A Parallel Approach for Multiprocessor Scheduling

Ishfaq Ahmad and Yu-Kwong Kwok

Department of Computer Science
The Hong Kong University of Science and Technology, Hong Kong

Abstract[1]

The objective of this research is to propose a low-complexity static scheduling and allocation algorithm for message-passing architectures by considering factors such as communication delays, link contention, message routing and network topology. As opposed to the conventional list-scheduling approach, our technique works by first serializing the task graph and "injecting" all the tasks to one processor. The parallel tasks are then 'bubbled up' to other processors and are inserted at appropriate time slots. The edges among the tasks are also scheduled by treating communication links between the processors as resources. The proposed approach takes into account the link contention and underlying communication routing strategy, and can self-adjust on regular as well as arbitrary network topologies. To reduce the complexity, our scheduling algorithm is itself parallelized. To our knowledge, this is the first attempt in designing a parallel algorithm for scheduling. The proposed approach implemented on an iPSC/860 hypercube, while yielding a high speedup in its execution, performs considerably better under a wide range of parameters including the task graph size, communication-to-computation ratio, and the target system topology. Comparisons are made with two other approaches.

1 Introduction

Scheduling of parallel programs represented by directed acyclic graphs (DAG) is an NP-complete problem in its general forms [4]. As a result, there has been a considerable research effort in designing efficient heuristic algorithms. Various heuristics using techniques such as list scheduling [6], [8], [11], [12], critical path methods [1], [2], [8], [10], clustering [5], [9], [11], [14], etc., have been proposed showing satisfactory performance. From a practical standpoint, however, there are two fundamental issues that need to be addressed: (i) does the heuristic make realistic assumptions and is it sophisticated enough to capture the architectural details of the system? and (ii) does the complexity of the heuristic permit it to be practically used for scheduling large task graphs?

The first question relates to the assumptions made by the scheduling algorithm about the program tasks and architecture models. Earlier scheduling heuristics [1], [2], [7] made simplifying assumptions such as equal times for all the nodes in the task graph, and ignoring the communication delays among tasks. The second question which is related to the complexity of the heuristic is an important consideration. In order to be of practical use, a scheduling algorithm must have low complexity. Most previous algorithms are evaluated by applying them on task graphs with a small number of nodes. In practice, a scheduling algorithm may be required to schedule task graphs with hundreds or thousands of nodes. Since most algorithms have complexity of $O(N^3)$ to $O(N^4)$, scheduling of graphs with a large number of nodes can take hours on a serial machine (for example, see Table 3 in Section 3).

In this paper, we present an algorithm that uses realistic assumptions such as arbitrary communication and computation costs in the task graph, performs scheduling and mapping, and takes into account link contention and communication routing strategy. There have been a few algorithms which meet all of the scheduling and mapping objectives mentioned above. Two such reported algorithms are the MH (Mapping Heuristic) proposed in [3], and the DLS (Dynamic Level Scheduling) proposed in [13].

The proposed algorithm can be used for any network topology and can adjust itself accordingly. The main feature of the proposed algorithm is that it is itself a parallel algorithm.

2 The Proposed Approach

For the purpose of presenting the philosophy behind it, we first describe its serial version which is based on a new technique that eventually leads to the parallel algorithm.

2.1 The Serial Algorithm

Before describing the serial BSA algorithm, we define some attributes and symbols which will be used in the subsequent discussion.

A parallel program can be represented by a directed acyclic graph $G = (V, E)$, where V is the set of nodes representing tasks and E is the set of edges representing communication messages ($|V| = v$ and $|E| = e$). Associated with each node n_i is a number indicating the amount of computation required, denoted by $w(n_i)$. Associated with each edge is a number indicating the amount of communication data from one task to another, denoted by c_{ij}. A node on the critical path is called *Critical Path Node (CPN)*. An *In-Branch node* (IBN) is a node, which is not a CPN, and from which there is a path reaching a CPN. An *Out-Branch Node* (OBN) is a node, which is neither a CPN nor an IBN. 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. If node n_i is scheduled to processor J, $ST(n_i, J)$ and $FT(n_i, J)$ denote the start time and finish time of n_i on processor J, respectively. It should be noted that $FT(n_i, J) = ST(n_i, J) + w(n_i)$. After all the nodes have been scheduled, the schedule length is defined as $max_i\{FT(n_i, J)\}$ across all processors. The goal of a scheduling algorithm is to minimize the schedule length for a given task graph.

