

MCP Revisited

Min-You Wu

Department of Electrical and Computer Engineering
The University of New Mexico

Abstract— MCP (Modified Critical-Path) algorithm [1] schedules Directed Acyclic Graphs (DAGs) with communication costs onto a bounded number of processors. It is one of the six most popular algorithms in this category and performs the best among them [2]. MCP has many implementations during the last decade, while most of them did not implement it efficiently. In this paper, we present the methods to reduce the complexity of MCP. A detailed algorithm of MCP is provided along with a complexity analysis. Variety of the standard MCP implementation is also discussed. This algorithm implementation may be used as a basis for comparison of newly invented algorithms in terms of schedule length, complexity, and running time.

1 Introduction

Task scheduling utilizes knowledge of problem characteristics to schedule parallel programs into multiprocessors. Although many people have conducted their research in various manners, they all share a similar underlying idea: taking a Directed Acyclic Graph (DAG) representing the parallel program as input and scheduling it into processors of a target machine to minimize the completion time. This is an NP-complete problem in its general form [3]. Therefore, many heuristic algorithms were proposed [1, 4, 5, 6]. A DAG scheduling algorithm should have the following features:

- **High quality** — minimize the completion time of a parallel program.
- **Low complexity** — minimize the time for scheduling a parallel program.

These two requirements are contradict in general. Usually, a high-quality scheduling algorithm has a high complexity. We have introduced an algorithm called the Modified Critical-Path (MCP) algorithm [1], which offered a good quality with a reasonable low complexity. This algorithm has been compared to other scheduling algorithms under the same assumption and shown that MCP performed best among six most popular algorithms, MCP, ISH, HLFET, LAST, DLS, and ETF [2].

MCP was originally introduced in our Hypertool paper [1], where task graph generation, partitioning, scheduling, synchronization, and code generation of DAG were discussed. MCP, as one of the scheduling algorithms, was only briefly described. Later works showed that MCP performed well [2] and recently, many people use MCP to compare and evaluate their new algorithms [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. MCP was also used as the basis of parallel scheduling of DAGs [20]. However, some MCP implementations and complexity analysis are not completely correct. It is the right time to revisit the MCP algorithm and present methods for efficient implementation and detailed complexity analysis that have not been presented in the original paper. In addition, the implementation details will help other people for their researches.

In this paper, we will revisit the MCP algorithm in details. It is carefully implemented to minimize its running time and the code is made available online. We will introduce some methods for efficient implementation of MCP. The complexity analysis will be given. Also, some misunderstanding in the implementation details will be clarified.

2 The Original MCP Algorithm

A DAG consists of a set of nodes $\{n_1, n_2, \dots, n_n\}$ connected by a set of edges, each of which is denoted by $e(n_i, n_j)$. Each node represents a task and the weight of a node $w_n(n_i)$ is the execution time of the task. Each edge represents a message transferred from one node to another node, and the weight of the edge $w_l(n_i, n_j)$ is equal to the transmission time of the message. When two nodes are scheduled to a single processing element (PE), the weight of the edge connecting them becomes zero. A DAG also has two virtual nodes of zero weight: a *start node* that is the start of the program and an *end node* that is the end of the program.

First, a few terms are defined:

- $T_{flevel}(n_i)$ is the length of the longest path (including the node weights and edge weights) from node n_i to the exit node, excluding the weight of node n_i ;
- $T_{blevel}(n_i)$ is the length of the longest path (including the node weights and edge weights) from node n_i to the exit node, including the weight of node n_i ;
- the critical path length L_{CP} is T_{blevel} (or T_{flevel}) of the start node; and
- *As-Late-As-Possible (ALAP)* time $T_{alap}(n_i)$ is

$$T_{alap}(n_i) = L_{CP} - T_{blevel}(n_i) = L_{CP} - T_{flevel}(n_i) - w_n(n_i).$$

The following algorithm has been designed for scheduling a DAG onto a bounded number of PEs [1].

The Original MCP Algorithm

Step 1. Compute the ALAP time of each node.

Step 2. For each node, create a list which consists of the ALAP times of the node itself and all its descendants in an ascending order, sort these lists in an ascending lexicographical order, and create a node list according to this order.

Step 3. Schedule the first node in the list to the PE that allows the earliest execution, using the insertion approach. Remove the node from the list and repeat Step 3 until the node list is empty.

