ISSN 1746-7659, England, UK
Journal of Information and Computing Science
Vol. 6, No. 2, 2011, *pp*. 083-096

An Extension of Edge Zeroing Heuristic for Scheduling Precedence Constrained Task Graphs on Parallel Systems Using Cluster Dependent Priority Scheme

Abhishek Mishra and Anil Kumar Tripathi

Deptartment of Computer Engineering, Institute of Technology, Banaras Hindu University, Varanasi, India, 221005

(*Received November 3, 2010, accepted December 20, 2010*)

(*An extended abstract of this paper appears in the Proceedings of 2010 IEEE International Conference on Computer & Communication Technology (ICCCT-2010), pages 647-651, ISBN: 978-1-4244-9034-9.*)

Abstract. Sarkar's edge zeroing heuristic for scheduling precedence constrained task graphs on parallel systems can be viewed as a priority based algorithm in which the priority is assigned to edges. In this algorithm, the priority is taken as the edge weight. This can also be viewed as a task dependent priority function that is defined for pairs of tasks. We have extended this idea in which the priority is a cluster dependent function of pairs of clusters (of tasks). Using this idea we propose an algorithm of complexity $O(|V||E|(|V|+|E|))$ and compare it with some well known algorithms.

Keywords: clustering, homogeneous systems, parallel processing, scheduling, task allocation.

1. Introduction

A parallel system is designed so that it can execute the applications faster than a sequential system. For this we need to parallelize the program. There are three steps involved in the parallelization of a program (Sinnen [27]). The first step is called *task decomposition* in which the application is divided into tasks. The *degree of concurrency* is the number of tasks that can be executed simultaneously (Grama et al. [10]).

The tasks generated may have interdependencies between them that will decide the partial execution order of tasks. The determination of precedence constraints between the tasks is the second step of parallelization and is called *dependence analysis* (Banerjee et al. [2], Wolfe [29]).

A dependence relation among the tasks is represented as a directed acyclic graph, also known as the *task graph*. Nodes in the task graph represent the tasks and have a weight associated with them that represents the execution time of the task. Edges in the task graph represent the dependence relation between the tasks and have a weight associated with them that represents the communication time between the tasks.

The final step of parallelization is the *scheduling* of tasks to the processors. By scheduling we mean both the spatial assignment (task allocation), and the temporal assignment (assigning start time) of tasks to the processors.

The problem of finding a scheduling for a given task graph on a given set of processors that takes minimum time is *NP-Complete* (Sarkar [26], Papadimitriou and Yannakakis [24]). Therefore several heuristics are applied for solving this problem in polynomial time (Yang and Gerasoulis [32], Gerasoulis and Yang [8], Dikaiakos et al. [7], Kim and Browne [16], Kwok and Ahmed [17], [18], Lo [19], Malloy et al. [22], Wu and Gajski [30], Kadamuddi and Tsai [14], Sarkar [26], Yang and Gerasoulis [33], Wu et al. [31], Sih and Lee [28]). The solutions generated by using these algorithms are generally suboptimal.

Our heuristic is an extension of Sarkar's edge zeroing heuristic [26] for scheduling precedence constrained task graphs on parallel systems. Sarkar's algorithm can be viewed as a task dependent priority based algorithm in which the priority (in this case edge weight) is a function of pairs of *tasks*. We extend this concept and define the priority as a cluster dependent function of pairs of *clusters*. Using this concept we propose an algorithm of complexity $O(|V||E|(|V|+|E|))$ and compare it with some well known algorithms.

The remainder of the paper is organized in the following manner. Section 2 presents an overview of the related literature. Section 3 defines a cluster dependent priority scheme that is dependent on cluster-pairs and also presents the proposed algorithm. Section 4 presents a detailed description of the algorithms used. Section 5 gives a sample run of the algorithm. Section 6 presents some experimental results. And finally in section 7 we conclude our work.

2. Literature Overview

Most scheduling algorithms for parallel systems in the literature are based on an idealized model of the target parallel system also referred to as the *classic model* (Sinnen [27]). It is a set of identical processors with fully connected dedicated communication subsystem. Local communications are cost-free and we also have concurrent inter-processor communications.

A fundamental scheduling heuristic is called the *list scheduling* heuristic. In list scheduling, first we assign a priority scheme to the tasks. Then we sort the tasks according to the priority scheme, while respecting the precedence constraints of the tasks. Finally each task is successively scheduled on a processor chosen for it. Some examples of list scheduling algorithms are: Adam et al. [1], Coffman and Graham [3], Graham [9], Hu [13], Kasahara and Nartia [15], Lee et al. [20], Liu et al. [21], Wu and Gajski [30], Yang and Gerasoulis [34].

Another fundamental scheduling heuristic is called *clustering*. Basically it is a scheduling technique for an unlimited number of processors. It is often proposed as an initial step in scheduling for a limited number of processors. A cluster is a set of tasks that are scheduled on the same processor. Clustering based scheduling algorithms generally consist of three steps. The first step finds a clustering of the task graph. The second step finds an allocation of clusters to the processors. The last step finds a scheduling of the tasks. Some examples of clustering based scheduling algorithms are: Mishra et al. [23], Yang and Gerasoulis [32], Kim and Browne [16], Kadamuddi and Tsai [14], Sarkar [26], Hanen and Munier [11].

3. The Cluster Dependent Priority Scheduling Algorithm

3.1. Notation

Let $\boldsymbol{N}$ denote the set of natural numbers: $\{1, 2, 3, ...\}$. Let $\boldsymbol{R}$ denote the set of real numbers, and let $\boldsymbol{R}^+$ denote the set of non-negative real numbers. For $(1 \leq i \leq n)$, let there be n tasks M_i. Let

$$\mathbf{M} = \{M_i \mid 1 \leq i \leq n\} \tag{1}$$

be the set of tasks. Then for $(1 \leq i \leq n)$, the clusters $C_i \in \boldsymbol{M}$ are such that for $i \neq j$ and $(1 \leq i \leq n, 1 \leq j \leq n)$:

$$C_i \cap_{i \neq j} C_j = \phi, \tag{2}$$

and

$$\cup^n_{i=1} C_i = \boldsymbol{M}. \tag{3}$$

Let

$$\boldsymbol{C} = \{C_j \mid C_j \in \boldsymbol{M}, 1 \leq i \leq n\} \tag{4}$$

be a decomposition of $\boldsymbol{M}$ into clusters. Note that some of the C_j's may be empty. Let

$$V = \{i \mid 1 \leq i \leq n\} \tag{5}$$

denote the set of vertices of the task graph. Let the directed edge from i to j be denoted as $(i \rightarrow j)$. Let

$$E = \{(i, j) \mid i \in V, j \in V, \exists(i \rightarrow j)\} \tag{6}$$

denote the set of edges of the task graph. Let $m_i \in \boldsymbol{R}^+$ be the execution time of the task M_i. If $(i, j) \in E$, then let $w_{ij} \in \boldsymbol{R}^+$ be the communication time from M_i to M_j. Let T be the adjacency list representation of the task graph.

Let $cluster : \boldsymbol{N} \rightarrow \boldsymbol{N}$ be a function such that:

$$cluster(i) = j \leftrightarrow M_i \in C_j. \tag{7}$$

For $C_j \in \boldsymbol{C}$, let $comp : \boldsymbol{C} \rightarrow \boldsymbol{R}^+$ be a function that gives the total computation time of a cluster:

$$comp(C_j) = \Sigma_{Mi \in Cj}\, w_i. \tag{8}$$

For $C_i \in \boldsymbol{C}$, and $C_j \in \boldsymbol{C}$, let $comm : \boldsymbol{C}\ X\ \boldsymbol{C} \rightarrow \boldsymbol{R}^+$ be a function that gives the total communication time from the first cluster to the second cluster:

$$comm(C_i, C_j) = \Sigma_{Mp \in Ci, Mq \in Cj, (p, q) \in E}\, w_{pq}. \tag{9}$$

We follow the convention that for a given function *f*, *f(...)* is a function notation (useful in definition) and *f[...]* (or *f[...][...]*) is the corresponding array notation (useful in algorithms for implementing functions as arrays).