Some additional attributes used in the BSA algorithm are described below.

- *EMST*: the earliest start time of a message on a link. This is computed for the message by scanning through the link to find the earliest idle time slot that can accommodate it.

1. This research was supported by Hong Kong Research Grants Council under contract number HKUST179/93E.

- *DAT*: the earliest possible data available time of a node. This is computed for the node by determining the maximum *EMST* among all the messages from its parent node.
- *ST*: the earliest possible start time of a node. This is computed for the node by taking the larger value among its *DAT* and the earliest large enough idle time slot on the processor.
- *VIP*: the *very important parent* of a node. It is a parent node that sends the data that arrives last.
- *Proc*: the processor holding a node.

For scheduling, we can construct an order of scheduling nodes in which the CPNs always get higher priorities. The rationale is that the CPNs are the nodes that potentially determine the final schedule length and should be considered for scheduling first so that they can occupy earlier time slots in the processors. We call this ordering method the *CPN-first ordering*. It is described below.

1) Beginning from the entry node of the CP, consider one CPN at a time. If all the IBNs reaching it have been scheduled, schedule the CPN to the most suitable processor; otherwise first schedule all the IBNs reaching it.
2) After all CPNs as well as IBNs are considered, the OBNs can be scheduled in topological order.

The BSA algorithm is then formalized below.

The BSA Algorithm:

```
(1)  Load processor topology and input task graph
(2)  Build_processor_list()
(3)  Serial_injection()
(4)  while Processor_list_not_empty do
(5)    Pivot_TPE ← first processor of Processor_list
(6)    for each $n_i$ on Pivot_TPE do
(7)      if $ST(n_i$ , Pivot_TPE) > DAT($n_i$, Pivot_TPE) or
         Proc(VIP($n_i$)) ≠ Pivot_TPE then
(8)        for each adjacent processor TPE' do
(9)          Determine DAT($n_i$, TPE')
(10)         Determine ST($n_i$, TPE')
(11)       end for
(12)       if there exists TPE' such that ST($n_i$, TPE') < ST($n_i$,
           Pivot_TPE) and migrating $n_i$ will not delay the ST of
           any of its succeeding CPN then
(13)         Migrate $n_i$ from Pivot_TPE to TPE'
(14)         Update start times of nodes and messages
(15)       else if ST($n_i$, TPE') ≥ ST($n_i$, Pivot_TPE) and
           Proc(VIP($n_i$)) = TPE' and migrating $n_i$ will not
           delay the ST of any of its succeeding CPN then
(16)         Migrate $n_i$ from Pivot_TPE to TPE'
(17)         Update start times of nodes and messages
(18)       end if
(19)     end if
(20)   end for
(21) end while
```

The BSA algorithm constructs a processor list in a breadth-first order from the processor having the highest degree (i.e., the one with the largest number of links). This processor is called the *Pivot_TPE* (pivot target processor). The BSA algorithm then constructs a schedule incrementally by first injecting all the nodes to the pivot processor. Then, it tries to improve the start time of each node (hence "bubbling" up nodes) by migrating it to the adjacent processors of the pivot processor if the migration can improve the start time of the node. After a node is migrated from the *Pivot_TPE* to another processor, not only is the node itself "bubbled up" but its successors is also moved with it. This is because after a node is migrated, the space occupied by it on the *Pivot_TPE* is released which can be used for its successor nodes on the *Pivot_TPE*. If a node can start at its *DAT* but its *VIP* is not resident on the pivot processor, it is still a candidate to be transferred. This is because if it can be transferred to the processor accommodating its *VIP*, its start time may further reduce. After all the nodes on the pivot processor are considered, the algorithm selects the next processor in the processor list to be the new pivot processor. This process is repeated until all the processors in the processor list have been considered.

