International Journal of Foundations of Computer Science

CRITICAL PATH SCHEDULING PARALLEL PROGRAMS ON AN UNBOUNDED NUMBER OF PROCESSORS

MOURAD HAKEM
LIPN, CNRS-UMR-7030, Université Paris-Nord,
Avenue Jean-Baptiste Clément
93430 Villetaneuse, France.

and

FRANCK BUTELLE
LIPN, CNRS-UMR-7030, Université Paris-Nord,
Avenue Jean-Baptiste Clément
93430 Villetaneuse, France.

Received
Revised
Communicated by

ABSTRACT

In this paper we present an efficient algorithm for compile-time scheduling and clustering of parallel programs onto parallel processing systems with distributed memory, which is called The Dynamic Critical Path Scheduling *DCPS*. The *DCPS* is superior to several other algorithms from the literature in terms of computational complexity, processors consumption and solution quality. *DCPS* has a time complexity of $O(e + v \log v)$, as opposed to *DSC* algorithm $O((e+v) \log v)$ which is the best known algorithm. Experimental results demonstrate the superiority of *DCPS* over the *DSC* algorithm.

Keywords: Scheduling; clustering; distributed computing; precedence task graphs; directed acyclic graphs DAGs; parallel scheduling

1. Introduction

The efficient execution of tasks, that constitute a parallel program on multiprocessors systems with distributed memory, highly depends on the scheduling algorithm used to distribute the tasks into processors. If the scheduling 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 objective of scheduling heuristics is to partition the program into appropriate size and number of tasks to balance communication overhead and parallelism so that the schedule length is minimized. The partitioning/scheduling problem has been shown to be NP-complete for a general task graph [2, 4, 15], and heuristics are required to find sub-optimal solutions.

In this paper we consider the scheduling problem for general parallel programs which can be represented by a Directed Acyclic Graph (DAG). The problem of clustering general task graphs with high communication delays ($g(G) < 1$ see below) has no good solution today. So, we focus our attention on this challenging case. We propose a new approach for scheduling DAGs which achieve better schedule lengths with minimum time complexity and processors consumption. A salient feature of our algorithm is that it computes the Makespan (schedule length or parallel time) of the partially scheduled graph incrementally at each refinement step of the scheduling process.

The remainder of the paper is organized as follows: Section 2 presents the basic definitions and assumptions adopted in this paper. We recall in Section 3 principles of the best existing scheduling algorithms. Section 4 describes *DCPS* algorithm. Some theoretical results are presented in Section 5 and Section 6 shows that our heuristic can achieve optimal solutions for Join and Fork DAGs. Before concluding, we report in Section 7 some experimental results that assess the good behavior of our algorithm.

2. Basic Definitions and Notations

The execution model for task graphs is called macro-dataflow. In the macro-dataflow model, a parallel program is represented as a weighted Directed Acyclic Graph (DAG) which is defined by $G = (V, E)$ where V is the set of task nodes, $v = |V|$ is the number of nodes, E is the set of edges corresponding to the precedence relations between the tasks and $e = |E|$ is the number of edges. Let ω be a cost function on the edges ($\omega(t_1, t_2)$ represents the communication cost between t_1 and t_2, which becomes zero if both tasks are mapped on the same processor). Let μ be a cost function on the nodes ($\mu(t)$ is the execution time of a task t). The length of a path in a DAG is defined as the sum of its edges weights plus the sum of its nodes weights. In the following we will use terms node or task interchangeably.

In a task graph, a node which does not have any predecessors is called an *entry* node while a node which does not have any successors is called an *exit(sink)* node. Each task first receives all the needed data from its predecessors, computes without interruption and then sends the results to its successors. The architecture is a network of an arbitrary number of homogeneous processors. We do not allow task duplication here. Several heuristics [1, 14] have been developed that take advantage of this option.

The objective function is to minimize both the Makespan (denoted by $\mathcal{M}$) and the processors consumption without violating the precedence constraints among the tasks.

Let $\Gamma^-(t_x)$ and $\Gamma^+(t_x)$ denote the sets of immediate predecessors and successors of t_x respectively. We call $g(G)$ the *granularity* of the task graph. We use the definition given in [6]:

$$g(G) = \min_{i=1..v}\left(\min\left\{\frac{\mu(t_i)}{\max\limits_{t_j\in\Gamma^-(t_i)}\omega(t_j,t_i)},\ \frac{\mu(t_i)}{\max\limits_{t_k\in\Gamma^+(t_i)}\omega(t_i,t_k)}\right\}\right)$$

If $g(G) \geq 1$ a task graph G is said *coarse grain*, otherwise *fine grain*. For *coarse grain* DAGs each task receives or sends a small amount of communication compared to the computation of its adjacent tasks.

3. Related Work

A large number of algorithms for scheduling and partitioning DAGs have been proposed in the literature. There exist mainly two classes:

List scheduling heuristics [8, 13, 14, 16]: These algorithms assign priorities to the tasks and schedule them according to a list priority scheme. In each step, a list scheduler selects one of the tasks in the list, assigns it to a suitable processor, and update the list.

Another technique is called Critical Path (CP) heuristics [5, 7, 11, 10, 12, 15, 18, 17]: The CP of a task graph (DAG) is defined to be the path having the largest sum of the weights of both nodes and edges from a source node to a sink node. These algorithms try to shorten the longest path in the DAG by removing communication requirements and mapping the adjacent tasks into a cluster (this is called zeroing an edge). This approach has received the most attention and a taxonomy of these techniques can be found in [5].

