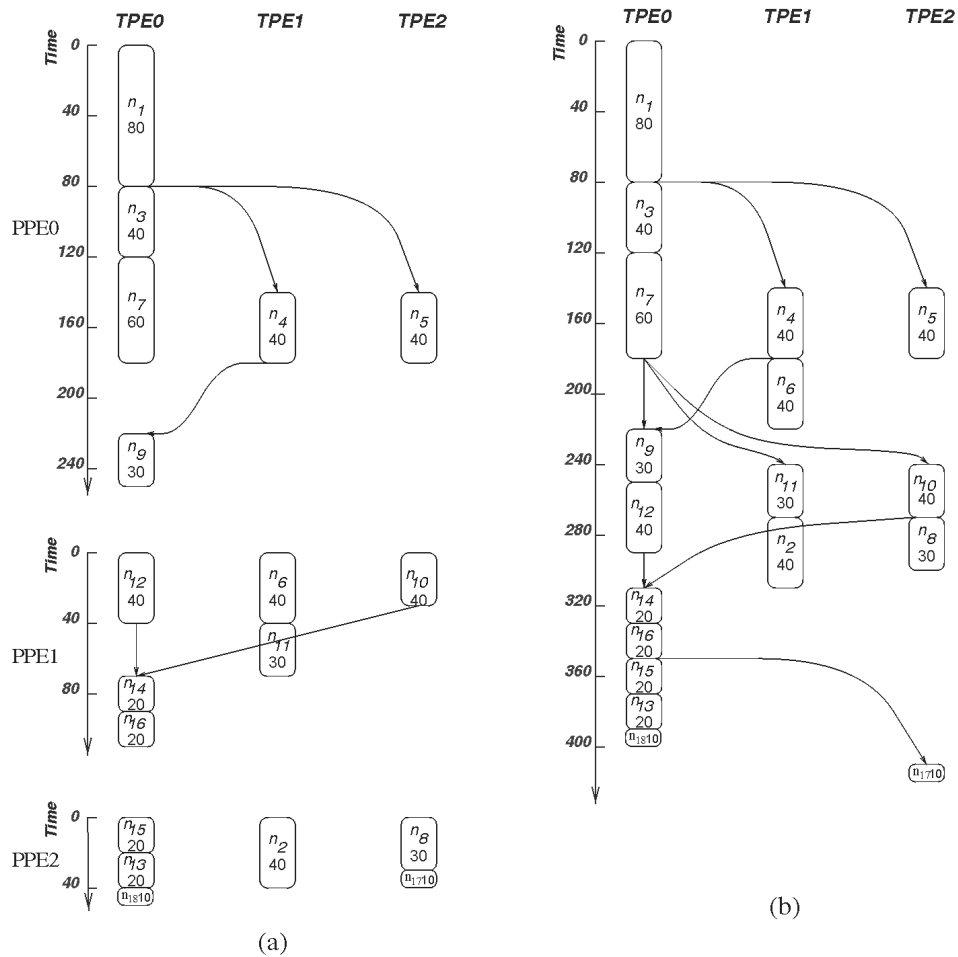


Fig. 13. Running example for the HPMCP algorithm.

TABLE 6
THE SCHEDULE LENGTHS PRODUCED BY VPMCP AND THEIR RATIOS TO THOSE OF MCP

CCR	No. of PPEs	Graph size (number of nodes)							
		500		1000		2000		4000	
		length	ratio	length	ratio	length	ratio	length	ratio
0.1	sequential	1256	-	2520	-	4984	-	9970	-
	2, 4	1261	1.004	2523	1.002	5005	1.004	10011	1.004
1	sequential	1262	-	2584	-	5096	-	10196	-
	2, 4	1272	1.009	2592	1.003	5116	1.004	10224	1.003
10	sequential	1988	-	4004	-	8141	-	16272	-
	2, 4	2027	1.02	4051	1.01	8236	1.01	16455	1.01

