Performance Comparison of Algorithms for Static Scheduling of DAGs to Multiprocessors[1]

Ishfaq Ahmad, Yu-Kwong Kwok

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

Min-You Wu

Department of Computer Science
State University of New York at Buffalo, Buffalo, New York

Abstract. In this paper, we evaluate and compare algorithms for scheduling and clustering. These algorithms allocate a parallel program represented by an edge-weighted directed acyclic graph (DAG), also called a task graph or macro-dataflow graph, to a set of homogeneous processors, to minimize the completion time. We examine several such classes of such algorithms and compare the performance of a class of algorithms known as the *arbitrary processor network (APN) scheduling* algorithms. We discuss the design philosophies and principles behind these algorithms and assess their merits and deficiencies. Experimental results have been obtained by testing the algorithms with a large number of test cases. Global and pair-wise comparisons are made within each group whereby these algorithms are ranked according to their performance.

1. Introduction

Scheduling and mapping of computations onto the processors is one of the crucial components of a parallel processing environment. Since the scheduling problem is NP-complete in its general forms [5] and optimal solutions are known only in restricted cases [3], [7], there has been considerable research effort in this area resulting in a myriad of heuristic algorithms. This work surveys static scheduling algorithms that schedule an edge-weighted directed acyclic graph (DAG), which is also called a task graph or macro-dataflow graph, to a set of homogeneous processors, to minimize the completion time or schedule length. We examine three classes of algorithms: *Bounded Number of Processors (BNP)* scheduling algorithms, *Unlimited Number of Clusters (UNC)* scheduling algorithms, and *Arbitrary Processor Network (APN)* scheduling algorithms. We provide qualitative analyses by examining the design philosophies and characteristics of these algorithms that can ensue some future guidelines for designing even better heuristics. Performance comparisons are made for the APN algorithms. Experimental results, using a large number of test cases, have been obtained by testing four APN algorithms. Global and pair-wise comparisons are made within each group whereby these algorithms are ranked according to their performance.

The paper is organized as follows. In the next section, we describe the generic DAG model and discuss its variations and suitability to different situations. We describe the basic strategies of scheduling algorithms in Section 3. The BNP and UNC algorithms are discussed in Section 4 and Section 5, respectively. The four APN algorithms are discussed in Section 4. Performance results and comparisons are presented in Section 7. Section 8 concludes the paper.

2. The DAG Model

The DAG is a generic model of a parallel program consisting of a set of processes among which there are dependencies. Each process is an indivisible unit of execution, expressed by an atomic node. An atomic node has one or more inputs. When all inputs are available, the node is triggered to execute. After its execution, it generates its outputs. In this model, a set of v nodes $\{n_1, n_2, \ldots, n_v\}$ are connected by a set of e directed edges, each of which is denoted

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

by (n_i, n_j), where n_i is called the parent and n_j is called the child. A node without parent is called an entry node and a node without child is called an exit node. The weight of a node, denoted by $w(n_i)$, is equal to the process execution time. Since each edge corresponds to a message transfer from one process to another, the weight of an edge, denoted by $c(n_i, n_j)$, is equal to the message transmission time. Thus, $c(n_i, n_j)$ becomes zero when n_i and n_j are scheduled to the same processor because intraprocessor communication time is negligible compared with the interprocessor communication time. The node and edge weights are usually obtained by estimation [19]. Some variations in the generic DAG model are described below.

Accurate model — In an accurate model, the weight of a node includes the computation time, the time to receive messages before the computation, and the time to send messages after the computation. The weight of an edge is a function of the distance between the source and the destination nodes, and therefore, depends on the node allocation and network topology. It also depends on network contention which can be difficult to model. When two nodes are assigned to a single processor, the edge weight becomes zero, so as the message receiving time and sending time.

Approximate model 1 — In this model, the edge weight is approximated by a constant, independent of the message transmission distance and network contention. A completely connected network without contention fits this model.

Approximate model 2 — In this model, the message receiving time and sending time are ignored in addition to approximating the edge weight by a constant.

These approximate models are best suited to the following situations: (i) the grain-size of the process is much larger than the message receiving time and sending time; (ii) communication is handled by some dedicated hardware so that the processor spends insignificant amount of time on communication; (iii) the message transmission time varies little with the message transmission distance, e.g., in a wormhole or circuit switching network; and (iv) the network is not heavily loaded.

In general, the approximate models can be used for medium to large granularity, since the larger the process grain-size, the less the communication, and consequently the network is not heavily loaded. The second reason for using the approximate models is that both the node and edge weights are obtained by estimation, which is hardly accurate. Thus, an accurate model is useless when the weights of nodes and edges are not accurate.

3. Basic Scheduling Strategies

A DAG can be scheduled to a network of processors in two ways:

Direct mapping: The DAG is directly mapped to the given network topology, using the accurate model. In this paper, we focus on the APN algorithms (will be discussed in Section 4), which are designed for direct mapping.

Indirect mapping: The DAG is scheduled to processors using one of the approximate models, ignoring the processor network topology (assumed to be fully-connected); then the processors are mapped to the given topology. The latter step is called *topology mapping*, which is a technique for alleviating the impact of network contention.

Most scheduling algorithms are based on the *list scheduling* technique [1], [8], [9], [17], [18]. List scheduling is a class of scheduling heuristics in which the nodes are assigned priorities and placed in a list arranged in a descending order of priority. The node with a higher priority will be examined for scheduling before a node with a lower priority. If more than one node has the same priority, ties are broken using some method.

Two major attributes for assigning priority are the *t-level* (top level) and *b-level* (bottom level). The *t-level* of a node n_i is the length of the longest path from an entry node to n_i in the DAG (excluding n_i). Here, the length of a path is the sum of all the node and edge weights along the path. The *t-level* of n_i highly correlates with n_i's earliest start time, denoted by $T_S(n_i)$, which is determined after n_i is scheduled to a processor. The *b-level* of a node n_i is the length of the longest path from node n_i to an exit node. The *b-level* of a node is bounded by the length of the *critical path*. A critical path (CP) of a DAG, is a path from an entry node to an exit

node, whose length is the maximum. The *t-level* of a node is inherently a dynamic attribute because the weight of an edge may be zeroed when the two incident nodes are scheduled to the same processor. Thus, the path reaching a node, whose length determines the *t-level* of the node, may cease to be the longest one. On the other hand, there are some variations in the computation of the *b-level* of a node. Most algorithms examine a node for scheduling only after all the parents of the node have been scheduled. In this case, the *b-level* of a node is a constant until after it is scheduled to a processor. Some scheduling algorithms allow the scheduling of a child before its parents [14], [19]. In that case, the *b-level* of a node is also a dynamic attribute. In these scheduling algorithms, the value of $T_S(n_i)$ for any node n_i is not fixed until all the nodes are scheduled. Doing so can allow the insertion of a node to a time slot created by pushing some earlier scheduled nodes downward. It should be noted that some scheduling algorithms do not take into account the edge weights in computing the *b-level* [10], [16]. To distinguish such definition of *b-level* from the one we described above, we call it the *static b-level*. These algorithms are further classified into three categories descrbed below.

4. BNP Scheduling Algorithms

These algorithms schedule the DAG to a bounded number of processors directly. The processors are assumed to be fully-connected. Most BNP scheduling algorithms are based on the *list scheduling* technique [1], [8]. List scheduling is a class of scheduling heuristics in which the nodes are assigned priorities and placed in a list arranged in a descending order of priority. The node with a higher priority will be examined for scheduling before a node with a lower priority. If more than one node has the same priority, ties are broken using some method.

Two major attributes for assigning priority are the *t-level* (top level) and *b-level* (bottom level). The *t-level* of a node n_i is the length of the longest path from an entry node to n_i in the DAG (excluding n_i). Here, the length of a path is the sum of all the node and edge weights along the path. The *t-level* of n_i highly correlates with n_i's earliest start time, denoted by $T_S(n_i)$, which is determined after n_i is scheduled to a processor. The *b-level* of a node n_i is the length of the longest path from node n_i to an exit node. The *b-level* of a node is bounded by the length of the *critical path*. A critical path (CP) of a DAG, is a path from an entry node to an exit node, whose length is the maximum. Examples of BNP algorithms are the HLFET (Highest Level First with Estimated Times) algorithm [1], the MCP (Modified Critical Path) algorithm [19], and the DLS (Dynamic Level Scheduling) algorithm [18].

5. UNC Scheduling Algorithms

These algorithms schedule the DAG to an unbounded number of clusters. The processors are assumed to be fully-connected. The technique employed by these algorithms is also called *clustering*. The basic technique employed by the UNC scheduling algorithms is called *clustering*. At the beginning of the scheduling process, each node is considered as a cluster. In the subsequent steps, two clusters are merged if the merging reduces the completion time. This merging procedure continues until no cluster can be merged. Usually, no backtracking is allowed in order to avoid formidable time complexity.

The clustering strategy is particularly designed for DAGs with non-zero edge weights. If all edge weights are zero, the CP length of the original DAG gives the shortest completion time. The clustering process is so designed that when two clusters are merged and the weights of the edges across the two clusters are zeroed, the new CP length of the resulting DAG becomes shorter than the one before the merging. An optimal clustering results in a number of clusters such that the CP length of the clustered DAG cannot be further reduced. At this point, the completion time is minimized. In order to facilitate the subsequent cluster mapping step, the secondary goal of the UNC scheduling algorithms is to minimize the number of clusters. Examples of UNC algorithms are the EZ (Edge-zeroing) algorithm [16], the DSC (Dominant Sequence Clustering) algorithm [6], the MD (Mobility Directed) algorithm [19] and the DCP (Dynamic Critical Path) algorithm [14].

6. APN Scheduling Algorithms

The algorithms in this class take into account specific architectural features such as the number of processors as well as their interconnection topology. These algorithms can schedule tasks on the processors and messages on the network communication links. Scheduling of messages may be dependent on the routing strategy used by the underlying network. The mapping, including the temporal dependencies, is therefore implicit — without going through a separate clustering phase. There are not many reported algorithms that belong to this class. In the following, we discuss four such algorithms..

6.1 The MH Algorithm

The MH (Mapping Heuristic) algorithm [4] assigns priorities by computing the *static b-level*s of all nodes. In calculating the start time of node, a routing table is maintained for each processor. The table contains the information as to which path to route messages from the parent nodes to the nodes under consideration. The shortcoming of the algorithm is the way it orders nodes for scheduling. Since it *appends* ready nodes to the ready list instead of inserting the nodes at appropriate positions in the list, a newly ready node, which may have higher priority, may get scheduled later than the lower priority nodes. The MH algorithm is briefly described below. Its complexity is shown to be $O(v(p^3v + e))$.

(1) Compute the *static b-level* of each node n_i in the task graph.
(2) Initialize a ready node list by inserting all entry nodes in the task graph. The list is ordered according to node priorities, with the highest priority node first.

Repeat

(3) $n_i \leftarrow$ the first node in the list
(4) Schedule n_i to the processor which gives the smallest start time.
(5) Append all ready successor nodes of n_i, according to their priorities, to the ready node list.

Until the ready node list is empty.

6.2 The DLS Algorithm

The DLS (Dynamic Level Scheduling) algorithm [18] uses an attribute called the *dynamic level* (DL). The DL is defined as the difference between the *static b-level* of a node and its *t-level* on a processor. Thus, the DL is an attribute for a node-processor pair. The DLS algorithm first computes the *static b-level* for each node and then computes the DL for every node in the ready pool on all processors. The DLS algorithm requires the message routing method to be supplied by the user. The *t-level* of a node is then computed according to how the messages from the parents of the node are routed. The node-processor pair that gives the largest value of DL is selected for scheduling. The DLS algorithm is briefly described below. Its time complexity is shown to be $O(v^3p^2)$.

(1) Calculate the *b-level* of each node.
(2) Initially, the ready node pool includes only the entry nodes.

Repeat

(3) Calculate the earliest start time for every ready node on each processor. Hence, compute the DL of every node-processor pair by subtracting the earliest start time from the node's *static b-level*.
(4) Select the node-processor pair that gives the largest DL. Schedule the node to the corresponding processor.
(5) Add the newly ready nodes to the ready pool.

Until all nodes are scheduled.

6.3 The BU Algorithm

The BU (Bottom-Up) algorithm [15] first finds out the CP of the DAG and then assigns all the nodes on the CP to the same processor at once. Afterwards, the algorithm assigns the remaining nodes in a reversed topological order to the processors. The node assignment is guided by a load-balancing processor selection heuristic which attempts to balance the load across all given processors. After all the nodes get assigned a processor, the BU algorithm

tries to schedule the communication messages among them using a channel allocation heuristic which tries to keep the hop count of every message roughly a constant constrained by the processor network topology. Different network topology requires different channel allocation heuristics. The main weakness of the BU algorithm is due to its method of assigning nodes to processors. It is obvious that achieving a balanced load across all processors is usually in conflict with the goal of minimizing the completion time. The BU algorithm is briefly described below. Its time complexity is shown to be $O(v^2 \log v)$.

(1) Find a critical path. Assign the nodes on the critical path to the same processor. Mark these nodes as assigned and update the load of the processor.
(2) Compute the *b-level* of each node. If the two nodes of an edge are assigned to the same processor, the communication cost of the edge is taken to be zero.
(3) Compute the *p-level* (precedence level) of each node, which is defined as the maximum number of edges along a path from an entry node to the node.
(4) In a decreasing order of *p-level*, for each value of *p-level*, do:
    (a) In a decreasing order of *b-level*, for each node at the current *p-level*, assign the node to a processor such that the processing load is balanced across all the given processors.
    (b) Re-compute the *b-level*s of all nodes.
(5) Schedule the communication messages among the nodes such that the hop count of each message is maintained constant.

6.4 The BSA Algorithm

The BSA (Bubble Scheduling and Allocation) algorithm [13] constructs a schedule incrementally by first injecting all the nodes to the *pivot processor*, defined as the processor with the highest degree. Then, the algorithm tries to improve the start time of each node (hence "bubbling" up nodes) by migrating it to one of the adjacent processor of the *pivot processor* if the migration can improve the start time of the node. Essentially, after a node is migrated from *pivot processor* to another processor, not only the node itself is "bubbled up" but its successors as well. This is because after a node is migrated, the space it occupies on the *pivot processor* is released and can be used for its successor nodes on the *pivot processor.* After all nodes on the *pivot processor* are considered, select the next processor in the processor list to be the new *pivot processor.* The process is repeated by changing the *pivot processor* in a breadth first order. The BSA algorithm is *self-adjusting* in that it analyses the structure of the target topology before scheduling starts. This can greatly enhance the accuracy of the subsequent scheduling process. This is because important nodes will not be scheduled (migrated) to inappropriate processors (processors that have low degrees or are remote from the first pivot processor). The BSA algorithm is briefly described below. Its complexity is $O(p^2 ev)$.

(1) Determine a Critical Path.
(2) Partition the task graph into three categories: *CPNs (Critical Path Nodes), IBN (In-Branch Nodes)* and *OBNs (Out-Branch Nodes*).
(3) Serialize the task graph to the *pivot processor* (defined as the processor with the largest number of links). The task graph is serialized in such an order that the *CPNs* start as early as possible.
(4) Migrate the tasks to adjacent processors of the *pivot processor* if the start times improve.
(5) Change the *pivot processor* to one of the adjacent processor in a breadth-first order.
(6) Repeat from step (4) until no more start time can be improved.

7. Performance and Comparison

In this section, we present a performance comparison of the APN scheduling algorithms described above. The algorithms were tested on a SUN SPARC IPX workstation. The performance comparison is based on the schedule length. We compare the normalized schedule lengths generated by the algorithms. The experimental results were obtained by using a set of 250 random task graphs which were generated as follows. The size of the graph was varied from 50 to 500 nodes with increments of 50. The weight of each node was randomly selected from a uniform distribution with mean equal to the specified average computation cost. The weight of each edge was also randomly selected from a uniform

distribution with mean equal to the product of the average computation cost and the communication-to-computation ratio (CCR). Five different values of CCR were selected: 0.1, 0.5, 1.0, 2.0 and 10.0. We also used another parameter called *parallelism* in generating the random task graphs. This parameter determined the average number of children nodes for each node. Five different values of parallelism were chosen: 1, 2, 3, 4 and 5.

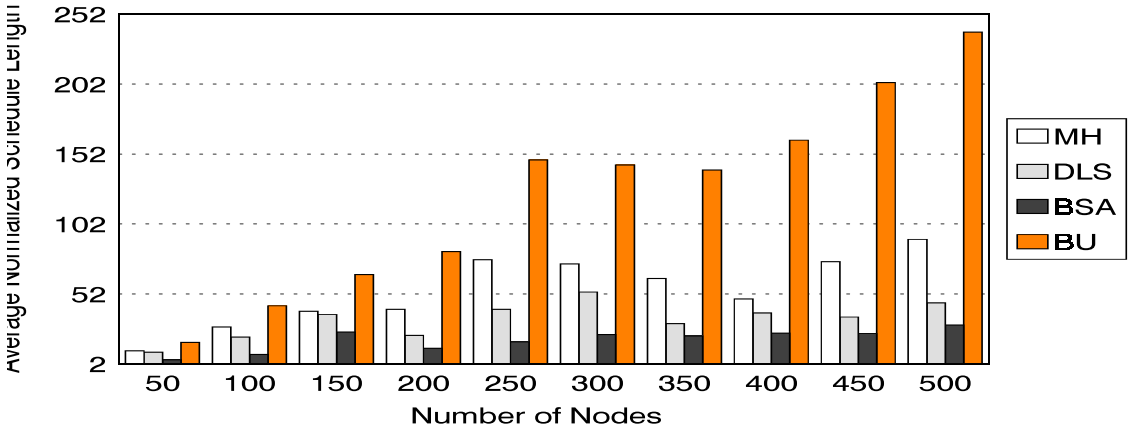

Figure 1: The average normalized schedule lengths produced by the APN scheduling algorithms for task graphs of various sizes on a 4x2 mesh.

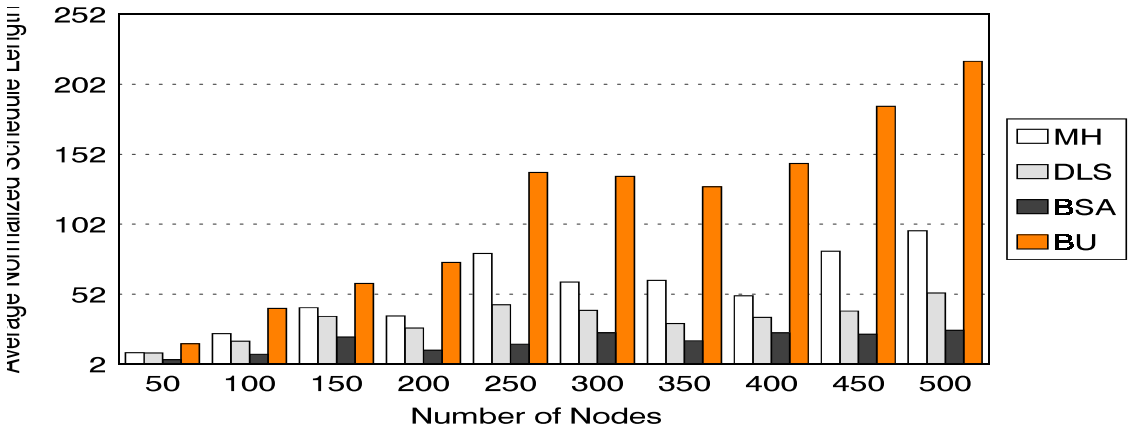

Figure 2: The average normalized schedule lengths produced by the APN scheduling algorithms for task graphs of various sizes on a 8-node hypercube.

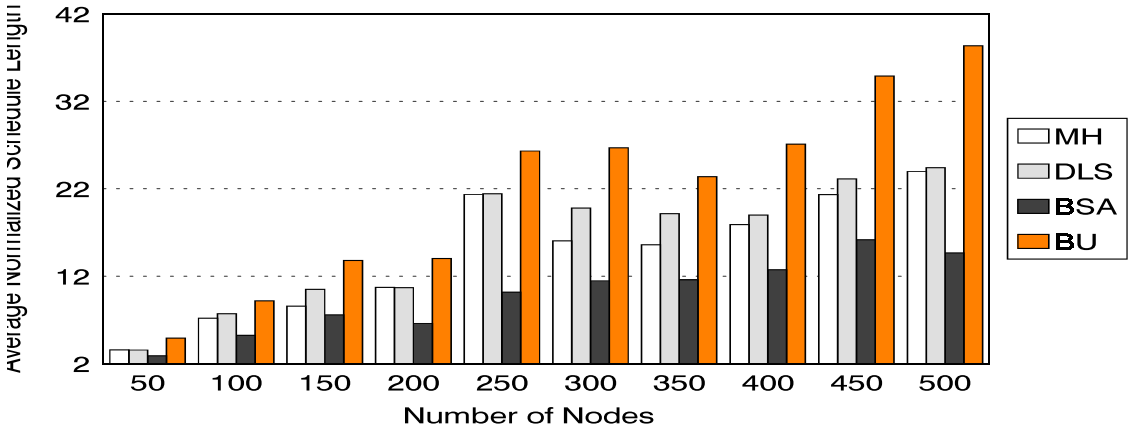

Figure 3: The average normalized schedule lengths produced by the APN scheduling algorithms for task graphs of various sizes on a 8-node clique.

The *normalized schedule lengths* (NSLs) for the APN algorithms are shown in Figure 1, Figure 1, and Figure 1. Each bar in the figure is the average of 25 cases with various values of

CCR and parallelism; the results showing the impact of CCR — and hence granularity — and the impact of parallelism is not included here due to space limitations. The NSLs were obtained by dividing the schedule lengths produced by each algorithm to the lower bounds, defined as the sum of node weights on the original critical path. It should be noted that the lower bounds may not always be possible to achieve, and the optimal schedule length may be longer than this bound. The target architectures for these tests included an 8-node hypercube, a 4×2 mesh, and an 8-node clique. One reason for the large values of NSLs in these cases is that the numbers of processors used were much smaller. We deliberately used very small number of processors to make the experiments more realistic. For example, a 500-node task graph is scheduled to 8 processors.

These results suggest that there can be substantial difference in the performance of these algorithms. For example, significant differences can be noticed between the NSLs of BSA and BU. The performance of DLS was relatively stable with respect to the graph size while MH yielded fairly long schedule lengths for large graphs. As can be noticed, the BSA algorithm performed admirably well for large graphs. One of the main reasons for the better performance of BSA is efficient scheduling of communication messages that can have a drastic impact on the overall schedule length. In terms of the impact of the topology, clearly all algorithms performed better on the networks with more communication links.

Next we present a pair-wise and a global comparison among these algorithms by observing the number of times each algorithm performed better, worse or the same compared to every other algorithm in 250 test cases. This comparison is given in a graphical form shown in Figure 4. Here, each box compares two algorithms — the algorithm on the left side and the algorithm on the top. Each box contains three numbers preceded by '>', '<' and '=' signs which indicate the number of times the algorithm on the left performed better, worse, and the same, respectively, compared to the algorithm shown on the top. For example, the BSA algorithm performed better than the MH algorithm in 840 cases, worse in 160 cases and the same in 0 case. For the global comparison, an additional box ("ALL") for each algorithm compares that algorithm with all other algorithms combined. Based on these results, we rank these BNP algorithms in the following order: BSA, DLS, MH, and BU. This ranking essentially indicates the quality of scheduling based on how often an algorithm performs better than the others, while the ranking with respect to NSLs indicates the quality of scheduling based on the average performance of the algorithm. An algorithm which outperforms other algorithms more frequently but has a lower rank based on the average NSL indicates that it produces very long schedule lengths in some cases. BSA outperformed the other three algorithms in a large number of cases while DLS performed better than MH. The BU algorithm was outperformed by all other algorithms.

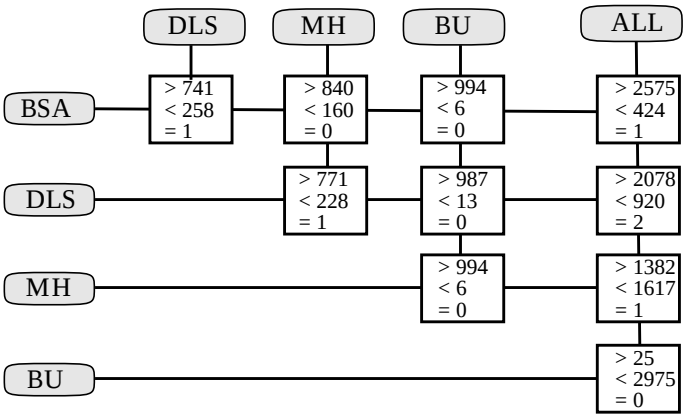

Figure 4: A comparison of the APN scheduling algorithms in terms of better, worse and equal performance across all topologies.

8. Conclusions and Future Work

The APN algorithms can be fairly complicated because they take into account more parameters. Further research is required in this area. The effects of topology and routing strategy needs to be determined. Further details on the performance of APN algorithms can be found in [12]. A number of research prototypes have been designed and implemented,

showing good performance on a group of carefully selected examples. The current researches concentrate on further elaboration of various techniques, such as reducing the scheduling complexities, improving computation estimations, and incorporating network topology and communication traffic.

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, no. 12, pp. 685-690, Dec. 1974.

[2] I. Ahmad and Y.K. Kwok, "A Parallel Approach to Multiprocessor Scheduling," Proc. of *Int'l Parallel Processing Symposium*, Apr. 1995, pp. 289-293.

[3] E.G. Coffman and R.L. Graham, "Optimal Scheduling for Two-Processor Systems," *Acta Informatica*, vol. 1, pp. 200-213, 1972.

[4] 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.

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

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

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

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

[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*, vol. II, pp. 243-250, Aug. 1994.

[10] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," Proc. of *Int'l Conference on Parallel Processing*, vol. II, pp. 1-8, Aug. 1988.

[11] B. Kruatrachue and T.G. Lewis, "Duplication Scheduling Heuristics (DSH): A New Precedence Task Scheduler for Parallel Processor Systems," Technical Report, Oregon State University, Corvallis, OR 97331, 1987.

[12] Y.K. Kwok, "Efficient Algorithms for Scheduling and Mapping of Parallel Programs onto Parallel Architectures," *M.Phil. Thesis*, Department of Computer Science, The Hong Kong University of Science and Technology, Hong Kong, Jun. 1994.

[13] Y.K. Kwok and I. Ahmad, "Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures," to appear in the Proc. of *Int'l Symposium of Parallel and Distributed Processing*, Oct. 1995.

[14] Y.K. Kwok and I. Ahmad, "A Static Scheduling Algorithm Using Dynamic Critical Path for Assigning Parallel Algorithms Onto Multiprocessors," Proc. of *Int'l Conf. on Parallel Processing*, vol. II, pp. 155-159, Aug. 1994.

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

[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.