The time complexity of the BSA algorithm is derived as follows. *Build_processor_list()* takes $O(p^2)$ *time* whereas *Serial_injection()* takes $O(v^2)$ time. Thus, the dominant step is the while loop from step (4) to step (21). In this loop, it takes $O(e)$ time to compute the *ST* and *DAT* values of the node on each adjacent processor. If migration is done, it also takes $O(e)$ time. Since there are $O(v)$ nodes on the *Pivot_TPE* and $O(p)$ adjacent processor, each iteration of the while loop takes $O(pev)$ time. Thus, the BSA algorithm takes $O(p^2ev)$ time.

2.2 The Parallel Algorithm

In this section, we describe the proposed Parallel BSA (PBSA) algorithm. In the following, we will call the processors which execute the PBSA algorithm the physical processing elements (PPEs) in order to distinguish them from the target processing elements (TPEs) to which the task graph is to be scheduled.

In the PBSA algorithm, we first partition the task graph according to the number of PPEs available. This is done after the serial injection process. Each partition of the task graph is then scheduled to the target system independently. After all the partitions are scheduled, the independently developed schedules are concatenated. The PBSA algorithm is written in a host-node programming style. The host PPE is responsible for all pre-scheduling and post-scheduling house keeping work. This includes the serial injection process, the task graph partitioning process, the concatenation of partial schedules and resolving any conflicts in partial schedules. All of the parallel PPEs concurrently schedule the partitions of the task graph assigned to them.

Due to the dependencies between the nodes of two adjacent partitions, each PPE needs the information about such dependencies in the scheduling process. For example, each node must know the finish time of a parent node belonging to another partition, called the *remote parent node* (RPN), so that it can determine its own earliest start time. In order to enable all the nodes in different partitions to know the finish times of their RPNs, a global information exchange among the PPEs is required. However, this can generate excessive amount of communication overhead. In the PBSA algorithm, only estimated information is available to each PPE so that inter-PPE communication is minimized. These estimates are given in the following definitions.

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

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

It is obvious that no node can start earlier than its EPST, and no node can be scheduled to start later than its LPST. An RPN can be scheduled to start at any time between the two extremes. Thus, the problem is to pick an accurate estimate for a parent node's start time from all values between the two extremes. Our approach is to take EPST for a parent node if it is a CPN; otherwise, the estimated start time will be EPST plus a fraction of the difference between LPST and EPST. The size of the fraction is determined by the relative importance of the parent node over a CPN. We formalize this in the following definition.

Definition 3: *The estimated start time (EST) of an RPN is given by* $\alpha\text{EPST} + (1-\alpha)\text{LPST}$, *where* α *is the importance factor and is equal to 1 if the RPN is a CPN; otherwise, it is equal to the length of the longest path from an entry node through the RPN to an exit node divided by the length of the CP. Here, length of a path is the sum of the computation and communication costs along the path.*

From the above definition, the *importance factor* α is always bounded above by 1. We measure the relative importance of an RPN by examining the length of the longest path passing through it, which may consist of large communication costs, large computation costs or both. Such a long path can potentially determine the schedule length. Therefore, the nodes lying on it are likely to be scheduled to the same TPE and as a result, to be scheduled at their EPSTs.

Given the estimated start time of an RPN, we still need to know on which TPE the RPN is scheduled. This is essential in determining the DAT of a node to be scheduled and, in turn, in choosing the most suitable TPE for the node. To estimate it, if the RPN is a CPN, then we assume that it will be scheduled to the same TPE as the highest level CPN in the local partition; otherwise, we just randomly pick one TPE to be the one to which the RPN is scheduled. We call this TPE of an RPN the *estimated* TPE (ETPE). It should be noted that both EST and ETPE of any RPN can be statically determined by the host program after the graph partitioning process. Using the above methods to obtain the estimated information about the RPNs of a partition, the node program of PBSA is formalized below.

