The Problem With Critical Path Scheduling Algorithms

C. L. McCreary, M. A. Cleveland, A. A. Khan
Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849
mccreary@eng.auburn.edu, clem@meng.auburn.edu

Abstract

Multi-processor scheduling is an NP-hard problem, so heuristic solutions must be found. Several critical path heuristics have been proposed, but these must be empirically tested to see if their solutions are adequate. This paper examines the performance of four critical path heuristics and finds that they are unable to perform well on the types of graphs which are likely to be created by compilers for parallel machines. The cause for this failure is examined.
1 Introduction

One of the primary problems in executing programs efficiently on multiprocessor systems with distributed memory is to partition the program into tasks that can be assigned to different processors for parallel execution. If the partitioning results in a high degree of parallelism, a greater amount of communication will be required among the tasks. On the other hand, if communication is restricted, potential parallelism will be lost. The goal of partitioning heuristics is to divide the program into appropriate size and number of tasks to balance communication overhead and parallelism so that the completion time is minimized. A sequential program is commonly represented as a Program Dependence Graph (PDG) which is a directed acyclic graph (DAG) with node and edge weights [1]. Each vertex in a PDG denotes a task and is associated with weight, which is its processing time. Each edge denotes the precedence relation between the two tasks, and the weight of the edge represents the communication cost incurred if the two tasks are assigned to different processors. Given a PDG, the graph is partitioned into appropriately sized groups of nodes (grain) which are assigned to processors of a parallel machine. The partitioning and assignment are known as the scheduling problem. The problem is also called grain size determination [2], the clustering problem [3, 4], and internalization pre-pass [1].

The partitioning/scheduling problem is intractable, and heuristics are required to find sub-optimal solutions. As the result, there are no performance guarantees for scheduling heuristics for general graphs. Many researchers have proposed algorithms, but little work has been done to determine the performance of the resulting schedules. Since neither analytic nor experimental results are known, the goal of this paper is to study some promising techniques, experimentally observe their performance and analyze their strengths and weaknesses.

In classical scheduling, communication costs are not considered [5, 6]. Introducing the communication cost is necessary because communication between processors does take considerable time in real parallel systems, especially in distributed memory systems where communication costs tend to be high relative to processor speed. The challenge in the extended scheduling problem is to consider the trade-off between communication time and degree of parallelism [7]. Many researchers have studied the problem and proposed solutions. Based on the techniques employed, the earlier methods can be classified into the following three categories:

- **Critical path heuristics** [1, 3, 4, 8, 10, 11]: For DAGs with edge weights and node weights, a path weight is defined to be the sum of the weights of both nodes and edges on the path. A critical path is a path of greatest weight from a source node to a sink node. Extending the critical path method due to Hu [12] in classical scheduling, these algorithms try to shorten the longest execution path in the DAG. Paths are shortened by removing communication requirements (zeroing edges) and combining the adjacent tasks into a grain. This approach has received the most attention. A taxonomy of these techniques as well as a comparison of four specific heuristics can be found in the work of Gerasoulis and Yang [8]. In this paper we include experimental results from four critical path algorithms, DSC, DCP, MCP and HU.
• **List scheduling heuristics** [2, 7, 14, 15, 16, 17, 18]: These algorithms assign priorities to the tasks and schedule them according to a list priority scheme. For example, a high priority might be given to a task with many heavily weighted incident edges or to a task whose neighbors have already been scheduled. Extending the list scheduling heuristic in classical scheduling [6], these algorithms use greedy heuristics and schedule tasks in a certain order. Task duplications have been used in [2, 14, 18] to reduce the communication costs.

• **Graph decomposition method** [20, 21]: Based on graph decomposition theory, the method parses a graph into a hierarchy (tree) of subgraphs representing the independent/precedence relationship among groups of tasks. Communication and execution costs are applied to the tree to determine the grain size that results in the most efficient schedule. The method can be conceptualized as orthogonal to the critical path methods in that it inspects graph sections that are independent and can be parallelized rather than concentrating on sequential paths. The only published method in this category is CLANS.

This paper describes the results of our experimentation with four critical path heuristics. All of these methods fail to produce a schedule that is as short as a single processor schedule under certain conditions. These conditions will be explained and the erroneous decisions of the schedulers will be shown. Since scheduling is an intractable problem, clearly, there will be cases where schedules generated by any heuristic will fail to be optimal. However, producing schedules that are worse than serial is unacceptable.