TABLE 7
RUNNING TIME (sec) AND SPEEDUP OF VPMCP

CCR	No. of PPEs	Graph size (number of nodes)							
		500		1000		2000		4000	
		time	S	time	S	time	S	time	S
0.1	sequential	0.78	-	1.58	-	4.59	-	11.5	-
	2	0.65	1.20	1.33	1.19	3.40	1.34	6.22	1.85
	4	0.61	1.28	1.22	1.29	2.81	1.63	4.95	2.32
1	sequential	0.71	-	1.02	-	2.90	-	12.1	-
	2	0.74	0.96	1.02	1.00	2.37	1.23	9.17	1.32
	4	0.76	0.93	1.01	1.01	2.14	1.36	5.15	2.35
10	sequential	0.65	-	1.25	-	3.11	-	11.4	-
	2	0.65	1.00	1.09	1.15	2.45	1.27	7.85	1.45
	4	0.64	1.02	1.00	1.25	2.12	1.47	5.02	2.27

TABLE 8
THE SCHEDULE LENGTHS PRODUCED BY HPMCP AND THEIR RATIOS TO THOSE OF MCP

CCR	No. of PEs	Graph size (number of nodes)							
		500		1000		2000		4000	
		length	ratio	length	ratio	length	ratio	length	ratio
0.1	sequential	1259	-	2520	-	4984	-	9970	-
	2	1267	1.006	2530	1.004	4992	1.002	9989	1.002
	4	1268	1.007	2531	1.004	5001	1.003	9992	1.002
	8	1273	1.01	2540	1.008	5007	1.004	10018	1.005
	16	1287	1.02	2551	1.01	5020	1.007	10037	1.007
1	sequential	1270	-	2582	-	5096	-	10191	-
	2	1276	1.005	2583	1.000	5100	1.001	10202	1.001
	4	1281	1.009	2590	1.003	5112	1.003	10223	1.003
	8	1297	1.02	2611	1.01	5142	1.009	10250	1.006
	16	1366	1.04	2635	1.02	5169	1.01	10279	1.009
10	sequential	1992	-	4003	-	8134	-	16470	-
	2	2010	1.01	4048	1.01	8201	1.008	16487	1.001
	4	2069	1.04	4092	1.02	8248	1.01	16518	1.003
	8	2110	1.06	4150	1.04	8410	1.03	16588	1.007
	16	2183	1.10	4278	1.07	8541	1.05	16822	1.020

TABLE 9
RUNNING TIME (sec) AND SPEEDUP OF HPMCP

CCR	No. of PPEs	Graph size (number of nodes)							
		500		1000		2000		4000	
		time	S	time	S	time	S	time	S
0.1	sequential	0.56	-	1.45	-	4.10	-	10.9	-
	2	0.30	1.87	0.97	1.49	2.28	1.80	4.70	2.32
	4	0.23	2.43	0.64	2.27	1.08	3.78	2.67	4.08
	8	0.19	2.94	0.44	3.29	0.90	4.56	1.60	6.82
	16	0.20	2.80	0.31	4.68	0.60	6.88	1.28	8.52
1	sequential	0.45	-	0.75	-	2.90	-	12.5	-
	2	0.30	1.50	0.32	2.34	1.41	2.06	5.90	2.12
	4	0.21	2.14	0.18	4.17	0.79	3.67	2.97	4.21
	8	0.19	2.37	0.12	6.25	0.55	5.30	2.12	5.89
	16	0.18	2.50	0.11	6.82	0.44	6.59	1.87	6.70
10	sequential	0.44	-	0.99	-	3.06	-	11.4	-
	2	0.28	1.57	0.54	1.83	1.37	2.23	4.84	2.36
	4	0.22	2.00	0.33	3.00	0.74	4.14	2.72	4.18
	8	0.16	2.75	0.24	4.11	0.52	5.88	1.59	7.17
	16	0.12	-3.67	0.18	5.50	0.39	7.85	1.27	8.98

Running time and speedup of the VPMCP algorithm are shown in Table 7. Speedup is defined by $S = T_S/T_p$, where T_S is the sequential execution time of the optimal sequential algorithm and T_p is the parallel execution time. The MCP running time on a single processor is a sequential version without parallelization overhead. The low speedup of VPMCP is caused by its large number of communications.

