On Parallelization of Static Scheduling Algorithms

Min-You Wu
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
wu@cs.buffalo.edu

Abstract— Most static scheduling algorithms that schedule parallel programs represented by directed acyclic graphs (DAGs) are sequential. This paper discusses the essential issues of parallelization of static scheduling and presents two efficient parallel scheduling algorithms. The proposed algorithms have been implemented on an Intel Paragon machine, and their performance has been evaluated. These algorithms produce high-quality scheduling and are much faster than existing sequential and parallel algorithms.

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 minimize the completion time. This is an NP-complete problem in its general form [7]. Therefore, many hueristic algorithms that produce satisfactory performance have been proposed [11, 13, 5, 14, 12, 4, 9].

Although these scheduling algorithms apply to parallel programs, the algorithms themselves are sequential, and are executed on a single processor system. A sequential algorithm is slow. Scalability of static scheduling is restricted since a large memory space is required to store the task graph. A natural solution to this problem is *using multiprocessors to schedule tasks to multiprocessors.* In fact, 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 minimize the completion time of a parallel program.
- **Low complexity** — it is able to minimize the time for scheduling a parallel program.

These two requirements contradict each other in general. Usually, a high-quality scheduling algorithm is of a high complexity. The Modified Critical-Path (MCP) algorithm was introduced [13], which offered a good quality with relatively low complexity. In this paper, we propose two parallelized versions of MCP. We will describe the MCP algorithm in the next section. Then, we will discuss different approaches for 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 comparison of the two algorithms will be presented in section 6.

2 The MCP Algorithm

A macro dataflow graph is a directed acyclic graph with a starting point and an end point [13]. A macro dataflow graph consists of a set of nodes $\{n_1, n_2, ..., n_n\}$ connected by a set of edges, each of which is denoted by $e(n_i, n_j)$. Each node represents a task, and the weight of a node is the execution time of the task. Each edge represents a message transferred from one node to another node, and the weight of the edge is equal to the transmission time of the message. When two nodes are scheduled to the same processing element (PE), the weight of the edge connecting them becomes zero.

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 the end point, including node n_i [6]. In fact, high-quality scheduling algorithms more or less rely on the ALAP time or level.

The Modified Critical-Path (MCP) algorithm was designed to schedule a macro dataflow graph on a bounded number of PEs.

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, and so on.

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. □

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 assumption, that are, ISH [10], ETF [8], DLS [12], and

LAST [3]. It has been shown that MCP performed the best [2].

The complexity of the MCP algorithm is O($n^2 logn$), 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.

3 Approaches for Parallelization of Scheduling Algorithms

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. In the following, 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 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 approaches for parallel scheduling. The first one is called 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. The second one is called 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. The vertical and horizontal schemes are illustrated in Figure 1. The task graph is mapped to the time-space domain. Here, 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.

The vertical scheme and the horizontal scheme are outlined in Figure 2 and 3, respectively. In the vertical scheme, PPEs exchange information and schedule graph nodes to TPEs. Frequent information exchange results in large communication overhead. With horizontal partitioning, each PPE can schedule its graph partition without exchanging information with another PPE. In the last step, PPEs exchange information of their sub-schedules and concatenate them to obtain the 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.

There is almost no work in designing a parallel algorithm for scheduling. In fact, there is no algorithm in the vertical scheme yet. The only algorithm in this area is in the horizontal

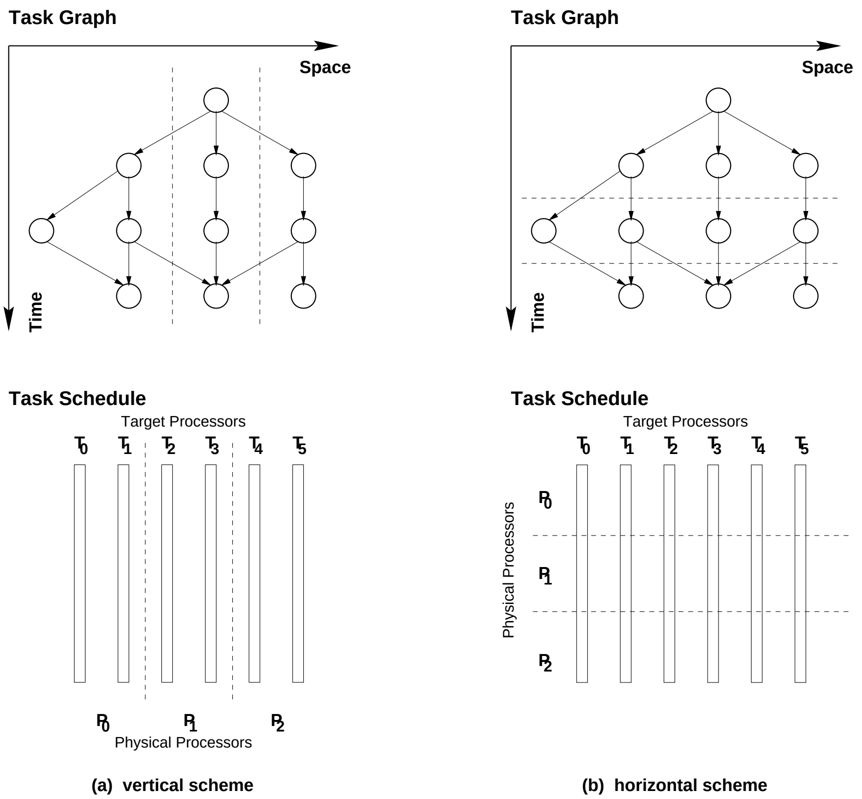

Figure 1: Vertical and Horizontal Schemes.

---

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.