Section 2 defines the problem and looks at assumptions we have made. Section 3 introduces the heuristics behind the algorithms we have tested. Section 4 describes the experiment. Section 5 appraises the performance of the algorithms. Section 6 looks at the reasons critical path heuristics fail, and section 7 examines expected performance on real machines, summarizing the implications of our findings.

## 2 Problem Definition and Assumptions

The problem of parallelizing the PDG to achieve minimal parallel time is NP-complete [1]. This process usually involves two steps that are either treated as distinct parts of the algorithm or are combined in a single step: partitioning and scheduling. Partitioning combines tasks from the PDG into groups of tasks called grains to be executed on the same processor. The optimal size of a grain is dependent on the characteristics of the target architecture. Processors whose interprocessor communication costs are high relative to processing costs require larger grains than processors with low inter-processor communication costs. Scheduling assigns grains to processors, insuring that all grain inputs are available at the scheduled start time. Associated with each scheduled grain is a processor number, a start time and a completion time (start time plus execution time).

The following simplifying assumptions are made in order to evaluate all the methods equally.

1. All methods use a DAG representing the PDG as input and consider the costs of inter-task communication. Two tasks scheduled on the same processor incur no communication costs, and any two tasks
scheduled on two different processors incur the communication cost specified by the edge weight in the PDG no matter which two processors are involved.

2. The architecture is a network of an arbitrary number of homogeneous processors.

3. Duplication of tasks in separate grains is not allowed. Several heuristics [2, 14, 18] have been developed that take advantage of this option, but duplication adds additional complexity to an already intractable problem that none of our competing methods use.

4. Tasks communicate only before starting and after completing their execution.

5. Communication occurs simultaneously with computation and tasks communicate via asynchronous mechanisms whether message passing or shared memory.

6. Tasks may send and receive data in parallel and may multicast messages to more than one receiver.

7. The objective function of all the heuristics is to minimize parallel time.

3 The Heuristics

This section introduces four critical path heuristics: the dominant sequence clustering (DSC) [22] algorithm by Gerasoulis and Yang, the modified critical path (MCP) [11] algorithm by Wu and Gajski, the Hu algorithm as modified by Lewis and El-Rewini [19], and the dynamic critical path (DCP) algorithm by Kwok and Ahmad [13].

In critical path techniques, partitions are formed by examining the current critical path, zeroing an edge (combining the incident nodes into a cluster) in that path and repeating the process on the new critical path. Gerasoulis and Yang use the term dominant sequence to mean the critical path after the zeroing of one or more edges. The heuristics differ in their method of selecting the edge(s) to be zeroed and identifying the new dominant sequence. The detailed algorithms are found in the Appendix.

3.1 Dominant Sequence Clustering

The two major goals of DSC [22] are to directly attempt to reduce the dominant sequence of the graph, and to reduce the complexity of the algorithm, to $O((v + e) \log v)$. The complexity is low because at each step the zeroing decision is based on a comparison of the affect on 2 values: the new start time of the node at the beginning of the dominant sequence and the start time of an unscheduled node most likely to be affected by the zeroing decision. If either node's start time is increased, the zeroing does not occur.

3.2 Modified Critical Path

MCP, proposed by Wu and Gajski, is similar to DSC in that its basic heuristic is to minimize the starting time of individual tasks. MCP considers the ALAP (as-late-as-possible) binding which refers to the latest
possible start times for each task. The ALAP binding is found by traversing the DAG from the sink nodes to the source nodes and assigning the latest possible start time to each node. A critical node is selected and placed on either a new processor or the best existing processor. The complexity of MCP is $O(n^2 \log v)$.

3.3 Hu

Lewis and El-Rewini modified the classic Hu algorithm to include communication costs. Hu obtains a priority by finding the level. All nodes that have no unscheduled predecessors are put in a free list in decreasing priority order. The top node from the free node list is then scheduled after all of its predecessors scheduled are put into the free list. [19]

3.4 Dynamic Critical Path

DCP by Kwok and Ahmad determines the order of node evaluation dynamically by finding the critical path after each step in scheduling. It avoids scheduling graph nodes until all nodes have been considered, allowing the partial schedule on a processor to change during the scheduling process. The algorithm also looks ahead to the start times of a node’s children when selecting a processor.[13]

4 The Experiment

For parallel programs, granularity as defined by Sarkar in [1] is: the average size of a sequential unit of computation in the program. This does not consider the weight of the communication between the tasks. To take into account of the edge weights, we follow the concept of granularity of a parallel program as proposed by Gerasoulis and Yang [10] to be the ratio of the computation cost to communication cost.