When nodes are scheduled to PEs, there may be some *holes* between nodes due to dependences among the nodes. In the above algorithm, the node is scheduled to the first available hole, called the *insertion approach*. If the node can only be scheduled after the last node scheduled in the PE without considering holes, it is called a *non-insertion approach*. The insertion approach performs much better than the non-insertion one. The complexity of the non-insertion approach for N nodes is $O(NP)$ and that of the insertion approach for N nodes is $O(N^2)$, where N is the number of nodes and P is the number of PEs.

The complexity of Step 1 is $O(E)$, the complexity of Step 2 is $O(N^2 \log N)$, and the complexity of Step 3 with the insertion approach is $O(N^2)$, where N is the number of nodes and E is the number of edges. Thus, the complexity of this algorithm is $O(N^2 \log N)$.

3 Efficient MCP Implementation

In this section, we present the details of an efficient MCP implementation. First, the original MCP algorithm is slightly modified to reduce its complexity. In Step 2 of the original algorithm, ties were broken by all the descendants of the node. Therefore, the complexity of Step 2 is $O(N^2 \log N)$. Experiments showed that it is not necessary to use all descendants to break ties. Instead, breaking ties by differentiating one level of descendants can produce almost the same

result. Thus, the complexity of Step 2 can be reduced to $O(E + N \log N)$. The simplified MCP algorithm is presented as follows.

The Simplified MCP Algorithm

Step 1. Compute the ALAP time of each node.

Step 2. Sort the nodes in the ascending order of ALAP times. Ties are broken by the child that has the smallest ALAP time.

Step 3. Schedule the first node in the list to the PE that allows the earliest execution, using the insertion approach. Remove the node from the list and repeat Step 3 until the node list is empty.

This algorithm can be implemented efficiently with a number of methods including the *extended ALAP time*, *ready* time calculation, and the *hole-list*. Figure 1 gives the details of the MCP algorithm implementation. In the first step, to compute the ALAP time $T_{alap}(n_i)$ for each node n_i , $T_{flevel}(n_i)$ is computed first. The T_{flevel} values are obtained by upwards-traversing through every edge starting from the exit node. At the same time, the critical path length L_{CP} is obtained, which is equal to T_{flevel} of the start node. Then, the ALAP time is obtained by $T_{alap}(n_i) = L_{CP} - T_{flevel}(n_i) - w_n(n_i)$.

In the second step, the nodes are sorted in an ascending order of ALAP times. Ties are broken by the *critical child* that has the smallest ALAP time. If the critical children have the same ALAP time, ties are broken randomly. This is done by sorting the *extended ALAP time* that concatenates the ALAP times of the node and its critical child.

The last and the most crucial step schedules all nodes onto m target PEs to minimize the schedule length, assigning each node a PE number and a start time at the PE. To describe this step succinctly the following terms are defined:

- for node n_i , the *message arrival time* from parent node n_j is $T_{msg}(n_j) = T_{start}(n_j) + w_n(n_j) + w_l(n_j, n_i)$, where $T_{start}(n_j)$ is the start time of parent node n_j ;
- from a node's perspective, the *ready time*, T_{ready} , is the earliest ready time considering all parents' message arrival times;

-
1. Compute ALAP time for each node n_i , initialize $T_{flevel}(n_i)=0$.
 - (a) for each node, set the reference count equal to the number of its children, and initialize the ready-list with all nodes that have no child
 - (b) while the ready-list is not empty, get a node n_k from the list
 - for each of node n_k 's parent node n_j
 - decrement n_j 's reference count, if it becomes zero, append n_j to the ready-list
 - let $T_{flevel}(n_j) = \text{MAX}(T_{flevel}(n_j), T_{flevel}(n_k)+w_n(n_k)+ w_l(n_j,n_k))$
 - update the critical path length $L_{CP} = \text{MAX}(L_{CP}, T_{flevel}(n_j)+w_n(n_j))$
 - (c) for each node n_i , $T_{alap}(n_i) = L_{CP} - T_{flevel}(n_i) - w_n(n_i)$
 2. Sort all nodes.
 - (a) for node n_i , compute its extended ALAP time $T'_{alap}(n_i)$, initially $T'_{alap}(n_i) = L_{CP}$
 - for each of its child node n_j $T'_{alap}(n_i) = \text{MIN}(T'_{alap}(n_i), T_{alap}(n_j))$
 - $T'_{alap}(n_i) = (L_{CP} + 1) \times T_{alap}(n_i) + T'_{alap}(n_i)$
 - (b) sort all nodes to generate a node-list with the node of the least T'_{alap} value at the top
 3. Schedule all nodes onto m target PEs, each node n_i has two attributes:

$PE(n_i) = k$, node n_i is scheduled on the target PE k , and $T_{start}(n_i)$, start time of node n_i

every PE has a hole-list, initialized by a hole with $T_{holestart} = 0$ and $T_{holeend} = \infty$

while the node-list is not empty, get a node n_i from the head of the list

 - (a) initially set $T'_{ready} = T''_{ready} = 0$, $T_{start}(n_i) = \infty$, $PE_l = 0$
 - (b) for each of node n_i 's parent node n_j
 - if $((t_{msg} = T_{start}(n_j)+w_n(n_j)+w_l(n_j, n_i)) > T'_{ready})$ $\{T'_{ready} = t_{msg}, PE_l = PE(n_j)\}$
 - (c) for each of node n_i 's parent node n_j , $t_{msg} = T_{start}(n_j) + w_n(n_j)$
 - if $(PE_l \neq PE(n_j))$ $t_{msg} = t_{msg} + w_l(n_j, n_i)$
 - if $(t_{msg} > T''_{ready})$ $T''_{ready} = t_{msg}$
 - (d) for each PE k
 - if $(k \equiv PE_l)$ then $T_{ready} = T''_{ready}$ else $T_{ready} = T'_{ready}$
 - traverse the hole-list of PE k from the beginning, for each hole
 - $T_{avail} = \text{MAX}(T_{ready}, T_{holestart})$
 - if $((T_{holeend} - T_{avail}) \geq w_n(n_i))$ exit from traverse
 - if $(T_{avail} < T_{start}(n_i))$ $\{T_{start}(n_i) = T_{avail}, PE(n_i) = k\}$
 - (e) update the start and end time of the hole, split it into two if needed

Figure 1: The detailed MCP algorithm.

- from a PE's perspective, the *available time*, T_{avail} , is the earliest available time considering all nodes that have been assigned to the PE; and
- the *earliest start time* T_{start} is the minimum of T_{avail} among all PEs.

The nodes are scheduled one by one in the order of the node list. For each node, the *ready time* is computed at every PE. Then, the *available time* T_{avail} at each PE is obtained by searching through the *hole-list*. Finally, the earliest start time T_{start} is assigned to the node. To compute the *ready time* of node n_i , the *message arrival time* from its parent n_j , $T_{msg}(n_j)$ is computed. The PE PE_l is the PE where the parent node that results in the largest T_{msg} value resides. All PEs except PE_l have the same *ready time* which is $MAX(T_{msg}(n_j))$. Node n_i may have a smaller *ready time* on PE_l since the edge weight between the parent node and the child node is zeroed. Thus, the *ready time* is computed twice, one for PE_l , and one for all other PEs. The *ready time* is the earliest possible time for a node considering all incoming messages, however, there must be a sufficient space to schedule the node. To search the first available slot, a *hole-list* is built for each PE. Every hole in the list has a start time $T_{holestart}$ and an end time $T_{holeend}$. If $(T_{holeend} - MAX(T_{ready}, T_{holestart})) \geq w_n(n_i)$, the node can be scheduled and $T_{avail}(n_i)$ is assigned. The earliest start time is the minimum of T_{avail} among all PEs and the node is scheduled to the PE that provides the earliest start time. Then the start and end time of the hole is updated. In certain case, that is, $T_{ready} > T_{holestart}$, the hole will be split into two holes.

The code can be found at <http://www.eece.unm.edu/~wu/mcp>.

4 Complexity Analysis

In the first step, the ALAP time is computed by traversing each edge in the graph and its complexity is $O(E)$. Step 2 sorts the nodes in the ascending order of ALAP times and the complexity is $O(N \log N)$. Ties are broken only by the first child that has the smallest ALAP time. The cost of comparing every child of a node is $O(|CHILD(n_i)|)$, and for the N nodes, the cost is $\sum O(|CHILD(n_i)|) = O(E)$. Therefore, the complexity becomes $O(E + N \log N)$. In Step 3, for each node n_i , to find the *ready time*, the cost is $O(|PARENT(n_i)|)$ since every parent of node n_i is examined. For N nodes, the cost is $\sum O(|PARENT(n_i)|) = O(E)$. Then node n_i looks for the earliest time T_{avail} by searching the hole-list in each PE. Since the total number of holes is less than N , the complexity of scheduling a single node is $O(n)$ and Step 3, to schedule all N nodes, is with complexity of $O(N^2)$. Therefore, the complexity of MCP is $O(E + N \log N + N^2)$ or $O(N^2)$ since $E < N^2$.

5 Concluding Remarks

The MCP algorithm for scheduling DAG with communication costs onto a bounded number of PEs is a high-performance, low-complexity algorithm. The lowest possible complexity of DAG scheduling is $O(E)$ while the complexity of MCP is $O(N^2)$. The rule of thumb in scheduling is that a slight more effort can lead to a performance that is slightly lower than optimum. It has been shown that this algorithm runs fast and performs reasonably well and is the best one among the six most popular algorithms in this category. A detailed algorithm description and its implementation, as well as the complexity analysis have been presented in this paper. This algorithm implementation may be used as a basis for comparison of newly invented algorithms in terms of schedule length, complexity, and running time.

References

- [1] M. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, pp. 330–343, July 1990.
- [2] Y. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 381–422, 1999.
- [3] M. Gary and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
- [4] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, June 1990.
- [5] T. Yang and A. Gerasoulis, "DSC: Scheduling parallel tasks on an unbounded number of processors," *IEEE Trans. Parallel and Distributed System*, vol. 5, pp. 951–967, Sept. 1994.
- [6] Y. Chung and S. Ranka, "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors," in *Supercomputer '92*, Nov. 1992.
- [7] V. Sarkar, "Unconstrained static scheduling with communication weights", *Journal of Scheduling*, vol.5, no.5, pp.359-377, September, 2002.
- [8] A. Radulescu, A. van Gemund, "Low-cost task scheduling for distributed-memory machines", *IEEE Transactions on Parallel and Distributed Systems*, vol.13, no.6, pp.648-658, June, 2002.
- [9] S. Fujita, M. Yamashita, "Approximation algorithms for multiprocessor scheduling problem", *IEICE Transactions on Information and Systems*, vol.E83D, no.3, pp.503-509, March, 2000.
- [10] M. Al-Mouhamed, H. Najjari, "Adaptive scheduling of computations and communications on distributed-memory systems", *Journal of Parallel and Distributed Computing*, vol.60, no.6, pp.716-740, June 2000.
- [11] T. Kalimowski, I. Kort, D. Trystram, "List scheduling of general task graphs under LogP", *Parallel Computing*, vol.26, no.9, pp.1109-1128, August, 2000.
- [12] J.N. Cao, O. De Vel, L. Shi, "Rapid prototyping of distributed algorithms", *Journal of Systems and Software*, vol.45, no.2, pp.141-154, March 1999.

- [13] S. Darbha, D.P. Agrawal, "Optimal scheduling algorithm for distributed-memory machines", *IEEE Transactions on Parallel and Distributed Systems*, vol.9, no.1, pp.87-95, January 1998.
- [14] J.C. Liou, M.A. Palis, "On the effectiveness of compiler-time scheduling approaches for distributed memory multiprocessor", *Journal of Information Science and Engineering*, vol.14, no.1, pp.7-26, March 1998.
- [15] A. Moukrim, A. Quilliot, "Scheduling with communication delays and data routing in message passing architectures", pp.438-451, 1998.
- [16] G.J. Lai, C. Chen, "New program model for program partitioning on NUMA multiprocessor systems", *IEE Proceedings-Computers and Digital Techniques*, vol.143, no.6, pp.431-435, November 1996.
- [17] T. Yang, A. Gerasoulis, "Executing scheduled task graphs on message-passing architectures", *International Journal of High Speed Computing*, vol.8, no.3, pp.271-294, September 1996.
- [18] . Chung, W.H. Ho, C.C. Liu, "A parallel programming tool for distributed-memory multiprocessors", *Journal of the Chinese Institute of Engineers*, vol.18, no.3, pp.365-378, May 1995.
- [19] Y. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406-471, 1999.
- [20] M. Wu and W. Shu, "On parallelization of static scheduling algorithms," *IEEE Transactions on Software Engineering*, vol. 23, pp. 517-528, Aug. 1997.