Figure 2: 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 sub-schedule.
3. PPEs exchange information to concatenate sub-schedules.

Figure 3: The Horizontal Scheme for Parallel Scheduling.

scheme, which is the parallel BSA (PBSA) algorithm [1]. The BSA algorithm takes into account link contention and communication routing strategy. A PE list is constructed in a breadth-first order from the PE having the highest degree (pivot PE). This algorithm constructs a schedule incrementally by first injecting all the nodes to the pivot PE. Then, 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 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 multiple nodes are scheduled simultaneously, the resultant schedule may not be the same as the one produced by MCP. The scheduling 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.

In the vertical scheme, multiple nodes may be selected to be scheduled 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. Therefore, the resultant schedule cannot be the same as the sequential one. 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 Figure 4. The nodes are first sorted by the ALAP time and cyclicly divided into P partitions. That is, the nodes in places $i, i+P, i+2P, \ldots$ 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.

---

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 obtains 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.

Figure 4: The VPMCP1 Algorithm.

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, 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 conflict with each other. Conflict may result in degradation of scheduling quality. The following lemma states the condition that allows some nodes to be scheduled in parallel without reducing scheduling quality.

Lemma 1: When a node is scheduled to its earliest *start_time* without conflicting with its former nodes, it is scheduled to the same place as it would be scheduled by sequential MCP.

Proof: 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. □

With this lemma, 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_time*s. 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. Therefore, 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 sub-optimal place. Therefore, when a node is found to be in conflict with its former nodes, it will be scheduled to the next non-conflict 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 is shown in Figure 5. Besides the sorted node list, a ready list is constructed and sorted by ALAP times. A set of nodes selected from the ready list are broadcast to all PPEs. The *start_time* of each node is calculated independently. Then, the *start_time*s 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 non-conflict place.

The time for calculation of the ALAP time and sorting is $O(e+n\,log\,n)$. The parallel scheduling step is of $O(n^2/P)$. Therefore, 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 $2n$ for VPMCP1 and $2n/p$ for VPMCP, where p is the number of TPEs.

The VPMCP algorithm is compared to VPMCP1 in Table I. The workload for testing consists of random graphs of various sizes. We use the same random graph generator in [1]. Three values of communication-computation-ratio (CCR) were selected to be 0.1, 1, and 10. The weights on the nodes and edges were generated randomly such that the average value of CCR corresponded to 0.1, 1, or 10. This set of graphs will be used in the subsequent experiment. In this section, we use a set of graphs, each of which has 2,000 nodes. Each graph is scheduled to four TPEs.

In the vertical scheme, the number of PPEs cannot be larger than the number of TPEs because

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 a ready list consisting of all nodes with no parent. Sort the list in an increasing ALAP order.
2. (a) The first p nodes in the ready 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 ready list, broadcast the entire ready list.
   (b) Each PPE obtains a *start_time* for each node on each of its TPEs. The *start_time*s of each node 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*, ans so on.
   (d) The parent information is updated for the children of the scheduled node. Delete these nodes from the ready list and update the ready list by adding nodes freed by these nodes. Repeat this step until the ready list is empty.

Figure 5: The VPMCP Algorithm.

Table I: Comparison of Vertical Strategies

CCR	Number of PPEs	Scheduling length VPMCP1	Scheduling length VPMCP	Running time (second) VPMCP1	Running time (second) VPMCP
0.1	1	4985	5007	4.62	4.62
	2	4985	5007	4.14	3.39
	4	4985	5007	4.02	2.81
1	1	5096	5111	2.91	2.91
	2	5096	5111	2.84	2.37
	4	5096	5111	2.81	2.12
10	1	8135	8231	3.11	3.11
	2	8135	8231	3.31	2.45
	4	8135	8231	3.38	2.12

each TPE must be maintained in a single PPE. It can be seen from the table that VPMCP1 produces a better scheduling quality. However, its heavy communication results in low speedup or no speedup. VPMCP reduces running times and still provides an acceptable scheduling quality. The scheduling lengths are between 0.3% and 1.2% longer than that produced by VPMCP1.

5 The HPMCP Algorithm

In a horizontal scheme, different partitions must be scheduled simultaneously for a parallel execution. We call a horizontal version of parallel MCP the HPMCP algorithm, which is shown in Figure 6.

---

1. (a) Compute the ALAP time of each node and sort the node list in an increasing ALAP order. Ties are broken randomly.
   (b) 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 sub-schedule. Edges between a node and its RPNs are ignored.
3. Concatenate each pair of adjacent sub-schedules. Walk through the schedule to determine the actual start time of each node.

Figure 6: The HPMCP Algorithm.

In the HPMCP algorithm, the nodes are first sorted by the ALAP time. Therefore, the node list is in a topological order and is 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 sub-schedule. Then, these sub-schedules will be concatenated to form the final schedule. Three problems are to be addressed for scheduling and concatenation: information estimation, concatenation permutation, and post-insertion.

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 partitions in parallel, each PPE needs schedule information of its former partitions. Since it is impossible to obtain such information before the schedules of former partitions have been produced, an estimation is necessary. Although the latter PPE does not know the exact schedules of its former partitions, an estimation can help a node to determine its earliest start time in the latter partition.

In the PBSA algorithm [1], the start time of each RPN (remote parent node) is estimated. It is done by calculating two parameters: the earliest possible start time (EPST), and the latest possible start time (LPST). The EPST of a node is the largest sum of computation times from the start point to the node, excluding the node itself. The LPST of a node is the sum of computation times of all nodes scheduled before the node, excluding the node itself. The estimated start time (EST) of an RPN is defined as $\alpha EPST + (1-\alpha)LPST$, where 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. Given the estimated start time of an RPN, it is still necessary to estimate to which TPE the RPN is scheduled. If the RPN is a critical path node, then it is assumed that it will be scheduled to the same TPE as the highest level critical path node in the local partition. Otherwise, a TPE is randomly picked to be the one to which the RPN is scheduled. We call this estimation the PBSA estimation. This estimation is not necessarily accurate or better than a simpler estimation used in HPMCP.