Definition: granularity of a weighted DAG is the average ratio of node weight to maximum adjacent outgoing edge weight. The formula for granularity is:

$$\text{Granularity} = \frac{1}{N - S} \left( \sum_{i=0}^{N-S} \frac{w_i}{\max[w_{e_{ij}}]} \right)$$

where:

- $N$ is the total number of nodes in the graph.
- $S$ is the total number of sink nodes in the graph.$^1$
- $w_i$ is the weight of the $i^{th}$ node that is not a sink.
- $\max[w_{e_{ij}}]$ is the maximally weighted outgoing edge from node $i$ to some node $j$.

$^1$Note that sink nodes do not contribute to communication delay. Therefore, they are left out of the average.
The granularity of a DAG determines the amount of useful parallelism available. In general, a granularity of 1 indicates an even trade-off between communication and execution costs. For granularities less than 1, more efficient schedules are produced when PDG nodes are combined more frequently to produce a clustered granularity greater than 1. When granularity is high, more parallelism can be extracted from the PDG. Gerasoulis et al. [10] call graphs with granularity > 1 coarse grained. It is proven that for any coarse grained graph, any list scheduling heuristic gives a schedule within a factor of 2 of that of the optimal schedule [6].

For the comparison of the heuristics, the following five granularity levels were chosen:

- Granularity < 0.08
- 0.08 < Granularity < 0.2
- 0.2 < Granularity < 0.8
- 0.8 < Granularity < 2.0
- 2.0 < Granularity

The primary goal of all the critical path algorithms is to reduce the time it takes to execute a program. This time is known as latency or parallel time and is the elapsed time beginning with the first processor to execute the program and ending with the last processor to complete the execution. The relative success of this goal can be determined by computing the speedup attained by a heuristic on a set of graphs. Speedup is defined as the ratio of the parallel time to the serial time:

$$\text{Speedup} = \frac{\text{ParallelTime}}{\text{SerialTime}}$$

Clearly, a ratio of one indicates that the serial and parallel times are identical. A ratio greater than one shows an actual slowdown of time for the parallel program. A slowdown can occur when communication overhead is excessive. Our experiments show that when communication costs are high relative to execution costs, all the critical path algorithms produce schedules which execute slower than the serial execution!

In this experiment, more than 500 DAGs were randomly generated for each of 5 granularity classes. Each of the four algorithms was run on all the graphs and the computed for each graph. Table 1 gives the percentage of graphs in each granularity class for which the speedup exceeded one. At low granularities the heuristics are extremely ineffective and often produce schedules that retard execution time. In particular, when granularity is less than 0.08, DSC, HU and MCP schedulers give a retardation instead of a speedup over 50

5 How Critical Path Algorithms Fail

The problem with critical path algorithms comes from the fact that they are greedy algorithms making choices based on localized information. These algorithms all give priority to scheduling nodes on the critical
<table>
<thead>
<tr>
<th>G&lt;0.08</th>
<th>dcp</th>
<th>dsc</th>
<th>hu</th>
<th>mcp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.08&lt;G&lt;0.2</td>
<td>38.2</td>
<td>61.2</td>
<td>100.0</td>
<td>54.3</td>
</tr>
<tr>
<td>0.2&lt;G&lt;0.8</td>
<td>14.7</td>
<td>32.4</td>
<td>97.4</td>
<td>24.9</td>
</tr>
<tr>
<td>0.8&lt;G&lt;2.0</td>
<td>0.0</td>
<td>0.2</td>
<td>55.0</td>
<td>0.5</td>
</tr>
<tr>
<td>2.0&lt;G</td>
<td>0.0</td>
<td>0.0</td>
<td>14.1</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 1: Percentage of graphs for which each heuristic gives a speedup less than 1 for the given granularity.

They examine nodes individually in order of their priority and decide if they should be scheduled with previously scheduled nodes or on a separate processor. Therefore, the order of node examination, or the evaluation of node priority is most important for these algorithms since missing the optimal scheduling at any step would affect the successive steps and thus lead the result far from the optimal one. Unfortunately, the evaluation of node priority in these algorithms is based only on the local information such as previously scheduled nodes and the scheduling node. Using additional information such as DCP's looking one step ahead can't overcome such drawback, because its mere intelligence lies in tentatively adding the current node into local information and then using the local information to examine current node's child. Decisions, while considering more information, are still local.

