Bubble Scheduling: A Quasi Dynamic Algorithm for Static Allocation of Tasks to Parallel Architectures

Yu-Kwong Kwok and Ishfaq Ahmad

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

Abstract[1]

We propose an algorithm for scheduling and allocation of parallel programs to message-passing architectures. The algorithm considers arbitrary computation and communication costs, arbitrary network topology, link contention and underlying communication routing strategy. While our technique is static, the algorithm is quasi dynamic because it is not specific to any particular system topology and thus can be used at run-time for the processor configuration avaiable at that time. The proposed algorithm, called Bubble Scheduling and Allocation (BSA) algorithm, *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 scheduling of messages on the links depends on the routing strategy, such as circuit switching and wormhole routing, of the underlying network. The proposed algorithm has admissible time complexity and is suitable for regular as well as irregular task graph structures.*

1 Introduction

The main objective of a static scheduling algorithm is to minimize the total execution time of a parallel program. Since scheduling a set of related tasks to a set of processors is known to be NP-complete problem in its most variants [4], [6], it is solved with a variety of heuristics. Though heuristics perform adequately with different assumptions, there are three fundamental questions to ask in scheduling: (1) does the heuristic make realistic assumptions? (2) is it sophisticated enough to capture the architectural details of the system? and (3) does the complexity of the heuristic permit it to be practically used for scheduling of large task graphs?

The first question relates to the assumptions made by the scheduling algorithm about the task and architecture models. As elaborated in the next section, earlier scheduling heuristics made simple assumptions: equal times for all the tasks in the task graph, simple graph structure such as trees or fork-joins, or no communication delays among tasks. The second question is concerned with the optimization of the scheduling strategy with respect to the target architecture. A scheduling algorithm tailored for one particular architecture may not generate efficient solutions on another architecture. The architectural attributes such as system topology, routing strategy, overlapped communication and computation, etc., if taken into account, can result in significantly different allocation decisions. The third question which is related to the complexity of the heuristic is an important consideration. A number of reported scheduling algorithms exhibit good performance by considering only a set of small task graphs. Such algorithms do not carry enough potential to be used for practical purpose.

The purpose of this paper is to propose an algorithm that simultaneously considers arbitrary communication and computation costs in the task graph, performs scheduling and mapping, and takes into account link contention and communication routing strategy. The proposed algorithm can be used for any network topology. The algorithm has reasonable complexity and is suitable for regular and irregular graph structures. This paper is organized as follows. The next section describes various approaches for solving this problem. In the same section, three previous closely related algorithms are described. In Section 3, we present the proposed algorithm and discuss some of the principles used in its design. In Section 4, we give an example illustrating the effectiveness of the algorithm. Section 5 presents the experimental results. The last section concludes the paper.

2 Types of DAG Scheduling Algorithms

A parallel program can be represented by a directed acyclic graph (DAG) $G = (V, E)$, where V is a set of v nodes and E is a set of e directed edges. The weight on a node is called the *computation cost* of a node n_i and is denoted by $w(n_i)$. The edges in the parallel program graph correspond to the communication messages and precedence constraints among the tasks. The weight on an edge is called the *communication cost* of the edge and is denoted by c_{ij}. Here, the subscript ij indicates that the directed edge emerges from the source node n_i and incidents on the destination node n_j. 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. The communication cost among two nodes assigned to the same processor is assumed to be zero. If node n_i is scheduled to processor P, $ST(n_i, P)$ and $FT(n_i, P)$ denote the start time and finish time of n_i on processor P, respectively. It should be noted that $FT(n_i, P) = ST(n_i, P) + w(n_i)$. After all nodes have been scheduled, the schedule length is defined as $max_i\{FT(n_i, P)\}$ across all processors.

The scheduling problem is defined to be an allocation of a set of tasks to a set of processors such that the cumulative schedule length or *makespan* is minimized without violating the precedence constraints among the tasks. However, the problem is NP-complete even in very simple cases [4], [6], [11]. There are only two special cases

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

for which optimal polynomial time algorithms exist: scheduling tree-structured task graphs with identical computation costs on arbitrary number of processors, and scheduling arbitrary task graphs with identical computation costs on two processors [4], [6], [8], [10]. However, even in these cases, no communication is assumed between tasks of the parallel program [1], [10], [11], [15]. It has been shown that scheduling an arbitrary task graph with inter-task communication onto two processors is NP-complete and scheduling a tree-structured task graph with inter-task communication onto many processors is also NP-complete [6]. In general, the algorithms for static scheduling of parallel programs represented by an edge-weighted directed acyclic graph (DAG), also called a task graph or macro-dataflow graph, to a set of homogeneous processors, can be classified into 3 categories:

- ***Bounded Number of Processors (BNP) scheduling:*** These algorithms schedule the DAG to a bounded number of processors directly [4], [7], [9], [10], [12], [19]. These algorithms are limited to fully connected networks and do not pay any attention to link contention or routing strategies used for communication
- ***Unbounded Number of Clusters (UNC) scheduling:*** These algorithms schedule the DAG to an unbounded number of clusters [2], [11], [16], [17], [19], [20]. The processors are assumed to be fully-connected. The technique employed by these algorithms is also called *clustering*.
- ***Arbitrary Processor Network (APN) scheduling:*** These algorithms perform scheduling and mapping on the target architectures in which the processors are connected via a network topology. These algorithms also schedule communication messages on the network channels, using some routing strategy and taking care of the link contention. Three such algorithms are the MH (Mapping Heuristic) algorithm [5], the DLS (Dynamic Level Scheduling) algorithm [18], and the BU (Bottom Up) algorithm [14].

3 The Proposed Algorithm

In this section, we describe the proposed algorithm, named the *Bubble Scheduling and Allocation (BSA) algorithm*. The BSA algorithm is designed with the following objectives.

- It computes node priorities accurately in order that important nodes will be scheduled to occupy earlier time slots.
- It makes an "intelligent" decision when scheduling a node to a processor by taking the effect of the descendant nodes into account.
- It is not designed for a specific topology or any routing strategy, rather it can adjust itself to the target system topology with any underlying routing strategy.
- It has reasonable time complexity and can be parallelized.

Before introducing our algorithm, we define some attributes and symbols which will be used in the subsequent discussion. The symbols and their meanings are shown in Table 1.

Table 1: Some symbols and their meanings.

Symbol	Meaning
n_i	The node number of a node in the parallel program task graph
$w(n_i)$	The computation cost of node n_i
c_{ij}	The communication cost of the directed edge from node n_i to n_j
v	Numer of nodes in the task grpah
e	Number of edges in the task graph
p	Number of processors available
CP	A critical path of the task graph
CPN	Critical Path Node
IBN	In-Branch Node
OBN	Out-Branch Node
$MST(e_x, L)$	The start time of message e_x on the link L
$MFT(e_x, L)$	The finish time of message e_x on the link L
$DAT(n_i,P)$	The possible data available time of n_i on processor P
$ST(n_i, P)$	The start time of node n_i on P
$FT(n_i, P)$	The finish time of node n_i on P
$VIP(n_i)$	The parent node of n_i that sends the data arrive last
$Pivot\_PE$	The processor from which nodes are migrated
$Proc(n_i)$	The processor accommodating node n
CCR	Communication-to-computation Ratio

3.1 Serialization

As discussed earlier, an inaccurate ordering of nodes for scheduling can lead to a poor schedule. This inaccuracy is much more detrimental to scheduling task graphs to an arbitrary network of processors in which link contention has to be considered. In such a network, the communication overhead should be carefully minimized. If relatively less important nodes get scheduled before the more important ones are considered, the early time slots in the communication links as well as the processors are occupied such that subsequently the more important nodes cannot get scheduled to start earlier.

To determine an accurate scheduling order, we have to identify which nodes in a task graph are more important in the sense that timely scheduling of such nodes can lead to a short schedule length. In a task graph, there is a set of nodes and edges that determines the schedule length of the graph on a processor network. This set of nodes is called the critical path (CP) of the task graph. Below is the definition of a CP.

Definition 1: *A Critical Path (CP) of a task graph, is a set of nodes and edges, forming a path from an entry node to an exit node, of which the sum of computation cost and communication cost is the maximum.*

If the nodes on the CP are not timely scheduled, the schedule length would be long because such nodes are on the longest path of the task graph so that their finish times determine the final schedule length. Thus, the CP nodes (CPNs) are the more important nodes in a task graph and should be examined for scheduling as early as possible in the scheduling process. However, we cannot examine all the CPNs at once without considering some other nodes in the intermediate scheduling steps. This is because in the scheduling process, we have to compute the start times of the CPNs on the processors which are determined by the parent nodes of the CPNs due to the precedence constraints. As a result, before we can examine a CPN for scheduling, we have to finish the scheduling of all its parent nodes. In order to formulate a systematic scheduling order in which all the CPNs get scheduled as early as possible, we devise a partitioning of the nodes in a task graph given in the following definition.

Definition 2: *In a connected graph, an In-Branch Node (IBN) is a node, which is not a CPN, and from which there is a path reaching a Critical Path Node (CPN). An Out-Branch Node (OBN) is a node, which is neither a CPN nor an IBN.*

In the above partitioning, the relative importance of nodes are in the order: CPNs, IBNs and OBNs. As discussed above, the CPNs are important because timely scheduling of these nodes can lead to a better schedule. The IBNs are also important because timely scheduling of these nodes can help reducing the start times of the CPNs. The OBNs are relatively less important because they usually do not affect the schedule length. Based on this partitioning, we have the following method of making a sequence of nodes for scheduling.

CPN-First Ordering:

```
(1) Initially, the sequence is empty. Make the entry
    CPN be the first node in the sequence. Set Position
    to 2. Let $n_x$ be the next CPN.
Repeat
(2) If $n_x$ has all its parent nodes in the sequence then
(3)     Put $n_x$ at Position in the sequence and increment
        Position.
(4) else
(5)     Suppose $n_y$ is the parent node of $n_x$ which is not
        in the sequence and has the largest
        communication with $n_x$. If $n_y$ has all its parent
        nodes in the sequence, put $n_y$ at Position in the
        sequence and increment Position. Otherwise,
        recursively make all the ancestor nodes of $n_y$ in
        the sequence so that the nodes with a larger
        communication are considered first.
(6)     Repeat the above step until all the parent nodes
        of $n_x$ are in the sequence. Append $n_x$ to the
        sequence.
(7) endif
(8) Make $n_x$ to be the next CPN.
Until all CPNs are in the sequence.
(9) Append all the OBNs to the sequence in a
    topological order of the task graph.
```

The CPN-First ordering does not violate the prcedence constraints among nodes as the IBNs reaching a CPN are always in front of the CPN in the node sequence. In addition, the OBNs are appended to the sequence in a topological order so that a parent OBN is always in front of a child OBN.

In the scheduling of a task graph to an arbitrary network of processors, message routing has to be considered. In previous approaches, the message routing strategy has to be supplied as a routing table for the scheduler. In our approach, the task graph is first serialized into a sequence of nodes which is then injected into a single processor. This process is called *serial injection.* The sequence of nodes for serial injection is constructed by using the CPN-First ordering described above. The nodes are then incrementally "bubbled-up" by migrating to the adjacent processors in a breadth first order. In order to optimize the scheduling and routing of messages, the processor chosen for serial injection is the one that has the largest number of links. Such a processor is called the *pivot processor.*

3.2 Node Migration and Scheduling

After serializing the task graph into the pivot processor, some of the nodes have to be migrated to other processors in order to improve the schedule length. A candidate to be considered for migration is a node that has its data arrival time (*DAT*) — defined as the time at which the last message from its parent nodes finishes delivery — greater than its current start time on the pivot processor. The goal of the migration is to schedule the node to an earlier time slot on an adjacent processor. Thus, a suitable adjacent processor for the migration is the one that allows the largest reduction of the start time of the node. To determine which adjacent processor can give the largest start time reduction, we have to compute the data available time and the start time of the node on each adjacent processor. The following rule governs the computation of the start time of a node on a processor. The data available time of a node on a processor will be discussed in the next subsection.

Rule I: *A node n_i can be scheduled to a processor P on which m nodes $\{n_{P_1}, n_{P_2}, \dots, n_{P_m}\}$ has been scheduled if there exists some k such that*

$$ST(n_{P_{k+1}}, P) - max\{FT(n_{P_k}, P), DAT(n_i, P)\} \geq w(n_i)$$

where $k = 0, \dots, m$, $ST(n_{P_{m+1}}, P) = \infty$, and $FT(n_{P_0}, P) = 0$. The start time of n_i on processor P is given by $max\{FT(n_{P_l}, P), DAT(n_i, P)\}$ with l being the smallest k satisfying the above inequality.

Intuitively, the above rule states that when determining the start time of a node on a processor, we start from examining the first idle time slot (if any) before the first node on that processor. We check whether the overlapped portion (if any) of the idle time slot and the time period in which the node can start execution is large enough to accommodate the node. If not, we proceed to try the next idle time slot. If there is indeed one such idle time slot, the start time for the node is the earliest one.

3.3 Message Routing and Scheduling

If a node and its parent node are scheduled to the same processor, the message arrival time of this parent node is simply its finish time on the processor because intra-processor communication is assumed to take zero time. On the other hand, if the parent node is scheduled on another processor, the message arrival time depends on how the message is routed and scheduled on the links. The following rule governs the scheduling of a message to a link.

Rule II: *A message $e_x = (n_i, n_j)$ can be scheduled to a link L on which m messages $\{e_1, \dots, e_m\}$ have been scheduled if there exists some k such that*

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

where $k = 0, \dots, m$ and $MST(e_{m+1}, L) = \infty$, $MFT(e_0, L) = 0$. The start time of e_x on L, denoted by $MST(e_x, L)$, is given by $max\{MFT(e_{r+1}, L), FT(n_i, Proc(n_i))\}$ with r being the smallest k satisfying the above inequality.

Intuitively, the above rule states that when determining the start time of a message on a link, we start from examining the first idle time slot (if any) before the first message on that link. We check whether the overlapped portion (if any) of the idle time slot and the time period in which the message can start transmission is large enough to accommodate the message. If not, we proceed to try the next idle time slot. If there is indeed one such idle time slot, the start time for the message is the earliest one.

Using the above rule, we can determine the message start time (and hence arrival time) for each message from the parent nodes to a node under consideration for migration. The data available time of the node on the processor is then simply the largest value of the message arrival times. The parent node that corresponds to this largest value of the message arrival time is called the *Very Important Parent* (VIP). Concerning the routing of messages between two communicating nodes scheduled to different processors, we employ an adaptive approach. Messages are automatically routed in the migration process of nodes from the pivot processors to other processors.

3.4 The BSA Algorithm

Using the techniques discussed above, the BSA algorithm can be formalized below. In the following, the procedure *Build_processor_list()* constructs a list of processors in a breadth-first order from the first pivot processor. The procedure *Serial_injection()* performs the CPN-First ordering of the nodes and injects all the nodes to the first pivot processor.

The Bubble Scheduling and Allocation Algorithm:

```
(1)  Load processor topology and input task graph
(2)  Pivot_PE ← the processor with the highest degree
(3)  Build_processor_list()
(4)  Serial_injection(Pivot_PE)
(5)  while Processor_list_not_empty do
(6)      Pivot_PE ← 1st proc. of Processor_list
(7)      for each $n_i$ on Pivot_PE do
(8)          if ST($n_i$, Pivot_PE) >
                 DAT($n_i$, Pivot_PE) or
                 Proc(VIP($n_i$)) ≠ Pivot_PE then
(9)              Determine DAT and ST of $n_i$ on each
                     adjacent processor PE'
(10)             if there exists a PE' such that ST($n_i$, PE')
                     < ST($n_i$, Pivot_PE) then
(11)                 Migrate $n_i$ from Pivot_PE to PE'
(12)                 Update start times of nodes and
                         messages
(13)             else if ST($n_i$, PE') ≥ ST($n_i$, Pivot_PE) and
                     Proc(VIP($n_i$)) = PE' then
(14)                 Migrate $n_i$ from Pivot_PE to PE'
(15)                 Update start times of nodes and
                         messages
(16)             end if
(17)         end if
(18)     end for
(19) end while
```

The time complexity of the BSA algorithm is derived as follows. The procedure *Build_processor_list()* takes $O(p^2)$ time while Serial_injection() takes $O(v^2)$ time. Thus, the dominant step is the while-loop from step (5) to step (19). In the 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_PE* and *O(p)* adjacent processor, each iteration of the while loop takes *O(pev)* time. Thus, the BSA algorithm takes $O(p^2ev)$ *time.*

While our technique is inherently static, the BSA algorithm is quasi dynamic because it is not specific to any particular system topology and thus can be used at run-time for the processor configuration available at that time. This can be useful in a large system with multiple users — such as Intel Paragon, nCube and CM-5 — where the next job is assigned to the available subset of processors.

4 An Example

In this section, we present an application example to demonstrate the effectiveness of the proposed algorithm. We use a small example task graph shown in Figure 1 and demonstrate the schedule generated by the BSA algorithm. The schedules generated by the other three algorithms mentioned earlier are also presented for comparison.

node	*Static Level*
$^*n_1$	11
n_2	8
n_3	8
n_4	9
n_5	5
n_6	5
$^*n_7$	5
n_8	5
$^*n_9$	1

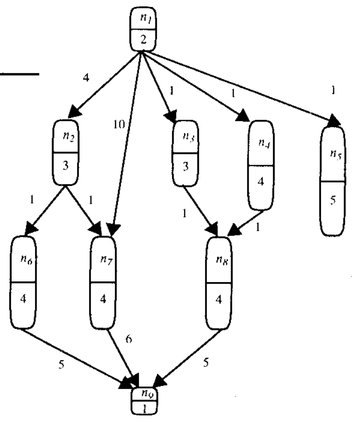

Figure 1: A task graph and the static levels of the nodes.

In Figure 1, the static levels of all the nodes are shown. The CPNs are marked by an asterisk. The target processor network for this task graph is a 4-node ring. The schedule generated by the MH and DLS algorithm is the same and is shown in Figure 2(a). Both algorithm schedules the nodes in the order: n_1, n_4, n_3, n_5, n_2, n_8, n_7, n_6, n_9. As can be seen from the schedule, both the MH and DLS algorithms commit the mistake that they schedule the node n_4 first before the more important nodes n_2 and n_7. The latter two nodes are more important because n_7 is a CPN and n_2 critically affects the start time of n_7. As n_4 has a larger static level, both algorithms examine n_4 first and schedules it to an early time slot on the same processor as n_1. But this decision has negative effect on the subsequent scheduling of n_2 because n_2 cannot be scheduled to an early time slot to start at the earliest possible time — it is the time just after n_1 finishes. This adverse effect propagates and eventually leads to a late start time of n_9 which would have been scheduled to start earlier if n_2 was scheduled to an earlier time slot. This example demonstrates the fact that inaccurate scheduling priorities can lead to very inefficient schedules. The schedule generated by the BU algorithm is shown in Figure 2(b). As

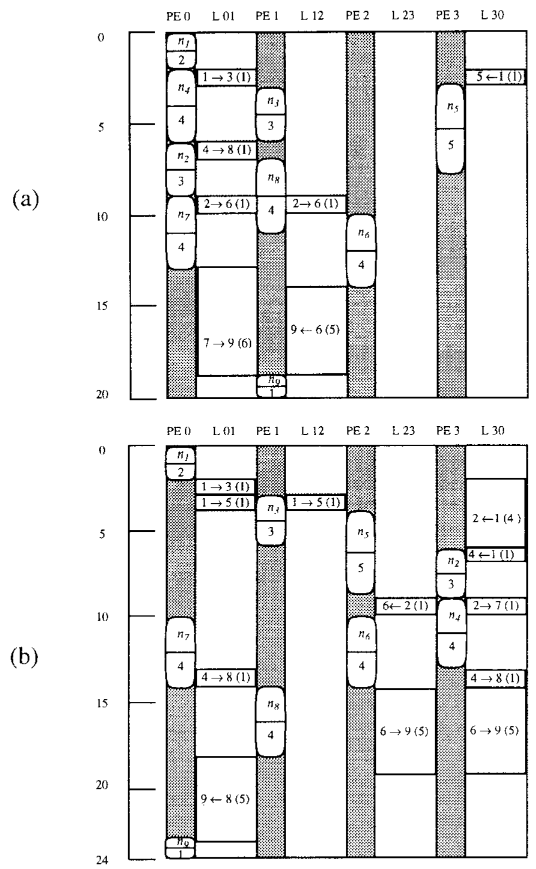

Figure 2: (a) The schedule generated by the MH and DLS algorithm (schedule length = 20, total comm. costs incurred = 16); (b) the schedule generated by the BU algorithm (schedule length = 24, total comm. costs incurred = 27).

can be seen, the schedule length is considerably longer than that of the MH and DLS algorithm. The reason for generating such a low quality schedule is that the BU[1] algorithm employs a processor selection heuristic which works by attempting to balance the load across all the processors. Obviously, such goal usually conflicts with the minimization of the schedule length.

The schedule generated by the BSA algorithm is shown in Figure 3(a). An intermediate schedule is also shown in Figure 3(b). The CPN-First ordering of the task graph is: n_1, n_2, n_7, n_4, n_3, n_8, n_6, n_9, n_5. The nodes are then serialized according to this order and injected into the first pivot processor PE 0. In the first phase, node n_4 is migrated to PE 1 because its start time improves. Similarly, nodes n_3 and n_6 are migrated to PE 3. Then, the only OBN n_5 is also migrated to PE 1 because its start time is earlier on PE 1 than on PE 3. In the second phase, PE 1 is chosen to be the pivot processor but only the OBN n_5 is migrated to PE 2 because all the other nodes are already scheduled to start at their earliest times. This example demonstrates that the

1. In the implementation of the BU algorithm, we have used the PSH2 processor selection heuristic with $\rho = 1.5$. Such a combination is shown [14] to give the best performance.

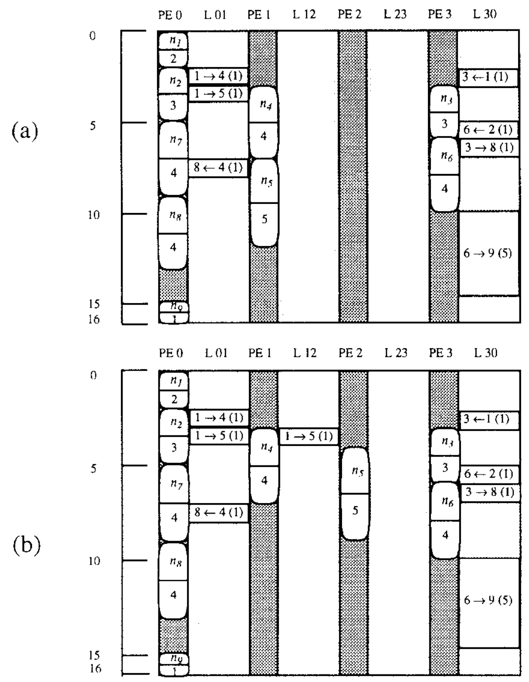

Figure 3: (a) an intermediate schedule generated by the BSA algorithm (schedule length = 16, total comm. costs incurred = 11); (b) the final schedule generated by the BSA algorithm (schedule length = 16, total comm. costs incurred = 12).

CPN-First ordering gives an accurate order for scheduling and the incremental node migration process makes efficient use of communication links to optimize the schedule length.

5 Performance Results

In this section, we present the performance of the proposed algorithm and compare it with the MH, DLS and BU algorithms. For this purpose, we generated two suites of task graph. The first suite consisted of regular graphs while the second suite consisted of irregular graphs. The regular graphs represent a number of parallel algorithms including the mean value analysis [3], Gaussian elimination [19], Laplace equation solver [19], and LU-decomposition [13], which contain regular patterns of nodes and edges. Since these algorithms operate on matrices, the number of nodes and edges in their task graphs depends on the size (N) of the matrix. The number of nodes in the task graph for each algorithm is roughly $O(N^2)$ which is about the same for all algorithms. Within each type of graph, we chose 3 values of CCR which are 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. A value of CCR equal to 0.1 represents a computation-intensive task graph, a value of 10.0 represents a communication-intensive task graph, and a value of 1.0 represents a graph in which computation and communication is equally balanced. The second suite of task graphs consisted of completely random graphs. Again, 3 values of CCR were selected (0.1, 1.0 and 10.0) for each graph size.

In our first experiment, we compared the schedules produced by the BSA algorithm with those of the MH, DLS and BU algorithms for a 500-node task graph representing the mean value analysis algorithm. We used 11 different network topologies. The results of Table 2 show comparisons for 3 different values of CCR. For each value of CCR, there are 3 columns: the first column shows the percentage improvement in the schedule length produced by the BSA algorithm over the MH algorithm, the second column shows the percentage improvement in the schedule length produced by the BSA algorithm over the DLS algorithm, and the third column shows the percentage improvement in the schedule length produced by the BSA algorithm over the BU algorithm.

Table 2: Percentage improvement by BSA over MH, DLS, and BU for a 500-node mean value analysis task graph on various topologies.

	CCR = 0.1			CCR = 1.0			CCR = 10.0		
Topology	BSA/ MH	BSA/ DLS	BSA/ BU	BSA/ MH	BSA/ DLS	BSA/ BU	BSA/ MH	BSA/ DLS	BSA/ BU
2 × 1	0.00	2.39	2.85	21.24	-2.97	25.51	10.43	50.39	59.80
2 × 2	0.52	6.21	6.43	14.04	1.16	16.81	43.31	51.47	55.14
4 node fully conn.	0.00	6.25	7.33	10.47	6.28	11.17	45.78	52.74	60.60
8 node hypercube	0.70	0.46	0.88	7.90	14.67	17.31	17.71	14.26	19.37
4 × 2 mesh	0.49	0.21	0.73	6.23	10.93	14.71	31.51	28.65	39.56
8 node ring	1.50	10.58	13.97	22.49	34.82	40.55	18.44	37.91	42.36
8 node fully conn.	0.00	-1.34	0.00	0.97	0.91	1.32	45.80	52.74	61.12
16 node hypercube	-0.09	-1.37	-0.09	2.63	10.66	11.14	2.05	14.26	16.28
4 × 4 torus	1.59	-0.64	2.11	-0.20	7.11	8.10	49.37	52.48	61.98
16 node ring	0.51	-2.98	0.53	17.97	34.19	39.18	18.44	37.91	38.23
16 node fully conn.	0.00	-0.38	0.00	-0.33	0.91	1.30	45.80	52.74	58.88

From this table we make three major observations. First, the performance of the BSA algorithm was close to the other three algorithms when CCR was equal to 0.1. Overall, the BSA algorithm, however, still yielded some improvement over the other three algorithms. Out of 33 entries in this table, the BSA algorithm was outperformed by the DLS algorithm in only 6 cases. Similarly, the BSA algorithm was outperformed by the MH algorithm in only 3 cases and by the BU algorithm in only 1 case. But, as can be seen, the magnitude of degradation was less than 3% in these case. Second, when the value of CCR was higher, the BSA algorithm performed significantly better than other algorithms. Third, the BSA algorithm yielded a larger improvement over the BU algorithm (up to 61%) as compared to the improvements over the DLS algorithm (up to 52%) and the MH algorithm (49%).

We repeated the same experiment for a 500-node task graph for Gaussian elimination, LU-decomposition, and Laplace equation solver. The experiment was also performed for a 500-node random task graph. The results of these experiments are presented in Table 3 to Table 6. An inspection of Table 3 reveals that, for Gaussian elimination task graph, the proposed BSA algorithm performed better than the MH, DLS, and BU algorithms in all cases. From the results of Laplace equation solver shown in Table 4, we can notice that the BSA algorithm performed better than the DLS algorithm in 26 out of 33 cases. Compared to the MH algorithm, it performed better in 22 out of 33 cases while compared to the BU algorithm, it performed better in 28 out of 33 cases. For the results of LU-decomposition shown in Table 5, it can be seen that the BSA algorithm performed than both the DLS and MH algorithms in all but one cases. For the random task graph, the BSA algorithm improved the schedule length by 40 to

Table 3: Percentage improvement by BSA over MH, DLS, and BU for a 500-node Gaussian elimination task graph on various topologies.

	CCR = 0.1			CCR = 1.0			CCR = 10.0		
Topology	BSA/ MH	BSA/ DLS	BSA/ BU	BSA/ MH	BSA/ DLS	BSA/ BU	BSA/ MH	BSA/ DLS	BSA/ BU
2 × 1	10.39	1.48	14.06	9.00	17.82	18.19	2.36	16.04	21.69
2 × 2	20.21	21.21	24.97	21.95	16.68	23.53	4.23	29.64	31.34
4 node fully conn.	18.48	3.67	24.27	24.85	18.15	29.08	6.64	34.00	36.77
8 node hypercube	22.22	27.64	32.95	27.55	24.80	30.77	6.64	18.89	20.37
4 × 2 mesh	22.84	27.64	35.05	26.86	21.99	35.44	7.64	18.89	21.07
8 node ring	22.89	41.05	42.54	27.71	23.55	31.61	4.23	29.64	38.08
8 node fully conn.	19.67	0.00	20.69	31.36	2.05	36.39	12.78	32.42	33.09
16 node hypercube	13.12	22.20	29.92	30.71	28.71	37.61	6.75	31.34	39.37
4 × 4 torus	13.85	22.43	24.28	32.25	21.70	35.30	6.75	25.98	33.36
16 node ring	22.21	42.64	45.04	27.71	23.55	35.65	4.23	29.64	37.71
16 node fully conn.	5.39	0.00	6.52	37.68	-4.73	45.06	19.64	23.19	30.62

Table 4: Percentage improvement by BSA over MH, DLS, and BU for a 500-node Laplace equation solver task graph on various topologies.

	CCR = 0.1			CCR = 1.0			CCR = 10.0		
Topology	BSA/ MH	BSA/ DLS	BSA/ BU	BSA/ MH	BSA/ DLS	BSA/ BU	BSA/ MH	BSA/ DLS	BSA/ BU
2 × 1	0.03	5.51	5.86	-0.41	4.75	6.02	31.51	38.84	45.82
2 × 2	0.13	7.03	9.93	6.25	1.93	6.57	27.31	46.01	49.86
4 node fully conn.	0.13	5.25	7.30	-0.06	1.93	2.17	47.02	37.70	49.15
8 node hypercube	0.52	12.35	15.17	3.57	-1.45	4.38	41.94	46.01	49.70
4 × 2 mesh	0.52	12.35	14.77	7.03	0.09	9.54	42.82	46.01	55.22
8 node ring	0.52	24.35	27.67	19.05	25.52	30.46	50.48	62.08	63.65
8 node fully conn.	0.54	3.10	3.15	-0.09	-6.43	-0.11	26.06	27.31	28.71
16 node hypercube	-0.34	1.93	2.10	-2.09	-6.89	-2.86	30.50	46.01	55.68
4 × 4 torus	-0.34	5.09	6.98	-2.11	-4.78	-2.81	34.79	46.01	55.65
16 node ring	-0.22	27.78	32.89	-6.63	-9.69	-8.81	24.59	62.08	62.26
16 node fully conn.	0.00	0.00	0.00	-0.89	-6.63	-1.22	26.06	27.31	32.68

Table 5: Percentage improvement by BSA over MH, DLS, and BU for a 500-node LU-decomposition task graph on various topologies.

	CCR = 0.1			CCR = 1.0			CCR = 10.0		
Topology	BSA/ MH	BSA/ DLS	BSA/ BU	BSA/ MH	BSA/ DLS	BSA/ BU	BSA/ MH	BSA/ DLS	BSA/ BU
2 × 1	3.83	3.71	4.04	10.03	6.79	13.29	19.69	20.12	22.27
2 × 2	9.08	31.63	38.71	17.04	27.77	28.67	20.26	27.64	34.06
4 node fully conn.	8.46	10.49	13.80	14.49	10.38	15.60	31.85	18.66	38.28
8 node hypercube	11.60	42.66	48.66	27.56	35.08	42.52	13.50	16.62	19.14
4 × 2 mesh	11.59	42.63	44.31	21.75	34.08	36.02	19.65	16.62	24.97
8 node ring	11.73	54.63	61.11	26.05	28.23	37.07	35.20	42.89	46.46
8 node fully conn.	13.90	19.36	26.12	21.24	16.36	21.91	27.47	9.57	30.13
16 node hypercube	0.71	34.09	34.36	20.77	31.20	37.92	26.20	15.41	32.10
4 × 4 torus	0.71	34.09	35.65	19.83	32.65	38.79	10.01	9.90	13.59
16 node ring	1.19	57.90	64.97	25.15	28.46	34.64	35.20	42.89	48.04
16 node fully conn.	0.71	0.00	0.93	6.49	0.00	8.80	13.48	6.57	14.70

Table 6: Percentage improvement of BSA over MH, DLS, and BU for a 500-node random task graph on various topologies.

	CCR = 0.1			CCR = 1.0			CCR = 10.0		
Topology	BSA/ MH	BSA/ DLS	BSA/ BU	BSA/ MH	BSA/ DLS	BSA/ BU	BSA/ MH	BSA/ DLS	BSA/ BU
2 × 1	6.15	5.71	8.36	2.15	10.83	14.41	31.92	38.94	39.94
2 × 2	37.86	48.79	58.40	14.66	32.40	41.35	15.87	9.35	21.98
4 node fully conn.	6.20	1.95	8.35	-4.41	18.41	25.22	12.47	55.56	59.84
8 node hypercube	52.68	41.19	61.57	12.72	13.47	16.43	29.12	45.29	50.22
4 × 2 mesh	62.64	50.35	66.79	36.18	4.30	42.04	15.87	1.01	20.27
8 node ring	58.21	54.46	62.19	1.66	2.81	2.94	10.96	52.85	58.47
8 node fully conn.	2.59	11.17	13.87	0.79	9.40	10.97	12.12	7.31	16.84
16 node hypercube	57.66	46.25	63.50	-5.69	21.25	26.90	53.03	62.56	62.95
4 × 4 torus	57.69	46.68	62.48	37.55	45.53	45.81	2.35	42.31	46.93
16 node ring	65.16	66.16	71.02	42.96	31.18	52.85	12.19	60.19	69.37
16 node fully conn.	6.37	12.74	17.64	-3.46	9.79	13.02	19.70	19.52	25.33

50% in many cases compared to both the MH and DLS algorithms and up to 71% compared to the BU algorithm. Based on the results of these experiments, the BSA algorithm was better than the other three algorithms in general when CCR was low but yielded significant improvement when CCR was high.

Our next experiment considered the combined effects of task graph size and network topology. For regular graphs, we varied the matrix size from 9 to 18 for each of the 3 values of CCR. For the random graphs, we varied the

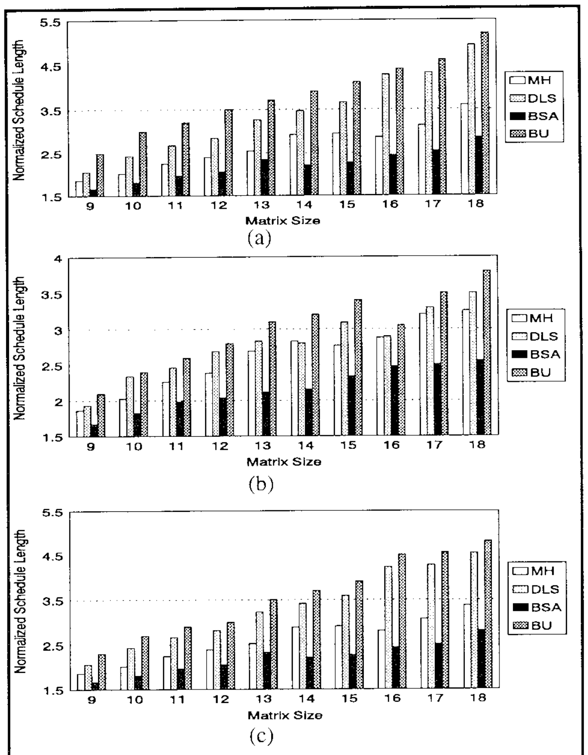

Figure 4: Normalized schedule lengths of regular graphs on (a) 8-node ring, (b) 8-node hypercube, (c) 16-node ring.

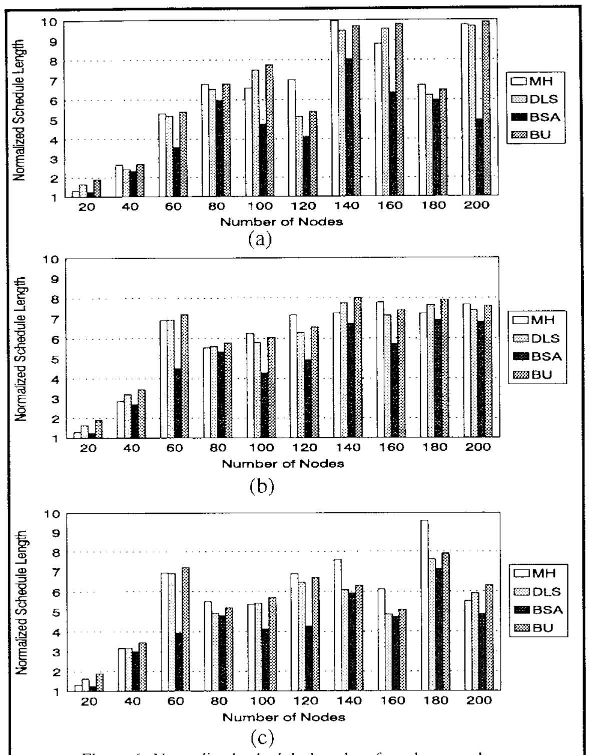

Figure 6: Normalized schedule lengths of random graphs on (a) 8-node ring, (b) 8-node hypercube, (c) 16-node ring.

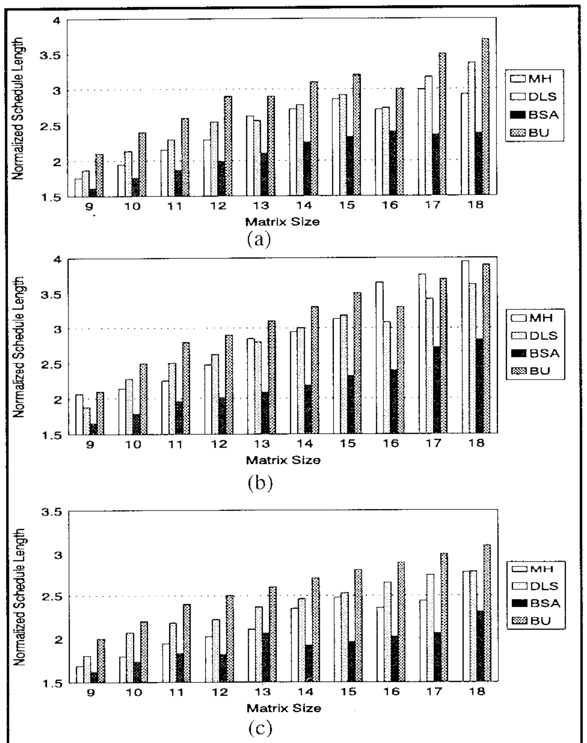

Figure 5: Normalized schedule lengths of regular graphs on (a) 16-node hypercube, (b) 8-node random, (c) 8-node fully connected.

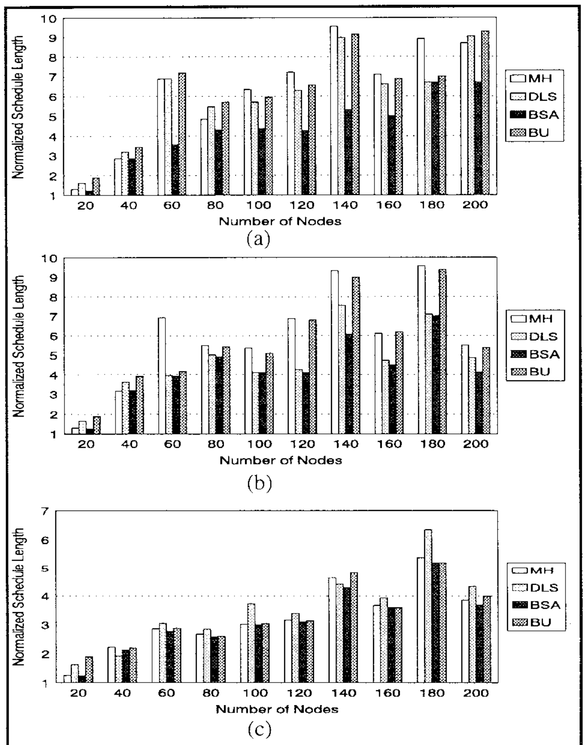

Figure 7: Normalized schedule lengths of random graphs on (a) 16-node hypercube, (b) 8-node random, (c) 8-node fully connected.

number of nodes from 20 to 200 with increments of 20. We compared the normalized schedule lengths produced by each algorithm on different network topologies and by varying the task graph size. The normalized schedule length was obtained by dividing the schedule length by the lower bound. The lower bound was determined by taking the sum of computation costs of the nodes on the critical path. It should be noted that the lower bound may not always be possible to achieve, and the optimal schedule length may be far greater than this bound.

We chose 6 different topologies to see how the degree of a processor could affect the schedule. For this purpose, we chose an 8-node ring (degree = 2), and 8-node hypercube (degree = 3), and an 8-node fully connected (degree = 7) topology. In addition, to see the effect of system size in terms of the number of processors, we selected a 16-node ring and 16-node hypercube to compare against their smaller counterparts. We also selected an 8-node random topology. These normalized scheduled lengths are shown in Figure 4(a) to Figure 4(c) and Figure 5(a) to Figure 5(c). Each bar in these figures was the average of the normalized scheduled lengths of task graphs for the mean value analysis, Gaussian elimination, Laplace equation solver and LU-decomposition algorithm, each with three different values of CCR (0.1, 1.0 and 10.0). Each bar, thus, was the average of 12 values. We can observe that the proposed BSA algorithm yielded better schedule lengths in all of the test cases. The worst performance is yielded by the BU algorithm. This mainly because it schedules all of the tasks on the critical-path to one processor with taking into account. This can lead to longer schedules when CCR is large.

Similar experiments were conducted for random graphs. The results of these experiments are shown in Figure 6(a) to Figure 6(c) and Figure 7(a) to Figure 7(c). For these figures, each bar was taken as the average of the schedule length for 3 random graphs with CCR equal to 0.1, 1.0, and 10.0. The BSA algorithm was again shown to better than the other two algorithms in almost all cases. On the other hand, the overall performance of MH was no longer better than DLS and was also outperformed by BU in some cases.

6 Conclusions

In this paper we have presented a new approach for scheduling and allocating parallel computations onto message-passing architectures. The objective is to simultaneously take into account realistic assumptions such as arbitrary computation and communication costs, network topology, contention on communication link. The complexity of the algorithm is not very high compared to other algorithms.

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] I. Ahmad and Y.K. Kwok, "A New Approach to Scheduling Parallel Programs Using Task Duplication," Proc. of *Int'l Conf. on Parallel Processing*, vol. II, pp. 47-51, Aug. 1994.

[3] V.A.F. Almeida, I.M. Vasconcelos, J.N.C. Arabe and D.A. Menasce, "Using Random Task Graphs to Investigate the Potential Benefits of Heterogeneity in Parallel Systems," Proc. of *Supercomputing'92*, pp. 683-691, 1992.

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

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

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

[7] 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.

[8] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan, "Optimization and Approximation in Deerministic Sequencing and Scheduling: A Survey," *Annals of Discrete Mathematics*, no. 5, pp. 287-326, 1979.

[9] 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.

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

[11] 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.

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

[13] R.E. Lord, J.S. Kowalik and S.P. Kumar, "Solving Linear Algebraic Equations on an MIMD Computer," *Journal of the ACM*, 30(1), pp. 103-117, Jan. 1983.

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

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

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

[17] 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.

[18] 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.

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

[20] T. Yang and A. Gerasoulis, "A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors," Proc. of *Supercomputing'91*, pp. 633-642, Nov. 1991.