Next, we study performance of the HPMCP algorithm. The number of TPEs is four in Tables 8 and 9. In Table 8, the ratio was obtained by running the HPMCP algorithm on 2, 4, 8, and 16 PPEs on Paragon and taking the ratios of the schedule lengths produced by it to those of sequential MCP running on one PPE. The deterioration in performance of HPMCP is due to estimation of the start time of RPNs and concatenation. Out of the 48 test cases shown in the table, there is only one case in which HPMCP performed 10 percent worse than sequential MCP, 28 of them are within 1 percent, and 19 of them are between 1 percent to 10 percent.

Running time and speedup of the HPMCP algorithm are shown in Table 9. There are some superlinear speedup cases in the table. That is because HPMCP has lower com-

plexity than MCP. The complexity of MCP is $O(n^2)$. The complexity of HPMCP on P PPEs is $O(e + n \log n + n^2/P^2)$. Therefore, speedup is bounded by P^2 instead of P . Speedup on 16 PPEs is not as good as expected, because the graph size is not large enough and the relative overhead is large.

We can now compare performance of VPMCP and HPMCP. Performance shown in Figs. 14 and 15 and is for graphs of 4,000 nodes. The number of TPEs is the same as the number of PPEs. Fig. 14 shows the percentage of the schedule length over that produced by MCP. For 2 or 4 PPEs, HPMCP produces shorter schedule lengths. However, for more PPEs, VPMCP produces better scheduling quality than that produced by HPMCP. In general, VPMCP provides a more stable scheduling quality. Fig. 15 compares the speedups of VPMCP and HPMCP algorithms and shows that HPMCP is faster than VPMCP.

7 CONCLUDING REMARKS

Parallel scheduling is faster and is able to schedule large macro dataflow graphs. Parallel scheduling is a new ap-

proach and is still under development. Many open problems need to be solved. High-quality parallel scheduling algorithms with low complexity will be developed. It can be achieved by parallelizing the existing sequential scheduling algorithms or by designing new parallel scheduling algorithms. We have developed the VPMCP and HPMCP algorithms by parallelizing the sequential MCP algorithm. Performance of this approach has been studied in this paper.

ACKNOWLEDGMENTS

We are very grateful to Yu-Kwong Kwok and Ishfaq Ahmad for providing their PBSA program and random graph generator for testing. This research was partially supported by the National Science Foundation under Grants CCR-9625784 and CCR-9505300.