In HPMCP, we simply ignore all dependences between partitions. Therefore, all entry nodes in its partition can start at the same time. Furthermore, we assume all schedules of the former PE end at the same time. We call this estimation the HPMCP estimation.

Table II: Comparison of Estimation Algorithms

CCR	Number of PPEs	Scheduling length		Running time (second)	
		HPMCP est.	PBSA est.	HPMCP est.	PBSA est.
0.1	1	4985	4985	4.62	4.62
	2	4994	5022	2.56	3.64
	4	4995	5078	2.04	2.90
	8	5006	5126	1.43	2.15
	16	5018	5232	0.98	1.59
1	1	5096	5096	2.91	2.91
	2	5096	5144	1.31	1.84
	4	5113	5276	0.79	1.17
	8	5142	5539	0.55	0.86
	16	5169	5740	0.44	0.73
10	1	8135	8135	3.11	3.11
	2	8214	8222	1.49	2.10
	4	8248	8471	0.93	1.38
	8	8416	9132	0.64	1.02
	16	8839	10023	0.47	0.81

Now, we will compare the two approaches of estimation. In this section, the number of nodes in a graph is 2,000 and each graph is scheduled to four TPEs. The comparison is shown in Table II.

The column "HPMCP est." shows the performance of HPMCP. The column "PBSA est." shows the performance of HPMCP with PBSA estimation. The scheduling lengths and running times are compared. The running time of PBSA estimation is longer. Notice that more PPEs produce longer scheduling lengths. That shows the trade-off between scheduling quality and parallelism. On the other hand, superlinear speedup was observed due to graph partitioning.

The scheduling 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.

Concatenation permutation

After each PPE produces its sub-schedule, the final schedule is constructed by concatenating these sub-schedules. Because there is no accurate information of former sub-schedules, it is not easy to determine the optimal permutation of TPEs between adjacent sub-schedules, that is, to determine which latter TPE should be concatenated to which former TPE. A hueristics is necessary. In the PBSA algorithm, a TPE with the earliest node is concatenated to the TPE of the former sub-schedule that allows the earliest execution. Then, other TPEs are concatenated to the TPEs in the former sub-schedule in a breadth-first order [1]. 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 sub-schedule is concatenated to the same TPE in the former sub-schedule. An alternative hueristics can be described as follows: each PPE finds out within its sub-schedule 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 non-permutation algorithm in Table III. The time spent on the permutation step causes this algorithm to be slower since extra time is spent on determine weather a node is on the critical path. In terms of the scheduling length, the permutation algorithm makes four of the test cases better than the non-permutation algorithm, and two cases worse. This permutation algorithm does not improve performance much. Therefore, no permutation is performed in the HPMCP algorithm.

Post-insertion

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 sub-schedules due to lack of their information. This leads to

Table III: Comparison of Permutation Algorithms

CCR	Number of PPEs	Scheduling length: No permut.	Scheduling length: Permut.	Running time (second): No permut.	Running time (second): Permut.
0.1	1	4985	4985	4.62	4.62
	2	4994	4994	2.56	2.80
	4	4995	4995	2.04	2.25
	8	5006	5006	1.43	1.64
	16	5018	5022	0.98	1.19
1	1	5096	5096	2.91	2.91
	2	5096	5096	1.31	1.40
	4	5113	5110	0.79	0.88
	8	5142	5141	0.55	0.64
	16	5169	5171	0.44	0.53
10	1	8135	8135	3.11	3.11
	2	8214	8214	1.49	1.59
	4	8248	8248	0.93	1.04
	8	8416	8145	0.64	0.75
	16	8839	8825	0.47	0.59

some performance loss. It can be partially corrected at the concatenation time by inserting the nodes of a latter sub-schedule into its former sub-schedules. Improvement of this post-insertion algorithm is shown in Table III. Compared to non-insertion, the post-insertion algorithm reduces scheduling length in eight test cases and increases it in four cases. Overall, this post-insertion algorithm can improve scheduling quality. However, it spends much more time for post-insertion. In the following, we do not perform this post-insertion.

The time for calculation of the ALAP time and sorting is $O(e + n\,log\,n)$. The second step of parallel scheduling is of $O(n^2/P^2)$, and the third step spends $O(e + n^2/p^2)$ time for post-insertion and $O(e)$ time for non-insertion. Therefore, the complexity of the HPMCP algorithm without post-insertion 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.

6 Performance

The VPMCP and HPMCP algorithms were implemented on Intel Paragon. We present its performance with three measures: scheduling length, running time, and speedup.

Table IV: Comparison of Post-insertion Algorithms

CCR	Number of PPEs	Scheduling length No insert.	Scheduling length Insert.	Running time (second) No insert.	Running time (second) Insert.
0.1	1	4985	4985	4.62	4.62
	2	4994	4993	2.56	3.88
	4	4995	4994	2.04	2.88
	8	5006	5003	1.43	2.21
	16	5018	5017	0.98	1.99
1	1	5096	5096	2.91	2.91
	2	5096	5111	1.31	2.07
	4	5113	5119	0.79	1.28
	8	5142	5124	0.55	1.05
	16	5169	5148	0.44	1.01
10	1	8135	8135	3.11	3.11
	2	8214	8218	1.49	1.96
	4	8248	8270	0.93	1.45
	8	8416	8352	0.64	1.19
	16	8839	8530	0.47	1.15

First, performance of the VPMCP algorithm is to be presented. In Tables V and VI, graphs of 1,000, 2,000, 3,000, and 4,000 nodes are scheduled to four TPEs. The scheduling length provides a measure of scheduling quality. The results shown in Table V are the scheduling lengths and the ratios of the scheduling lengths produced by the VPMCP algorithm to the scheduling lengths produced by the MCP algorithm. The ratio was obtained by running the VPMCP algorithm on 2 and 4 PPEs on Paragon and taking the ratios of the scheduling lengths produced by it to those of MCP running on one PPE. As one can see from this table, there was almost no effect of graph size on scheduling quality. In most cases, the scheduling lengths of VPMCP are not more than 1% longer than that produced by MCP.