3.2. Sarkar's Edge Zeroing Heuristic

When the two tasks that are connected through a large weight edge, are allocated to different processors, then this will make a large communication delay. To avoid large communication delays, we generally put such tasks together on the same machine, thus avoiding the communication delay between them. This concept is called edge zeroing.

Sarkar's alorithm [26] uses the concept of edge zeroing for clustering of tasks. Edges are sorted in decreasing order of edge weights. Initially each task is in a separate cluster. Edges are examined one-by-one in decreasing order of edge weight. The two clusters connected by the edge are merged together if on doing so, the parallel execution time does not increase. Sarkar's algorithm uses the level information to determine the parallel execution time and the levels are computed for each step. This process is repeated until all the edges are examined. The complexity of Sarkar's algorithm is $O(|E|(|V|+|E|))$.

We can define a task dependent priority scheme for Sarkar's algorithm. Let $P_m : \boldsymbol{M} \; X \; \boldsymbol{M} \rightarrow \boldsymbol{R}$ be a function such that

$$P_m(M_i, M_j) = w_{ij} \leftrightarrow (i,j) \in E, \tag{10}$$

and

$$P_m(M_i, M_j) = -\infty \leftrightarrow (i,j) \notin E. \tag{11}$$

Now if we make a descending priority queue of pairs of tasks (M_i, M_j) based on the value of P_m such that

$$P_m(M_i, M_j) \geq 0, \tag{12}$$

then this is equivalent to the first step of Sarkar's algorithm in which the edges are sorted in decreasing order of edge weight.

3.3. An Example of Cluster Dependent Priority Scheme

In order to get a good scheduling algorithm, we should define a good cluster dependent priority scheme P_c. We use two observations. The first observation is that by merging two clusters that are heavily communicating with each other, we can expect to reduce the parallel execution time by using the edge zeroing principle. The second observation is that we can exploit parallelism in a better way by keeping two heavily computing clusters separated. Therefore a good priority scheme should reflect these two observations.

One such cluster dependent priority scheme is defined by:

$$P_c(C_i, C_j) = comm(C_i, C_j) + comm(C_j, C_i) - comp(C_i) - comp(C_j). \tag{13}$$

3.4. The Cluster-Pair-Priority-Scheduling Algorithm

```
Cluster-Pair-Priority-Scheduling(T)
01 flag ← true
02 for k ← 1 to |V|
03     do cluster[k] ← k
04 P_c[...][...] ← Evaluate-Priority(T, cluster[...])
05 (p, array[...]) ← Sort(T, P_c[...][...], cluster[...])
06 min ← Evaluate-Time(T, cluster[...])
07 reach[...][...] ← Reachability (T)
08 (topology[...], trank[...]) ← Topological-Sort(T)
09 while flag = true
10     do flag ← false
11         for m ← 1 to p
12             do if flag = true
13                     then break
```

```
14            (i, j) ← array[m]
15            Merge(T, i, j, cluster[...], topology[...], trank[...], reach[...][...])
16            time ← Evaluate-Time(T, cluster[...])
17            if time < min
18               then flag ← true
19                    min ← time
20                    $P_c$[...][...] ← Evaluate-Priority(T, cluster[...])
21                    (p, array[...]) ← Sort(T, $P_c$[...][...], cluster[...])
22               else undo step 15
23 return (min, cluster[...])
```

Cluster-Pair-Priority-Scheduling(...) (CPPS) is the proposed algorithm that uses the cluster dependent priority scheme. Let $P_c : \boldsymbol{C}\ X\ \boldsymbol{C} \rightarrow \boldsymbol{R}$ be a cluster dependent priority function. *flag* is a Boolean variable that is used to determine when a clustering is able to reduce the parallel execution time of the system. In line 01, *flag* is initialized to *true*. In the ***for*** loop from lines 02 to 03, each task is initially kept in a separate cluster. In line 04, the priority of each cluster-pair is calculated using the function *Evaluate-Priority(...)* that is connected by at least one edge and stored in the array P_c*[...][...]*. In line 05, the cluster-pairs are sorted using the function *Sort(...)* in non-increasing order of their P_c*[...][...]* values. *Sort(...)* returns *p* and *array[...]*. *p* is the number of cluster-pairs between which P_c*[...][...]* is defined. *array[...]* is an array that stores the cluster-pairs between which P_c*[...][...]* is defined. In line 06, the parallel execution time of the clustering is evaluated using the function *Evaluate-Time(...)*, and stored in the variable *min*. *min* is used to obtain the clustering that gives the minimum parallel execution time. In line 07, the reachability matrix *reach[...][...]* is evaluated using the function *Reachability(...)*. We have *reach[i][j] = true* if and only if there exists a path from M_i to M_j. In line 08, the task graph *T* is topologically sorted using the function *Topological-Sort(...)* and stored in the array *topology[...]*. The topological ranks of the tasks are stored in the array *trank[...]*.

In lines 09 to 22, in the ***while*** loop, the cluster-pairs are examined one-by-one in non-increasing order of their P_c*[...][...]* values (the ***for*** loop from lines 11 to 22). In line 15, the cluster-pair is merged using the function *Merge(...)*. In line 16, the parallel execution time of the current clustering is evaluated using the function *Evaluate-Time(...)* and the value is stored in the variable *time*. In lines 18 to 21, if the parallel execution time is reduced, then the cluster-pairs are again sorted in non-increasing order of their P_c*[...][...]* values, and the ***while*** loop from lines 09 to 22 is restarted. In line 22, if the parallel execution time of the clustering is not reduced, then the cluster-pair is kept separated. This process is repeated until no further reduction in parallel execution time is possible. Line 23 returns the parallel execution time and the clustering.

Line 01 has complexity $O(1)$. The ***for*** loop from lines 02 to 03 has complexity $O(|V|)$. In line 04, the algorithm *Evaluate-Priority(...)* has complexity $O(|V| + |E|)$ (section 4.1). In line 05, the algorithm *Sort(...)* has complexity $O(|E| \log (|E|))$ (Horowitz et al. [12]). In line 06, the algorithm Evaluate-Time(...) has complexity $O(|V| + |E|)$ (section 4.3). In line 07, the algorithm *Reachability(...)* has complexity $O(|V|(|V| + |E|))$ (Papadimitriou [25]). In line 08, the algorithm *Topological-Sort(...)* has complexity $O(|V| + |E|)$ (Cormen et al. [4]). The complexity of the CPPS algorithm is dominated by the ***while*** loop from lines 09 to 22 that can iterate a maximum of $|V|$ times, since after each merging of clusters, one cluster is reduced. The ***for*** loop from lines 11 to 22 can iterate a maximum of $|E|$ times, since P_c*[...][...]* is defined between the clusters that are having edges between them. Each iteration of the ***for*** loop has complexity that is dominated by lines 15 and 16, each of which has complexity $O(|V| + |E|)$ (Sections 4.2 and 4.3 respectively). Therefore, the ***while*** loop has complexity $O(|V||E|(|V| + |E|))$ that is also the complexity of CPPS algorithm.

4. A Detailed Description of the Algorithms Used

4.1. Evaluate-Priority

Evaluate-Priority(T, cluster[...])

```
01 for each (i,j) $\in$ E
02     do k ← cluster[i]
03        l ← cluster[j]
04        if k $\neq$ l
```

```
05          then $P_c[k][l] \leftarrow 0$
06 for each $(i,j) \in E$
07     do $k \leftarrow cluster[i]$
08        $l \leftarrow cluster[j]$
09        if $k \neq l$
10          then $P_c[k][l] \leftarrow P_c[k][l] + w_{ij}$
11 for $i \leftarrow 1$ to $|V|$
12     do $comp[i] \leftarrow 0$
13 for $i \leftarrow 1$ to $|V|$
14     do $k \leftarrow cluster[i]$
15        $comp[k] \leftarrow comp[k] + w_i$
16 for each $(i,j) \in E$
17     do $k \leftarrow cluster[i]$
18        $l \leftarrow cluster[j]$
19        if $k \neq l$
20        then if $(comp[k] + comp[l])$ was previously not subtracted from $P_c[k][l]$
21                then $P_c[k][l] \leftarrow P_c[k][l] - (comp[k] + comp[l])$
22 return $P_c[...][...]$
```

In lines 01 to 05, the *$P_c[...][...]$* values between different cluster-pairs are initialized to *0*. In lines 06 to 10, the edge weights between the corresponding cluster-pairs are added to the *$P_c[...][...]$* values between them. The *comp[...]* value of each cluster is initialized to *0* in lines 11 to 12. The *comp[...]* value of clusters are calculated in lines 13 to 15. In lines 16 to 21, the *comp[...]* value of corresponding clusters are subtracted from the sum of weights between the clusters to get the *$P_c[...][...]$* values between the cluster-pairs. Line 22 returns the array *$P_c[...][...]$*.