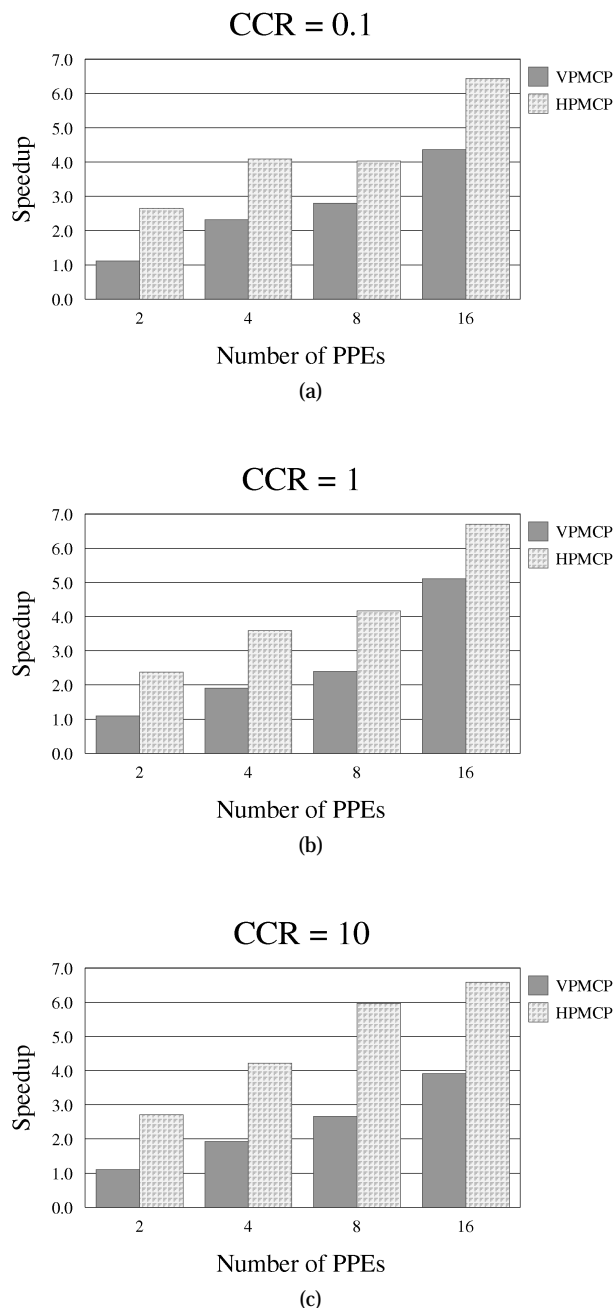
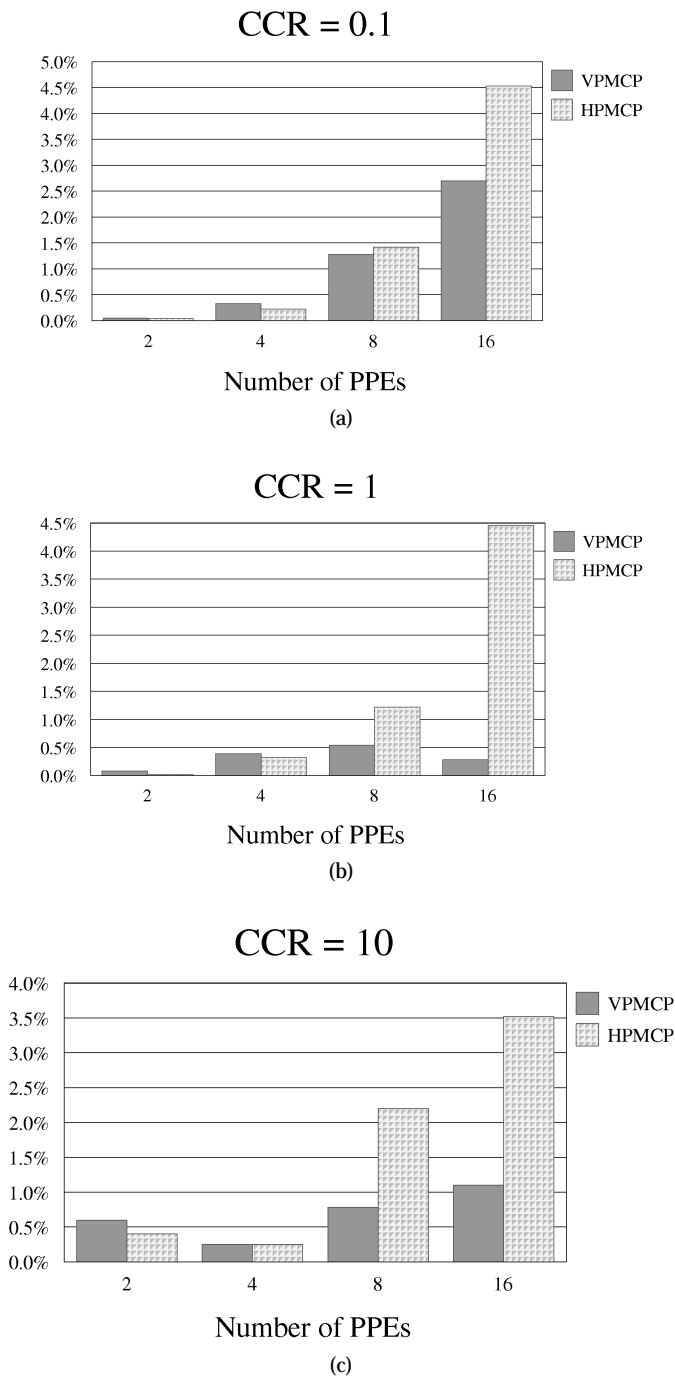
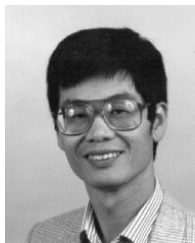


Fig. 14. Comparison of scheduling quality produced by VPMCP and HPMCP.