Running time and speedup of the VPMCP algorithm are shown in Table VI. 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 running times of VPMCP on more than one PPE are compared with MCP running time on one PPE. 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.

The next experiment is to study VPMCP performance of different numbers of TPEs. Tables VII and VIII show the scheduling lengths and running times of graphs of 4,000 nodes for 2, 4,

Table V: The scheduling lengths produced by VPMCP and their ratios to those of MCP

CCR	Number of PPEs	Graph size (number of nodes) 1000 length	ratio	2000 length	ratio	3000 length	ratio	4000 length	ratio
0.1	1	2518	–	4985	–	7574	–	9914	–
	> 1	2522	1.002	5007	1.004	7601	1.004	9947	1.002
1	1	2585	–	5096	–	7466	–	10107	–
	> 1	2593	1.003	5111	1.003	7468	1.000	10146	1.003
10	1	4007	–	8135	–	11060	–	16294	–
	> 1	4137	1.03	8231	1.01	11121	1.006	16335	1.003

Table VI: Running time (in second) and speedup of VPMCP

CCR	Number of PPEs	Graph size (number of nodes) 1000 time	S	2000 time	S	3000 time	S	4000 time	S
0.1	1	1.48	–	4.62	–	6.01	–	11.5	–
	2	1.24	1.19	3.39	1.36	4.50	1.34	6.21	1.85
	4	1.15	1.28	2.81	1.64	3.74	1.61	4.95	2.32
1	1	0.69	–	2.91	–	9.54	–	13.6	–
	2	0.73	0.95	2.37	1.23	6.80	1.39	10.3	1.32
	4	0.77	0.90	2.12	1.37	5.55	1.72	7.11	1.91
10	1	0.94	–	3.11	–	6.61	–	11.4	–
	2	0.92	1.02	2.45	1.27	4.96	1.33	7.85	1.45
	4	0.90	1.04	2.12	1.47	3.98	1.66	5.91	1.93

8, and 16 TPEs. The difference between the scheduling lengths of VPMCP and MCP are within 1% in most cases. As can be noticed from this table, when the number of TPEs increases, the scheduling lengths decrease and the ratio only increases slightly. Therefore, scheduling quality scales up quit well. Higher speedups are obtained with more PPEs.

Next, we study performance of the HPMCP algorithm. The number of TPEs is four in Tables IX and X. In Table IX, the ratio was obtained by running the HPMCP algorithm on 2, 4, 8, and 16 PPEs on Paragon and taking the ratios of the scheduling lengths produced by it to those of 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 more than 10% worse than MCP, 31

Table VII: The scheduling lengths and ratios for different number of TPEs produced by VPMCP

CCR	Number of PPEs	Number of TPEs: 2 length	2 ratio	4 length	4 ratio	8 length	8 ratio	16 length	16 ratio
0.1	1	19790	–	9914	–	5003	–	2626	–
	> 1	19800	1.001	9947	1.003	5067	1.01	2697	1.03
1	1	20147	–	10107	–	5179	–	2824	–
	> 1	29163	1.001	10146	1.003	5207	1.005	2832	1.003
10	1	22659	–	16294	–	13528	–	12547	–
	> 1	22795	1.006	16335	1.003	13633	1.008	12686	1.01

Table VIII: VPMCP running time (in second) and speedup for different number of TPEs

CCR	Number of PPEs	Number of TPEs: 2 time	2 S	4 time	4 S	8 time	8 S	16 time	16 S
0.1	1	10.3	–	11.5	–	14.2	–	19.7	–
	2	9.25	1.11	6.21	1.85	9.16	1.55	11.7	1.68
	4	–	–	4.95	2.32	6.23	2.28	7.16	2.75
	8	–	–	–	–	5.07	2.80	5.14	3.83
	16	–	–	–	–	–	–	4.51	4.36
1	1	11.6	–	13.6	–	17.7	–	26.4	–
	2	10.6	1.09	10.3	1.32	12.8	1.38	19.0	1.39
	4	–	–	7.11	1.91	9.15	1.93	12.8	2.06
	8	–	–	–	–	7.35	2.48	7.30	3.62
	16	–	–	–	–	–	–	4.88	5.41
10	1	10.2	–	11.4	–	13.9	–	19.2	–
	2	9.27	1.10	7.85	1.45	9.86	1.41	14.8	1.30
	4	–	–	5.91	1.93	6.59	2.11	8.89	2.16
	8	–	–	–	–	5.23	2.66	6.17	3.11
	16	–	–	–	–	–	–	4.91	3.91

of them are within 1%, and 16 of them are between 1% to 10%. It is of interest that in two cases, HPMCP produced even better results than MCP. Since MCP is a heuristic algorithm, this is possible. Sometimes, a parallel version could produce a better result than its corresponding sequential one.

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

Tables XI and XII show the HPMCP scheduling lengths and running times of graphs of 4,000 nodes for 2, 4, 8, and 16 TPEs. The speedups decrease with the number of TPEs. That is caused by increasing dependences between TPEs.

Now, we compare performance of VPMCP and HPMCP. Performance shown in Figures 7 and 8 is for graphs of 4,000 nodes. The number of TPEs is the same as the number of PPEs. Figure 7 shows the percentage of the scheduling length over that produced by MCP. There are some negative numbers which indicate that scheduling lengths are shorter than that produced by MCP. For 2 or 4 PPEs, HPMCP produces shorter scheduling lengths. However, for more PPEs, VPMCP produces better scheduling quality than that produced by HPMCP. In general, VPMCP provides a more stable scheduling quality. Figure 8 compares the speedups of VPMCP and HPMCP algorithms. HPMCP is faster than VPMCP with a higher speedup.

After scheduling, the nodes are not in the PPEs where they are to be executed in the horizontal scheme. A major communication step is necessary to move nodes. However, when the number of PPEs is equal to the number of TPEs, the vertical scheme can avoid this communication step because the nodes reside in the PPEs where they are to be executed. It becomes more important when the scheduling algorithms are used at runtime.

Next, we compare two algorithms in the horizontal scheme, HPMCP and PBSA. The PBSA algorithm takes into account link contention and communication routing strategy, but HPMCP does not consider these factors. Therefore, the edge weights in PBSA vary with different topologies, whereas in HPMCP they are constant. For comparison purposes, we have implemented a simplified version of PBSA, which assumes that the edge weights are constant. PBSA is much slower than HPMCP, because its complexity is much higher. The complexity of PBSA is $O(p^2en/P^2)$ and that of HPMCP is $O(e + n\ log\,n + n^2/P^2)$. HPMCP is about 50 to 120 times faster than PBSA for this set of graphs. Then, we compare the scheduling lengths produced by HPMCP and PBSA. The results are shown in Table XIII. In this table, the scheduling lengths produced by the sequential MCP and BSA algorithms running on a single processor are also compared.

Table IX: The scheduling lengths produced by HPMCP and their ratios to those of MCP

CCR	Number of PPEs	Graph size (number of nodes)							
		1000		2000		3000		4000	
		length	ratio	length	ratio	length	ratio	length	ratio
0.1	1	2518	–	4985	–	7574	–	9914	–
	2	2530	1.005	4994	1.002	7587	1.002	9930	1.002
	4	2527	1.004	4995	1.002	7598	1.003	9936	1.002
	8	2542	1.01	5006	1.004	7603	1.004	9957	1.004
	16	2565	1.02	5018	1.007	7630	1.007	9981	1.007
1	1	2585	–	5096	–	7466	–	10107	–
	2	2593	1.003	5096	1.000	7476	1.001	10118	1.001
	4	2593	1.003	5113	1.003	7486	1.003	10139	1.003
	8	2641	1.02	5142	1.009	7500	1.005	10169	1.006
	16	2663	1.03	5169	1.01	7516	1.007	10193	1.009
10	1	4007	–	8135	–	11060	–	16294	–
	2	4064	1.01	8214	1.01	11102	1.004	16257	0.998
	4	4222	1.05	8248	1.01	11183	1.01	15875	0.974
	8	4361	1.09	8416	1.03	11359	1.03	16410	1.007
	16	4513	1.13	8839	1.09	11660	1.05	16850	1.03

Table X: Running time (in second) and speedup of HPMCP

CCR	Number of PPEs	Graph size (number of nodes)							
		1000		2000		3000		4000	
		time	S	time	S	time	S	time	S
0.1	1	1.48	–	4.62	–	6.01	–	11.5	–
	2	0.99	1.49	2.56	1.80	2.43	2.47	4.99	2.30
	4	0.66	2.27	2.04	2.26	1.32	4.55	2.81	4.09
	8	0.44	3.36	1.43	3.23	0.85	7.07	1.70	6.76
	16	0.33	4.48	0.98	4.71	0.65	9.25	1.34	8.58
1	1	0.69	–	2.91	–	9.54	–	13.6	–
	2	0.29	2.38	1.31	2.22	4.68	2.04	6.45	2.11
	4	0.17	4.06	0.79	3.68	2.49	3.83	3.79	3.59
	8	0.11	6.27	0.55	5.29	1.85	5.16	2.32	5.86
	16	0.09	7.67	0.44	6.61	1.48	6.45	2.05	6.63
10	1	0.94	–	3.11	–	6.61	–	11.4	–
	2	0.51	1.84	1.49	2.09	2.84	2.33	4.82	2.37
	4	0.34	2.76	0.93	3.34	1.56	4.24	2.70	4.22
	8	0.26	3.62	0.64	4.86	1.01	6.54	1.59	7.17
	16	0.19	4.95	0.47	6.62	0.77	8.58	1.28	8.91

Table XI: The scheduling lengths and ratios for different number of TPEs produced by HPMCP

CCR	Number of PPEs	Number of TPEs: 2 length	2 ratio	4 length	4 ratio	8 length	8 ratio	16 length	16 ratio
0.1	1	19790	–	9914	–	5003	–	2626	–
	2	19798	1.000	9930	1.002	5018	1.003	2644	1.007
	4	19805	1.001	9936	1.002	5034	1.006	2664	1.01
	8	19813	1.001	9957	1.004	5074	1.01	2704	1.03
	16	19838	1.002	9981	1.007	5102	1.02	2745	1.05
1	1	20147	–	10107	–	5179	–	2824	–
	2	20150	1.000	10118	1.001	5191	1.002	2831	1.002
	4	20167	1.001	10139	1.003	5213	1.007	2853	1.01
	8	20172	1.001	10169	1.006	5242	1.01	2899	1.03
	16	20186	1.002	10193	1.009	5297	1.02	2950	1.04
10	1	22659	–	16294	–	13528	–	12547	–
	2	22185	0.979	16257	0.998	13550	1.002	12567	1.002
	4	22159	0.978	15875	0.974	13421	0.992	12601	1.004
	8	22375	0.987	16410	1.007	13826	1.02	12787	1.02
	16	23255	1.03	16850	1.03	14093	1.04	12989	1.04

Table XII: HPMCP Running time (in second) and speedup for different number of TPEs

CCR	Number of PPEs	Number of TPEs: 2 time	2 S	4 time	4 S	8 time	8 S	16 time	16 S
0.1	1	10.3	–	11.5	–	14.2	–	19.7	–
	2	3.88	2.65	4.99	2.30	7.22	1.97	11.6	1.70
	4	2.03	5.07	2.81	4.09	4.33	3.28	7.22	2.73
	8	1.28	8.05	1.70	6.76	3.52	4.03	4.15	4.75
	16	1.06	9.72	1.34	8.58	1.92	7.40	3.06	6.44
1	1	11.6	–	13.6	–	17.7	–	26.4	–
	2	4.88	2.38	6.45	2.11	9.66	1.83	15.9	1.66
	4	2.81	4.13	3.79	3.59	5.90	3.00	10.3	2.56
	8	2.38	4.87	2.32	5.86	4.24	4.17	7.03	3.76
	16	1.60	7.25	2.05	6.63	2.85	6.21	3.94	6.70
10	1	10.2	–	11.4	–	13.9	–	19.2	–
	2	3.77	2.71	4.82	2.37	6.97	1.99	11.2	1.71
	4	1.94	5.26	2.70	4.22	4.19	3.32	7.12	2.70
	8	1.21	8.43	1.59	7.17	2.33	5.97	3.79	5.07
	16	1.00	10.2	1.28	8.91	1.83	7.60	2.92	6.58