PBSA_Node():

```
(1) Receive the target processor network from PBSA_Host().
(2) Receive graph partition together with the RPN's
    information (i.e., estimated start times and TPEs) from
    PBSA_Host().
(3) Apply the serial BSA algorithm to the graph partition. For
    every RPN, its EST and ETPE are used for determining the
    DAT of a node to be scheduled in the local partition.
(4) Send the resulting sub-schedule to PBSA_Host().
```

Suppose that there are m nodes in the local partition of *PBSA_Node()*. As step (3) in *PBSA_Node()* is the dominant step, the complexity of *PBSA_Node()* is $O(p^2e'm)$, where e' is the number of edges in the local partition.

After all the *PBSA_Node()* processes finish, the host program constructs the resulting schedule from all the sub-schedules by resolving the conflicts among them. Essentially, the host program concatenates the one sub-schedule after another in such a way that the resulting schedule is as short as possible. Since the serial part of the PBSA algorithm should not dominate, two methods are used in the host program for the concatenation of sub-schedules.

First, for every sub-schedule, the earliest node among all the TPEs is determined. Call this node the *leader* node and the TPE to which the leader node is scheduled the *leader* TPE. The leader node, together with all its succeeding nodes on the leader TPE, are concatenated to a TPE of the previous sub-schedule such that the start time of the leader node is as early as possible. Such TPE is called the leader TPE image. Then, the nodes on the neighboring TPEs of the leader TPE are concatenated to a neighbor TPE of the leader TPE image in the previous sub-schedule. This is done to all other remaining TPEs in a breadth-first order. In the concatenation process, nodes may need to be pushed down because of the scheduling of the inter-partition communication messages.

Second, after a current sub-schedule is merged with the previous sub-schedule, all exit nodes in the current sub-schedule (i.e., the nodes with no successors in the current graph partition) are considered for re-scheduling. For every such exit node, all the TPEs are examined and the node will be re-scheduled to the one which allows the minimum start time.

We formalize these methods in the following host program procedure *Concat_Schedules()*.

Concat_Schedules():

```
(1) for every pair of adjacent sub-schedules, do
(2)     Determine the earliest node in the latter sub-schedule.
        Call this the leader node. Call its TPE the leader TPE.
(3)     Concatenate all nodes, which are scheduled on the same
        TPE as the leader node, to a TPE in the former sub-
        schedule so that the leader node can start as early as
        possible.
(4)     Concatenate the nodes on all other TPEs to the TPEs of
        the former sub-schedule in a breadth-first order
        beginning from the neighbors of the leader TPE.
(5)     Re-schedule the exit nodes in the latter sub-schedule so
        that they can start as early as possible.
(6)     Walk through the whole concatenated schedule to resolve
        any conflict between the actual start times and the
        estimated start times.
(7) end for
```

Suppose that there are at most m nodes in every sub-schedule. The complexity of *Concat_Schedules()* is then $O(Pm^2)$, where P is the number of PPEs (i.e., the number of sub-schedules). This is because steps (2) and (5) take $O(m)$ time; while steps (3), (4) and (6) take $O(m^2)$ time. With *Concat_Schedules()*, the PBSA algorithm can then be formalized below.

PBSA_Host():

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

If there are P PPEs, the maximum size m of each partition will then be $\lceil v/P \rceil$. The dominant steps in *PBSA_Host()* are steps (6) and (7). As described above, step (6) takes $O(p^2e'm)$ and step (7) takes $O(Pm^2)$. The complexity of *PBSA_Host()* is $O(p^2e'm + Pm^2)$.

3 Performance Results

In this section, we present the experimental results and performance comparisons. We implemented both the BSA and PBSA algorithms as well as the MH [3] and the DLS algorithms [13]. The parallel PBSA algorithms was implemented on an iPSC/860 hypercube at Syracuse University. The three serial algorithms were also implemented on a single processor of the iPSC/860.

We first compared the performance of the BSA algorithm with the MH and DLS algorithms, and then compared the BSA and PBSA algorithms. Two performance parameters were used: the schedule length produced by the algorithm, and the running time of the algorithm.