The ***for*** loop from lines 01 to 05, and 06 to 10, each have complexity $O(|E|)$. The ***for*** loop from lines 11 to 12, and 13 to 15, each have complexity $O(|V|)$. The ***for*** loop from lines 16 to 21 has complexity $O(|E|)$. Line 22 has complexity *O(1)*. Therefore the algorithm *Evaluate-Priority(...)* has complexity $O(|V| + |E|)$.

4.2. Merge

Merge(T, i, j, cluster[...], topology[...], trank[...], reach[...][...])

```
01 $flag_1 \leftarrow flag_2 \leftarrow false$
02 $k \leftarrow l \leftarrow 0$
03 $ablevel[...] \leftarrow$ Allocated-Bottom-Level(T, cluster[...])
04 while $k \leq |V|$ and $l \leq |V|$
05     do if $flag_1 = false$
06          then while $k \leq |V|$
07                 do $k \leftarrow k + 1$
08                 if $cluster[topology[k]] = cluster[i]$
09                   then $p_1 \leftarrow topology[k]$
10                        $flag_1 \leftarrow true$
11                        break
12        if $flag_2 = false$
13          then while $l \leq |V|$
14                 do $l \leftarrow l + 1$
15                   if $cluster[topology[l]] = cluster[j]$
16                     then $p_2 \leftarrow topology[l]$
```

```
17                              flag_2 ← true
18                              break
19         if flag_1 = true and flag_2 = true
20           then if reach[p_1][p_2] = true
21                  then flag_1 ← false
22                if reach[p_2][p_1] = true
23                  then flag_2 ← false
24                if flag_1 = true and flag_2 = true
25                  then if ablevel[p_1] > ablevel[p_2]
26                         then add a pseudo-edge of cost 0 from p_1 to p_2
27                              flag_1 ← false
28                       if ablevel[p_1] < ablevel[p_2]
29                         then add a pseudo-edge of cost 0 from p_2 to p_1
30                              flag_2 ← false
31                       if ablevel[p_1] = ablevel[p_2]
32                         then if trank[p_1] < trank[p_2]
33                                then add a pseudo-edge of cost 0 from p_1 to p_2
34                                     flag_1 ← false
35                                else add a pseudo-edge of cost 0 from p_2 to p_1
36                                     flag_2 ← false
37 for m ← 1 to |V|
38     do if cluster[m] = cluster[j]
39          then cluster[m] ← cluster[i]
```

In line 01, $flag_1$ and $flag_2$ are initialized to *false*. $flag_1$ is used for searching the next task that is in the cluster C_i in topological order (line 05 and the inner ***while*** loop from lines 06 to line 11). The task searched is stored in p_1 (line 09). $flag_2$ is used for searching the next task that is in the cluster C_j in topological order (line 12 and the inner ***while*** loop from lines 13 to line 18). The task searched is stored in p_2 (line 16). In line 02, k and l are initialized to 0. k and l are used as indices for searching the tasks in topological order that belong to the clusters C_i and C_j respectively. In line 03, the allocated bottom level (Sinnen [27]) of the tasks of the task graph T for the current allocation *cluster[...]* (before merging C_i and C_j) is evaluated using the function *Allocated-Bottom-Level(...)* and stored in the array *ablevel[...]*. Given an allocation of tasks, the *allocated bottom level* of a task M_n is the length of the longest path starting with M_n in which the weight of an edge is taken as 0 if the two tasks corresponding to the edge are allocated on the same processor.

Given an allocation, a *scheduled DAG* (Yang and Gerasoulis [32]) is a DAG in which there are no independent tasks in any cluster. If there are independent tasks in a cluster, then we can make them dependent by adding 0-cost pseudo-edges between them. We follow the approach used by Sarkar (Sarkar [26]) in which we add a pseudo-edge of cost 0 from a task of higher allocated bottom level to a task of lower allocated bottom level if the two tasks are independent (lines 25 to 30). If the allocated bottom levels of the tasks are same, then we add a pseudo-edge of cost 0 from a task that comes earlier in topological order to the task that comes later in topological order (lines 31 to 36). By following this strategy, it will ensure that there does not exist any cycle in the task graph so that the resulting graph is a scheduled DAG. This is because in a DAG, the successor of a task has lower allocated bottom level and also it comes topologically later. If we always add pseudo-edges of cost 0 from tasks having a higher allocated bottom level to a task having a lower allocated bottom level, then if a cycle exists then some successor of a node will have a higher allocated bottom level than the node itself which is not possible (for the case of cycles having at least one non-pseudo-edge). For the case of equal allocated bottom levels (for the case of cycles having all edges as pseudo-edges), since we are always adding a pseudo-edge of cost 0 from a task that comes earlier in topological order to the task that comes later in topological order, consider the task in the cycle that comes earliest in topological order. Now the predecessor of that task should come even earlier than the task itself implying a contradiction. Therefore a cycle is not possible.

We are evaluating the reachability matrix *reach[...][...]* using the function *Reachabilty(...)* at the initial time only to reduce the complexity of the *Merge(...)* algorithm. This makes no difference because adding some extra *0*-cost pseudo-edges will make no difference to the scheduling. If p_2 is reachable from p_1, then we again search for the next p_1 in topological order (lines 20 to 21). If p_1 is reachable from p_2, then we again search for the next p_2 in topological order (lines 22 to 23). If neither p_2 is reachable from p_1, nor p_1 is reachable from p_2, then we draw a *0*-cost pseudo-edge from the task with higher *ablevel[...]* to the task with lower *ablevel[...]* (lines 24 to 30). For the case of equal *ablevel[...]*, we draw a *0*-cost pseudo edge from the task with lower *trank[...]* to the task with higher *trank[...]*. We continue in this manner until all the nodes of the two clusters are examined (the outer ***while*** loop from lines 04 to 36). And finally in lines 37 to 39, we merge the two clusters C_i and C_j.

Lines 01 to 02 have complexity *O(1)*. In line 03, *Allocated-Bottom-Level(...)* has complexity $O(|V| + |E|)$ (Sinnen [27]). The ***while*** loop from lines 04 to 36 has complexity $O(|V|)$. The ***for*** loop from lines 37 to 39 has complexity $O(|V|)$. Therefore the algorithm *Merge(...)* has complexity $O(|V| + |E|)$.