Fig. 15. Comparison of speedups of VPMCP and HPMCP.

REFERENCES

- [1] M. Gary and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.
- [2] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, 1989.
- [3] M.Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, pp. 330-343, July 1990.
- [4] H. El-Rewini and T.G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *J. Parallel and Distributed Computing*, June 1990.
- [5] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, pp. 951-967, Sept. 1994.
- [6] 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, pp. 175-187, Feb. 1993.
- [7] Y. Chung and S. Ranka, "Applications and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessors," *Supercomputer '92*, Nov. 1992.
- [8] A. Khan, C. McCreary, and M. Jones, "A Comparison of Multiprocessor Scheduling Heuristics," *Proc. Int'l Conf. Parallel Processing*, vol. 2, pp. 243-250, Aug. 1994.
- [9] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9, no. 6, pp. 841-848, 1961.
- [10] T.L. Adam, K. Chandy, and J. Dickson, "A Comparison of List Scheduling for Parallel Processing Systems," *Comm. ACM*, vol. 17, pp. 685-690, Dec. 1974.
- [11] B. Kruatrachue and T. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software*, pp. 23-32, Jan. 1988.
- [12] H. El-Rewini, T.G. Lewis, and H.H. Ali, *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, 1994.
- [13] B. Kruatrachue, "Static Task Scheduling and Grain Packing in Parallel Processor Systems," PhD thesis, Dept. of Electrical and Computer Engineering, Oregon State Univ., Corvallis, 1987.
- [14] J. Hwang, Y. Chow, F. Anger, and C. Lee, "Scheduling Precedence graphs in Systems with Interprocessor Communication Times," *SIAM J. Computing*, vol. 18, pp. 244-257, Apr. 1989.
- [15] J. Baxter and J.H. Patel, "The LAST Algorithm: A Heuristics-Based Static Task Allocation Algorithm," *Proc. Int'l Conf. Parallel Processing*, vol. 2, pp. 217-222, Aug. 1989.
- [16] I. Ahmad, Y. Kwok, and M. Wu, "Performance Comparison of Algorithms for Static Scheduling of DAGs to Multiprocessors," *Second Australasian Conf. Parallel and Real-time Systems*, Sept. 1995.
- [17] I. Ahmad and Y. Kwok, "A Parallel Approach to Multiprocessor Scheduling," *Proc. Int'l Parallel Processing Symp.*, pp. 289-293, Apr. 1995.



Min-You Wu received the MS degree from the Graduate School of Academia, Sinica, Beijing, China, and the PhD degree from Santa Clara University, California. Before he joined the Department of Electrical and Computer Engineering, University of Central Florida, where he is currently an associate professor, he has held various positions at the University of Illinois at Urbana-Champaign, University of California at Irvine, Yale University, Syracuse University, and State University of New York at Buffalo. His

research interests include parallel and distributed systems, compilers for parallel computers, programming tools, VLSI design, and multimedia systems. He has published over 60 journal and conference papers in the above areas and edited two special issues on parallel operating systems. He is a member of ACM and a senior member of the IEEE.



Wei Shu received the PhD degree from the University of Illinois at Urbana-Champaign in 1990, the MS degree from Santa Clara University in 1984, and the BS degree from Hefei Polytechnic University, China, in 1982. Since then, she worked at Yale University and the State University of New York at Buffalo. She is currently an associate professor in the Department of Electrical and Computer Engineering, University of Central Florida. Her current research interests include dynamic scheduling, runtime support

systems for parallel processing, and parallel operating systems. She is a member of the IEEE and the ACM.