The workload for the testing purpose consisted of random graphs of various sizes. We selected 3 values of CCR which were 0.1, 1.0, and 10.0. The weights on the nodes and edges were generated randomly such that the average value of CCR corresponded to 0.1, 1.0 or 10.0.

In our first experiment, we compared the schedules produced by the MH, DLS and BSA algorithms for a 500-node random task graph. Eleven different target system topologies were selected representing various combinations of processors and communication links. These results are shown in Table 1 for 3 different values of CCR. For each value of CCR, there are 3 columns. The first column shows the ratios of the schedule lengths produced by the MH algorithm to those of the DLS algorithm, the second column shows the ratios of the schedule lengths produced by the BSA algorithm to those of the DLS algorithm, and the third column shows the ratios of the schedule lengths produced by the BSA algorithm to those of the MH algorithm.

Table 1: A relative comparison of MH, DLS and BSA algorithms for a 500-node random task graph on various topologies

	CCR = 0.1			CCR = 1.0			CCR = 10.0		
Topology	MH/ DLS	BSA/ DLS	BSA/ MH	MH/ DLS	BSA/ DLS	BSA/ MH	MH/ DLS	BSA/ DLS	BSA/ MH
2 × 1	1.00	0.95	0.94	0.92	0.90	0.98	0.91	0.72	0.76
2 × 2	0.85	0.67	0.73	0.83	0.76	0.87	1.08	0.91	0.86
4 node fully conn.	1.05	0.98	0.94	0.82	0.84	1.05	0.62	0.64	1.16
8 node hypercube	1.32	0.71	0.65	0.99	0.88	0.89	0.81	0.69	0.77
4 × 2 mesh	1.49	0.67	0.61	0.79	0.77	0.96	0.73	0.73	0.99
8 node ring	1.10	0.65	0.63	0.99	0.97	0.98	0.68	0.65	0.90
8 node fully conn.	0.92	0.90	0.97	0.92	0.91	0.99	0.78	0.93	1.14
16 node hypercube	1.37	0.68	0.63	0.80	0.82	1.06	0.83	0.62	0.65
4 × 4 torus	1.35	0.68	0.63	0.89	0.69	0.73	0.71	0.70	0.98
16 node ring	0.97	0.60	0.61	1.26	0.76	0.70	0.62	0.62	1.02
16 node fully conn.	0.94	0.89	0.94	0.89	0.91	1.04	1.00	0.84	0.84

Out of the 66 comparisons shown in this table, there were only 6 cases in which BSA performed worse than MH (these are indicated by the ratios greater than 1). No case of BSA performing worse than DLS was observed. The schedule lengths produced by BSA were about 60-70% of those of DLS in most cases. DLS was shown to be in general better than MH when CCR was low while MH performed better than DLS when CCR was high. The proposed BSA algorithm was better than both the MH and DLS algorithms in general when CCR was low and in particular when CCR was high.

Next, we considered relatively larger task graphs by varying the number of nodes from 200 to 2000 with increments of 200. Here, two topologies were chosen: a 2 by 2 mesh and a 4 by 4 mesh. The results provided in Table 2 are the ratios of schedule lengths by MH to those of DLS, ratios of schedule lengths by BSA to those of DLS and ratios of schedule lengths by BSA to those of MH. As can be noticed from this table, there was no effect of task graph size on the relative performance of the three algorithms, and BSA is shown to be better than both the MH and DLS algorithms.

Table 2: A relative comparison of the schedule lengths produced by MH, DLS and BSA algorithms for random task graphs of various sizes on two topologies.