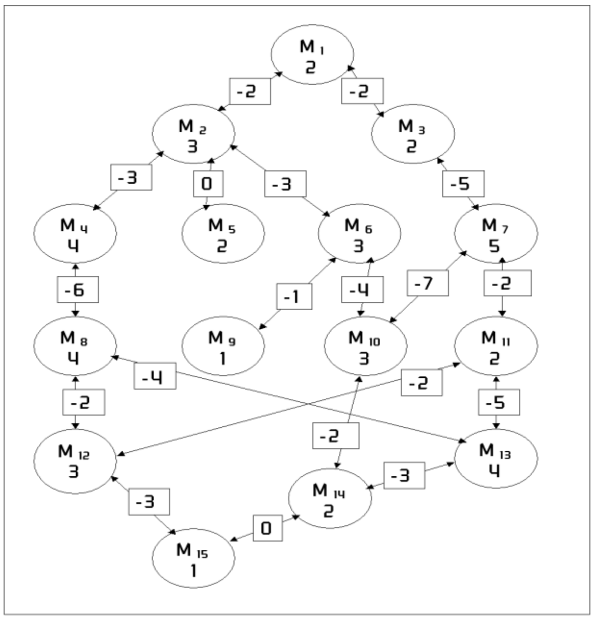
4.3. Evaluate-Time

```
Evaluate-Time(T, cluster[...])
01 stack ← empty
02 for k ← 1 to |V|
03     do atlevel[k] ← 0
04        time[k] ← 0
05        backlink[k] ← 0
06 for each (k, m) ∈ E
07     do backlink[m] ← backlink[m] + 1
08 for k ← 1 to |V|
09     do if backlink[k] = 0
10          then Push(stack, k)
11 while stack ≠ empty
12    do i ← Pop(stack)
13       if atlevel[i] ≥ time[cluster[i]]
14          then time[cluster[i]] ← atlevel[i] + $w_i$
15          else time[cluster[i]] ← time[cluster[i]] + $w_i$
16       for each (i, j) ∈ E
17            do backlink[j] ← backlink[j] - 1
18       if cluster[i] = cluster[j]
19          then c ← 0
20          else c ← $w_{ij}$
21       if time[cluster[i]] + c > atlevel[j]
22          then atlevel[j] ← time[cluster[i]] + c
23       if backlink[j] = 0
24          then Push(stack, j)
25 return Max(time[...])
```

In line 01, *stack* is initially empty. *stack* is used to traverse the nodes of the task graph. In the ***for*** loop from lines 02 to 05, the arrays *atlevel[...]*, *time[...]*, and *backlink[...]* are initialized to *0*. *atlevel[k]* is the allocated top level (Sinnen [27]) of task M_k. Given an allocation of tasks, the *allocated top level* of a task M_k is the length of the longest path ending with M_k (excluding the computation weight of M_k that is w_k) in which the weight of an edge is taken as *0* if the two tasks corresponding to the edge are allocated on the same processor. *time[k]* is the finish time of the processor *k* at which it finishes all of its allocated tasks. *backlink[k]* is the number of edges that are incident on the task M_k. In the ***for*** loop from lines 06 to 07, *backlink[...]* of the tasks is evaluated. In the ***for*** loop from lines 08 to 10, the initial tasks that are not having any incident edge are pushed onto the stack using the function Push(...). In the ***while*** loop from lines 11 to 24, we repeatedly pop a task using the function *Pop(...)* from the stack and compare its allocated top level with the partial execution time of the processor on which it is allocated. We update the *time[...]* value of the processor depending on whether processor has to remain idle for some time or not (lines 12 to 15). In the ***for*** loop from lines 16 to 17, for each outgoing edge we decrement the corresponding *backlink[...]* value indicating the completion of communication corresponding to the edges. In lines 18 to 20, *c* is the communication time from the task M_i to the task M_j depending upon whether the two tasks are allocated on the same processor or not. In lines 21 to 22, the allocated top level of successors is updated according to the definition. In lines 23 to 24, tasks are pushed onto the stack if they have completed all of their communications. And finally in line 25, the maximum of partial execution time of processors (using the function *Max(...)*) is returned that is the parallel execution time of the given scheduling if the task graph *T* is a scheduled DAG.

Line 01 has complexity *O(1)*. The ***for*** loop from lines 02 to 05, and 08 to 10, each have complexity $O(|V|)$. The ***for*** loop from lines 06 to 07 has complexity $O(|E|)$. The ***while*** loop from lines 11 to 24 has complexity $O(|E|)$ because of the ***for*** loop from lines 16 to 17. Line 25 has complexity $O(|V|)$. Therefore the algorithm *Evaluate-Time(...)* has complexity $O(|V| + |E|)$.

5. A Sample Run of the CPPS Algorithm



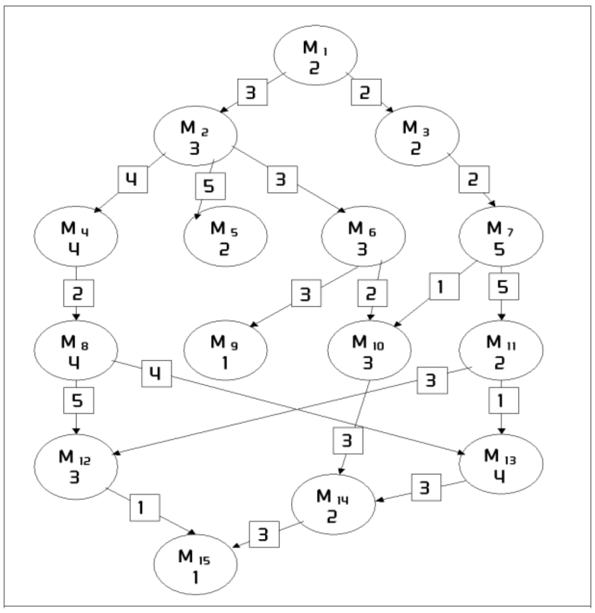


Fig. 1 (left): The example task graph. Fig. 2 (right): The initial clusters and the priorities between them. Parallel execution time is *39*.

As an example for explaining the steps involved in the CPPS algorithm, we will consider the task graph in Fig. 1, that is taken from Kadamuddi and Tsai [14], for clustering. Tasks are shown in circles with their labels and execution times. Communication delays (edge weights) are shown in rectangular boxes along their respective edges.

Initially each task is kept in a separate cluster as shown in Fig. 2, in which clusters are shown in circles listing their constituent tasks, along with the weight of the cluster. In rectangular boxes, along the edges, are shown the priorities between different cluster-pairs. Parallel execution time for this clustering comes out to be *39*.

In Fig. 2, the cluster-pairs $\{M_2\}$ and $\{M_5\}$; and $\{M_{14}\}$ and $\{M_{15}\}$ are examined for merging in non-increasing order of priorities between them. After merging, their parallel execution time comes out to be *39*, and *36* respectively. Therefore the cluster-pair $\{M_{14}\}$ and $\{M_{15}\}$ is merged in Fig. 3, since it is the first cluster-pair that is reducing the parallel execution time from the previous value of *39* (Fig. 2) to *36* (Fig. 3).

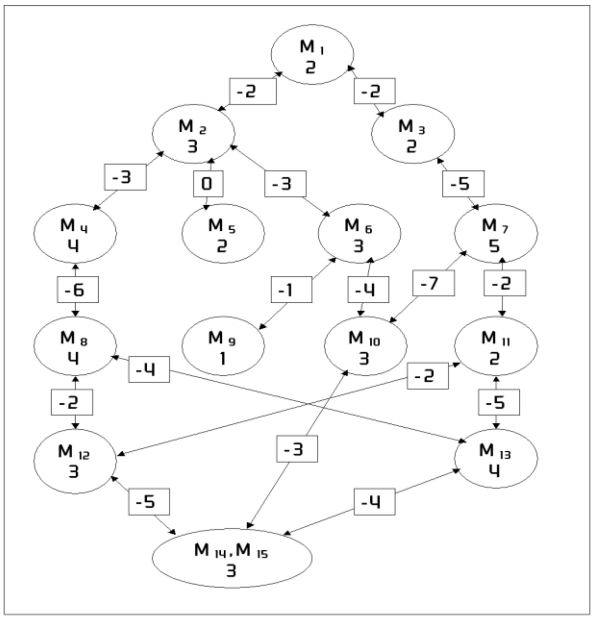
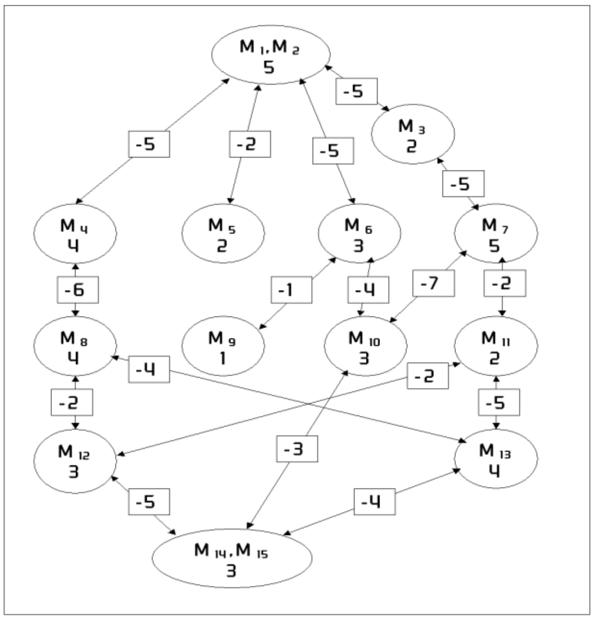



Fig. 3 (left): Clustering after merging *{M_{14}}* and *{M_{15}}*. Parallel execution time is *36*. Fig. 4 (right): Clustering after merging *{M_1}* and *{M_2}*. Parallel execution time is *33*.