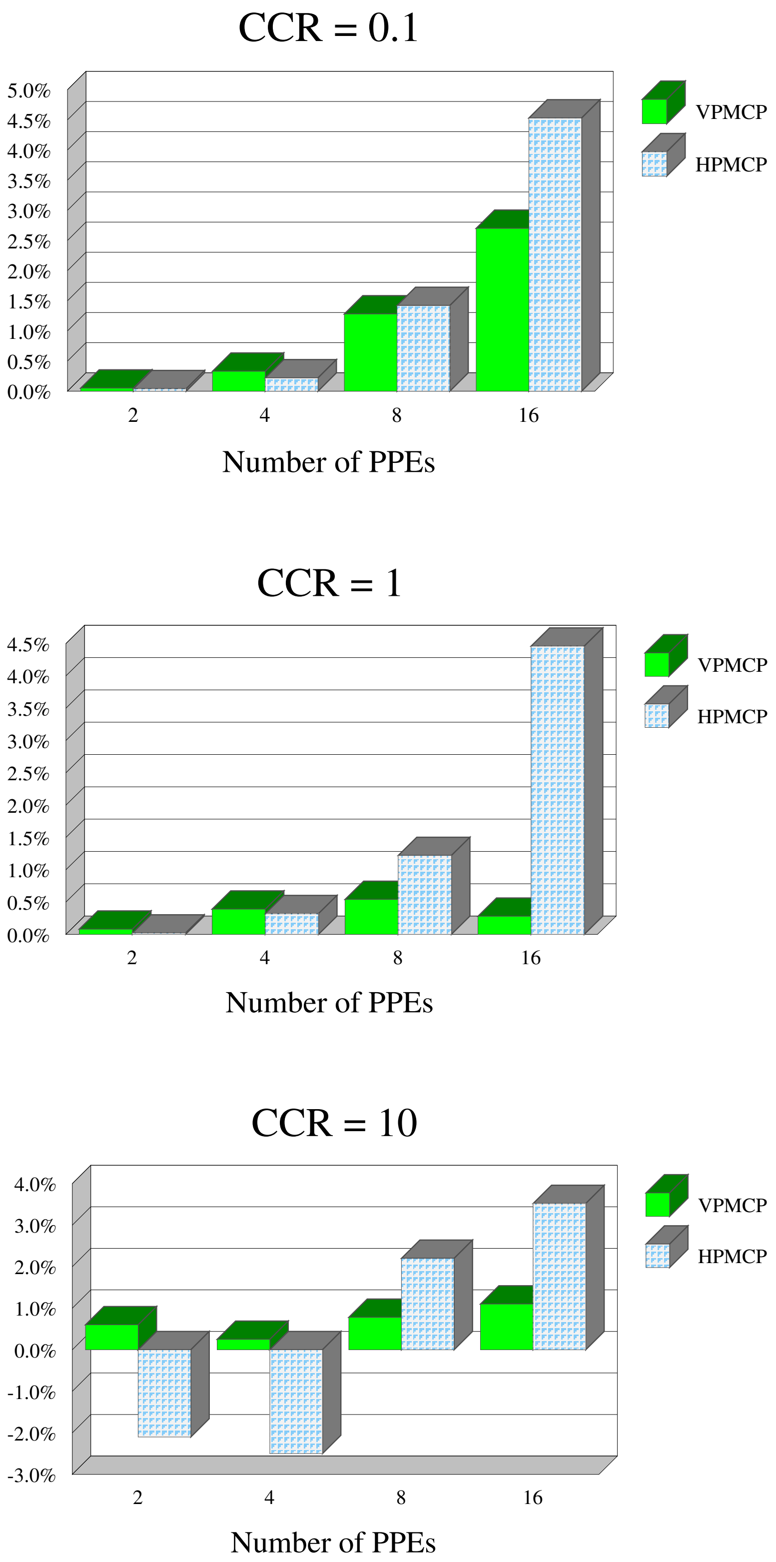

Figure 7: Comparison of scheduling quality produced by VPMCP and HPMCP

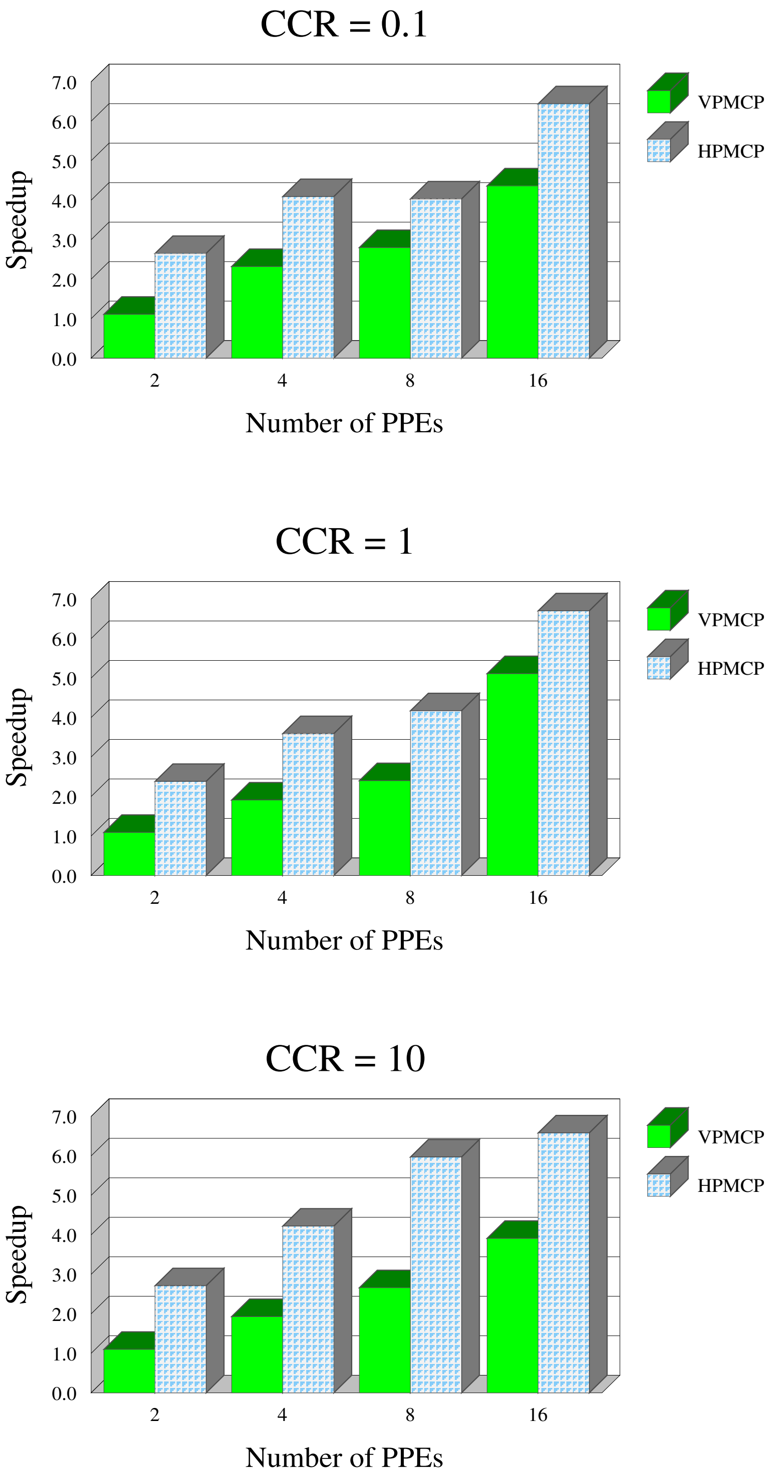

Figure 8: Comparison of speedups of VPMCP and HPMCP

When CCR is 0.1 or 1, the scheduling lengths produced by MCP are slightly shorter than those produced by BSA. When CCR is 10, MCP is much better than BSA. The parallel versions of the two algorithms perform very differently. When the number of PPEs increases, the scheduling lengths produced by HPMCP increases only slightly, but that of PBSA increases significantly. Figure 9 compares HPMCP and PBSA by the sum of scheduling lengths of four graphs of 1,000, 2,000, 3,000, and 4,000 nodes for different CCRs and different number of PPEs.

Table XIII: The scheduling lengths produced by HPMCP and PBSA

CCR	Number of PPEs	Graph size (number of nodes)							
		1000		2000		3000		4000	
		HPMCP	PBSA	HPMCP	PBSA	HPMCP	PBSA	HPMCP	PBSA
0.1	1	2518	2532	4985	4995	7574	7602	9914	9922
	2	2530	2658	4994	5010	7587	7728	9930	10255
	4	2527	2893	4995	6025	7598	7816	9936	10247
	8	2542	3417	5006	6114	7603	8046	9957	11174
	16	2565	3680	5018	7872	7630	8468	9981	12532
1	1	2585	2595	5096	5128	7466	7459	10107	10106
	2	2593	2724	5096	5217	7476	7484	10118	10174
	4	2593	2933	5113	5380	7486	7850	10139	10893
	8	2641	3089	5142	5905	7500	8323	10169	11049
	16	2663	3218	5169	6265	7516	9669	10193	12115
10	1	4007	4666	8135	10342	11060	14488	16294	21867
	2	4064	4681	8214	10300	11102	14527	16257	21858
	4	4222	4955	8248	11172	11183	14426	15875	22155
	8	4361	5261	8416	12123	11359	15651	16410	24527
	16	4513	5516	8839	13442	11660	16022	16850	24161

7 Concluding Remarks

Parallel scheduling is faster and is able to schedule large macro dataflow graphs. Parallel scheduling is a new approach 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. Both VPMCP and HPMCP algorithms are much faster then PBSA. They produce high-quality scheduling in terms of the scheduling length.