	2 x 2 mesh			4 x 4 mesh		
Graphs Size	MH/ DLS	BSA/ DLS	BSA/ MH	MH/ DLS	BSA/ DLS	BSA/ MH
200	0.93	0.80	0.84	0.86	0.78	0.87
400	0.88	0.79	0.87	0.87	0.79	0.88
600	0.87	0.78	0.87	0.93	0.79	0.84
800	0.91	0.79	0.84	0.88	0.78	0.85
1000	0.88	0.80	0.87	0.89	0.78	0.85
1200	0.88	0.79	0.87	0.86	0.78	0.87
1400	0.91	0.78	0.84	0.88	0.80	0.87
1600	0.88	0.80	0.87	0.93	0.80	0.84
1800	0.89	0.79	0.85	0.91	0.79	0.85
2000	0.88	0.78	0.85	0.89	0.78	0.85

For the same set of experiments, the running times of the DLS, MH and BSA algorithms are provided in Table 3. This table provides the exact times (in seconds) for running these serial algorithms on a single node of the iPSC/860 hypercube. As can be seen from this table, the running times of these algorithms approached thousands of seconds for large task graphs when the number of nodes was more than 800. The running times were also higher for 4 by 4 mesh as compared to those of 2 by 2 mesh. The results indicate that MH was about 30% faster than DLS, BSA was about 20% faster than DLS and roughly 40% slower than MH.

Table 3: Running times (in seconds) of MH, DLS and BSA algorithms for random task graphs of various sizes on two topologies

	2 x 2 mesh			4 x 4 mesh		
Graphs Size	DLS	MH	BSA	DLS	MH	BSA
200	11.9	7.1	9.3	13.9	8.1	10.9
400	158.9	93.3	126.1	182.7	112.5	146.1
600	222.6	125.0	171.2	258.2	153.3	201.7
800	565.3	338.29	428.2	598.6	373.5	478.9
1000	1617.7	908.4	1244.4	1584.2	985.4	1247.4
1200	1781.2	1074.4	1413.7	1809.0	1031.7	1413.3
1400	3139.7	1888.7	2452.9	3316.0	2122.2	2652.8
1600	6834.9	4409.6	5512.0	7139.4	4340.8	5711.5
1800	8149.8	4738.3	6317.7	8339.0	4682.7	6414.6
2000	9409.7	5762.6	7294.4	10002.7	6155.5	7694.4

Table 4: The ratios of the schedule lengths produced by the PBSA algorithm to those of the BSA algorithm for various graph sizes on two topologies.

	2 x 2 mesh				4 x 4 mesh			
Graphs Size	2 PPEs	4 PPEs	8 PPEs	16 PPEs	2 PPEs	4 PPEs	8 PPEs	16 PPEs
200	1.08	1.10	1.13	1.16	1.07	1.09	1.09	1.12
400	1.10	1.11	1.13	1.13	1.09	1.11	1.11	1.12
600	1.07	1.07	1.08	1.10	1.08	1.11	1.13	1.13
800	1.09	1.12	1.15	1.17	1.09	1.11	1.13	1.13
1000	1.07	1.08	1.11	1.12	1.07	1.07	1.07	1.09
1200	1.06	1.07	1.09	1.10	1.07	1.08	1.09	1.10
1400	1.06	1.09	1.12	1.12	1.08	1.10	1.10	1.13
1600	1.09	1.10	1.13	1.14	1.06	1.08	1.10	1.13
1800	1.07	1.07	1.09	1.12	1.08	1.08	1.10	1.12
2000	1.06	1.07	1.09	1.11	1.09	1.10	1.12	1.14

Next, we examine the performance of the proposed parallel PBSA algorithm by making a comparison with the

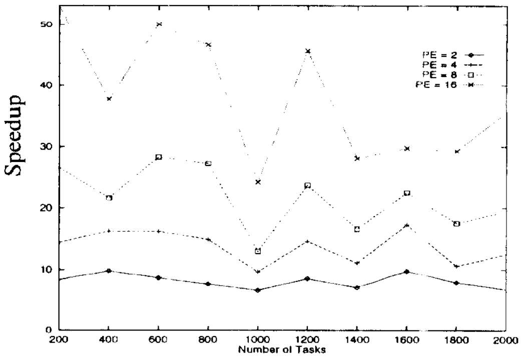
BSA algorithm. The results shown in Table 4 are the ratios of the schedule lengths produced by the PBSA algorithm to the schedule lengths produced by the BSA algorithm. This was done by running the PBSA algorithm on 2, 4, 8 and 16 processors on the iPSC/860 and taking ratios of the schedule lengths produced by it to those of BSA running on one processor. The slight deterioration in performance of PBSA is primarily due to the simple procedure of merging the partial schedules. However, it can be noticed that, in more than half of the cases, the schedule lengths produced by PBSA were within 10% of those produced by BSA. The topology of the target architecture did not seem to have any bearing on this observation.