The nature of the existing problem with critical path algorithms is that shortening the critical path by merging a few local edges doesn't correspond to finding near-optimal schedule. After a step of reducing the critical path length, the critical path may change into another set of edges. Therefore, after any step of local analysis, these algorithms may distort the goal from finding an optimal schedule to reducing the consequent critical path by merging a few local edges.

To demonstrate, in figure 1 we present a simple nine node graph where each algorithm finds a schedule worse than serial. All of the critical path algorithms produce a schedule with length 180 for this graph, but the serial schedule is 159, and the optimal schedule is 149. Like most graphs which cause these algorithms to produce worse than serial schedules, this graph has small granularity.

Let's see how the order of node evaluation is a major and unavoidable source of error in critical path heuristics. Our sample graph provides a good example of this with the DSC scheduler. For the graph as presented, DSC like the others, creates a worse than serial schedule. Changing the value of the edge $(7,8)$ which should be unimportant when scheduling this graph, changes the DSC schedule dramatically. With a value of 30 instead of 10 for the edge DSC finds the optimal schedule. Interestingly, the optimal schedule is the same in the two graphs. The order in which DSC examines the nodes is changed, and so different decisions are made when scheduling. This happens despite the fact that DSC determines examination order during evaluation, rather than pre-determining evaluation order.

These algorithms all give priority to scheduling nodes on the critical path in an effort to reduce the critical path length, and this can cause the algorithms to fail. For example, the DCP scheduler uses a heuristic to dynamically determine the order of node evaluation, and it looks ahead to the start times of child nodes in order to make better scheduling choices, but it is still flawed in the same basic way. The first
Serial Schedule = 159
Critical Path Length = 382

Optimal Schedule = 149

Critical Path Schedule = 180
(for DCP, DSC, HU, and MCP)

Figure 1: Example of scheduling failure.
few steps of scheduling the graph will illustrate this. After examining the trivial case of node 1, the DCP scheduler examines node 2 which is seen to have the highest priority since it is on the Dynamic Critical Path. In the local view of nodes 1 and 2 which are connected by an edge of weight 20 it makes sense to combine the nodes and zero that edge, but in relation to the whole graph this decision already misses the optimal schedule. Next node 5 is grouped with nodes 1 and 2. Now node 4 is examined and determined to make a good fit into the cluster with 1, 2 and 5. This decision, after examining less than half of the nodes has doomed the final schedule to be worse than serial.

In graphs with small granularity the edge weights are so large relative to the node weights that critical path algorithms chose to remove almost every edge they come to until they reach edges whose removal increases the overall schedule length. This ignores the fact that many large edges will also be encountered later in the scheduling process, and ensures an inefficient schedule early in the process. From the start, critical path algorithms seek to shorten the critical path. As seen in the example they all reduce the critical path by over 200, but focusing on the critical path is much too local since drastically reducing the critical path length may or may not correspond to finding a better than serial schedule. As long as an algorithm follows the critical path method this problem will exist because decisions are made relative to only a few local edges. Algorithms which take a more global view of the graph must be developed.

6 Hardware Granularity for Real Systems

We have demonstrated that for low granularities, critical path and list schedulers should not be used, while for granularities exceeding 1, all the critical path heuristics are good. On real systems, it is fair to ask what granularities are expected.

For distributed memory multiprocessor systems, the overhead required for message passing can be measured and there are several studies that do just that [23, 24]. Dunigan measures round-trip time for 1000 iterations and computes one-hop and six-hop data rates on three distribute memory machines, the mesh-connected touchstone DELTA and two hypercubes, iPSC/860 and Ncube 6400. The hop penalty is negligible on the DELTA and barely significant on the hypercubes. The communication time is nearly linear in proportion to the message size, N and is given by the equation, $T = \alpha + \beta N + (1-1)\gamma$ where $\alpha$ is a startup time, $\beta$ is a transport time, and $\gamma$ is a per hop delay. Dunigan computes the coefficients of the timing equation using a least-squares estimate of the data collected from the testing on an otherwise idle system. In testing with contention, the expected slowing is much more significant in the hypercubes than the mesh. For the three systems the cost of an 8-byte transfer on an idle system ranges from 62 to 161 $\mu$s. Compare that with the time required for computation in an 8-byte multiply[Table 2], a time which ranges from 0.08 to 1.5 $\mu$s on these systems, and it's clear that the granularity is very low. If an execution block contained 100 multiplication steps, the granularity would range from 0.1 to 0.932 $\mu$s on these systems.

With multiprocessing capabilities on workstation networks through tools such as PVM, the communication/execution imbalance is even more exacerbated.
Table 2: Communication Performance of Distributed Memory Multiprocessor System.

<table>
<thead>
<tr>
<th></th>
<th>touchstone DELTA</th>
<th>iPSC/860</th>
<th>Ncube 6400</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-byte transfer time(μs)</td>
<td>62</td>
<td>80</td>
<td>161</td>
</tr>
<tr>
<td>8-byte multiply time(μs)</td>
<td>0.08</td>
<td>0.08</td>
<td>1.5</td>
</tr>
<tr>
<td>Comp./Comm.</td>
<td>0.00129</td>
<td>0.001</td>
<td>0.00932</td>
</tr>
</tbody>
</table>

Summary

Our experiments have shown that even some of the most respected critical path algorithms produce multiprocessor schedules for a program that take time longer than if the program were run on a single processor. Designers of multiprocessor scheduling algorithms should be aware of these possible flaws and carefully check for them. Problems occur most frequently when communication costs are high relative to execution costs, and this can easily be the case on distributed memory multiprocessors. Furthermore, for newer systems the problem is worse than that for older systems because with new technology, processing speeds are increasing at a faster rate than communication speed.

Acknowledgments

The DSC algorithm was provided by Apostolos Gerasoulis and Tao Yang of Rutgers University.

References


Appendix A

A.1 Dominant Sequence Clustering (DSC)

To describe DSC we need to specify certain constraints and definitions.

Node types:

- **scheduled**: A node is scheduled if it has been assigned to a processor.
- **free**: A node is free if it is unscheduled and all its predecessors are scheduled.
- **partial free**: A node is partial free if it is unscheduled and at least one of its predecessors is unscheduled.

The reduction in the focus range is achieved by maintaining lists of free and partial free nodes. The lists are ordered according to priority as defined below. At any one iteration in the algorithm, the focus is on the first elements in these lists and the outgoing edges of the free node in question. This reduces the complexity by confining the range of edges to be zeroed.

Timing Values:

- **ST(n_x)**: the starting time for n_x when scheduled on an independent cluster.
- **ST(c_i,n_x)**: starting time for n_x when scheduled in cluster/processor c_i.
- **ET(n_x)**: the execution time of n_x.
- **level(n_x)**: the length of the longest path from the start of n_x to an exit node.
- **arrivet ime(n_j,n_x)** ≡ **ST(n_j) + ET(n_j) + e_jx** where n_j is a scheduled predecessor of n_x, and e_jx is the cost of edge (n_j,n_x).
- **startbound(n_x)** ≡ **max(arrivet ime(n_j,n_x))**, where n_j ∈ **PRED(n_x)**. This is the lower bound for starting n_x.

Using these definitions, we define priority for nodes in the free and partial free lists as follows.

\[
priority(n_x) ≡ startbound(n_x) + level(n_x)
\]

\[
PT ≡ ParallelTime
\]

Figure 2 gives the DSC algorithm and Figure 3 illustrates the steps in the execution of DSC with an example.
The Algorithm (DSC):
Mark all nodes as *unscheduled*

WHILE there are *unscheduled* nodes DO

In descending order of priority, sort list of *free* nodes

In descending order of priority, sort list of *partial* free nodes

Let \( n_x \) be the *free* task with highest priority.

Let \( n_y \) be the *partial* free task with highest priority.

IF(priority\( n_x \) = priority\( n_y \)) THEN

IF ( CT1) THEN accept zeroing for edge (\( c_i, n_x \)) where \( c_i \) gives min\( ST(c_i, n_x) \)

ELSE schedule \( n_x \) on new cluster

ENDIF

ELSE

IF ( CT2 & CT1) THEN accept zeroing for edge (\( c_i, n_x \)) where \( c_i \) gives min\( ST(c_i, n_x) \)

ELSE schedule \( n_x \) on new cluster

ENDIF

ENDIF

Mark \( n_x \) as *scheduled*

Recalculate priorities & find new *free* & *partial* free nodes

ENDWHILE

DEFINE CT1: /* Constraint 1 guarantees that parallel time is not increased */

For all clusters \( c_i \) that are parents of \( n_x \) (min_start_time = MAXINT)