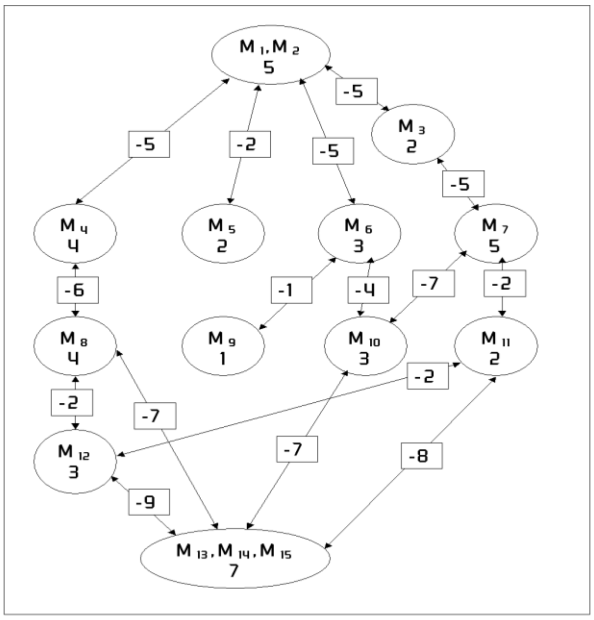
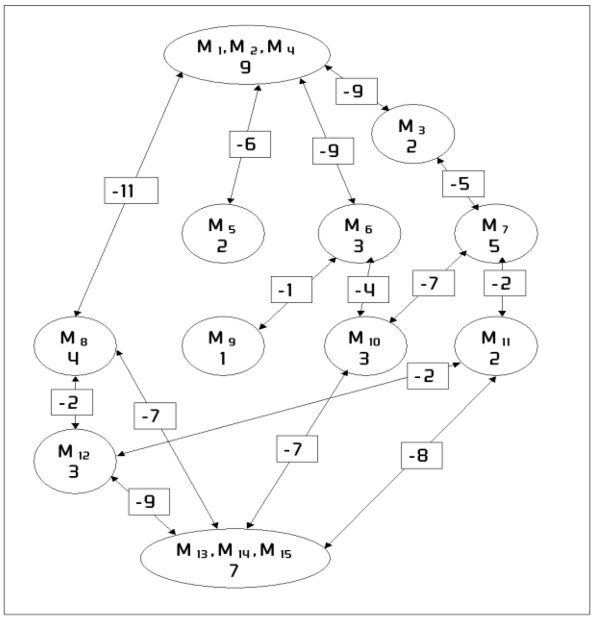



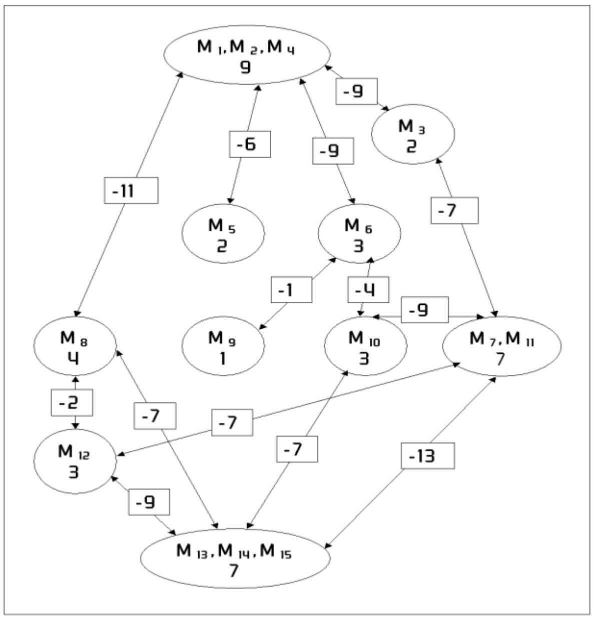
Fig. 5 (left): Clustering after merging *{M_{13}}* and *{M_{14}, M_{15}}*. Parallel execution time is *30*. Fig. 6 (right): Clustering after merging *{M_1, M_2}* and *{M_4}*. Parallel execution time is *28*.

In Fig. 3, the cluster-pairs *{M_2}* and *{M_5}*; *{M_6}* and *{M_9}*; *{M_7}* and *{M_{11}}*; *{M_8}* and *{M_{12}}*; *{M_{11}}* and *{M_{12}}*; and *{M_1}* and *{M_2}* are examined for merging in non-increasing order of priorities between them. After merging, their parallel execution time comes out to be *36*, *36*, *36*, *36*, *36*, and *33* respectively. Therefore the cluster-pair *{M_1}* and *{M_2}* is merged in Fig. 4, since it is the first cluster-pair that is reducing the parallel execution time from the previous value of *36* (Fig. 3) to *33* (Fig. 4).

In Fig. 4, the cluster-pairs *{M_6}* and *{M_9}*; *{M_8}* and *{M_{12}}*; *{M_{11}}* and *{M_{12}}*; *{M_1, M_2}* and *{M_5}*; *{M_7}* and *{M_{11}}*; *{M_{10}}* and *{M_{14}, M_{15}}*; and *{M_{13}}* and *{M_{14}, M_{15}}* are examined for merging in non-increasing order of priorities between them. After merging, their parallel execution time comes out to be *33*, *33*, *33*, *33*, *33*, *33*, and *30* respectively. Therefore the cluster-pair *{M_{13}}* and *{M_{14}, M_{15}}* is merged in Fig. 5, since it is the first cluster-pair that is reducing the parallel execution time from the previous value of *33* (Fig. 4) to *30* (Fig. 5).

In Fig. 5, the cluster-pairs *{M_6}* and *{M_9}*; *{M_8}* and *{M_{12}}*; *{M_{11}}* and *{M_{12}}*; *{M_7}* and *{M_{11}}*; *{M_1, M_2}* and *{M_5}*; *{M_6}* and *{M_{10}}*; *{M_3}* and *{M_7}*; and *{M_1, M_2}* and *{M_4}* are examined for merging in non-increasing order of priorities between them. After merging, their parallel execution time comes out to be *30*, *30*, *30*, *30*, *30*, *30*, *30*, and *28* respectively. Therefore the cluster-pair *{M_1, M_2}* and *{M_4}* is merged in Fig. 6, since it is the first cluster-pair that is reducing the parallel execution time from the previous value of *30* (Fig. 5) to *28* (Fig. 6).

In Fig. 6, the cluster-pairs *{M_6}* and *{M_9}*; *{M_{11}}* and *{M_{12}}*; and *{M_7}* and *{M_{11}}* are examined for merging in non-increasing order of priorities between them. After merging, their parallel execution time comes out to be *28*, *28*, and *26* respectively. Therefore the cluster-pair *{M_7}* and *{M_{11}}* is merged in Fig. 7, since they are the first pair of clusters that is reducing the parallel execution time from the previous value of *28* (Fig. 6) to *26* (Fig. 7).



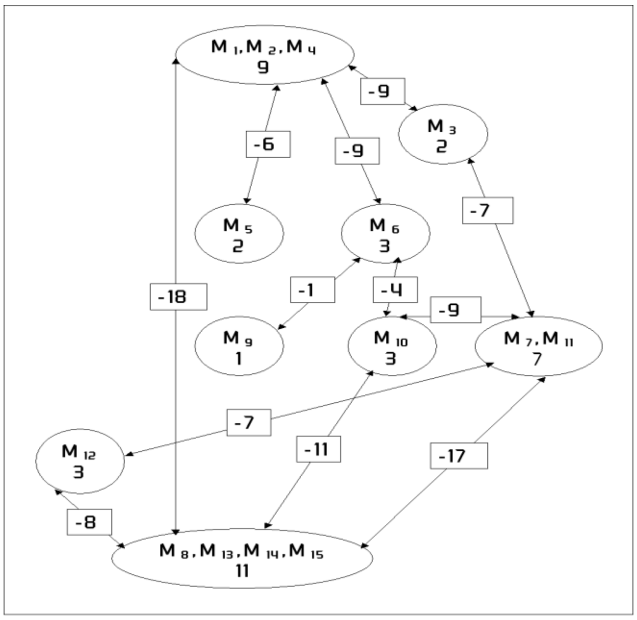


Fig. 7 (left): Clustering after merging *{M_7}* and *{M_{11}}*. Parallel execution time is *26*. Fig. 8 (right): Clustering after merging *{M_8}* and *{M_{13}, M_{14}, M_{15}}*. Parallel execution time is 25.

In Fig. 7, the cluster-pairs *{M_6}* and *{M_9}*; *{M_8}* and *{M_{12}}*; *{M_7}* and *{M_{11}}*; *{M_1, M_2, M_4}* and *{M_5}*; *{M_7, M_{11}}* and *{M_{12}}*; *{M_{10}}* and *{M_{13}, M_{14}, M_{15}}*; and *{M_8}* and *{M_{13}, M_{14}, M_{15}}* are examined for merging in non-increasing order of priorities between the cluster-pairs. After merging, their parallel execution time comes out to be *26*, *26*, *26*, *26*, *26*, *26*, and *25* respectively. Therefore, the cluster-pair *{M_8}* and *{M_{13}, M_{14}, M_{15}}* is merged in Fig. 8, since it is the first cluster-pair that is reducing the parallel execution time from the previous value of *26* (Fig. 7) to *25* (Fig. 8).

In Fig. 8, the cluster-pairs *{M_6}* and *{M_9}*; *{M_6}* and *{M_{10}}*; *{M_1, M_2, M_4}* and *{M_5}*; *{M_7, M_{11}}* and *{M_{12}}*; *{M_3}* and *{M_7, M_{11}}*; *{M_8, M_{13}, M_{14}, M_{15}}* and *{M_{12}}*; *{M_1, M_2, M_4}* and *{M_3}*; *{M_7, M_{11}}* and *{M_{10}}*; *{M_1, M_2, M_4}* and *{M_6}*; *{M_8, M_{13}, M_{14}, M_{15}}* and *{M_{10}}*; *{M_7, M_{11}}* and *{M_8, M_{13}, M_{14}, M_{15}}*; and *{M_1, M_2, M_4}* and *{M_8, M_{13}, M_{14}, M_{15}}* are examined for merging in non-increasing order of priorities between the cluster-pairs. After merging, their parallel execution time comes out to be *25*, *25*, *25*, *25*, *25*, *26*, *28*, *25*, *25*, *25*, *27*, and *23* respectively. Therefore the cluster-pair *{M_1, M_2, M_4}* and *{M_8, M_{13}, M_{14}, M_{15}}* is merged in Fig. 9, since it is the first cluster-pair that is reducing the parallel execution time from the previous value of *25* (Fig. 8) to *23* (Fig. 9).

In Fig. 9, the cluster-pairs *{M_6}* and *{M_9}*; *{M_6}* and *{M_{10}}*; *{M_3}* and *{M_7, M_{11}}*; *{M_7, M_{11}}* and *{M_{12}}*; *{M_7, M_{11}}* and *{M_{10}}*; *{M_1, M_2, M_4, M_8, M_{13}, M_{14}, M_{15}}* and *{M_5}*; *{M_1, M_2, M_4, M_8, M_{13}, M_{14}, M_{15}}* and *{M_{12}}*; *{M_1, M_2, M_4, M_8, M_{13}, M_{14}, M_{15}}* and *{M_6}*; *{M_1, M_2, M_4, M_8, M_{13}, M_{14}, M_{15}}* and *{M_{10}}*; *{M_1, M_2, M_4, M_8, M_{13}, M_{14}, M_{15}}* and *{M_3}*; and *{M_1, M_2, M_4, M_8, M_{13}, M_{14}, M_{15}}* and *{M_7, M_{11}}* are examined for merging in non-increasing order of priorities between the cluster-pairs. After merging, their parallel execution time comes out to be *23*, *23*, *23*, *23*, *24*, *25*, *26*, *26*, *24*, *25*, and *28* respectively. Therefore we cannot merge any cluster-pair in Fig. 9, since no cluster-pair is able to reduce the parallel execution time from the current value of *23* (Fig. 9) to a smaller value.

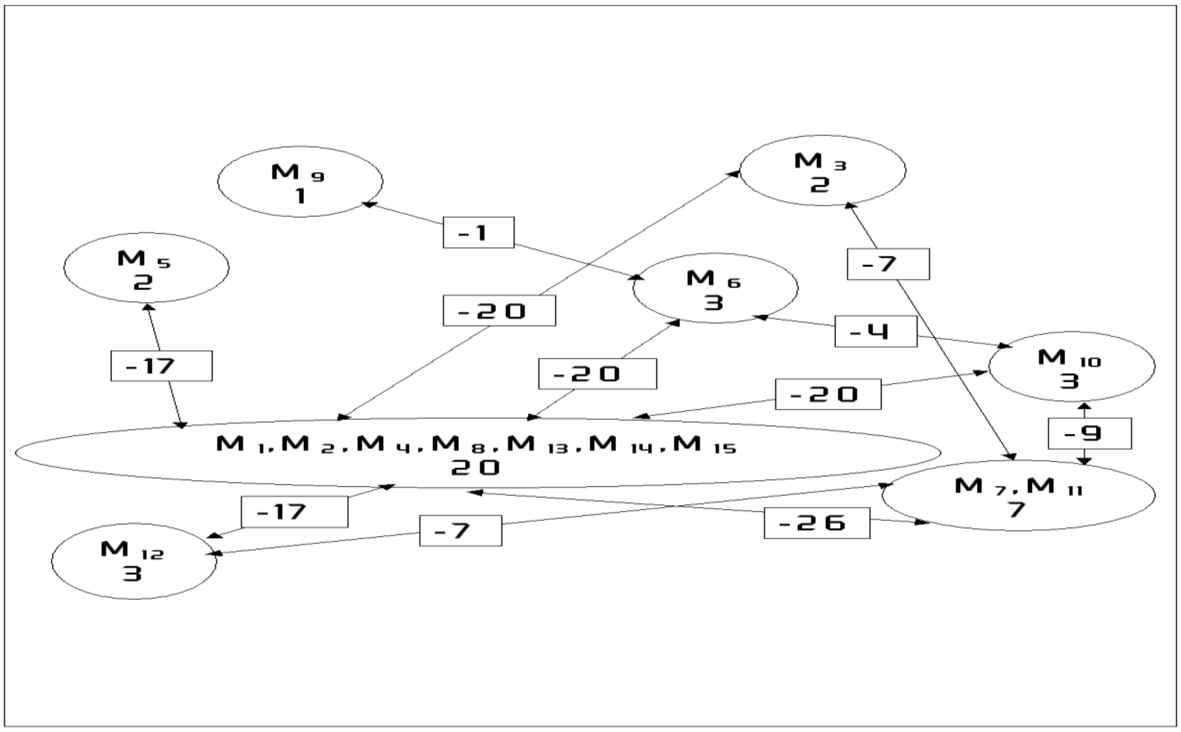


Fig. 9: Clustering after merging *{M_1, M_2, M_4}* and *{M_8, M_{13}, M_{14}, M_{15}}*. Parallel execution time is *23*.

The CPPS algorithm stops at this point. The clusters generated are: *{M_1, M_2, M_4, M_8, M_{13}, M_{14}, M_{15}}*, *{M_3}*, *{M_5}*, *{M_6}*, *{M_7, M_{11}}*, *{M_9}*, *{M_{10}}*, and *{M_{12}}*. Parallel execution time comes out to be *23*.

6. Experimental Results

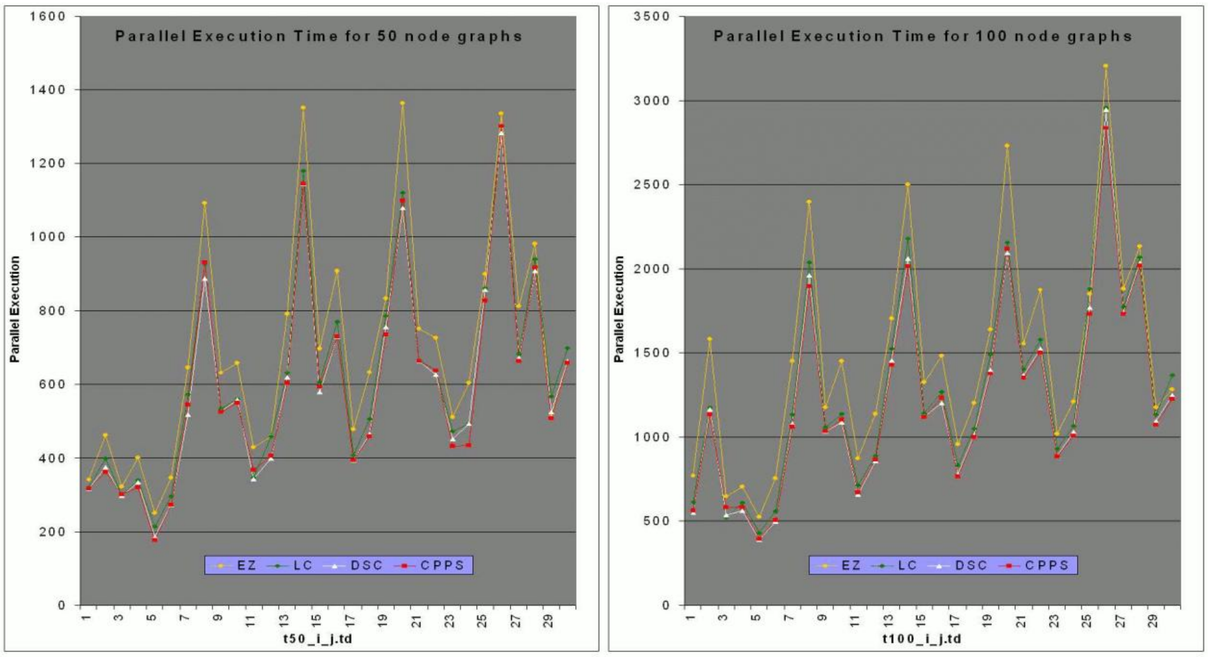
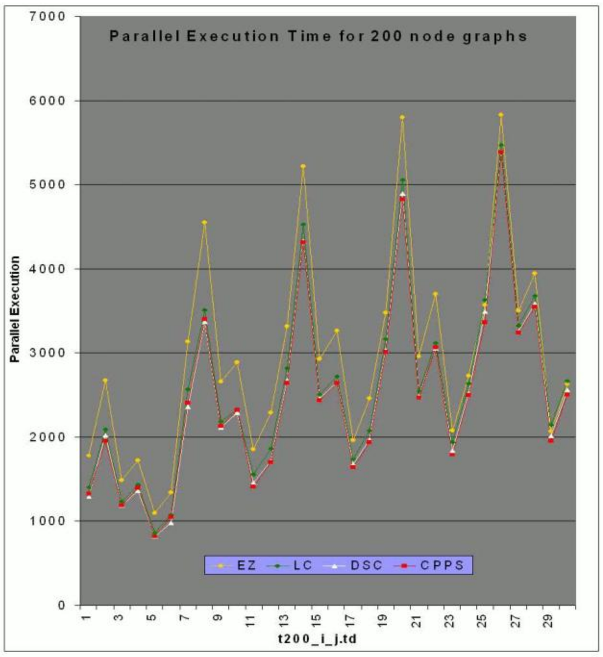


Fig. 10 (left): Parallel execution time for 50 node task graphs. Average improvement of CPPS over EZ is *15.01%*; over LC is *4.57%*; and over DSC is *0.59%*. Fig. 11 (right): Parallel execution time for 100 node task graphs. Average improvement of CPPS over EZ is *17.43%*; over LC is *4.72%*; and over DSC is *0.32%*.



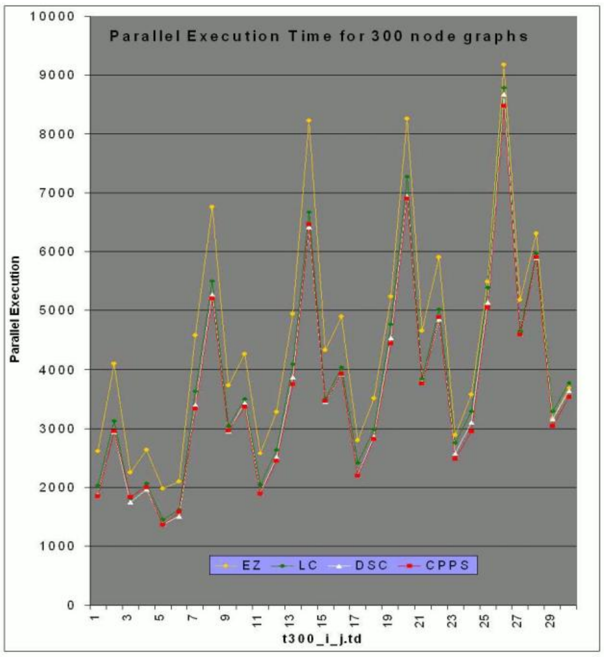
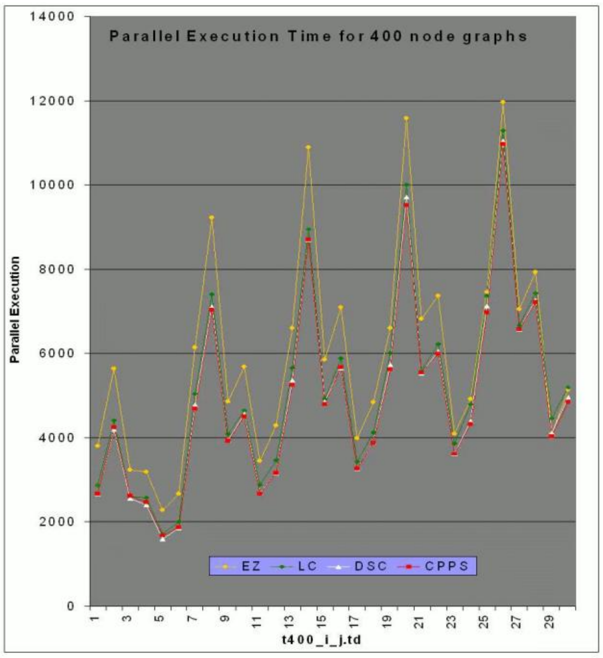


Fig. 12 (left): Parallel execution time for 200 node task graphs. Average improvement of CPPS over EZ is *17.16%*; over LC is *4.68%*; and over DSC is *0.45%*.Fig. 13 (right): Parallel execution time for 300 node task graphs. Average improvement of CPPS over EZ is *18.88%*; over LC is *5.07%*; and over DSC is *0.77%*.



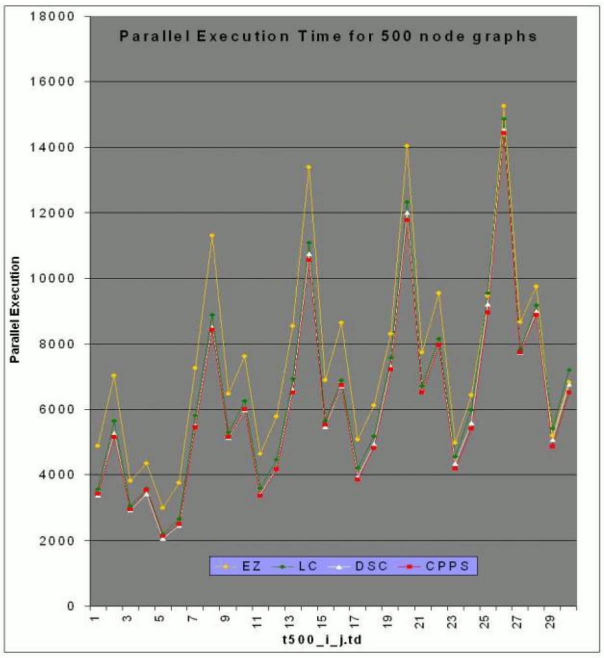


Fig. 14 (left): Parallel execution time for 400 node task graphs. Average improvement of CPPS over EZ is *17.92%*; over LC is *4.86%*; and over DSC is *0.32%*. Fig. 15 (right): Parallel execution time for 500 node task graphs. Average improvement of CPPS over EZ is *19.13%*; over LC is *5.11%*; and over DSC is *0.86%*.

The CPPS algorithm is compared with three well known algorithms: Sarkar's Edge Zeroing (EZ) algorithm [26], Kim and Browne's Linear Clustering (LC) algorithm [16], and Yang and Gerasoulis' Dominant Sequence Clustering (DSC) algorithm [32]. The algorithms are tested on benchmark task graphs of Tatjana and Gabriel [5, 6]. We have tested for *180* task graphs having the number of nodes as *50*, *100*, *200*, *300*, *400*, and *500* respectively. Each task graph has a label as *tn_i_j.td*. Here *n* is the number of nodes. *i* is a parameter depending on the edge density. Its possible values are: *20*, *40*, *50*, *60*, and *80*. For each combination of *n* and *i*, there are *6* task graphs that are indexed by *j*. *j* ranges from *1* to *6*. Therefore, for each *n* there are *30* task graphs.

We define the *performance improvement ratio* of an algorithm *A* over an algorithm *B* (*PI(A, B)*) for a given instance of task graph as follows:

$$PI(A, B) = 1 - PT(A) / PT(B), \tag{14}$$

where *PT(X)* is the *parallel execution time* of algorithm *X*. For performance evaluation we take the average value of *PI(A, B)* for a given set of task graphs.

For the values of *n* having *50*, *100*, *200*, *300*, *400*, and *500* respectively, Fig. 10 to Fig. 15 show the comparison between the four algorithms: EZ, LC, DSC, and CPPS for the parallel execution time. The average improvement of CPPS algorithm over EZ algorithm ranges from *15.01%* to *19.13%*. The average improvement of CPPS algorithm over LC algorithm ranges from *4.57%* to *5.11%*. The average improvement of CPPS algorithm over DSC algorithm ranges from *0.32%* to *0.86%*.

7. Conclusion

We formulated Sarkar's EZ algorithm [26] for scheduling precedence constrained task graphs on a parallel system as a task dependent priority scheme in which the priority is defined for pairs of tasks. We then extended this idea to cluster dependent priority scheme in which the priority is defined for pairs of clusters. Using this idea we developed the CPPS algorithm. Using a suitable cluster dependent priority scheme of complexity $O(|V||E|(|V| + |E|))$ we compared our algorithm with some well known scheduling algorithms. Experimental results demonstrated good performance of the CPPS algorithm.

We give two research directions for future work. The first line of research is to reduce the complexity of the CPPS algorithm without compromising its performance. The second line of research is to study the performance of the CPPS algorithm for more intelligent priority schemes. The example given in this paper is a static priority scheme. We can also think of dynamic priority schemes.

8. References

[1] T.L. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*. 1974, **17**: 685-689.

[2] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. Automatic program parallelization. *Proceedings of the IEEE*. 1993, **81**: 211-243.

[3] E.G. Coffman, and R.L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*. 1972, **1**: 200-213.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*(2nd edition). The MIT Press, 2001.

[5] T. Davidovic, and T.G. Crainic. Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multiprocessor systems. *Computers & Operations Research*. 2006, **33**: 2155-2177.

[6] T. Davidovic. Benchmark task graphs available online at http://www.mi.sanu.ac.rs/~tanjad/sched_results.htm.

[7] M.D. Dikaiakos, A. Rogers, and K. Steiglitz. A Comparison of Techniques Used for Mapping Parallel Algorithms to Message-Passing Multiprocessors. *Technical Report.* Princeton Univ, 1994.

[8] A. Gerasoulis and T. Yang. A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors. *J. Parallel and Distributed Computing*. 1992, **16**: 276-291.

[9] R.L. Graham. Bounds for multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*. 1969, **17**: 416-419.

[10] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing* (2nd edition). London: Pearson Addison Wesley, 2003.

[11] C. Hanen, and A. Munier. An approximation algorithm for scheduling dependent tasks on m processsors with small communication delays. *ETFA 95 (INRIA/IEEE Symposium on Emerging Technology and Factory Animation).* IEEE Press. 1995, pp. 167-189.

[12] E. Horowitz, S. Sahni, and S. Rajasekaran. Fundamentals of Computer Algorithms. W. H. Freeman, 1998.

[13] T. Hu. Parallel sequencing and assembly line problems. *Operations Research*. 1961, **9**: 841-848.

[14] D. Kadamuddi, and J.J.P. Tsai. Clustering Algorithm for Parallelizing Software Systems in Multiprocessors Environment. *IEEE Transations on Software Engineering*. 2000, **26**: 340-361.

[15] H. Kasahara, and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*. 1984, **33**: 1023-1029.

[16] S.J. Kim, and J.C. Browne. A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures. *Proc. 1988 Int'l Conf. Parallel Processing*. 1998, **3**: 1-8.

[17] Y.K. Kwok, and I. Ahmad. Dynamic Critical-Path Scheduling: an Effective Technique for Allocating Task Graphs to Multiprocessors. *IEEE Trans. Parallel and Distributed Systems*. 1996, **7**: 506-521.

[18] Y.K. Kwok, and I. Ahmad. FASTEST: A Practical Low-Complexity Algorithm for Compile-Time Assignment of Parallel Programs to Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*. 1999, **10**: 147-159.

[19] V.M. Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Trans. Computers*. 1988, **37**: 1384-1397.

[20] C.Y. Lee, J.J. Hwang, Y.C. Chow, and F.D. Anger. Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*. 1988, **7**: 141-147.

[21] Z. Liu. A note on Graham's bound. *Information Processing Letters*. 1990, **36**: 1-5.

[22] B.A. Malloy, E.L. Lloyd, and M.L. Soffa. Scheduling DAG's for Asynchronous Multiprocessor Execution. *IEEE Trans. Parallel and Distributed Systems*. 1994, **5**: 498-508.

[23] P.K. Mishra, K.S. Mishra, and A. Mishra. A Clustering Heuristic for Multiprocessor Environments using Computation and Communication Loads of Modules. *International Journal of Computer Science & Information Technology (IJCSIT)*. 2010, **2**: 170-182.

[24] C. Papadimitriou, and M. Yannakakis. Towards an Architecture Independent Analysis of Parallel Algorithms. *SIAM Journal on Computing*. 1990, **19**: 322-328.

[25] C. Papadimitriou. *Computational Complexity.* Addison Wesley Longman, 1994.

[26] V. Sarkar. Partitioning and Scheduling Parallel Programs for Multiprocessors. *Research Monographs in Parallel and Distributed Computing*. MIT Press, 1989.

[27] O. Sinnen. *Task Scheduling for Parallel Systems*. John Wiley & Sons, 2007.

[28] G.C. Sih, and E.A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*. 1993, **4**: 175-187.

[29] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

[30] M.Y. Wu, and D.D. Gajski. Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Trans. Parallel and Distributed Systems*. 1990, **1**: 330-343.

[31] M.Y. Wu, W. Shu, and J. Gu. Efficient Local Search for DAG Scheduling. *IEEE Transactions on Parallel and Distributed Systems*. 2001, **12**: 617-627.

[32] T. Yang, and A. Gerasoulis. A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors. *Proc. Fifth Int'l Conf. Supercomputing*. 1991, pp. 633-642.

[33] T. Yang, and A. Gerasoulis. PYRROS: Static Scheduling and Code Generation for Message Passing Multiprocessors. *Proc. Sixth Int'l Conf. Supercomputing*. 1992, pp. 428-437.

[34] T. Yang, and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing.* 1993, **19**: 1321-1344.