The performance of PBSA was also compared with MH and DLS, for which the results are not provided here due to lack of space. There was no single case in which PBSA performed worse than either MH or DLS. Using smaller number of PPEs, the schedule lengths generated by PBSA were roughly 90% of those produced by MH and about 80% of those produced by DLS. Using a larger number of PPEs, the schedule lengths produced by PBSA slightly increased due to some inaccuracy in the global information exchange.

The speedup in the running times of PBSA over BSA using 2, 4, 8 and 16 processors with various sizes of task graphs are plotted in Figure 1, when target topologies were 2 by 2 mesh and 4 by 4 mesh, respectively. Note that these speedups were obtained by comparing the running times of parallel PBSA with the serial BSA and not comparing parallel PBSA with serial PBSA (by running it on one processor). The plots indicate that the parallel PBSA on 2 processors was about 6 to 10 times faster than the serial BSA. By using more PPEs, the speedup increased almost linearly. With 16 PPEs, the speedup was sometimes more than 50. Due to the availability of only 16-node hypercube, we could test our algorithm on at most 16 PPEs. But we expect the speedup to further increase on 32 and 64 PPEs.

4 Conclusions

We presented a scheduling approach that drastically reduces the running time of the algorithm through parallelization. Comparisons with two related algorithms indicated that our proposed serial and parallel algorithms perform better under a wide range of parameters. We have observed substantial speedup when the parallel algorithm is implemented on the iPSC/860 hypercube. Some degradation in the performance of PBSA as compared to the serial BSA is due to the estimation of the start times of nodes assigned to other PPEs. Further improvement may be possible by dealing with this problem through better information exchange.



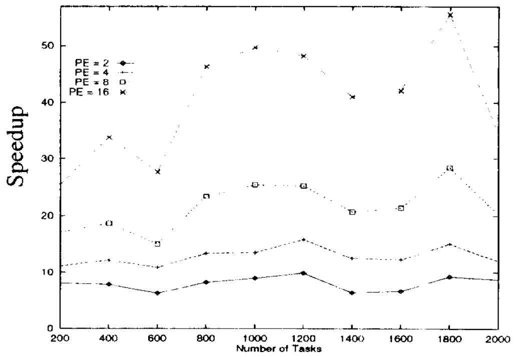


Figure 1: (Upper) The speedup in the running times of PBSA over BSA for a 2 x 2 mesh target architecture; (lower) target architecture is a 4 x 4 mesh.

References

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

[2] E.G. Coffman, *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.

[3] H. El-Rewini and T. Lewis, "Scheduling Parallel Programs onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 138-153, Jun. 1990.

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

[5] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAG's on multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 276-291, Dec. 1992.

[6] D.S. Hochbaum and D.B. Shmoys, "Using Dual Approximation Algorithms for Scheduling Problems: Theoretical and Practical Results," *Journal of the ACM*, 34(1), pp. 144-162, Jan. 1987.

[7] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Oper. Research*, vol. 19, no. 6, pp. 841-848, Nov. 1961.

[8] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. on Computers*, vol. C-33, pp. 1023-1029, Nov. 1984.

[9] A.A. Khan, C.L. McCreary and M.S. Jones, "A Comparison of Multiprocessor Scheduling Heuristics," Proc. *of Int'l Conf. on Parallel Processing*, Aug. 1994, vol. II, pp. 243-250.

[10] W.H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Trans. on Computers*, vol. C-24, pp. 1235-1238, Dec. 1975.

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

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

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

[14] T. Yang and A. Gerasoulis, "A Fast Static Scheduling Algorithm for DAGs on a unbounded Number of Processors," proceedings of *Supercomputing'91*, pp. 633-642, Nov. 1991.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE