Link Contention-Constrained Scheduling and Mapping of Tasks and Messages to a Network of Heterogeneous Processors

YU-KWONG KWOK[1] AND ISHFAQ AHMAD[2]

[1]Department of Electrical and Electronic Engineering
The University of Hong Kong, Pokfulam Road, Hong Kong

[2]Department of Computer Science
The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong

Email: *ykwok@eee.hku.hk, iahmad@cs.ust.hk*

Abstract†—In this paper, we consider the problem of scheduling and mapping precedence-constrained tasks to a network of heterogeneous processors. In such systems, processors are usually physically distributed, implying that the communication cost is considerably higher than in tightly coupled multiprocessors. Therefore, scheduling and mapping algorithms for such systems must schedule the tasks as well as the communication traffic by treating both the processors and communication links as important resources. We propose an algorithm that achieves these objectives and adapts its tasks scheduling and mapping decisions according to the given network topology. Just like tasks, messages are also scheduled and mapped to suitable links during the minimization of the finish times of tasks. Heterogeneity of processors is exploited by scheduling critical tasks to the fastest processors. Our extensive experimental study has demonstrated that the proposed algorithm is efficient, robust, and yields consistent performance over a wide range of scheduling parameters.

Keywords: algorithms, parallel processing, heterogeneous systems, scheduling, link contention, task graphs.

1 Introduction

One of the major goals of using a heterogeneous system is to minimize the completion time of a parallel application by exploiting the heterogeneous processing requirements within the application [5]. To achieve this goal, a judicious scheme is needed to properly schedule and allocate the tasks of the application to the most suitable processors. In this study, we are interested in the static scheduling of precedence-constrained tasks to a network of heterogeneous processors. Static scheduling is normally done at compile-time with available information about the structure of the parallel application in terms of its task execution times, task dependencies, communication, and synchronization [4], [9]. The goal of static scheduling is to allocate a set of tasks to a set of processors such that the overall completion time of the application, called the *schedule length*, is minimized while the precedence constraints among the tasks are preserved. Since this scheduling problem is NP-complete [4], [6], it is commonly tackled by using heuristics [7]. While each heuristic may perform well under different circumstances, there are three important criteria that must be considered for evaluating a heuristic: (1) Does the heuristic make realistic assumptions about the application and architecture of the system? (2) Is it problem-specific or can it work under a wide range of parameters without compromising the solution quality? (3) Does the complexity of the heuristic permit it to be practically used for compile-time scheduling?

The first criterion relates to the assumptions made by the scheduling algorithm about the program tasks and architecture models. Indeed, to simplify the design of the scheduling method, earlier approaches usually rely on simplifying assumptions such as assuming all tasks to have equal execution times, or ignoring the communication delays among tasks altogether [4], [9]. With the emergence of a wide variety of architectures in recent years, the architectural attributes such as system topology, message routing strategy, overlapped communication and computation, and processors heterogeneity, must also be taken into account by a scheduling algorithm. The second criterion dictates that the scheduling algorithm should generate good solutions for a variety of applications and target systems. A scheduling algorithm tailored for one particular application and architecture may not generate efficient solutions on another architecture [8]. The third criterion which is related to the execution time of the heuristic itself is an important consideration for effectively using it for compile-time scheduling of large-scale applications [1].

We are interested in scheduling algorithms that both schedule tasks and messages on arbitrary networks consisting of heterogeneous processors and communication links. Scheduling tasks while considering link contention for a heterogeneous system is a relatively less explored research topic and very few

† This research was supported by the Hong Kong Research Grants Council under contract number HKUST619/94E and a grant from the HKU CRCG.

algorithms for this problem have been designed. One well-known algorithm is the *dynamic level scheduling* (DLS) algorithm [11], which employs a dynamic list scheduling approach. In this paper, we propose a new algorithm, the primary objective of which is to generate efficient solutions while simultaneously handles arbitrary communication and execution costs in the parallel application, schedules tasks and messages by considering link contention as well as processors heterogeneity, and adapts to arbitrary network topology. The algorithm has a practicable complexity and is suitable for regular and irregular parallel program structures.

In a traditional algorithm, the tasks are first arranged as a list using some priority measure and then each task is scheduled one after another to a processor which allows the earliest finish time [2], [4], [8], [9], [10], [11]. To find such a processor in a heterogeneous target system where message scheduling has to be handled, a routing table is also needed, as in the DLS, for determining the most suitable route for messages in order to minimize the data ready time of each task. The problem with using a routing table is two-fold: (i) the routing table has to be pre-determined, usually using shortest-path algorithm, for the input target topology; (ii) during the scheduling process, the routing table, which has to be frequently updated, may not give optimized routes. Checking such routing information for every candidate processors inevitably results in high time complexity. Furthermore, the routing information is usually maintained for only a few common network topologies which may not be useful for an arbitrary network.

The proposed algorithm is different from traditional scheduling schemes in several aspects. First, in the algorithm, the tasks are not fixed in one single list throughout the entire scheduling process as in the traditional approach. Initially, the tasks are all scheduled to a single processor—effectively the parallel program is serialized. Then, each task is considered in turn for possible migration to the neighbor processors. The objective of this process is to improve the finish times of tasks because a task migrates only if it can "bubble up". If a task is selected for migration, the communication messages from its predecessors (some of which may remain in the original processor while others may have also migrated) are scheduled to the communication link between the new processor and the original processor. After all the tasks in the original processor are considered, the first phase of scheduling completes. In the second phase, the same process is repeated on one of the neighbor processor. Thus, a task migrated from the original processor to a neighbor processor may have an opportunity to migrate again to a processor one more hop away from the original processor. This incremental scheduling by migration process is repeated for all the processors in a breadth-first fashion. The advantage of this incremental approach is that no pre-specified routing table is needed because the algorithm adapts its scheduling decisions to each input topology, which may be arbitrary. More importantly, the incremental scheduling of tasks and messages can lead to optimized routes.

The remainder of this paper is organized as follows. In the next section, we provide a formal problem statement, followed by a detailed description and explanation of the proposed algorithm. An illustrative example is used throughout to explicate the features of the algorithm. Section 3 presents the experimental results. The last section concludes the paper.

2 The Proposed Algorithm

In this section, we first formally define the scheduling problem and the model used. We then outline our proposed algorithm, called *Bubble Scheduling and Allocation* (BSA). A small example is used for illustrating the algorithm's characteristics.

2.1 The Scheduling and Mapping Model

A parallel program is composed of n tasks $\{T_1, T_2, \ldots, T_n\}$ in which there is a partial order: $T_i < T_j$ implies that T_j cannot start execution until T_i finishes due to the data dependency between them. Thus, a parallel program can be represented by a directed acyclic *task graph* [2]. Parallelism exists among independent tasks—T_i and T_j are said to be independent if neither $T_i < T_j$ nor $T_j < T_i$. Each task T_i is associated with a nominal execution cost τ_i which is the execution time required by T_i on a reference machine in the heterogeneous system. Similarly, a nominal communication cost c_{ij} is associated with the message M_{ij} from T_i to T_j. Assume there are e messages where $(n-1) \le e < n^2$ so that the task graph is a connected graph.

To model heterogeneity of the target system which consists of m processors $\{P_1, P_2, \ldots, P_m\}$, *heterogeneity factors* are used. For example, if a task T_i is scheduled to a processor P_x, then its actual execution cost is given by $h_{ix}\tau_i$ where h_{ix} is the heterogeneity factor which is determined by measuring the difference in processing capabilities (e.g., speed) of processor P_x and the reference machine with respect to task T_i. Similarly, if a message M_{ij} is scheduled to the communication link L_{xy} between processors P_x and P_y, its actual communication cost is given by $h'_{ijxy}c_{ij}$. An example parallel program graph is shown in Figure 1.

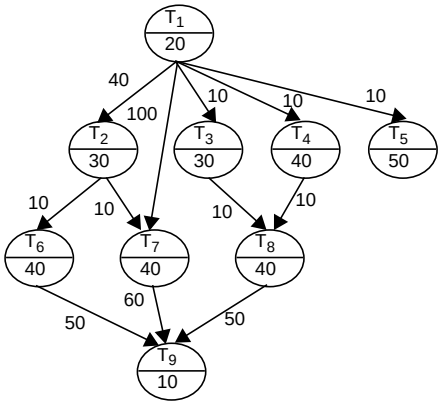


Figure 1: A parallel program task graph.

The start time and finish time of a message M_{ij} from T_i to T_j on a communication link L_{xy} are denoted by $MST(M_{ij}, L_{xy})$ and $MFT(M_{ij}, L_{xy})$, respectively. Thus, we have $MFT(M_{ij}, L_{xy}) = MST(M_{ij}, L_{xy}) + h'_{ijxy}c_{ij}$. The start time of a task T_i on processor P_x is denoted by $ST(T_i, P_x)$ which critically depends on the task's *data ready time* (DRT). The DRT of a task is defined as the latest arrival time of messages from its predecessors. The finish time of a task T_i is given by $FT(T_i, P_x) = ST(T_i, P_x) + h_{ix}\tau_i$. The objective of scheduling is to minimize the maximum FT, which is called the *schedule length* (SL).