{ IF ( min_start_time > ST\( (c_i, n_x) \)) THEN min_start_time = ST\( (c_i, n_x) \) } 

IF ( startbound\( (n_x) \) < min_start_time ) THEN

return TRUE

ELSE

return FALSE

ENDIF

ENDDEF

DEFINE CT2: /* Constraint 2 guarantees that the start time of partial free nodes is never increased */

For all scheduled clusters \( c_i \) that are parents of \( n_y \)

{ IF (startbound\( (n_y) \) < ST\( (c_i, n_y) \)) THEN

return FALSE

ENDIF

} 

return TRUE

ENDDEF

Figure 2: DSC algorithm.
Figure 3: DSC example.
A.2 Modified Critical Path

MCP, proposed by Wu and Gajski, is similar to DSC in that its basic heuristic is to minimize the starting time of individual tasks using an ALAP strategy. The term ALAP (as-late-as-possible) binding, refers to the latest possible start times for each task. The ALAP binding is found by traversing the DAG from the sink nodes to the source nodes and assigning the latest possible start time $T_L(n_i)$ to each node. The ALAP start times allow MCP to locate the critical path. For each node, the algorithm maintains a list $l(n_i)$ which consists of a $T_L$ for each $n_i$ and all its descendants. The lists $l(n_i)$ are in turn ordered and the list $L$ of nodes is created according to the order of lists $l(n_i)$. See Figure 4 for the algorithm and Figure 5 for an example of its execution steps. The complexity of MCP is $O(n^2\log n)$.

The Algorithm (MCP):

List_of_PEs = $\emptyset$

1. Perform ALAP binding and assign the resulting ALAP time $T_L$ to each node in the graph.

2. For each node $n_i$ create a list $l(n_i)$ which consists of $T_L$’s of $n_i$ and all it’s descendants, sort $l(n_i)$ in decreasing order of $T_L$.

3. Create $L$ by concatenating the $T_L$ values for each $l(n_i)$ and sort in decreasing order.

4. Schedule $head(L)$ to a Processing Element (PE) and remove $head(L)$
   List_of_PEs $\leftarrow$ PE$(head(L))$

5. While $L$ is not empty
   Compute min(start_time) when $head(L)$ is placed on PE in List_of_PEs.
   If min(start_time exist_cluster) > start time of $head(L)$ when placed on a new PE
      then
      add new PE to List_of_PEs and place $head(L)$ on new PE
   else
      place $head(L)$ on PE that gives min(starttime)

Figure 4: MCP algorithm.

A.3 Hu

Lewis and El-Rewini modified the classic Hu algorithm to include communication costs. Hu obtains a priority by finding the level. All nodes that have no unscheduled predecessors are put in a free list in decreasing priority order. The top node from the free node list is then scheduled after all of its predecessors scheduled are put into the free list. [19]. See Figure 6 and Figure 7 for an example of its execution steps.
TL(n1) = 0
TL(n2) = 12
TL(n3) = 57
TL(n4) = 48
TL(n5) = 62
TL(n6) = 87
TL(n7) = 78
TL(n8) = 87
TL(n9) = 99

l(n1) = (n9, 99), (n8, 87), (n6, 87), (n7, 78), (n5, 62),
(n3, 57), (n4, 48), (n2, 12)
l(n2) = (n9, 99), (n8, 87), (n7, 78), (n5, 62), (n4, 48),
(n2, 12)
l(n3) = (n9, 99), (n6, 87)
l(n4) = (n9, 99), (n8, 87)
l(n5) = (n9, 99), (n5, 62)
l(n6) = (n9, 99), (n6, 87)
l(n7) = (n9, 99), (n7, 78)
l(n8) = (n9, 99), (n8, 87)
l(n9) = (n9, 99)

L = {n1, n2, n4, n7, n5, n3, n8, n6, n9}

head(L) = n1 => PE1 <- n1
head(L) = n2 => min(start_time(n2)) => PE1 <- n2
head(L) = n4 => min(start_time(n4)) => PE1 <- n4
head(L) = n7 => min(start_time(n7)) => PE2 <- n7
head(L) = n5 => min(start_time(n5)) => PE1 <- n5
head(L) = n3 => min(start_time(n3)) => PE2 <- n3
head(L) = n8 => min(start_time(n8)) => PE1 <- n8
head(L) = n6 => min(start_time(n6)) => PE3 <- n6
head(L) = n9 => min(start_time(n9)) => PE1 <- n9

PT = 124

Figure 5: MCP example.
The Algorithm (HU)

Find the level for each task and use it as the task’s priority
Put into free list all nodes which have no predecessors
   in order by priority
Get task t off the free list
Assign t to the first processor
repeat
   update free list
   if (free list not empty)
      get next t off the free list
      Find processor with earliest start time
      Assign t to this processor
until all tasks have been assigned to processors
Figure 7: Hu example.
A.4 Dynamic Critical Path

To identify the nodes for clustering at each step, the DCP has using the following lower and upper bound on the start time of a node:

The absolute earliest start time of a node $n_i$ in a linear cluster $J$, denoted by $\text{AEST}(n_i, J)$, is recursively defined as $\max_{1 \leq k \leq p} \{\text{AEST}(n_i, \text{LC}(n_i)) + w(n_{ik}) + c_{ik}\}$, where $n_i$ has $p$ parent nodes and $n_{ik}$ is the $k$-th parent node; $\text{LC}(n_i)$ denotes the linear cluster containing the node $n_i$; $w(n_{ik})$ denotes the computation cost while $c_{ik}$ the communication cost. Similarly the absolute latest start time $\text{ALST}(n_i, J)$ is defined as $\min_{1 \leq k \leq q} \{\text{ALST}(n_i, \text{LC}(n_i)) - c_{ii} - w(n_i)\}$, where $n_i$ has $q$ children nodes and $n_{ik}$ is the $k$-th child node. Based on the definition of AEST and ALST, the DCP node is denoted as a node whose AEST and ALST are equal.

The DCP algorithm first clusters all DCP nodes and then updates the AEST and ALST values dynamically after each clustering step to identify the next DCP node. DCP nodes are processed differently by the algorithm from NOT-DCP nodes. If a node is DCP node, only the linear clusters containing its parent and children nodes are considered. If it is not a DCP node, all linear clusters are considered. The algorithm, therefore, is composed of two procedures, the Cluster to schedule nodes and the FindSlot to determine the AEST and ALST value for the scheduling node.

Figure 8 gives the DCP algorithm and Figure 9 illustrates the steps in the execution of DCP with an example.
The Algorithm (DCP)

compute AEST and ALST for all nodes and let each node be an Unexamined Linear Cluster (ULC)
WHILE there exists some ULC DO
    \( ni \leftarrow \) the highest node with the smallest value of \( ALST(ni,ULC)-AEST(ni,ULC) \);
    break ties by choosing the one with a smallest AEST
    IF AEST(ni,ULC)=ALST(ni,ULC) THEN
        Cluster(ni,On_DCP)
    ELSE
        Cluster(ni,Not_On_DCP)
    END IF
    update all AEST’s and ALST’s
END WHILE
allocate each cluster to a distinct processors so that each node starts at its AEST

Cluster(ni,Location)
IF Location != On_DCP THEN
    Cluster(Stack)←all examined clusters
ELSE
    push on Cluster(Stack) all examined clusters containing the children nodes of ni followed by
    the ones containing the parent nodes of ni
END IF
Best_Cluster←NULL, Mixed_Cluster←infinite
WHILE Cluster(Stack) is not empty DO
    J←top of Cluster(Stack)
    This_AEST←Find_Slot(ni,J,DONT_PUSH)
    IF This_AEST=\( \infty \) AND Location = On_DCP THEN
        This_AEST←Find_Slot(ni,J,PUSH)
    END IF
    IF This_AEST \( \neq \) \( \infty \) THEN
        nc←critical child node of ni
        Tentatively insert ni into J
        Child_AEST←Find_Slot(nc,J,DONT_PUSH)
        IF Child_AEST +This_AEST<Mixed_AEST THEN
            Mixed_AEST←Child_AEST+This_AEST
            Best_Cluster←J
        END IF
    END IF
END IF
END WHILE

Find_Slot(ni,J,Condition)
Determine AEST(ni,J) and ALST(ni,J)
IF Condition = DONT_PUSH THEN
    return the earliest start time of ni in J by checking the inequality in CR without delaying
    any nodes’ AEST in J
ELSE
    return the earliest start time of ni in J by checking the inequality in CR by possibly delaying
    some nodes’ AEST in J
END IF

Figure 8: DCP algorithm.
Figure 9: DCP example.