Efe's [3] article is one of the earliest works to consider task graph clustering in distributed computing. Kim and Browne [9] studied linear clustering which is an important special case for clustering. In [10], the Dynamic Critical Path (*DCP*) algorithm is proposed. This algorithm uses a look-ahead strategy for the start times of a node's children when selecting a processor. The *DCP* algorithm have the following features: - It assigns dynamic priorities to the nodes at each step in the scheduling process, - The start times of the nodes are not fixed until all nodes have been scheduled, - It selects a processor for a node by looking ahead the potential start time of the node's *critical child* node on that processor. In this paper, a DCP node is identified by checking for equality of its $AEST$ (Absolute Earliest Start Time) and $ALST$ (Absolute Earliest Start Time) attributes. The computation of these values requires the traversal of the entire DAG at each step. Repeating this computation for all steps will result in at least $O(v^2)$ complexity. Unlike DCP algorithm, the computation of CP node is done incrementally from step to step in our algorithm (DCPS), in order to reduce the time complexity. The time complexity of the *DCP* algorithm is shown to be $O(v^3)$. For scheduling arbitrary task graphs without duplication, the fastest known algorithm to date was proposed by Gerazoulis and Yong [7] who considered the Dominant Sequence (DS) instead of CP to represent the longest path of the partially clustered graph. Their algorithm called *DSC* (Dominant Sequence clustering), has a low complexity of $O((e+v)\log v)$. Our approach in this paper is based on the principle of Critical Path scheduling.

4. *DCPS* Algorithm

To describe *DCPS* we need to specify certain constraints and definitions. First the node types:

- *scheduled:* A task is scheduled if it has been assigned to a processor.
- *free:* A task is called free if it is unscheduled and all of its successors are scheduled.
- *partially free:* A task is partially free if it is unscheduled and at least one of its successors is scheduled but not all of them have been scheduled.

During the execution of *DCPS*, the graph consists of two parts, the examined (scheduled) tasks $\mathcal{S}$ and the unscheduled tasks U. Initially $U = V$.

Timing values:

- $\mathcal{T}(t_x)$: *top level.* It is the length of the longest path from an entry (top) node to t_x (excluding the execution time of t_x) in a DAG. Thus, $\mathcal{T}(t_x)$ is the starting time of t_x prior to any clustering of a DAG. The $\mathcal{T}$ values are computed according to the topological order of a graph. The $\mathcal{T}$ value of every entry node is zero. Let t_x be a task such that all of its immediate predecessors have been assigned $\mathcal{T}$ values. Then,

$$\mathcal{T}(t_x) = \max\{\mathcal{T}(t_i) + \mu(t_i) + \omega(t_i, t_x) \mid t_i \in \Gamma^-(t_x)\} \tag{1}$$

- $\mathcal{B}(t_x)$: *bottom level.* It is the length of the longest path from the start of t_x to an exit node in the current partially clustered DAG. The $\mathcal{B}$ values are computed according to the reverse topological order of a graph. The $\mathcal{B}$ value of every exit node is equal to its execution time. Let t_x be a task such that all of its immediate successors have been scheduled (and hence been computed $\mathcal{B}$ values). Then,

$$\mathcal{B}(t_x) = \max\{\mu(t_x) + \omega(t_x, t_j) + \mathcal{B}(t_j) \mid t_j \in \Gamma^+(t_x)\} \tag{2}$$

  Note that the $\mathcal{B}$ value of a partial free task can be computed using only the $\mathcal{B}$ from its immediate scheduled successors. Because only part of successors are considered, so we define the $\mathcal{B}$ value of a partial task:

$$\mathcal{B}(t_x) = \max\left\{\mu(t_x) + \omega(t_x, t_j) + \mathcal{B}(t_j) \mid t_j \in \Gamma^+(t_x) \cap \mathcal{S}\right\} \tag{3}$$

- Using these formulas (1, 2 and 3), we define the priority for tasks in the free and partial free lists (to be defined in the following) as follows:

$$\mathcal{P}(t_x) = \mathcal{T}(t_x) + \mathcal{B}(t_x) \tag{4}$$

The *constraining successor* and the *constraining predecessor* of a task t_x are defined respectively as the task which determines $\mathcal{B}(t_x)$ and $\mathcal{T}(t_x)$ values.

We maintain two priority list's α and β (that contains respectively free and partially free tasks) which are implemented respectively by using a balanced search tree data structure (*AVL*) and a simple linked list. At the beginning, α and β are empty. The Head function $H(\alpha)$ returns the first task in the sorted list α, which is the task with the highest priority (if two tasks have the same priority we choose one of them randomly). If $\alpha = \emptyset, H(\alpha) = NULL$ and $\mathcal{P}(NULL) = 0$.

Let $\mathcal{M}_i$ be the Makespan at step i. A partially free task t is inserted at the head of β once and only if $\mathcal{P}(t) = \mathcal{M}_i$. When a task of β becomes free, the first task of this list is deleted.

The *DCPS* algorithm that we propose in this work consists of a sequence of refinement steps, where each step creates a new cluster or grows an existing cluster. In the beginning, *DCPS* assumes that every task in the DAG is assigned to a different processor (cluster). Unlike *DSC*, the *DCPS* topological traversing order of the graph is bottom-up, i.e., it constructs the clusters by starting from the sink task. Our heuristic is guided by an unbounded number of processors scheduling mechanism. This control mechanism performs v steps and, at each refinement step, it selects a free task and tries to schedule it by zeroing one of its outgoing edges.

4.1. Policies of task clustering

The policies of task clustering are described below:

The criterion of accepting a zeroing is that the value $\mathcal{B}$ of the highest free task does not increase by such a zeroing, otherwise, we impose some rules (see below) to allow this increase. By reducing $\mathcal{B}(t_x)$ values all paths passing through t_x could be compressed and as a result the *CP* length could be reduced. When an edge is zeroed then a free task t_x is merged to the cluster where its *constraining successor* resides. Note that our scheduling scheme adds a pseudo edge from t_x to the last task of this cluster if they are independent.

To describe the following rules we need to specify some definitions and constraints: assume that t_x is the current task ($t_x = H(\alpha)$) and let $\delta(t)$ be the *constraining predecessor* of t. Let $\mathcal{F}$ be the future cluster of $\delta(t_x)$ where it will be merged when it becomes free and t_y be the last task scheduled to $\mathcal{F}$.

Rule 1: Rule 1 can be applied in the case of fork component structure in the DAG as follows : t_x is placed in $\mathcal{F}$ if and only if

$$\left(\Gamma^-(t_x) \bigcap \Gamma^-(t_y) = \delta(t_x)\right) \wedge (\mathcal{L}(\mathcal{F}) \leq \omega(\delta(t_x), t_x))$$

is carried out. Where $\mathcal{L}(\mathcal{F})$ is the load of the cluster $\mathcal{F}$.

Rule 2: If rule 1 is not carried out, rule 2 can be tested: let $\mathcal{C}(t_y)$ be the cluster containing t_y (t_y is the last task in the cluster) , $\mathcal{B}^{\mathcal{C}}(t_x)$ the $\mathcal{B}$ value if t_x is scheduled to $\mathcal{C}(t_y)$. If $|\Gamma^-(t_x)| = |\Gamma^-(t_y)| = 1$ then t_x is merged to $\mathcal{C}(t_y)$ if and only if the following formula is satisfied:

$$(\mathcal{F}(\delta(t_y)) \neq \mathcal{C}(t_y)) \wedge (\delta(t_x) = \delta(t_y)) \wedge \left(\mathcal{B}^{\mathcal{C}}(t_x) + \mathcal{T}(t_x) \leq \mathcal{M}_i\right)$$

Note that by applying the preceding rules we can reduce considerably the number of processors used in the scheduling process.

Definition 1 *The final Makespan, denoted by $\mathcal{M}^*$, is defined as :*

$$\mathcal{M}^* = \max\{\mathcal{B}(t_i) \mid t_i \in \mathcal{S}\}$$

The value $\mathcal{M}^*$ is simply computed by taking the maximum value across all the $\mathcal{B}$ values of the scheduled tasks. Note that the *DCPS* algorithm can detect $\mathcal{M}^*$ value at the intermediate step of the scheduling process (see theorem 5).

Rule 3: If the final schedule length $\mathcal{M}^*$ of the DAG is detected at some step i in the scheduling process, we try to schedule t_x to the cluster used in step $i-1$ (let's call it $\mathcal{C}_{i-1}$) provided that the following condition is checked

$$\mathcal{B}^{\mathcal{C}_{i-1}}(t_x) + \mathcal{T}(t_x) \leq \mathcal{M}^*$$

Since $\mathcal{M}^*$ will not change in subsequent clustering steps, we reverse the function head $H(\alpha)$ in $\overline{H}(\alpha)$ to return the task with the smallest priority, doing so, the number of processors used for the DAG will be decreased.

If none of the preceding rules is checked, the task t_x remains in its cluster. The formal description of the *DCPS* algorithm is given below.

4.2. An application example

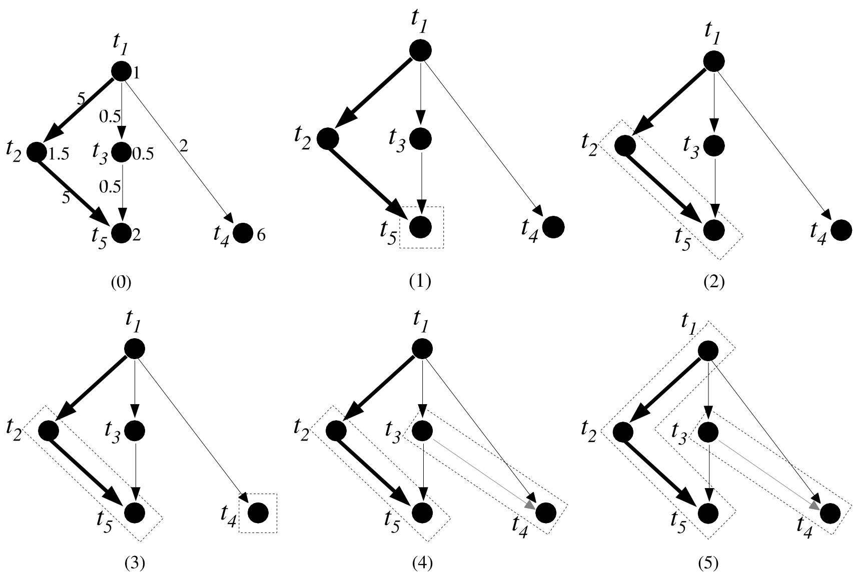

Figure 1: *DCPS* scheduling algorithm steps

As an example, a running trace of *DCPS* is shown in Fig. 1. The thick paths are the CPs and dashed pseudo edges are the execution order within a cluster. In the following, the superscript of a task in α or β denotes its priority value.

Initially all the tasks are in a separate unit cluster, $\mathcal{M}_0 = 14.5$, $\alpha = \{t_4^9, t_5^{14.5}\}$, $\beta = \emptyset$. At step 1, t_5 is selected, $\mathcal{B}(t_5) = 2$, it cannot be reduced so $\mathcal{C}(t_5)$ remains a unit cluster. Then $\mathcal{M}_1 = 14.5$, $\alpha = \{t_2^{14.5}, t_3^{4.5}, t_4^9\}$, $\beta = \emptyset$. At step 2, t_2 is selected, $\mathcal{B}(t_2) = 8.5$. By zeroing the outgoing edge (t_2, t_5) of t_2, $\mathcal{B}(t_2)$ reduces to 3.5. This zeroing is accepted and after that step, $\mathcal{M}_2 = 9.5$, $\alpha = \{t_3^{4.5}, t_4^9\}$, $\beta = \{t_1^{9.5}\}$ (t_1 is inserted into β because $\mathcal{P}(t_1) = 9.5 = \mathcal{M}_2$). At step 3, t_4 is examined, $\mathcal{B}(t_4) = 6$. Since none of the rules is satisfied, $\mathcal{B}(t_4)$ remains the same and $\mathcal{C}(t_4)$ remains a unit cluster as shown in Fig. 1. At step 4, t_3 is selected, it cannot be merged with its *constraining successor* t_5 because the increase of its $\mathcal{B}$ value as well as rule 1 is not checked. Therefore according to rule 2, t_3 is scheduled to $\mathcal{C}(t_4)$. Finally t_1 is selected and (t_1, t_2) is zeroed, so that $\mathcal{B}(t_1)$ is reduced from 9.5 to 9. As we can see in Fig. 1 two clusters are generated with $\mathcal{M} = 9$.

5. Fully Detailed Algorithm and Theoretical Results

5.1. Our Algorithm

Algorithm 1 The *DCPS* Algorithm

Compute $\mathcal{T}(t)$ for each task t and set $\mathcal{B}(t) = \mu(t)$ for each exit task ;
$\mathcal{S} = \emptyset$; $U = V$; *(*Mark all tasks as unscheduled*)*
while $U \neq \emptyset$ **do**
    $t_x = H(\alpha)$; *(*Select a free task with the highest priority from* α **)*
    Try to merge t_x with the cluster of its *constraining successor* t_y ;
    if $\mathcal{B}(t_x)$ does not increase **then**
        Zero the edge (t_x, t_y) ;
    else
        schedule t_x by checking one of the rules 1, 2 and 3 in the order ;
        if none of the rules is satisfied, schedule t_x to a new cluster;
    end if
    insert t_x in $\mathcal{S}$ and update the priority values of t_x's predecessors;
    insert free predecessors of t_x in α
end while

5.2. Complexity Analysis

Theorem 1 *The time complexity of DCPS is* $O(e + v \log v)$.

Proof. Note that at some step i a partially free task t is inserted in β once and only if $\mathcal{P}(t) = \mathcal{M}_i$, and when a task of β becomes free, the first task of this list is deleted. Thus the number of partially free tasks that are inserted or deleted from β is at most equal to v. The cost of each operation (insertion, deletion) is $O(1)$.

The overhead occurs only to maintain the proper order among tasks when a task is inserted or deleted from α. This operation costs $O(\log |\alpha|)$ where $|\alpha| \leq v$. Since each task in a DAG is inserted into α once and only once and is removed once and only once during the entire execution of *DCPS*, the total complexity for maintaining α is at most in order of $2v \log v$. The main computational cost of *DCPS* is spent in the while loop (Line 3). The number of loops is v. Line 4 costs $O(\log v)$ for finding the head of α. Line 5 costs $O(|\Gamma^+(t_x)|)$ in examining the immediate successors of task t_x. For the whole v loops the cost of this line is $\sum_{i=1}^{v} O(|\Gamma^+(t_i)|) = O(e)$. Line 12 costs $O(|\Gamma^-(t_x)|)$ to update the priority values of the immediate predecessors of t_x, and similarly the cost for the v loops of this line is $O(e)$. Thus the total cost of *DCPS* is $O(e + v \log v)$. □

5.3. Algorithm Analysis

Theorem 2 *For each step i of DCPS, $\mathcal{M}_{i-1} \geq \mathcal{M}_i$*

Proof. By definition $\mathcal{M}_{i-1}$ is the length of the critical path at step $i-1$, assume that at step i, $H(\alpha) = t_x$. If $\mathcal{B}(t_x)$ is reduced then it cannot be greater than the sum of the costs of both tasks execution time and communication time along the critical path from the sink task to t_x. In addition and according to rule 1, 2 and 3, the intermediate Makespan will not be increased even if $\mathcal{B}(t_x)$ increases. It follows that $\mathcal{M}_{i-1} \geq \mathcal{M}_i$ □

Property 1 *For the DCPS algorithm, $\mathcal{B}(t_x)$ remains constant if $t_x \in \mathcal{S}$ and $\mathcal{T}(t_x)$ remains the same if $t_x \in U$.*

Proof. If $t_x \notin \mathcal{S}$, then the topological traversal implies that all predecessors of t_x are not in $\mathcal{S}$. Since t_x is in a separate unit cluster, $\mathcal{T}(t_x)$ remains unchanged before it is examined. Also for task's in $\mathcal{S}$, when a free task is merged to a new cluster it is always attached to the last task of that cluster. Thus $\mathcal{B}(t_x)$ remains unchanged after t_x has been scheduled. □

Lemma 1 *Assume that $t_x = H(\alpha)$ after some step i. If there are CPs which pass through free tasks in α, then $\mathcal{P}(t_x) = \mathcal{M}_i$.*

Proof. After step i, $\mathcal{M}_i = \mathcal{P}(t_y)$, where t_y is a critical task. Assume that no critical path pass through t_x. Then one critical path must pass through another non-head free task t_z. This implies that $\mathcal{P}(t_z) = \mathcal{M}_i > \mathcal{P}(t_x)$. Since $t_x = H(\alpha)$, there is a contradiction. □

Lemma 2 *After some step i, assume that $t_x = H(\alpha)$ and there are* CP*s passing through not-scheduled task U. If $\mathcal{P}(t_x) < \mathcal{M}_i$, then these critical paths pass only through β.*

Proof. Assume on the contrary that no critical path pass through β and by definition, a partially free task (let call it t_y) is inserted in β if and only if $\mathcal{P}(t_y) = \mathcal{M}_i$. Thus $\mathcal{P}(\beta) = 0 < \mathcal{M}_i$. In addition we have $\mathcal{P}(t_x) < \mathcal{M}_i$. This contradicts the assumption that there are *CP*s passing through not-scheduled task U. □

Theorem 3 *Assume that $t_x = H(\alpha)$ and $\beta \neq \emptyset$ after step i and that there is a critical path passing through not-scheduled tasks U.*

- *if $\mathcal{P}(t_x) = \mathcal{M}_i$, then a CP passes through t_x.*
- *if $\mathcal{P}(t_x) < \mathcal{M}_i$, then a CP passes only through β.*

Proof. This result follows from the previous lemmas:

- if $\mathcal{P}(t_x) = \mathcal{M}_i$, then we will show that a critical path passes through t_x. First assume that a Critical Path passes through α or both α and β. Then according to Lemma 1, it must pass through t_x. Next assume that a *CP* passes only through β's tasks and according to Lemma 2 we have $\mathcal{P}(\beta) = \mathcal{M}_i > \mathcal{P}(t_x)$ which is a contradiction since $\mathcal{P}(t_x) = \mathcal{M}_i$.
- If $\mathcal{P}(t_x) < \mathcal{M}_i$, suppose that a *CP* goes through a free task. Then according to Lemma 1, $\mathcal{P}(t_x) = \mathcal{M}_i \geq \mathcal{P}(\beta)$ which is a contradiction. Therefore the *CP*s must pass through β by Lemma 2.

□

Theorem 4 *The* Makespan *for the partially scheduled graph after step i is :*

$$\mathcal{M}_i = \max\{\mathcal{P}(H(\alpha)), \mathcal{P}(\beta), \max\{\mathcal{B}(t_y) \mid t_y \in \mathcal{S}\}\}$$

Proof. There are two cases, either there is a critical path that passes through U or there is not. If not, that indicates that all *CP*s have been examined and are only within $\mathcal{S}$, then by Definition 1 $\mathcal{M}_i = \max\{\mathcal{B}(t_y) \mid t_y \in \mathcal{S}\}$, which is the final schedule length of the partially scheduled graph. If there is a critical path going through U then a *CP* must go either through the head of α or the tasks in β if $\beta \neq \emptyset$. Therefore we have the following two cases:

i) If a *CP* is going through the head of α, then $\mathcal{M}_i = \mathcal{P}(H(\alpha))$ by Lemma 1.

ii) If this *CP* is only passing through β, then $\mathcal{M}_i = \mathcal{P}(\beta)$ by Lemma 2.

□

Theorem 5 *Assume that $\beta = \emptyset$ and $t_x = H(\alpha)$ after some step i. If $\mathcal{P}(t_x) < \mathcal{M}_i$, then $\mathcal{M}_i = \mathcal{M}^*$ is the final* Makespan *of the partially scheduled graph.*

Proof. Assume on the contrary that there exist a *CP* passing through α. Then, according to lemma 1, it must pass through t_x. This implies that $\mathcal{P}(t_x) = \mathcal{M}_i$, which contradicts with our assumption that $\mathcal{P}(t_x) < \mathcal{M}_i$. □

6. Optimality for Fork and Join Graphs

6.1. Join Graphs

Since any task graph can be decomposed into a collection of *Fork* and *Join* graph, it is useful to consider how the algorithms work on these two primitive graph structures. In the following, we derive the optimal schedule lengths for these primitive structures. Fig. 2 shows the clustering steps of *DCPS* for a Join DAG. Without loss of generality, assume that for the *Join* structure, we have :

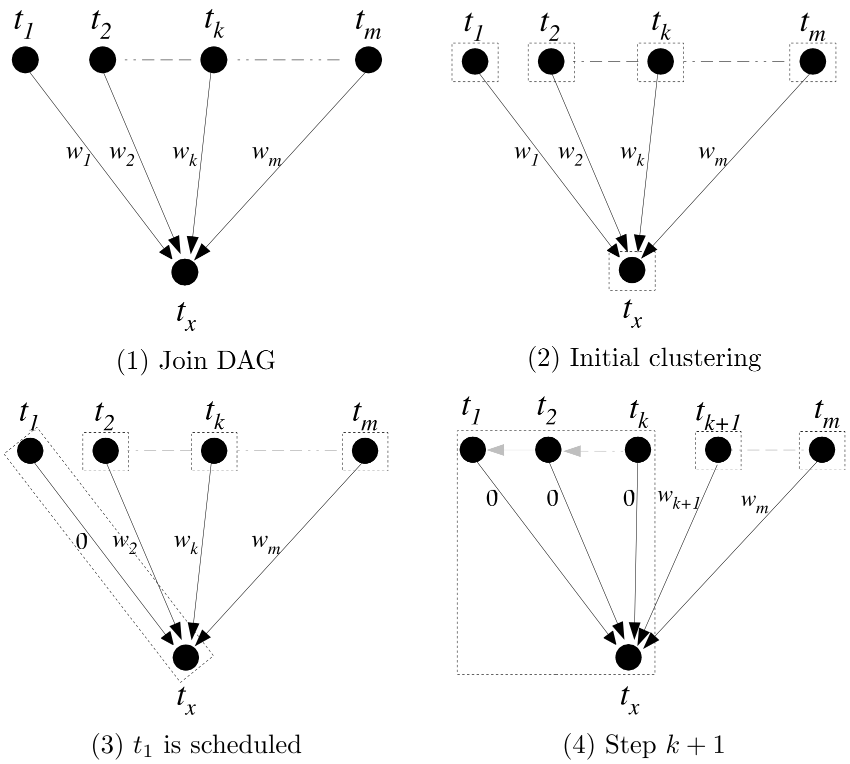

Figure 2: Scheduling steps for a Join DAG

$$\mu(t_1) + \omega(t_1, t_x) \geq \mu(t_2) + \omega(t_2, t_x) \geq \ldots \geq \mu(t_m) + \omega(t_m, t_x)$$

Initially, each task is in a unit cluster as shown in Fig. 2(2). At step 1, t_x is the only free task in U and $\mathcal{P}(t_x) = \mu(t_1) + \omega(t_1, t_x) + \mu(t_x)$, t_x is selected and it remains in a unit cluster. At step 2 shown in Fig. 2(3), $t_1, t_2, \ldots, t_m$ become free and t_1 has the highest priority, $\mathcal{P}(t_1) = \mu(t_1) + \omega(t_1, t_x) + \mu(t_x)$, t_1 is selected and merged to the cluster of t_x and $\omega(t_1, t_x) = 0$. At step $k+1$, t_k is selected. The edge (t_k, t_x) is zeroed only if attaching t_k to the begin of a linear chain $t_{k-1}, t_{k-2}, \ldots, t_1, t_x$ does not increase $\mathcal{B}(t_k)$. The cluster will keep growing until the following condition cannot be satisfied:

$$\sum_{i=1}^{k-1} \mu(t_i) \leq \omega(t_k, t_x)$$

So the optimal schedule length for the Join DAG is equal to:

$$\max \left\{ \sum_{i=1}^{j} \mu(t_i) + \mu(t_x), \mu(t_{j+1}) + \omega(t_{j+1}, t_x) + \mu(t_x) \right\}$$

Where j (the optimal zeroing stopping point) is given by the following conditions:

$$\left(\sum_{i=1}^{j}\mu(t_i) \leq \mu(t_j) + \omega(t_j, t_x)\right) \wedge \left(\sum_{i=1}^{j+1}\mu(t_i) > \mu(t_{j+1}) + \omega(t_{j+1}, t_x)\right) \tag{5}$$

6.2. Fork Graphs

Assume that for *Fork* structure we have:

$$\omega(t_x, t_1) + \mu(t_1) \geq \omega(t_x, t_2) + \mu(t_2) \geq \ldots \geq \omega(t_x, t_m) + \mu(t_m)$$

Then the optimal schedule length for the Fork DAG is equal to:

$$\max\left\{\mu(t_x) + \sum_{i=1}^{j}\mu(t_i), \mu(t_x) + \omega(t_x, t_{j+1}) + \mu(t_{j+1})\right\}$$

Where j is given by the following conditions :

$$\left(\sum_{i=1}^{j}\mu(t_i) \leq \omega(t_x, t_j) + \mu(t_j)\right) \wedge \left(\sum_{i=1}^{j+1}\mu(t_i) > \omega(t_x, t_{j+1}) + \mu(t_{j+1})\right)$$

6.3. Optimality on Fork and Join DAGs

Theorem 6 *DCPS achieves optimal solutions for* Fork *and* Join *DAGs.*

Proof. Let $\mathcal{M}_{opt}$ be the optimal MakeSpan and k be the zeroing stopping point of *DCPS*. We will prove that $\mathcal{M}_{opt} = \mathcal{M}_{DCPS}$ by contradiction. Suppose that $j \neq k$ and $\mathcal{M}_{opt} < \mathcal{M}_{DCPS}$. There are two cases :

i) if $j < k$, then $\sum_{i=1}^{j}\mu(t_i) < \sum_{i=1}^{k}\mu(t_i)$ and $\mu(t_{j+1}) + \omega(t_{j+1}, t_x) \geq \mu(t_k) + \omega(t_k, t_x)$. ¿From condition 5, we have $\sum_{i=1}^{k}\mu(t_i) \leq \mu(t_k) + \omega(t_k, t_x)$, this implies that $\mu(t_{j+1}) + \omega(t_{j+1}, t_x) \geq \sum_{i=1}^{k}\mu(t_i)$.

   Thus $\mathcal{M}_{opt} = \mu(t_{j+1}) + \omega(t_{j+1}, t_x) + \mu(t_x) \geq \mu(t_k) + \omega(t_k, t_x) + \mu(t_x) \geq \max\left\{\sum_{i=1}^{k}\mu(t_i) + \mu(t_x), \mu(t_{k+1}) + \omega(t_{k+1}, t_x) + \mu(t_x)\right\} = \mathcal{M}_{DCPS}$.

ii) if $j > k$, then $\sum_{i=1}^{j}\mu(t_i) \geq \sum_{i=1}^{k+1}\mu(t_i)$ and $\mu(t_{j+1}) + \omega(t_{j+1}, t_x) \leq \mu(t_{k+1}) + \omega(t_{k+1}, t_x)$. ¿From condition 5, we have $\sum_{i=1}^{k+1}\mu(t_i) > \mu(t_{k+1}) + \omega(t_{k+1}, t_x)$, this implies that $\sum_{i=1}^{j}\mu(t_i) > \mu(t_{k+1}) + \omega(t_{k+1}, t_x)$.

   Thus $\mathcal{M}_{opt} = \sum_{i=1}^{j}\mu(t_i) + \mu(t_x) \geq$
   $\max\left\{\sum_{i=1}^{k}\mu(t_i) + \mu(t_x),\ \mu(t_{k+1}) + \omega(t_{k+1}, t_x) + \mu(t_x)\right\} = \mathcal{M}_{DCPS}$.

There is a contradiction in both cases. The proof applied in *Join* structure can be applied to *Fork* structure by just reversing the *Fork* graph into a *Join* graph. □

7. Experimental Results

Due to the NP-completeness of this scheduling problem, the proposed algorithm cannot always lead to an optimal solution. Thus it is necessary to compare the performance of different algorithms using randomly generated graphs. So a random DAG generator has been developed. To generate a random connected DAG, we begin by generating a random spanning tree in an iterative way. We assume that, at the step i, tasks $t_v, t_{v-1}, \ldots, t_i$ are in the tree. We then add vertex t_{i-1} to the tree by adding the directed edge $(t_{i-1}, t_{rand(i,v)})$ linking from t_{i-1} to $t_{rand(i,v)}$, where $rand(i, v)$ is a random generator function which generates a integer in the $[i, v]$ interval. Finally we add additional random edges $\{(t_i, t_j), i < j\}$ to produce a DAG with edges between $v-1$ and $2v$. Task t_v is the unique bottom task in a DAG. This algorithm generates DAGs that are quite close to some of those occurring in practical applications.

In our study, we compare our algorithm only with *DSC* because it is the best known algorithm to date in terms of speed and solution quality of the schedule length. A comparative study of the various algorithms in the literature can be found in [7, 10].

We have generated 594 random graphs as follows: we classified the all the DAGs into 11 groups of 54 DAGs each according to their *granularity* ($g(G) = 0.1, 0.2, \ldots, 1.1$). In every group we vary the number of tasks from 100 to 1000 with increments of 100 and in each interval we generate 6 different graphs. The performance comparison is carried out in three contexts:

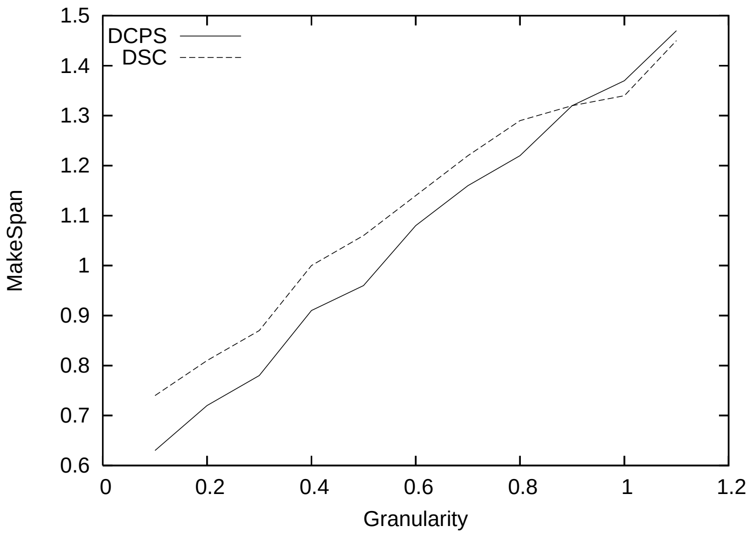

Figure 3: Average normalized schedule lengths

First we compare the Makespan produced by each algorithm for various sizes and types of *granularity*. Fig. 3, shows that *DCPS* is better than *DSC* for $g(G) \leq 0.9$ and its performance increases as the DAG becomes increasingly *fine grain* (the communication delay are high). This is because sequentializing a set of tasks on the same processor can produce a better Makespan than executing them in parallel with more processors instead of one. For $g(G) > 0.9$ we can see that *DSC* becomes competitive in terms of solution quality. But for *coarse grain* DAG there exist a linear clustering which achieve the best Makespan (see [6]).

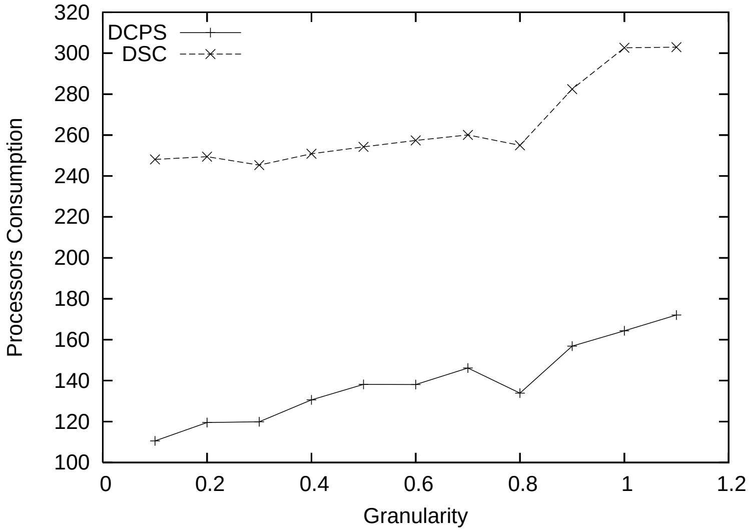

Figure 4: Average processors consumption

Another quality of measure is the number of processors used. In Fig. 4, we show the average number of processors used by each algorithm for different kind of graph sizes and values of *granularity*. We observe that *DSC* uses considerably large number of processors compared to our algorithm. However, this is due to a deficiency of *DSC*: it tries to schedule tasks on as much as processors as possible to minimize the schedule length, thus *DSC* finds several clusters with only one task. As a result, the schedules generated by *DSC* are not well load balanced. Our heuristic cures this deficiency of *DSC* and produces better Makespans by performing some load balancing by minimizing the number of processors. Note that *DCPS* consumes also less processors even if $g(G) > 1$.

Finally, we compare the efficiency of these algorithms which are given in Fig. 5. Efficiency reveals the average percent of time the processors are active. The definition of this measure is given by the following formula:

$$Efficiency = \frac{SpeedUp}{Number\ \ of\ \ Processors},$$

$$\text{where}\ \ SpeedUp = \frac{SerialTime}{ParallelTime}$$

As we can observe, *DCPS* is much more efficient than *DSC* because it consistently uses fewer processors than the *DSC* algorithm.

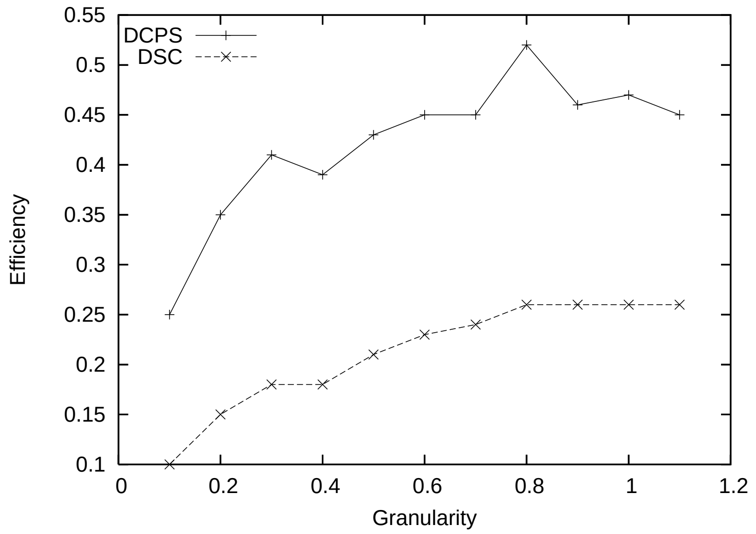

Figure 5: Average efficiency

8. Conclusion

In this paper, we have presented a new algorithm based on a critical path approach for scheduling parallel programs onto multiprocessors, we have demonstrated that the solution quality, the number of processors used and the time complexity of the proposed algorithm makes it a viable choice for compile-time scheduling of general task graphs.

References

1. F. D. Anger, J.-J. Hwang, and Y.-C. Chow, "Scheduling with sufficient loosely coupled processors," *Journal of Parallel and Distributed Computing*, **9** (1990) 87–92.
2. P. Chretienne, "Task scheduling over distributed memory machines," *Proc. International Workshop on Parallel and Distributed Algorithms*, N. Holland, (1989) 165–176.
3. K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer*, **6** (1982) 50–56.
4. M. Garey and D. Johnson, *Computer and Intractability: A guide to the Theory of NP-Completness.* (W.H. Freeman & Co, 1979).
5. A. Gerasoulis and T. Yang, "A comparison of clustering heuristics for scheduling DAGs on multiprocessors," *Journal of Parallel and Distributed Computing*, **16** (1992) 276–291.
6. A. Gerasoulis and T. Yang, "On the granularity and clustering of directed acyclic task graphs," *IEEE Transactions on Parallel and Distributed Systems*, **4** (1993) 686–701.
7. A. Gerasoulis and T. Yang, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, **5** (1994)

951–967.

8. J. J. Hwang, Y. C. Chow, F. D. Anger, and B. Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal on Computing*, **18** (1989) 244–257.
9. S. J. Kim, J. C. Browne. "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *International Conference on Parallel Processing*, **3** (1988) 1–8.
10. Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, **7** (1996) 506–521.
11. Y.K. Kwok and I. Ahmad, "Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures," *Proc. 7th IEEE Symposium on Parallel and Distributed Processing*, (1995), 36–43.
12. C. Lavarenne and Y. Sorel, "Performance Optimization of Multiprocessor Real-Time Applications by Graphs Transformations," *Proc. Parallel Computing*, Grenoble, (1993).
13. Y.-K. Kwok and I. Ahmad, "Towards an architecture-independent analysis of parallel algorithms," *SIAM Journal on Computing*, **19** (1990) 322–328.
14. H. E. Rewini and T. G.Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, **9** (1990) 138–153.
15. V. Sarkar, *Partitionning and Scheduling Parallel Programs for Execution on Multiprocessors.* (MIT Press, 1989).
16. T. Yang and A. Gerasoulis, "List scheduling with and without communication delays," *Parallel Computing*, **19** (1993) 1321–1344.
17. M. Y. Wu and D. Gajski, "A programming aid for hypercube architectures," *Journal of Supercomputing*, **2** (1988) 349–372.
18. M.Y. Wu and D. Gajski, "Hypertool: Aprogramming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, **1** (1990) 330–343.