2.2 Serialization

The serialization process, which determines the order of subsequent tasks migration, is a crucial step of the algorithm. A parallel program can be serialized using many different methods because there are many total orders which do not violate the original partial order. In the BSA algorithm, the serialization process is centered around a *critical path* of the parallel program.

DEFINITION 1**:** A critical path (CP) is defined as the set of tasks and messages forming a path with the largest sum of execution costs and communication costs.

In the case that there are multiple CPs, we select the one with a larger sum of execution costs and ties are broken randomly. The CP is a crucial structure of a parallel program because it is the longest execution path and thus, timely scheduling of its tasks can potentially lead to a shorter schedule length. However, to preserve the precedence constraints among tasks, we cannot arrange all the CP tasks first. Instead, in the serialization process, we have to first consider a CP task's predecessors, which need not be CP tasks themselves. Such predecessors are called *in-branch* (IB) tasks. The remaining tasks, which are neither CP tasks nor IB tasks, are called *out-branch* (OB) tasks. This partitioning of the tasks induces a serial order of the parallel program, in which CP tasks are arranged to occupy the earliest possible positions, with IB tasks inserted among them, and OB tasks are appended at the end.

To determine whether a task is a CP task, we can use two attributes: *t-level* (top level) and *b-level* (bottom level). The *b-level* of a task is the length of the longest path beginning with the task. The *t-level* of a task is the length of the longest path reaching the task. Thus, all tasks on the CP have the same value of (*t-level* + *b-level*), which is equal to the length of the CP. Based on this observation, we can easily partition the parallel program into CP, IB, and OB tasks by in $O(e)$ time because the *t-level* and *b-level* of all tasks can be computed by using depth-first search. A task with a larger *b-level* implies that it is followed by a longer chain of tasks, and thus, is given a higher priority. The serialization process can be performed by an $O(e)$ time algorithm outlined below.

SERIALIZATION:

Input: a program task graph with n tasks $\{T_1, T_2, \ldots, T_n\}$

Output: a serial order of the tasks

1. compute the *t-level* and *b-level* of each task by using depth-first search;
2. identify the CP; if there are multiple CPs, select the one with the largest sum of execution cost and ties are broken randomly;
3. put the CP task which does not have any predecessor to the first position of the serial order;
4. $i \leftarrow 2$; $T_x \leftarrow$ the next CP task
5. while not all the CP tasks are included do
6. if T_x has all its predecessors in the serial order then
7. put T_x at position i and increment i;
8. else let T_y be the predecessor of T_x which is not in the serial order and has the largest *b-level* (ties are broken by choosing the predecessor with a smaller *t-level*);
9. if T_y has all its predecessors in the serial order then put T_y at position i and increment i; otherwise, recursively include all the ancestors of T_y in the serial order such that the tasks with a larger *b-level* are included first;
10. repeat the above step until all the predecessors of T_x are in the serial order;
11. put T_x at position i and increment i;
12. $T_x \leftarrow$ the next CP task ;
13. append all the OB tasks to the serial order in descending order of *b-level*;

For example, consider the parallel program graph shown earlier in Figure 1. Based on the nominal execution and communication costs, the *t-level*s and *b-level*s of the tasks can be computed and the tasks $\{T_1, T_7, T_9\}$ form the CP. Since T_1 is the first CP task, it is placed in the first position in the serial order. The second task is T_2 because it is another unexamined predecessor of the next CP task T_7. After T_2 is appended to the serial order, all predecessors of T_7 have been

considered and, therefore, it can also be added. Now, the last CP task, T_9 is considered. It cannot be appended to the serial order because some of its predecessors (i.e., the IB tasks) have not been examined yet. Since both T_6 and T_8 have the same value of *b-level* and T_8 has a smaller *t-level*, T_8 is considered first. However, both predecessors of T_8 have not been examined. Thus, its two predecessors, T_3 and T_4 are appended to the list first. Next, T_8 is appended followed by T_6. The only OB task, T_5, is the last task in the serial order. The final serialized list is: $\{T_1, T_2, T_7, T_4, T_3, T_8, T_6, T_9, T_5\}$.

In the serialization process, the tasks are all scheduled to a single processor, called the *pivot* processor, which is selected as follows. The first processor in the heterogeneous system is considered and the corresponding heterogeneity factor is multiplied to the nominal execution cost of each task. Based on the set of actual execution costs, the CP is constructed. This process is repeated for other processors and eventually the processor that gives the shortest CP length based on actual execution costs is selected as the first pivot processor. To illustrate, consider the actual execution costs of the tasks on the four processor heterogeneous system as shown in Table 1. Given the actual execution costs, the CPs with respect to P_1, P_2, P_3, and P_4 are $\{T_1, T_7, T_9\}$, $\{T_1, T_2, T_6, T_9\}$, $\{T_1, T_2, T_7, T_9\}$, and $\{T_1, T_2, T_6, T_9\}$, respectively. The CP lengths are 240, 226, 235, and 260, respectively. Thus, the first pivot processor is P_2 because the CP is shortest with respect to this processor. The serial order is $\{T_1, T_2, T_6, T_7, T_3, T_4, T_8, T_9, T_5\}$, which is different from that determined earlier using nominal execution costs.

Table 1: The task execution cost of each task on a four heterogeneous processors.

task	P_1	P_2	P_3	P_4
T_1	39	7	2	6
T_2	21	50	57	56
T_3	15	28	39	6
T_4	54	14	16	55
T_5	45	42	97	12
T_6	15	20	57	78
T_7	33	43	51	60
T_8	51	18	47	74
T_9	8	16	15	20

2.3 Tasks Migration

After the parallel program is serialized to the first pivot processor, tasks have to be considered for possible migration to the neighbor processors in order to improve their finish times (bubble up). To determine whether a migration is beneficial, we have to compute the finish time of the task on a neighbor processor. To compute the start time, we need to know the DRT of the task, which in turn depends on the scheduling of messages. We outline below an algorithm for computing the finish time of a message on a communication link between two processors. Using a procedure called *ComputeMFT*, we can determine the finish times of every incoming messages of the task on a neighbor processor. The maximum finish time is then the DRT of the task. The corresponding predecessor which sends this latest message is called the *very important predecessor* (VIP) of the task.

After the DRT of the task on a neighbor processor is computed, the potential finish time of the task can also be determined. Then, using another procedure called *ComputeFT*, we can determine whether a task can improve its finish time through migrating to a neighbor processor of the pivot processor. If the finish time does improve, the task is rescheduled to the neighbor processor and its incoming and outgoing messages are also rearranged. If the finish time does not improve, then a task will also migrate if its VIP is scheduled to that neighbor processor. The rationale behind this heuristic decision is that if a task and its VIP are scheduled to the same processor, the successors of the task may subsequently improve their finish times also. This process is repeated for all the remaining tasks on the pivot. Then a neighbor processor is chosen to be a new pivot. Thus, each processor in the heterogeneous system in turn will be assigned as the pivot in a breadth-first manner. Throughout the entire bubbling up process, messages are automatically routed in the migration process of tasks from the pivot processor to other processors. There is no need to use a routing table. If the routing of messages has to be static (as in some commonly used networks, such as a hypercube that uses the E-cube routing method), we can just put a constraint on the destinations a task can migrate to. Moreover, the routes taken by such messages are optimized routes in that, at every step, a task migrates if its finish time is not increased.

Using the techniques discussed above, the BSA algorithm can be formalized below. In the following, the procedure *BuildProcessorList* constructs a list of processors in a breadth-first order from the first pivot processor.

BSA ALGORITHM:

Input: a parallel program graph with n tasks $\{T_1, T_2, \ldots, T_n\}$ and a heterogeneous system with m processors $\{P_1, P_2, \ldots, P_m\}$

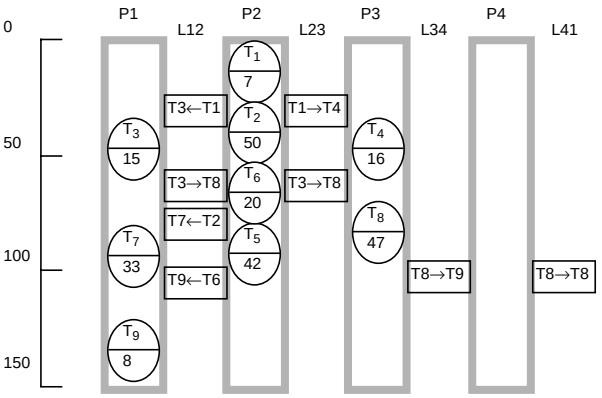
Output: a program schedule

1. initial *Pivot* ← the processor that gives the shortest CP length;
2. *Serialization*(*Pivot*);
3. *BuildProcessorList*(*Pivot*);
4. while *ProcessorList* is not empty do
5. *Pivot* ← remove the first processor from *ProcessorList*;
6. for each T_i on *Pivot* do
7. if $FT(T_i, Pivot) > DRT(T_i, Pivot)$ or VIP of T_i is not scheduled to *Pivot* then
8. for each neighbor processor P_y of *Pivot*, compute $DRT(T_i, P_y)$ and $FT(T_i, P_y)$;
9. if there is a neighbor processor P_y' such that $FT(T_i, P_y') < FT(T_i, Pivot)$ then
10. make T_i migrate from *Pivot* to P_y';
11. else if $FT(T_i, P_y') = FT(T_i, P_y')$ and VIP of T_i is scheduled to P_y' then
12. make T_i migrate from *Pivot* to P_y';

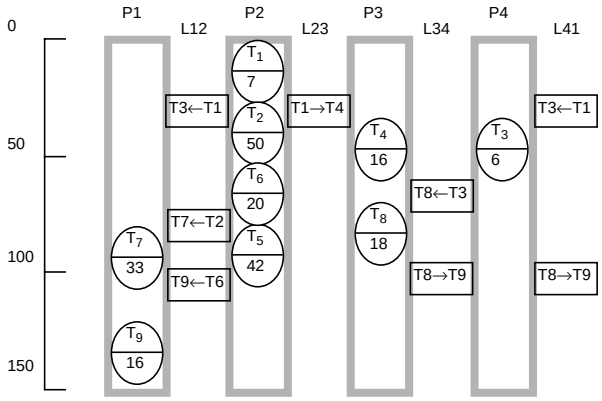
The time complexity of the BSA algorithm is derived as follows. The procedure *BuildProcessorList* takes $O(m^2)$ time while *Serialization* takes $O(n^2)$ time. Thus, the dominant step is the while-loop, which takes $O(e)$ time to compute the *FT* and *DRT* values of the task on each neighbor processor. If migration is done, it also takes $O(e)$ time. Since there are $O(n)$ tasks on the *Pivot* and $O(m)$ neighbor processor, each iteration of the while loop takes $O(men)$ time. Thus, the BSA algorithm takes $O(m^2en)$ time.

2.4 An Example

To illustrate the novel characteristics of the BSA algorithm, let us consider applying it to schedule the parallel program graph shown in Figure 1 to a four-processor heterogeneous ring system with the actual execution costs depicted in Table 1. For simplicity, we assume that the communication links are homogeneous; that is, $h'_{ijxy} = 1$ for all messages M_{ij} and links L_{xy}. Initially, the tasks are injected by the procedure *Serialization* to the first pivot processor P_2 in the order: $T_1, T_2, T_6, T_7, T_3, T_4, T_8, T_9, T_5$, as we have shown in Section 2.2. Note that the actual execution costs on P_2 are quite different from the nominal execution costs. Then, tasks are considered for possible migration. In the first phase, T_1, being the first CP task, does not migrate because its migration is not beneficial. Also, T_2 and T_6 do not migrate because their finish times cannot be improved by migration. However, T_3 and T_4 migrate to P_1 and P_3, respectively as their finish times are improved. Note that the reduction of T_3 's finish time is contributed not only by the "bubbling up" process but also by the heterogeneity of the processors—the execution cost of T_3 on P_2 is 28 while on P_1 is only 15. Similarly, T_7 also migrates to P_1 since it can also be "bubbled up" and its execution cost is reduced. After two more migrations from the first pivot processor P_2, the first phase is completed; the intermediate schedule at this point is shown in Figure 2(a). In the second phase, the pivot processor is P_1. Only T_3 migrates while the other tasks cannot improve their finish times. No more migration can be performed after this stage and the final schedule is shown in Figure 2(b). The schedule length is only 138 which is considerably smaller than that can be achievable on homogeneous processors.



(a) Intermediate schedule after T8 and T9 migrate to neighbor processors (schedule length = 147, total communication costs = 200)



(b) final schedule after T3 migrates from P1 to P4 (schedule length = 138, total communication costs = 200)

Figure 2: Schedules generated by the BSA algorithm.

3 Performance Results

In this section, we present the experimental performance of the BSA algorithm and also compare it with a previous algorithm, called the *dynamic level scheduling* (DLS) algorithm, which was also designed

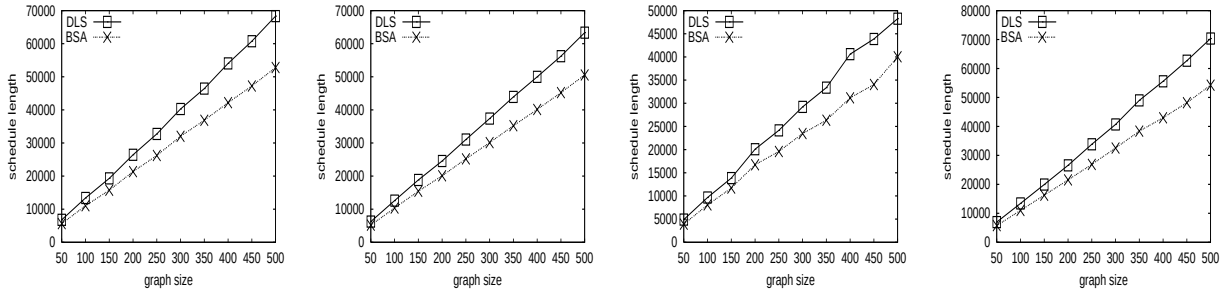
for heterogeneous systems. The DLS algorithm is also a greedy algorithm in that it chooses a task for scheduling if its potential start time is the earliest and it has the largest *b-level*.

In our experiments, we applied the two algorithms to two suites of task graphs using a Sun Ultrasparc workstation. The first suite consisted of regular graphs representing a number of parallel applications including the mean value analysis [1], Gaussian elimination [3], Laplace equation solver [1], LU-decomposition [3], containing regular patterns of tasks and communication messages. Since these applications operate on matrices, the number of tasks (and messages) in their task graphs depends on the matrix dimension N. Each application has its own equation in terms of N for determining the exact number of tasks but all of the equations are $O(N^2)$. We generated ten graphs for each application by varying N such that the graph size varies from approximately 50 to 500 with increments of 50. The average execution cost each task of the applications is about 150. Note that the graph structure and relative magnitudes of the execution costs in these applications are fixed according to the underlying algorithm modeled by the graph. However, the communication costs can be varied. We used a parameter called *granularity*, which is defined as the average execution cost divided by the average communication cost in a graph. Within each type of graph, we used three granularities: 0.1, 1.0, and 10.0. Thus, in a fine-grained (i.e., granularity = 0.1) application, the average communication cost is about ten times the average task execution cost. On the other hand, in a coarse-grained (i.e., granularity = 10.0) application, the average communication cost is only about 10% of the average task execution cost. In summary, the regular graphs suite contained 90 graphs (three graph types, ten sizes, and three granularities). The second suite of task graphs consisted of randomly structured graphs with sizes also varied from 50 to 500 with increments of 50. The execution cost of each task was randomly selected from a uniform distribution with range [100, 200]. Again, three granularities (0.1, 1.0, and 10.0) were selected for each graph size. Unless otherwise state, the heterogeneity factors (i.e., h_{ix} and h'_{ijxy}) were selected randomly from a uniform distribution with range [1, 50]. Thus, the nominal execution and communication costs in each graph represented the costs of the fastest processor.

To investigate the effect of processor network topology (i.e., processor connectivity), we used four different topologies in the experiments: 16-processor ring, 16-processor hypercube, 16-processor fully-connected network, and 16-processor randomly structured topology. The random topology was generated such that the degree of each processor ranged from two to eight.

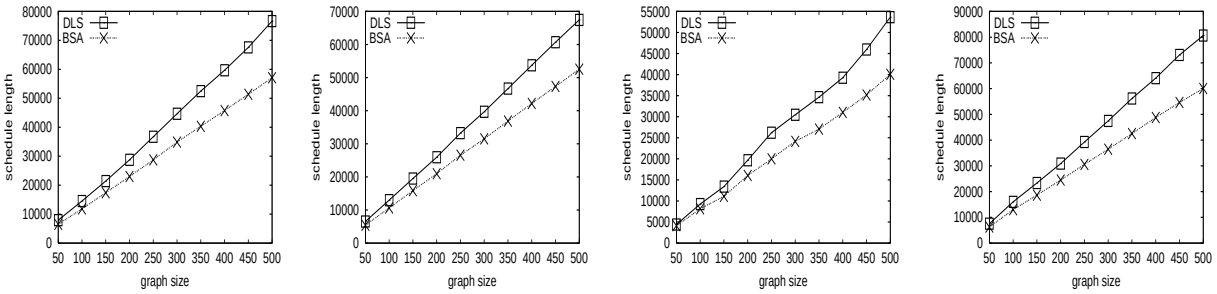
In our first experiment, we compared the schedule lengths produced by the BSA algorithm with those by the DLS algorithm. For the regular graphs, it turned out that each algorithm generated similar performance for the three types of applications and thus, we computed the average schedule lengths across different applications. To examine the effect of graph size, we also computed the average schedule lengths across the three granularities. These average schedule lengths for the four topologies are shown in Figure 3. From the plots, we make a number of observations:

- the BSA algorithm consistently outperformed the DLS algorithm;
- the improvement was about 20% and increased slightly with graph size;
- the improvement was slightly larger for lower processor connectivity (e.g., a ring); and
- both algorithms gave shorter schedule lengths for higher processor connectivity (e.g., a clique).



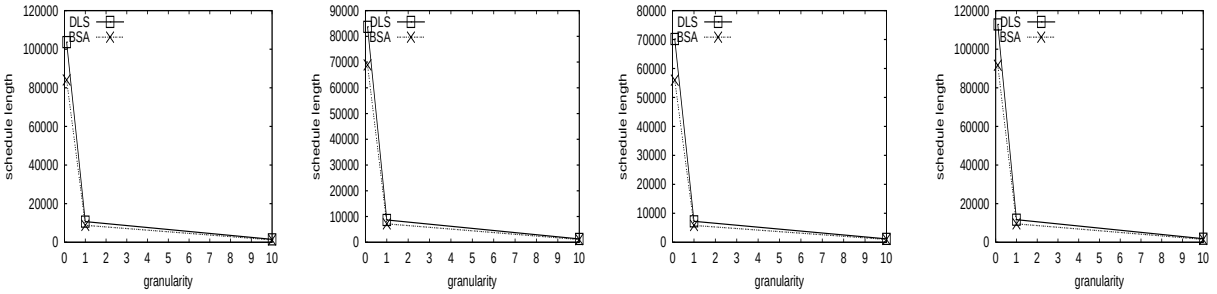
(a) 16-processor ring (b) 16-processor hypercube (c) 16-processor clique (d) 16-processor random topology

Figure 3: Average schedule lengths for the regular graphs with different graph sizes using four different network topologies.



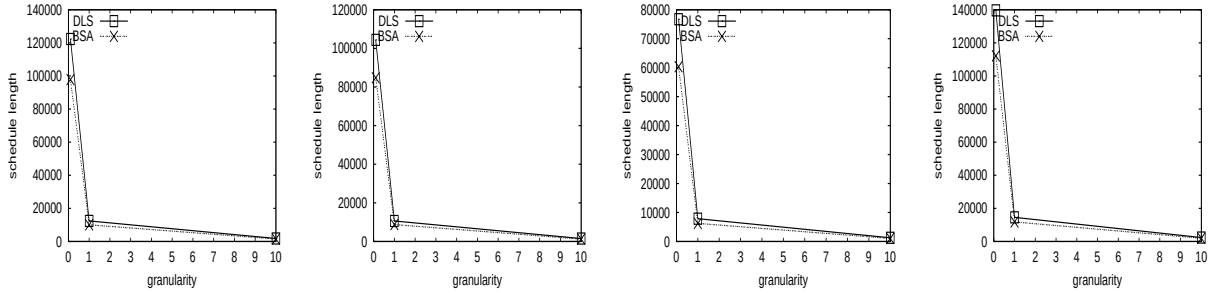
(a) 16-processor ring (b) 16-processor hypercube (c) 16-processor clique (d) 16-processor random topology

Figure 4: Average schedule lengths for the random graphs with different graph sizes using four different network topologies.



(a) 16-processor ring (b) 16-processor hypercube (c) 16-processor clique (d) 16-processor random topology

Figure 5: Average schedule lengths for the regular graphs with different granularities using four different network topologies.



(a) 16-processor ring (b) 16-processor hypercube (c) 16-processor clique (d) 16-processor random topology

Figure 6: Average schedule lengths for the random graphs with different granularities using four different network topologies.

These observations can be explained as follows. First, notice that the DLS algorithm selects a task for scheduling if its start time is the earliest. This greedy decision is made without regard to the scheduling of subsequent tasks and hence, such a decision may be too "local" in that the communication links are not properly utilized leading to inefficient scheduling of communication messages of subsequent tasks. Indeed, when we looked into the schedules produced by the DLS algorithm more closely, we found that there were many cases in which a task could not be scheduled to a better time slot due to the inefficient scheduling of messages of previous tasks. The adverse effect of inefficient scheduling of messages and tasks was also more profound for increasing graph size and decreasing processor connectivity. In this aspect, the BSA algorithm has a better design because the messages are incrementally scheduled to suitable slots such that the finish times of tasks can be improved. When the connectivity was high, both algorithms generated shorter schedules because the message scheduling was easier to handle.

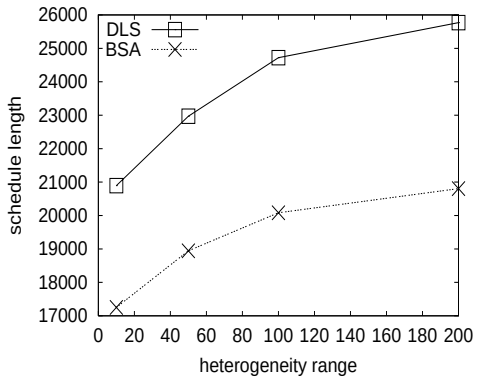


Figure 7: Effect of heterogeneity.

The results for randomly structured graphs are shown in Figure 4. From these results, we can see that the BSA algorithm is robust in that it also consistently outperformed the DLS algorithm, despite that both algorithms generated longer schedules compared with the regular graphs. Next, we investigated the effect of granularity by computing the average schedule lengths across the graph sizes. The results for regular graphs are shown in Figure 5. We can see that the granularity had significant impact on the performance of the scheduling algorithms. First, the schedule lengths increased sharply with decreasing granularity. At a low granularity (e.g., 0.1), the message scheduling was a dominant factor in determining the schedule length. Thus, the improvement of the BSA algorithm over the DLS algorithm was also larger for lower granularity. Finally, it is interesting to note that the effect of network topology was less significant from a granularity perspective. Similar conclusions can be drawn from the results for randomly structured graphs, which are shown in Figure 6.

We also investigated the effect of heterogeneity. For this purpose, we used ten different randomly structured task graphs with 500-task each (the granularity was 1.0). We chose the 16-processor hypercube topology and varied the range of heterogeneity as follows: [1, 10], [1, 50], [1, 100], and [1, 200]. Thus, a large range implies that there are more slow processors in the system. Again we computed the average schedule lengths, which are shown in Figure 7. As can be seen, both algorithms generated longer schedules as the heterogeneity range increased. However, the rate of increase in schedule lengths generated by the BSA algorithm was lower than that of the DLS algorithm. This indicates that the BSA algorithm is more adaptive to a highly heterogeneous system. We also measured the running times of both algorithms, which were about the same because the two algorithms are of comparable time complexity.

4 Conclusions

In this paper we have presented a new algorithm, called the BSA algorithm, for scheduling and allocation of parallel tasks onto message-passing heterogeneous architectures using a novel task ordering strategy. The objective is to generate efficient solutions while simultaneously taking into account realistic parameters such as arbitrary execution and communication costs, network topology, contention on communication links, and heterogeneity of processors. The distinctive feature of the BSA algorithm is that it can adapt its tasks and messages scheduling decisions according to the given network topology. Messages are incrementally scheduled to suitable links during the optimization of the finish times of tasks. Heterogeneity of processors is also exploited by scheduling critical tasks to the fastest processors. Our extensive performance evaluation study has demonstrated that the BSA algorithm is efficient, robust, and able to give consistent performance over a wide range of parameters.

References

[1] I. Ahmad, Y.-K. Kwok, M.-Y. Wu, and W. Shu, "Automatic Parallelization and Scheduling of Programs on Multiprocessors Using CASCH," *Proceedings of the 1997 International Conference on Parallel Processing*, pp. 288-291, Aug. 1997.

[2] M. Cosnard and M. Loi, "Automatic Task Graphs Generation Techniques," *Parallel Processing Letters*, vol. 5, no. 4, pp. 527-538, Dec. 1995.

[3] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystam, "Parallel Gaussian Elimination on An MIMD Computer," *Parallel Computing*, vol. 6, pp. 275-296, 1988.

[4] H. El-Rewini, T.G. Lewis, and H.H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Englewood Cliffs, New Jersey: Prentice Hall, 1994.

[5] R.F. Freund and H.J. Siegel, "Heterogeneous Processing," *Computer*, pp. 13-17, June 1993.

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

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

[8] —, "Benchmarking the Task Graph Scheduling Algorithms," *Proceedings of the 12th International Parallel Processing Symposium*, pp. 531-537, Mar. 1998.

[9] —, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys*, accepted for publication and to appear.

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

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