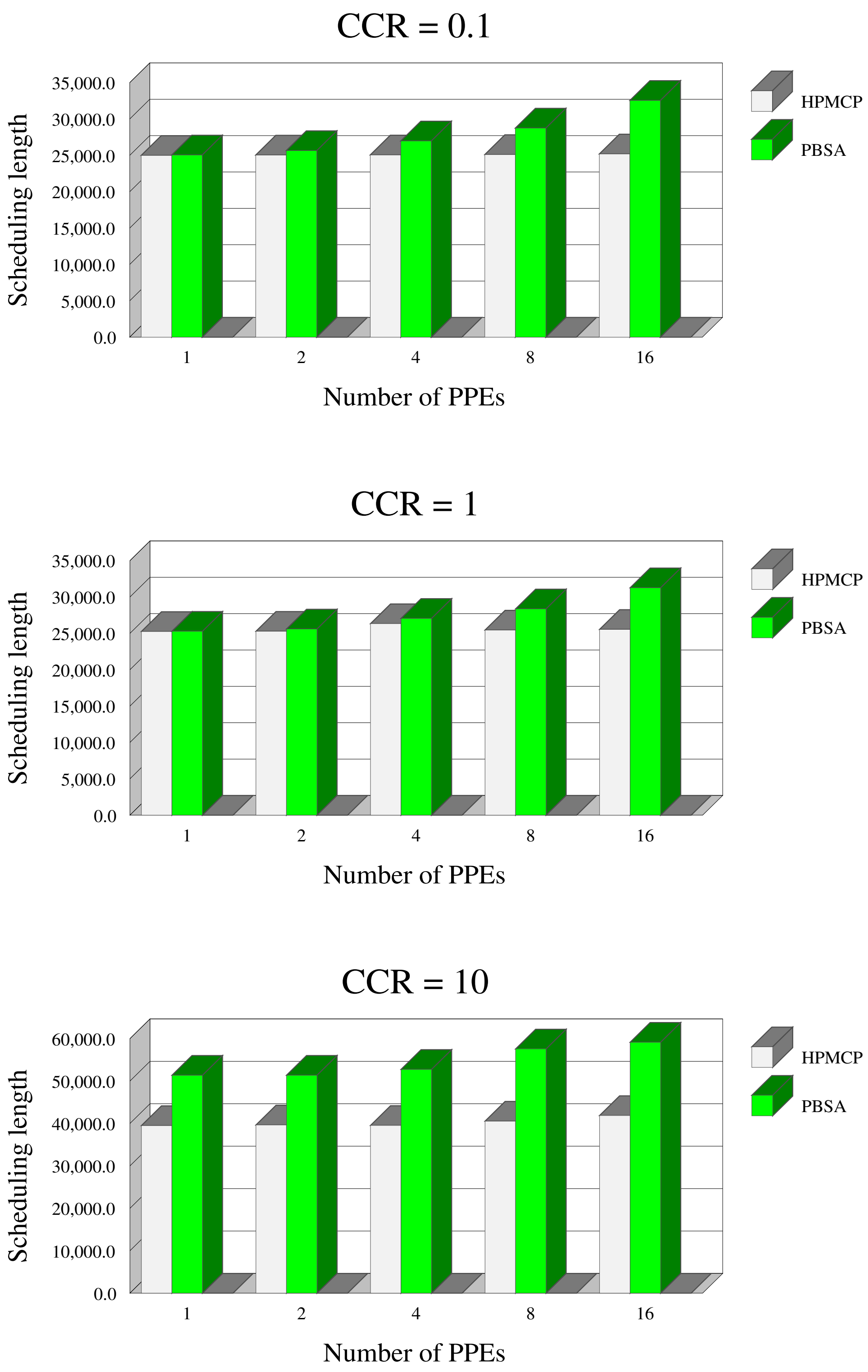

Figure 9: Comparison of scheduling lengths produced by HPMCP and PBSA

Acknowledgments

We are very grateful to Yu-Kwong Kwok and Ishfaq Ahmad for providing their PBSA program and random graph generator for testing.

References

[1] I. Ahmad and Y.K. Kwok. A parallel approach to multiprocessor scheduling. In *Int'l Parallel Processing Symposium*, pages 289–293, April 1995.

[2] I. Ahmad, Y.K. Kwok, and M.Y. Wu. Performance comparison of algorithms for static scheduling of DAGs to multiprocessors. In *Second Australasian Conference on Parallel and Real-time Systems*, September 1995.

[3] J. Baxter and J. H. Patel. The LAST algorithm: A heuristics-based static task allocation algorithm. In *Int'l Conf. on Parallel Processing*, volume II, pages 217–222, August 1989.

[4] Y.C. Chung and S. Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In *Supercomputer '92*, November 1992.

[5] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, June 1990.

[6] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, 1994.

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

[8] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 18(2):244–257, April 1989.

[9] A.A. Khan, C.L. McCreary, and M.S. Jones. A comparison of multiprocessor scheduling heuristics. In *Int'l Conf. on Parallel Processing*, volume II, pages 243–250, August 1994.

[10] B. Kruatrachue and T.G. Lewis. Duplication scheduling heuristics (dsh): A new precedence task scheduler for parallel processor systems. Technical report, Oregon State University, Corvallis, 1987.

[11] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, 1989.

[12] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel and Distributed System*, 4(2):175–187, February 1993.

[13] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel and Distributed Systems*, 1(3):330–343, July 1990.

[14] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel and Distributed System*, 5(9):951–967, September 1994.