Parallel Computing 38 (2012) 175–193

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Scheduling for heterogeneous systems using constrained critical paths

Minhaj Ahmad Khan

Department of Computer Science, Bahauddin Zakariya University Multan, Pakistan

ARTICLE INFO

Article history:
Received 7 January 2011
Received in revised form 17 October 2011
Accepted 8 January 2012
Available online 20 January 2012

Keywords:
Scheduling algorithms
Heterogeneous scheduling
List scheduling
Static scheduling
Parallel applications

ABSTRACT

A complex computing problem may be efficiently solved on a system with multiple processing elements by dividing its implementation code into several tasks or modules that execute in parallel. The modules may then be assigned to and scheduled on the processing elements so that the total execution time is minimum. Finding an optimal schedule for parallel programs is a non-trivial task and is considered to be NP-complete.

For heterogeneous systems having processors with different characteristics, most of the scheduling algorithms use greedy approach to assign processors to the modules. This paper suggests a novel approach called constrained earliest finish time (CEFT) to provide better schedules for heterogeneous systems using the concept of the *constrained critical paths* (*CCPs*). In contrast to other approaches used for heterogeneous systems, the CEFT strategy takes into account a broader view of the input task graph. Furthermore, the statically generated CCPs may be efficiently scheduled in comparison with other approaches.

The experimentation results show that the CEFT scheduling strategy outperforms the well-known HEFT, DLS and LMT strategies by producing shorter schedules for a diverse collection of task graphs.

1. Introduction

Modern computer systems employ multiple processors which may work in parallel to enhance the performance of an application. The parallelism may then be fully exploited if the parallel modules of the application are effectively scheduled. The modules of the application may communicate and synchronize at several points during execution. For heterogeneous systems with a combination of CELL processors, GPUs and multi-core processors, or distributed memory systems, a large overhead of communication may incur thereby limiting the overall performance of the application. An effective schedule for executing modules on multiple processors may result in overlapping the communication with the execution of code. Consequently, the scheduling may contribute to a large gain in the performance of the application.

In general, scheduling defines the execution order of the modules of the application together with the assignment of the processors to these modules. The schedule length encompasses the entire execution and communication cost of all the modules, and is also termed as *makespan*. Finding optimal schedules with the minimum schedule length is considered to be NP-complete even with the homogeneous processors [1–4]. The factor of heterogeneity therefore results in additional complexity for finding optimal solution to this problem. There are however classes of graphs for which polynomial solutions of scheduling exist such as series–parallel graphs, coarse-grain trees, fork and join graphs [5–8]. For arbitrary task graphs, while taking into account the communication cost, the problem is NP-complete even with two processors [9]. This is why, different heuristics have been proposed for arbitrary number of tasks to be assigned to arbitrary number of processors which work with low complexity and are still able to produce very effective schedules.

The tasks of the application are represented as nodes in a task graph which is essentially a *directed acyclic graph* (*DAG*). The edges in the DAG are used to describe dependencies among the tasks. These edges are labeled with communication cost that

E-mail address: minhajahmad@gmail.com

0167-8191/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2012.01.001

incurs when the tasks send and receive information to/from other tasks. A task may start execution only when all its predecessors have completed execution and all the required data have been communicated to it. The scheduling strategies arrange the tasks so that the precedence constraints are respected and the most appropriate processor is assigned to each task.

Most of the existing scheduling algorithms are categorized into list scheduling, duplication based scheduling, and clustering. The list scheduling [10–19] in its simplest form, produces a schedule in two major phases. In the first phase, the tasks are assigned priorities so that they are processed in a sequence. In the second phase, the processors are assigned to the tasks. The priorities are normally assigned based on the execution and communication costs associated with the tasks. In contrast to the list scheduling approach, the duplication based algorithms [20–26] attempt to duplicate the tasks in order to minimize the total execution cost. For heterogeneous systems, the duplication may effectively reduce the schedule length as the communication cost is eliminated by placing the tasks on the same processor.

The clustering algorithms [5,27–30] consider collections of tasks termed as *clusters* to be mapped to appropriate processors. In general, these algorithms work for homogeneous type of systems with unbounded number of processors. The clusters are then processed further to adapt for a bounded number of processors. On the one hand, the list scheduling based algorithms lack the consideration of a global view of the task graph, and on the other than hand, the duplication based algorithms incur a large overhead and are considered inappropriate for systems with small communication cost. Similarly, the clustering algorithms incorporate the complex merging steps thereby increasing the complexity of scheduling and its implementation.

This paper targets scheduling for heterogeneous systems and proposes an approach that considers a global view of the task graph. It is based on the notion of a *constrained critical path* (*CCP*) which is a small task window representing ready tasks at one instance. The critical paths in the DAG are initially found, and subsequently, the tasks in the CCPs are scheduled using the finish time of the entire CCP. This approach not only takes a larger view of the task graph but also facilitates in producing schedules with shorter makespan and works with small complexity.

The rest of the article is organized as follows. Section 2 discusses the context of the scheduling approach. Section 3 elaborates the terms used for the suggested algorithm and also describes the CEFT scheduling approach. An execution trace of the algorithm is described in Section 4. The experimental results for various graphs are presented in Section 5. Section 6 describes the related work in comparison with the suggested approach. A summary of the findings and future work are given in Section 7.

2. Overview and context of the CEFT scheduling algorithm

The scheduling problem takes as input a directed acyclic graph $G = (V,E)$, with $|V| = n$ nodes representing tasks, and $|E| = m$ edges representing dependencies among the tasks. The edges of the graph are labeled with communication overhead that might incur during data transfer if two modules are executed on different processors. The communication overhead is zero when two communicating modules are executed on the same processor. It is also assumed that the execution and communication may be performed simultaneously by the processors. With heterogeneous processors, we use the unrelated model which implies that a processor can be faster for some tasks while being slower for other tasks. Similarly, the precedence constraints are defined by the edges which imply that for a node to start execution, all its predecessors should have completed their execution. A *start* node (with no predecessors) and an *exit* node (with no successors) to represent the beginning and end of execution respectively are added to the task graph.

In general, a critical path is the longest path from the *start* node to the *exit* node in the application task graph. The length of a path is computed as the sum of averages of the execution costs of the nodes and the communication cost along that path. Intuitively, an effective schedule is produced when the tasks that exist on the critical path are scheduled first. However, all the tasks on the critical path may not be ready. A task for which all the predecessors have been processed for scheduling is termed as a ready task. The collection of tasks comprising only the tasks ready for scheduling forms a *constrained critical path* (CCP). Each CCP may be assigned a single processor based on the finish time of its tasks.

Following the execution order, a task is considered ready for scheduling when its predecessors have been scheduled. For scheduling, the graph is initially traversed and critical paths based on average execution costs and communication cost are found. The graph is then pruned of the nodes that constitute a critical path. The subsequent traversals of the pruned graph produce the remaining critical paths.

While the nodes are being removed from the task graph, some pseudo-edges may be required to be added to the graph so that the graph remains to be connected. For a node having turned into a *free* node (not existing in any path from the *start* node to the *exit* node) after pruning, a pseudo-edge to the *start* node is added if it has no predecessors. Similarly, a pseudo-edge to the *exit* node is added if the node has no successors. The CCPs are subsequently formed by selecting ready nodes in the critical paths in a round-robin fashion.

Consider the graph with the nodes A, B, C, D, E & F as shown in Fig. 1. The first critical path w.r.t. the sum of average execution costs and the communication cost, from *start* to *exit* contains task nodes A, B & F. These nodes are then removed from the graph[1] together with the edges incident on them. The pseudo-edges from *start* node to the node C, from node C to the *exit* node and from the node E to the *exit* node, are then added to the pruned graph. It ensures that for each node there is always a path from *start* node to *exit* node that passes through that node. The resulting graph is shown in Fig. 2(a).

[1] Modification and traversal for CCPs is applied to a temporary clone of the task graph instead of the original graph.

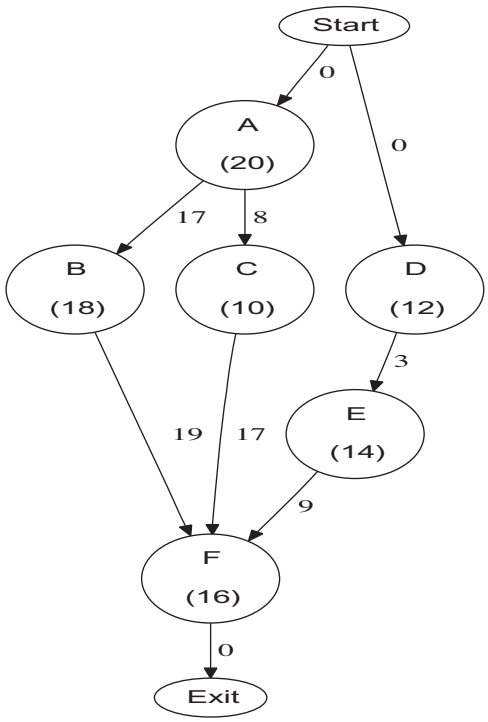
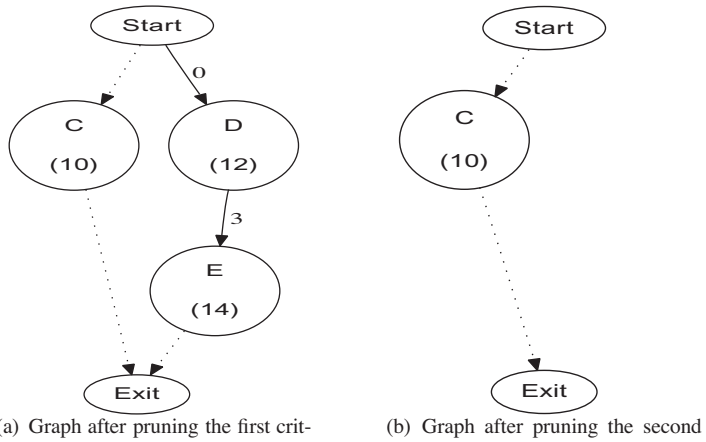


Fig. 1. Graph with nodes $A - F$ labeled with average execution costs and edges labeled with communication costs.



(a) Graph after pruning the first critical path $A - B - F$

(b) Graph after pruning the second critical path $D - E$

Fig. 2. States of the (temporary) graph during generation of critical paths.

Critical Paths
A-B-F
D-E
C

(a) The critical paths found using average execution costs

Constrained Critical Paths
A-B
D-E
C
F

(b) CCPs corresponding to critical paths

Fig. 3. CCPs and the critical paths for the example graph.

Table 1
Specific terms and their usage for the CEFT algorithm.

$T_{P_r}(w)$	Execution cost of a node w using processor P_r
$M(w,P_r,v,P_x)$	Communication cost from node v to w, if P_x has been assigned to node v and P_r is assigned to node w
$S_{P_r}(w,v)$	Possible start time of node w which is assigned the processor P_r with the v node being any predecessor of w which has already been scheduled
$Z_{P_r}(w)$	Finish time of node w using processor P_r
F_w	Actual finish time of node w
$E_{P_r}(Q_j)$	Finish time of the constrained critical path Q_j when processor P_r is assigned to it
A_{P_r}	Availability time of P_r
$Pred(w)$	Set of predecessors of node w
$Succ(w)$	Set of successors of node w
$AEC(w)$	Average execution cost of node w

Continuing in the similar fashion, the nodes *D* & *E* of the second critical path are removed to produce the graph shown in Fig. 2(b). The third critical path would therefore contain only the node *C*.

A node may be scheduled only if all its predecessors have been scheduled following the same order as that of execution. Therefore, the queues ready for scheduling need to be formed which are represented by CCPs. The critical paths are arranged in order of the length, and CCPs are found by traversing these critical paths in a round-robin fashion.

Fig. 3(a) and (b) show respectively the critical paths and the CCPs found for the example being considered. The first CCP contains the nodes A & B because these nodes can start execution. Since there are no more ready nodes in the first critical path, the second critical path is used to produce the second CCP containing the nodes D & E. Similarly, the third CCP contains the node C taken from the third critical path. Continuing in the round-robin fashion, we obtain the node E to produce the fourth CCP.

For all the tasks in a CCP, a single processor is assigned. Consideration of all the tasks in a CCP not only reduces the communication cost, but also benefits from a broader view of the task graph. This is why, it produces better scheduling results as compared to other scheduling approaches being used for heterogeneous systems.

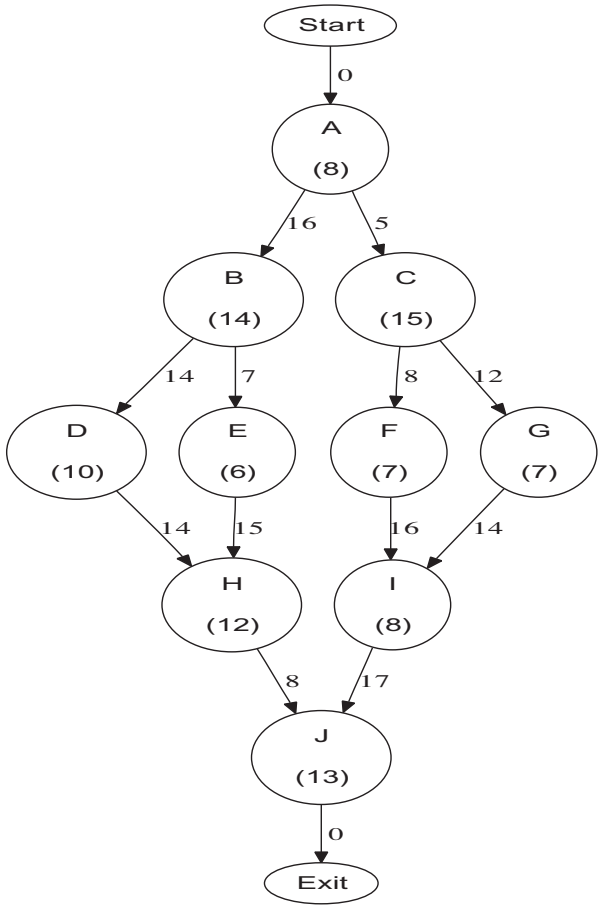


Fig. 4. Example graph containing task nodes *A* – *J* labeled with average execution costs and edges labeled with communication costs.

3. The constrained earliest finish time (CEFT) algorithm

The CEFT algorithm generates schedules for n tasks with $|P|$ heterogeneous processors. Some specific terms and their usage (indicated in Table 1) are elaborated as follows. The execution cost corresponding to a node w on processor P_r is represented as $T_{P_r}(w)$. Similarly, the communication cost associated with the node w through its predecessor v is represented as $M(w,P_r,\ v,P_x)$, when the processor P_x is assigned to the node v and P_r is assigned to the node w. In essence, if P_r and P_x are the same, the communication cost is considered to be zero. Corresponding to a predecessor v of the node w, let $S_{P_r}(w, v)$ represent the possible execution start time of the node w if the processor P_r is assigned to it. The value Z_{P_r} represents the finish time of node w using processor P_r, whereas, the actual finish time is represented by F_w. Similarly, the value $E_{P_r}(Q_j)$ is used to represent the finish time of constrained critical path Q_j when processor P_r is assigned to all of its tasks. After assignment, the processor availability time is updated, let A_{P_r} represent the time at which the processor P_r is available using insertion based policy. The sets $Pred(w)$ and $Succ(w)$ represent respectively the predecessors and successors of node w.

Algorithm 1. Algorithm *gen_CP*( ) for generating critical paths

1: // Let G_{tmp} contain vertices V_T in topological order and edges E
2: $count = 0$; $G_{tmp} = (V_T, E)$
3: **while** there are task nodes in G_{tmp} **do**
4: // Find the length of the critical path using
5: Initialize $L(v_i) = 0$, for each $v_i \in V_T$
6: **for** each $v_i \in V_T$ **do**
7: **for** each edge $v_i \rightarrow w_j$ **do**
8: $AEC(w_j) = \frac{\sum_{k=1}^{|P|} T_{P_k}(w_j)}{|P|}$
9: // Let $\Pi(v_i, w_j)$ be the weight of the edge $v_i \rightarrow w_j$
10: **if** $L(w_j) \leqslant L(v_i) + AEC(w_j) + \Pi(v_i, w_j)$**then**
11: $L(w_j) = L(v_i) + AEC(w_j) + \Pi(v_i, w_j)$
12: $PredC(w_j) = v_i$
13: **end if**
14: **end for**
15: **end for**
16: $loc = 1$, $maxL = 0$
17: **for** $k = 1$ to $|V_T|$ **do**
18: **if** $L(v_k) \geqslant maxL$ **then**
19: $maxL = L(v_k)$
20: $loc = k$
21: **end if**
22: **end for**
23: Let $node = V_{loc}$// Last node of the critical path
24: // Add nodes to the current critical path CP_{count}
25: $CP_{count} = \{V_{loc}\}$
26: **while** $node \neq start$ **do**
27: $node = PredC(node)$
28: $CP_{count} = CP_{count} \bigcup node$
29: **end while**
30: // Prune the graph and add pseudo-edges
31: $G' = (CP_{count}, E_{CP})$// E_{CP} are the edges of the nodes in CP_{count}
32: $G_{tmp} = G_{tmp} - G'$
33: **for** each $u_k \in CP_{count}$ **do**
34: **if** $|Pred(u_k)| == 0$ **then**
35: $G_{tmp} = G_{tmp} \bigcup (null, start \rightarrow u_k)$
36: **elsif** $|Succ(u_k)| == 0$ **then**
37: $G_{tmp} = G_{tmp} \bigcup (null, u_k \rightarrow exit)$
38: **end if**
39: **end for**
40: $count+ = 1$
41: **end while**
42: return CP

Initially, the critical paths are found as described in Algorithm 1. Using the graph nodes in topological order, V_T, the length of the critical path is computed (Steps 5–22). In a top-down fashion, the sum of average execution costs (*AEC* values) is computed for each node. The nodes along the path producing the largest value of the sum of *AEC* values and the communication cost form the critical path CP_{count}. A list of predecessors is also maintained in order to keep track of the nodes in the critical path. If two paths produce the same length, the last path (in topological traversal of the nodes) is selected as the critical path. The graph is then pruned of the nodes along the critical path (Steps 30–32). Pseudo-edges are subsequently added to the graph so that there is always a path from the *start* node to the *exit* node while traversing any of the task nodes of the graph (Steps 33–39). The procedure of finding critical paths is iterated until all the task nodes have been processed.

Algorithm 2. Algorithm for finding schedule

1: //PHASE 1: Find the constrained critical paths (CCPs)
2: //Find set of critical paths C
3: $C = gen\_CP()$
4: $j = 1$
5: **for** $i = 1$ to $|C|$ **do**
6: **while** there are ready nodes in C_i **do**
7: Insert ready node V_k into constrained critical path Queue (Q_j).
8: **end while**
9: $j \leftarrow j + 1$
10: $i \leftarrow i\%|C|$
11: **end for**
12: // PHASE 2: Assign and schedule tasks
13: **for** $j = \{1,2,\ldots,|Q|\}$ **do**
14: **for** each processor $P_r \in P$ **do**
15: **for** each node $w \in Q_j$ **do**
16: Find the start time of w w.r.t. predecessor node k

$$S_{P_r}(w,k) = \max((F_k + M(w,P_r,k,P_x)), A_{P_r}) \tag{1}$$

17: Find the finish time of the node

$$Z_{P_r}(w) = \max\left((S_{P_r}(w,k))_{\forall k \in Pred(w)}\right) + T_{P_r}(w) \tag{2}$$

18: **end for**
19: Find the finish time of the constrained critical path Q_j

$$E_{P_r}(Q_j) = \max\left((Z_{P_r}(w))_{\forall w \in Q_j}\right) \tag{3}$$

20: **end for**
21: Assign the processor to constrained critical path Q_j which minimizes the $E_{P_r}(Q_j)$ value.
22: Let P_x be the processor assigned, update the actual finish time

$$(F_w)_{\forall w \in Q_j} = (Z_{P_x}(w))_{\forall w \in Q_j} \tag{4}$$

23: **end for**

The CEFT scheduling approach (Algorithm 2) works in two phases. In the first phase, the CCPs are found which are a manifestation of queues that are ready for scheduling. At the beginning, the critical paths are generated using the *gen_CP*( ) method given in Algorithm 1. The critical paths are then traversed and the ready nodes are inserted into the CCP queues $Q_j, \forall j = 1,2,\ldots,|Q|$. A ready node in a critical path is the one that has all its predecessors processed. If there are no more ready nodes in a critical path, the CCP takes nodes from the next critical path following round-robin traversal of the critical paths.

In the second phase of the scheduling part, the CCPs are assigned the processors. All the CCP based queues which contain tasks ready for scheduling, are traversed in order (Step 13). The earliest start time S_{P_r} of each node w with respect to each processor P_r is then found (Eq. (1)). Corresponding to a predecessor of node w, the S_{P_r} value is the maximum of the possible start time and availability time A_{P_r}.

The finish time Z_{P_r} of a node w is computed as the sum of the maximum of the possible start times (corresponding to the predecessors of node w) and the execution cost of w on processor P_r (Eq. (2)). Since a CCP may contain several nodes, the finish time of the CCP is the largest value of the finish times of all the tasks of the CCP using the same processor P_r (Eq. (3)). The processor with the minimum finish time of the CCP is then assigned to all of its nodes (Step 21). Subsequently, the actual finish time of the task is updated and the processor assignment continues iteratively for all the CCPs found.

The overall complexity of the CEFT algorithm for a task graph with n nodes and m edges is $O(n * |P|(n + m + deg_{in}))$, where $|P|$ is the number of processors considered for scheduling and deg_{in} represents the in-degree of the task graph.

4. Application of the algorithm

Consider the example shown in Fig. 4. Its edges are labeled with the communication costs, whereas the execution costs are shown in Table 2.

Initially, the critical path $A - B - D - H - J$ with length 109 is found. The graph is then pruned of these nodes and pseudo-edges are added as described in Section 2. The resulting graph is shown in Fig. 5(a). Subsequently, the critical paths, $C - G - I$ with length 56, F with length 7 and E with length 6, are found. The graph is pruned of the corresponding nodes in each critical path as shown in Fig. 5(b) and (c) respectively.

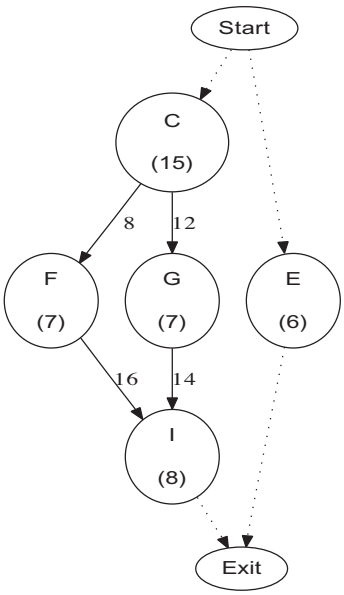
After having determined the critical paths, the CCPs are found as shown in column 1 of Table 3. For all the CCPs, the finish time for each processor is also shown in columns 2–4.

Each CCP corresponds to a queue ready for scheduling at an instance of scheduling, and is processed in order. For the first CCP, the processors P_1, P_2 & P_3 produce the finish times of 28, 30 & 38 respectively. The processor P_1 producing the minimum finish time is then assigned to the first CCP. The processors producing the minimum finish time P_2, P_3, P_3, P_3, P_3 & P_3 are assigned respectively to the subsequent CCPs. Consequently, the CEFT algorithm produces the schedule of length 73.

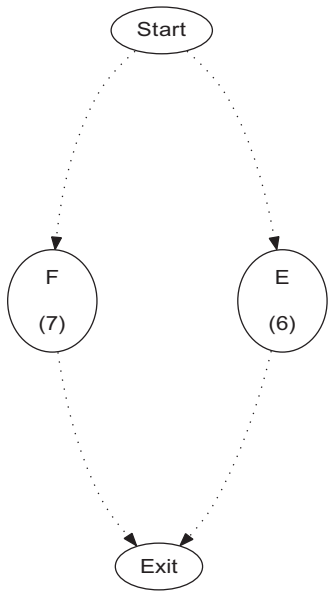
In contrast to CEFT, the HEFT algorithm initially orders the tasks w.r.t their ranks. The processor assignment takes int account the *Earliest Finish Time (EFT)* of each task. For each task, the processor with the minimum value of EFT is assigned. For the task graph given in Fig. 4, the HEFT algorithm processes the nodes in the order given as A, C, B, F, G, D, E, I, H & J, and based on the EFT values, assigns the processors P_1, P_1, P_1, P_3, P_1, P_1, P_3, P_3, P_3 & P_3, respectively. It produces a schedule of length 85, which is 16.4% larger than that produced by the CEFT algorithm.

Table 2
Execution costs for the task graph given in Fig. 4.

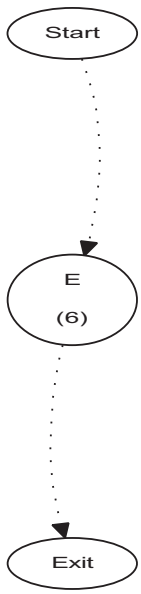
Tasks	P_1	P_2	P_3
A	7	8	9
B	11	14	17
C	12	15	18
D	10	8	12
E	5	7	6
F	9	7	5
G	6	8	7
H	14	12	10
I	10	8	6
J	11	13	15



(a) Graph after pruning the first critical path $A-B-D-H-J$ found using average communication costs.



(b) Graph after pruning the second critical path $C - G - I$.



(c) Graph after pruning the task nodes of the third critical path F.

Fig. 5. Generation of critical paths and graph pruning.

Table 3
CCPs and their finish times corresponding to each processor.

CCP	P_1	P_2	P_3
$A-B-D$	28	30	38
$C-G$	46	35	37
F	44	42	40
E	33	42	31
H	60	58	52
I	66	64	58
J	86	88	73

The HEFT algorithm has a complexity of $O(m * q)$ with m edges and q processors, that becomes $(n^2 * q)$ for dense graphs with n nodes. The CEFT strategy produces better schedules than HEFT but at the cost of a complexity increased almost by a factor of n over HEFT, for both the sparse and the dense task graphs.

5. Experimental results

The experimentation has been performed using several graph topologies including the Gaussian-elimination, random, LU decomposition, fork-join and out-tree graphs. We compare the performance of CEFT with the HEFT [10], DLS [17] and LMT [31] algorithms used for scheduling heterogeneous systems.

Fig. 6(a) shows the Gaussian elimination graph for input size of 4, whereas Fig. 6(b) shows a task graph for LU decomposition [32] for input matrix of size 3. For the fork-join and out-tree graphs, we use the parameters *depth*,*width* and *degree* to refer respectively to the number of subgraphs from the *start* node to *exit* node, the number of subgraphs at one level of the graph and the maximum number of edges leaving a node in the graph. A fork-join graph with *degree* = 3, *depth* = 1 & *width* = 3, is shown in Fig. 6(c) and an out-tree graph with parameters *degree* = 3 & *depth* = 2, is shown in Fig. 6(d). A random graph with 7 nodes is shown in Fig. 6(e).

The scheduling strategies are evaluated in terms of a metric termed as schedule length ratio (SLR), which is the ratio of the schedule length or *makespan* to the length of critical path of the task graph.

$$\mathrm{SLR} = \frac{makespan}{Length\ of\ Critical\ Path}.$$

As there are heterogeneous processors with diverse processing capabilities, the length of the critical path is measured with the minimum execution latency, i.e. considering the fastest processor for each task. The length of the critical path with minimum execution latencies also represents a minimum threshold for the schedule length of a task graph. The scheduling strategy producing the smaller SLR value therefore implies to produce better schedule.

Due to a large number of experiments, an average SLR value has been computed corresponding to each value of the parameter on x-axis of the graphs depicting performance results. We assume multiple system configurations with 2 to 16 heterogeneous processors. For an accurate evaluation not inclined towards a specific scheduling strategy, the random data has been used for task execution latencies corresponding to each processor. For data transfer, the communication cost is taken into account by using the communication to computation cost ratio (CCR). The communication cost is assigned by using execution latencies and the CCR values as follows:

$$Communication\ Cost = CCR * Computation\ Cost$$

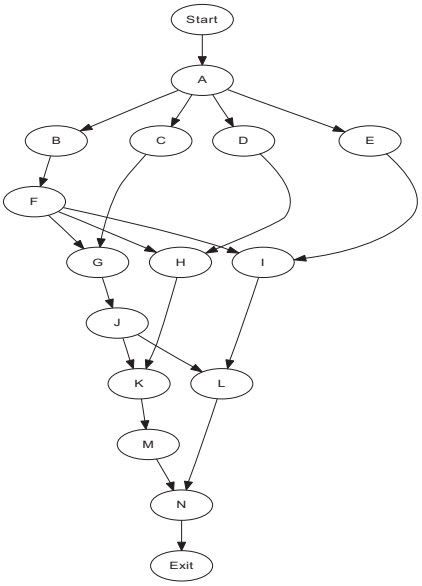
5.1. Fork-join graphs

The *width*, *degree* and *depth* parameters are used to generate the fork-join graphs. The set of possible parameter values for the fork-join graph are shown in Table 4.

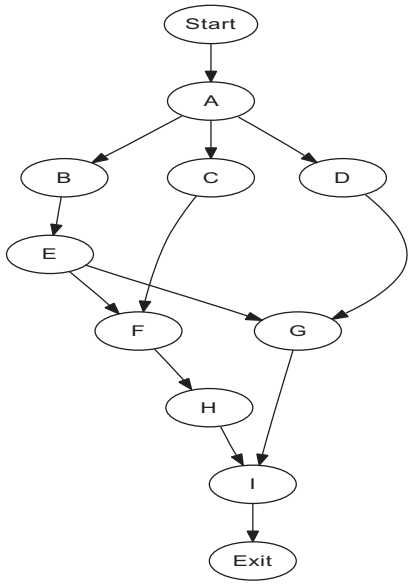
The results of fork-join graphs corresponding to various CCR values are shown in Fig. 7. For different values of CCR, the average SLR produced by CEFT is much smaller than the one produced by other scheduling strategies. For initial values however, the CEFT performance is almost equivalent to HEFT. The main reason for this is the fact that CEFT assigns a single processor to multiple task nodes which deducts the communication cost overhead from the *makespan*. A very small communication cost therefore does not impact to a large extent.

Fig. 8 presents the average SLR values corresponding to various values of the *depth* parameter. The average SLR values as produced by CEFT for all the values of the *depth* parameter are much smaller than those produced by other scheduling strategies.

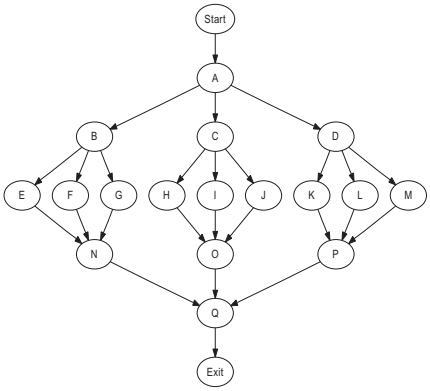
Fig. 9 shows the scheduling results corresponding to different number of processors. In all the cases, the CEFT algorithm results in smaller SLR values. Even with a large number of processors, the CEFT algorithm continues to outperform other scheduling algorithms.



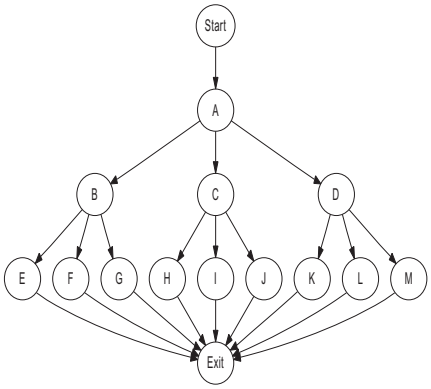
(a) Gaussian-Elimination graph (matrix size=4)



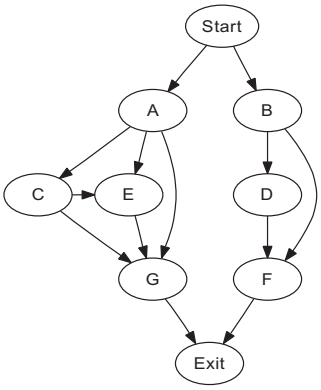
(b) LU decomposition (matrix size=3)



(c) Fork-join graph (*degree*=3, *depth*=1 & *width*=3)



(d) Out-tree graph (*degree*=3 and *depth*=2)



(e) Random graph

Fig. 6. Examples of various graph topologies used for experimentation.

The overall performance of the CEFT algorithm for the fork-join graphs is 12.21%, 21.82% & 39.07% better than HEFT, DLS & LMT respectively.

Table 4
Configuration parameters for the fork-join graph.

Parameter	Possible values
CCR	{0.5,1,2,...,10}
Number of processors	{2,3,...,10,12,14,16}
Depth	{1,2,...,10}
Width	{2,3}
Degree	{2,3,4}

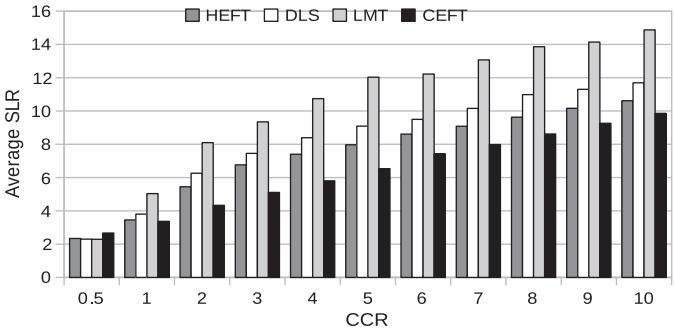


Fig. 7. Results of the fork-join graphs for the given CCR values.

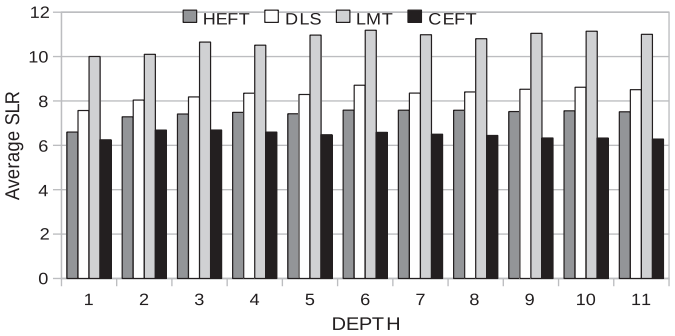


Fig. 8. Results of the fork-join graphs for the given *depth* values.

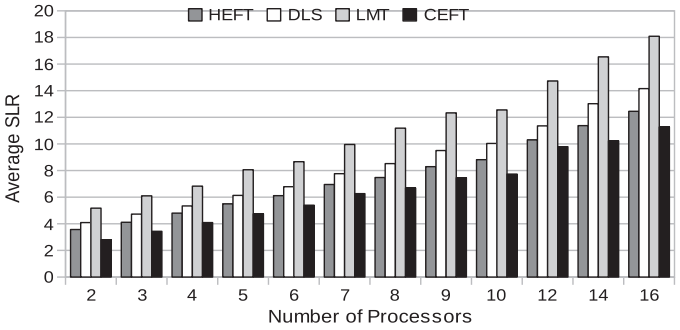


Fig. 9. Results of the fork-join graphs for the given number of processors.

5.2. Random graphs

A mechanism of generating random graph as defined in [33] is effective for various applications which are not inclined toward a specific graph topology. A random graph is generated using the probability for an edge between any two nodes of the graph.

Table 5
Configuration parameters for random graphs.

Parameter	Possible values
CCR	{0.5, 1, 2, ..., 10}
Number of processors	{2, 3, ..., 10, 12, 14, 16}
Number of nodes	{10, 20, ..., 100}
Probability	{0.2, 0.4, 0.5, 0.6, 0.8, 1.0}

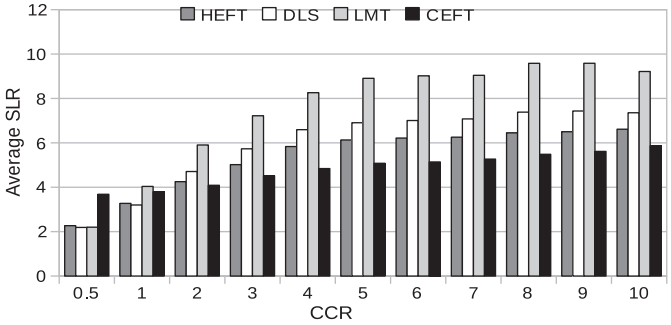


Fig. 10. Results of the random graphs for the given CCR values.

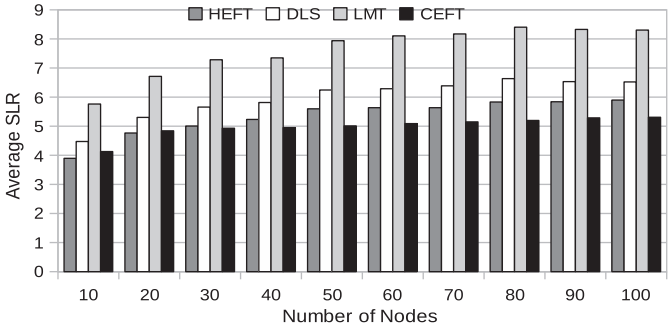


Fig. 11. Results of the random graphs for the given number of nodes.

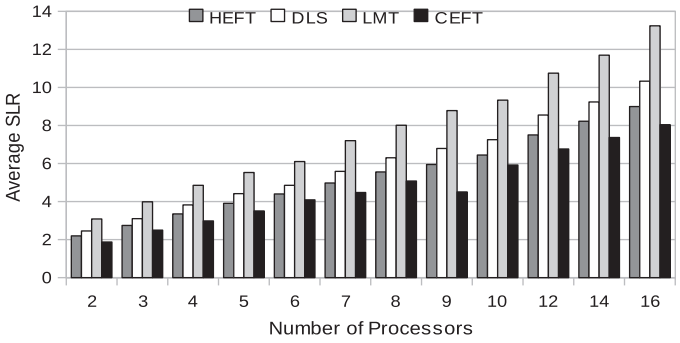


Fig. 12. Results of the random graphs for the given number of processors.

Table 6
Configuration parameters for the Gaussian elimination task graph.

Parameter	Possible values
CCR	{0.5, 1, 2, ..., 10}
Number of processors	{2, 3, ..., 10, 12, 14, 16}
Size	{5, 6, ..., 12, 15, 20}

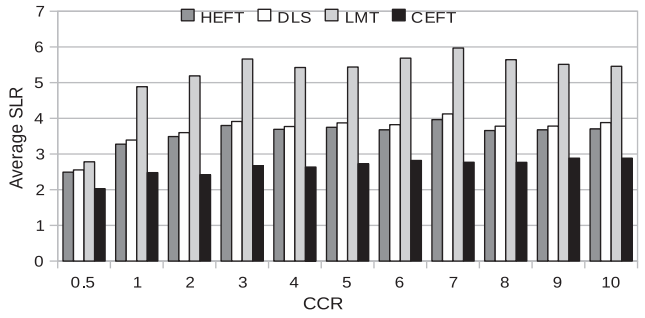


Fig. 13. Results of the Gaussian-elimination graphs for the given CCR values.

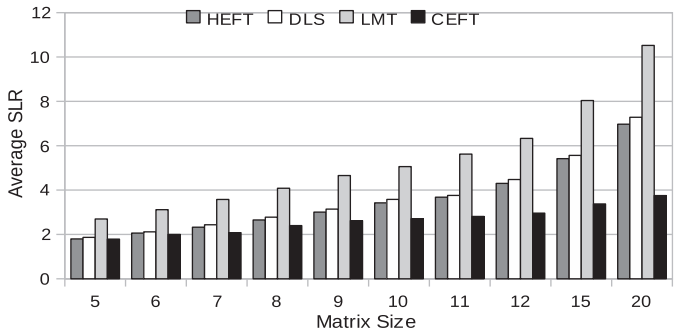


Fig. 14. Results of the Gaussian-elimination graphs for the given sizes of the input matrix.

In our implementation of random graphs, the precedence constraints are respected using topological order, i.e. an edge exists between two nodes m and n only if $m < n$. For probability p, $\lfloor |V| * p \rfloor$ edges are created from every node m to other node $(m + 1/p * i) \bmod |V|$, where $1 \leqslant i \leqslant \lfloor |V| * p \rfloor$, and $|V|$ is the total number of task nodes in the graph.

The experimentation for the random graphs is performed using the parameters and their values as given in Table 5.

As shown in Fig. 10, the CEFT scheduling strategy produces smaller average SLR values as compared to HEFT, DLS & LMT. For smaller CCR values, HEFT, DLS & LMT perform better, however with an increase in the CCR value, the average SLR value produced by CEFT becomes smaller than that produced by HEFT, DLS & LMT.

The impact of the number of nodes for the random graphs is shown in Fig. 11. The CEFT strategy outperforms HEFT, DLS & LMT when the number of nodes becomes large. CEFT produces a very consistent average SLR value in contrast with other strategies for which the SLR value increases with an increase in the number of nodes. For very small sizes however, the HEFT strategy performs marginally better than the CEFT strategy.

For various number of processors, the scheduling results for the random graphs are shown in Fig. 12. The CEFT strategy continues to perform better than HEFT as in previous cases. The difference between the SLR values of CEFT and HEFT is almost constant despite a change in the number of processors. On the contrary, for DLS & LMT, the difference in SLR values increases gradually with the increase in the number of processors.

For the random graphs, the CEFT algorithm, on average performs 8.95%, 18.92% & 36.22% better than HEFT, DLS & LMT respectively.

5.3. Gaussian-elimination graphs

The Gaussian-elimination is a well known method used in mathematics for solving a system of equations. It takes as input a matrix representing equations, and returns the values of the unknowns in the equations.

The task graph depicting the Gaussian-elimination method for a matrix of size n, contains $\frac{(n^2+n-2)}{2}$ nodes. The parameters used for experimentation of Gaussian-elimination graphs are shown in Table 6.

As shown in Fig. 13, the CEFT strategy outperforms the HEFT, DLS & LMT scheduling strategies for all the CCR values. The performance gap between CEFT and HEFT is large for medium CCR values in comparison with small and large CCR values. For DLS and LMT, the performance gap increases with an in increase in the CCR values.

For various sizes of the input matrix, the scheduling results are presented in Fig. 14. The average SLR value for small sizes of the input matrix is almost the same for CEFT and HEFT scheduling strategies. However, with the increasing size of the input matrix, the CEFT performance becomes better. For very large sizes, the CEFT strategy outperforms the HEFT strategy by a big margin. The performance of DLS and LMT scheduling strategies deteriorates for large sizes of the input matrix.

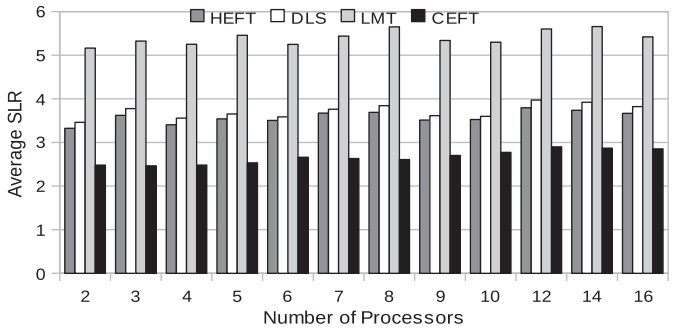


Fig. 15. Results of the Gaussian-elimination graphs for the given number of processors.

Table 7
Configuration parameters for the LU decomposition task graphs.

Parameter	Possible values
CCR	{0.5,1,2,...,10}
Number of processors	{2,3,...,10,12,14,16}
Size	{5,6,...,12,15,20}

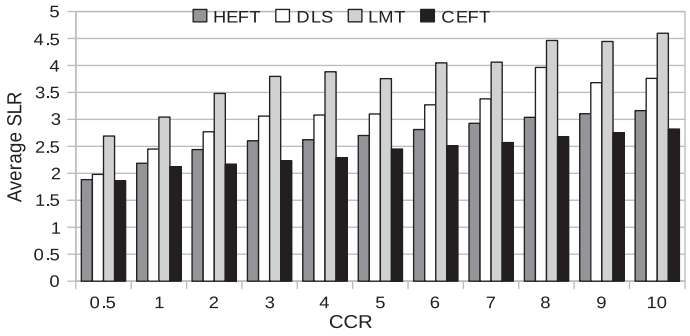


Fig. 16. Results of the LU decomposition graphs for the given CCR values.

Fig. 15 shows the results of the Gaussian-elimination graphs corresponding to different number of processors. For most of the configurations, the CEFT strategy performs consistently better than HEFT, DLS and LMT.

The overall performance of the CEFT algorithm for the Gaussian-elimination graphs is 25.72%, 28.31% & 50.33% better than HEFT, DLS & LMT respectively.

5.4. LU decomposition graphs

The LU decomposition is widely used for solving mathematical equations and works by transforming a matrix into a product of lower and upper triangular matrices.

The parameters used for experimentation of Gaussian-elimination graphs are shown in Table 7.

As shown in Fig. 16, the CEFT strategy performs better than other strategies for all values of the CCR parameter. For small and large CCR values, the CEFT algorithm consistently produces very low SLR values. In contrast, the HEFT, DLS and LMT algorithms produce large SLR values, and are outperformed by CEFT, especially for large CCR values.

For various sizes of the input matrix, the scheduling results for are presented in Fig. 17. The performance gap with other scheduling strategies increases with the increase in the matrix size.

Fig. 18 shows the results of the LU decomposition graphs corresponding to different number of processors. The CEFT strategy performs better for all the cases with different number of processors.

For the LU decomposition graphs, the CEFT algorithm, on average performs 10.62%, 23.65% & 36.89% better than HEFT, DLS & LMT respectively.

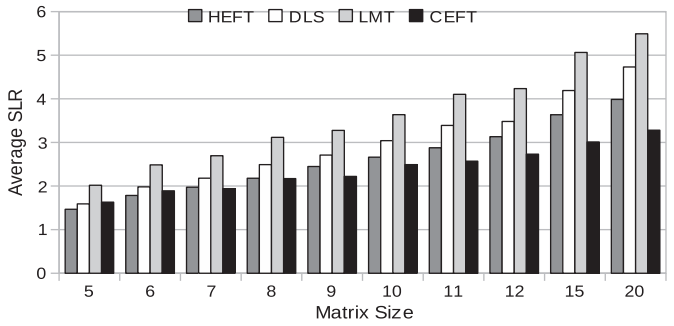


Fig. 17. Results of the LU task graphs for the given sizes of the input matrix.

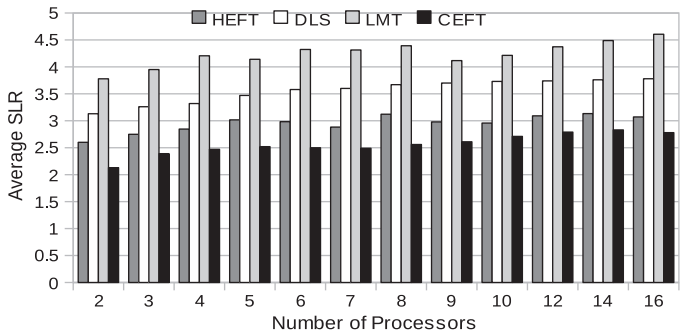


Fig. 18. Results of the LU decomposition task graphs for the given number of processors.

Table 8
Configuration parameters for the out-tree and in-tree graphs.

Parameter	Possible values
CCR	{0.5, 1, 2, . . . , 10}
Number of processors	{2, 3, . . . , 10, 12, 14, 16}
Depth	{1, 2, 3, 4, 5}
Degree	{2}

5.5. Out-tree graphs

The out-tree graphs contain the edges leaving the nodes from top to bottom. With *degree u* and *depth v*, the out-tree graphs would contain $\sum_{i=0}^{v} u^i$ number of nodes. The parameters used for experimentation are shown in Table 8.

For the out-tree graphs, the experimentation results corresponding to various CCR values are shown in Fig. 19. The CEFT algorithm performs better than the HEFT algorithm with a small difference in the SLR values. The reason for producing a very close SLR value is that the CEFT strategy performs allocation of a processor to more than one task along a single path. Since there is a large number of paths, the HEFT strategy is also able to produce competitive *makespan*. The CEFT and HEFT strategies perform better than the DLS and LMT scheduling strategies, though the difference is not very large as in previous graphs.

Corresponding to the *depth* parameter and the number of processors, the scheduling results are shown in Figs. 20 and 21 respectively.

For initial *depth* values, the CEFT strategy performs almost equivalent to the HEFT strategy. For larger *depth* values however, the CEFT performance becomes better as the schedule length produced by it is much smaller than that produced by the HEFT strategy. For the DLS and LMT algorithms, the performance gap increases with the increase in the depth.

The increase in the number of processors does not impact to a large extent the average SLR value produced for the scheduling strategies. Although the CEFT algorithm performs better than other scheduling strategies, the performance of CEFT, HEFT and DLS is very close.

The overall performance of the CEFT algorithm for the out-tree graphs is 6.83%, 11.89% & 30.7% better than HEFT, DLS & LMT respectively.

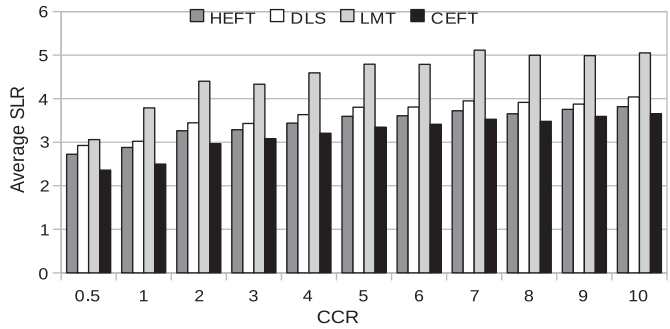


Fig. 19. Results of the out-tree graphs for the given CCR values.

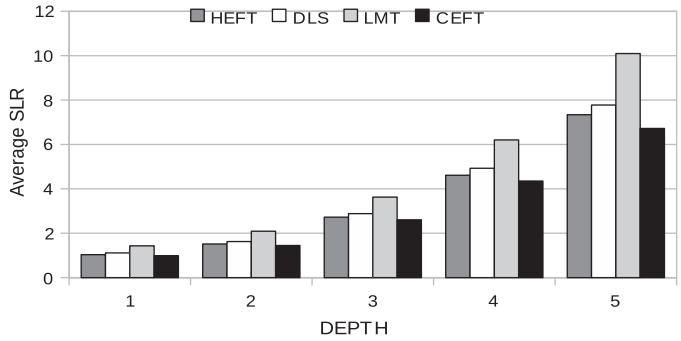


Fig. 20. Results of the out-tree graphs for the given *depth* values.

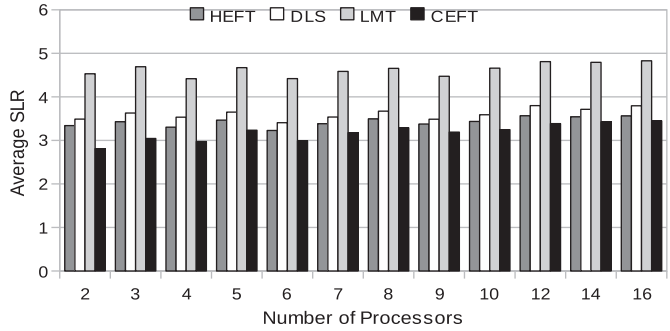


Fig. 21. Results of the out-tree graphs for the given number of processors.

5.6. Performance analysis

A comparison summary for the experiments performed on the fork-join, random, Gaussian-elimination, LU decomposition and out-tree graphs is provided in Table 9. The combination of parameters used for all the topologies produces 57,420 graphs.

Although the CEFT algorithm has a high complexity in comparison with HEFT, it performs better than HEFT for 41,203 (71.76%) cases. The HEFT algorithm, in contrast, performs better than CEFT for 15,359 (26.75%) cases. Similarly, in comparison with the DLS and LMT algorithms, the CEFT algorithm performs better for 45,185 (78.69%) and 50,937 (88.71%) cases respectively.

In comparison with all the algorithms (HEFT, DLS and LMT), the CEFT algorithm performs better in 79.72% cases and performs worse in 19.13% cases.

5.7. CEFT in comparison with optimal scheduling

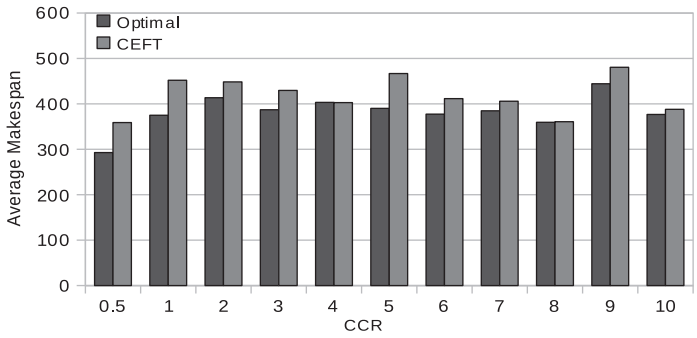
For small graphs, the optimal results may be obtained by testing all the combinations of processor assignments for all the tasks. We compared thus obtained optimal scheduling results with the results obtained by CEFT scheduling. The experimentation has been performed with fork-join graphs using the configuration parameters given in Table 10.

Table 9
The number of cases CEFT performed better, worse and equal in comparison with other scheduling strategies.

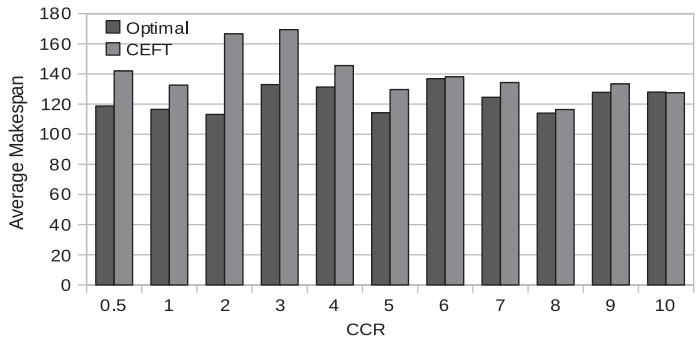
	HEFT	DLS	LMT	All strategies (%)
CEFT				
Better	41,203	45,185	50,937	79.72
Worse	15,359	11,468	6131	19.13
Equal	858	767	352	1.15

Table 10
Configuration parameters for comparison of CEFT with the optimal results.

Parameter	Possible values
CCR	{0.5, 1, 2, ..., 10}
Number of processors	{2, 3, 4}
Degree	{2}



(a) *Depth*=1



(b) *Depth*=2

Fig. 22. Comparison of CEFT with optimal schedules for the fork-join graphs.

The optimal results and the CEFT scheduling results for the fork-join graph with *depth* = 1 are shown in Fig. 22(a). The CEFT strategy produces near optimal results achieving 91.29% of the optimal *makespan*. In general, the CEFT strategy produces better results for higher CCR values.

The scheduling results with *depth* = 2 are shown in Fig. 22(b). In some cases with the large CCR values, the CEFT scheduling produces optimal results. However, with small CCR values, the *makespan* is slightly larger than the optimal results. Overall, with *depth* = 2, the CEFT strategy is able to achieve 88.44% of the optimal results.

6. Related work

Most of the scheduling algorithms are categorized into list scheduling, cluster scheduling, duplication based scheduling and random search based scheduling. These strategies may be used to generate schedules at compile time, i.e. before execution of the application. This section summarizes the approach adopted by different scheduling strategies in comparison with the CEFT scheduling approach.

6.1. HEFT

The heterogeneous earliest finish time (HEFT) scheduling strategy [10] incorporates the concept of assigning priorities to the tasks. Subsequently, the tasks are assigned to the processors using the criteria based on the finish time of the tasks. The HEFT algorithm has a complexity of $O(n^2 * q)$, for task graph with n nodes and q processors.

The HEFT algorithm uses a recursive approach in the bottom-to-top direction to determine the ranks of the nodes which are based on execution costs. The tasks are processed in order of their ranks. The rank based processing is a notion of the critical paths since a task along the critical path possesses a higher rank. This approach is to some extent similar to that of CEFT, however, the processor assignment of HEFT uses a greedy approach and takes into account only one node at a time.

The greedy approach of HEFT makes it work with a low complexity. In contrast, the CEFT approach takes into account the collection of nodes at one instance, and also attempts to eliminate the communication cost. The small increase in the complexity of CEFT due to consideration of a broader view of the task graph, is amortized by the fact that CEFT is able to outperform HEFT for a large number of cases.

6.2. DCP

The dynamic critical path (DCP) scheduling [11] initially finds out a critical path of the task graph and then schedules a node dynamically. A final schedule is not produced until all the nodes have been processed. The DCP algorithm works with a complexity of $O(n^3)$ for n nodes in the task graph.

In contrast to the CEFT scheduling, the DCP based scheduling uses dynamic critical paths and attempts to minimize the schedule length at each step. It takes into account the start times of the remaining nodes for processor assignment. Furthermore, the start times for nodes in the task graph are dynamic thereby producing a schedule that changes dynamically at each step.

The DCP uses the absolute earliest start time (AEST) and the Absolute latest start time (ALST) that represent respectively the possible execution of a task at the earliest or the latest time. These values are computed through breadth-first traversal of the task graph. A critical child node having the smallest value of $ALST - AEST$ is then determined. Instead of considering children along a path, the DCP takes into account only the critical child node for the processor assignment. Subsequently, the processor producing the minimum value of the sum of AESTs of the current node and its critical child node is assigned to the current node.

6.3. DSC

Dominant sequence clustering (DSC) algorithm [5] schedules tasks for unbounded number of processors by creating clusters of tasks. In contrast to CEFT, it works for homogeneous processing systems and also requires the clusters to be merged so as to adapt schedule to the existing number of processors. The complexity of the DSC algorithm is $O((n + m) * log(n))$.

In contrast to the CEFT approach, the DSC approach is based on the notion of a dominant sequence (DS) that represents the critical path of the scheduled graph. The DSC algorithm initially assigns priorities computed as the sum of *pt-level* and *b-level* values. A *pt-level* value represents the length of the longest path from *entry* node to any immediate predecessor of the current node, whereas, the *b-level* value is the length of the longest path from the current node to the *exit* node. The node with the highest priority becomes a DS node. Subsequently, the edge-zeroing takes place in order to reduce the communication cost.

6.4. MCP

The modified critical path (MCP) algorithm [15] considers only one critical path of the task graph. It performs As Late As Possible (ALAP) binding for each node and computes the ALAP time which represents the latest possible start time for execution of a task. It is found by traversing the task graph backwards. The overall scheduling algorithm has the complexity of $O(n^2 * log n)$.

The MCP algorithm maintains a list of descendant tasks and their ALAP time is created and arranged in increasing order. The node is then scheduled and assigned to the processor resulting in earliest execution time.

The CEFT approach differs from the MCP approach in that the CEFT makes use of several critical paths and then assigns the processor to ready nodes in these critical paths.

6.5. LMT

The LMT algorithm [31] works by performing level sorting. The level sorting arranges the tasks in an order so that the tasks at one level of the task graph, are independent of each other.

The LMT algorithm allocates the processor using the minimum value of the finish time of a level. At one level, the tasks with low execution costs are merged to conform to the number of processors. The processor assignment then uses the criteria based on minimizing the sum of execution costs and the communication costs.

The CEFT approach, in contrast with LMT, makes use of a CCP instead of performing level sorting. It produces better results as it considers a large number of nodes to be assigned to a single processor.

6.6. Dynamic level scheduling (DLS)

The DLS scheduling algorithm [17] uses the current states of the communication and computation resources. It incorporates calculation of static and dynamic levels. A static level of a node is the longest path in terms of execution costs from that node to the exit node of the graph. Given the static level of a node, S_{V_i}, the dynamic level of that node with a processor P_j is computed as the difference between static level and the earliest execution start time of that node on a processor.

$$D(V_i, P_j) = S_{V_i} - EST(V_i, P_j)$$

After having calculated the dynamic levels for all the nodes in the graph, scheduling takes place with dynamic level being used as the priority for selection of a node. The dynamic level values therefore change as the scheduling proceeds, an important characteristic that makes it different from single step rank computation strategies. In contrast to DLS, the CEFT approach uses static critical path and also reduces the communication cost through CCPs, while working with low complexity.

For heterogeneous processors, the DLS variant makes use of the average value of execution times, while calculating the static levels. In addition, the average execution cost of one descendent (to which the largest amount of data is communicated) is also considered while calculating the dynamic levels. If all the nodes are considered at an instance, the algorithm works with the complexity of $O(n^3 + n^2 * q * f(q))$, where n is the number of nodes, q is the number of processors and $f(q)$ is the complexity of calculating the *execution start time* of the node.

6.7. CPFD

The critical path fast duplication (CPFD) algorithm [20] also works in two phases. It first assigns priority to each node depending upon the category a node lies into. Subsequently, the task duplication is applied and the task is scheduled to the processor that produces the minimum value of the *earliest start time*. In contrast to CEFT, the CPFD is a duplication based algorithm that uses a greedy approach to find a critical path.

The nodes in the DAG are categorized into *in-branch node* (*IBN*), *critical path node* (*CPN*) and *out-branch node* (*OBN*). A node existing on the critical path is termed as CPN. A node is categorized IBN if it does not exist on critical path and may be connected to a CPN through any path. All other nodes are OBN nodes. The nodes are processed in the order given as CPN, OBN and IBN. A CPN-dominant list is constructed, and subsequently, all the OBNs are appended to the list. The task duplication is performed by finding the *very important parent* (*VIP*) of a node which represents the parent node due to which the communication of data completes at the latest time. The start time of a node is minimized by recursive duplication of the ancestor nodes starting with the VIP node.

For a bounded number of processors, an economical version of the algorithm termed as ECPFD is proposed. It makes use of the same duplication mechanism as described for CPFD, but differs in that the duplication for OBNs is delayed and the processor resulting in the minimum schedule length is assigned to the OBN.

The time complexities of the CPFD and ECPFD algorithms are $O(m * n^2)$ and $O(q * m * n)$ respectively.

7. Conclusion

This paper contributes towards a novel scheduling approach called CEFT that works for heterogeneous computing systems. It is based on the concept of *constrained critical paths (CCPs)*, which are a notion of ready queues at an instance of scheduling. All the tasks in a CCP are assigned the same processor thereby reducing the communication cost. The consideration of multiple nodes at a time makes our approach perform better by producing near optimal schedules with a very low complexity.

We have performed experimentation with a large number of task graphs. The results show that the CEFT in most of the cases outperforms the well-known HEFT, DLS and LMT strategies which also schedule tasks for heterogeneous systems.

As future work, we intend to adapt this strategy for various groups of machines. It would require multiple CCPs being considered for assignment to a subset of machines instead of taking into account all the existing machines.

References

[1] J.L.R.L. Graham, E.L. Lawler, A.R. Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, Annals of Discrete Mathematics (5) (1979) 287–326.
[2] C. Papadimitriou, M. Yannakakis, Towards an architecture-independent analysis of parallel algorithms, in: STOC'88: Proceedings of the Twentieth Annual ACM Symposium on Theory of computing, ACM, New York, NY, USA, 1988, pp. 510–513. <http://doi.acm.org/10.1145/62212.62262>.
[3] V. Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessors, MIT Press, Cambridge, MA, USA, 1989.
[4] P. ChrTtienne, Task scheduling with interprocessor communication delays, European Journal of Operational Research 57 (3) (1992) 348–354, doi:10.1016/0377-2217(92)90346-B. <http://www.sciencedirect.com/science/article/B6VCT-48NBG6V-1SV/2/95468ef9c192edb0a47f2f29707afdb0>.
[5] T. Yang, A. Gerasoulis, Dsc: scheduling parallel tasks on an unbounded number of processors, IEEE Transactions on Parallel and Distributed Systems 5 (1994) 951–967, doi:10.1109/71.308533.

[6] M. Veldhorst, A linear time algorithm to schedule trees with communication delays optimally on two machines, Technical Report RUU-CS-93-04, Utrecht University, Netherlands, 1993.
[7] K. Takamizawa, T. Nishizeki, N. Saito, Linear-time computability of combinatorial problems on series–parallel graphs, Journal of the ACM 29 (3) (1982) 623–641, doi:10.1145/322326.322328.
[8] H.M. Abdel-Wahab, T. Kameda, Scheduling to minimize maximum cumulative cost subject to series–parallel precedence constraints, Operations Research (26) (1978) 141–158.
[9] T. Lewis, H. ElRewini, Introduction to Parallel Computing, Prentice-Hall, New-York, USA, 1992.
[10] H. Topcuouglu, S. Hariri, M.-y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems 13 (2002) 260–274, doi:10.1109/71.993206.
[11] Y. Kwong Kwok, 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.
[12] F. Suter, F. Desprez, H. Casanova, From heterogeneous task scheduling to heterogeneous mixed parallel scheduling, in: Euro-Par, Vivien, 2004, pp. 230–237.
[13] C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, F.D. Anger, Multiprocessor scheduling with interprocessor communication delays, Operations Research Letters 7 (3) (1988) 141–147, doi:10.1016/0167-6377(88)90080-6. <http://www.sciencedirect.com/science/article/B6V8M-48MPVK2-1M/2/6ce4b43d61181989ed65585f59e9f408>.
[14] M. Maheswaran, H.J. Siegel, A dynamic matching and scheduling algorithm for heterogeneous computing systems, in: Proceedings of the Seventh Heterogeneous Computing Workshop, HCW'98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 57–69.
[15] M. You Wu, D.D. Gajski, Hypertool: a programming aid for message-passing systems, IEEE Transactions on Parallel and Distributed Systems 1 (1990) 330–343.
[16] H. El-Rewini, T.G. Lewis, Scheduling parallel program tasks onto arbitrary target machines, Journal of Parallel and Distributed Computing 9 (2) (1990) 138–153, doi:10.1016/0743-7315(90)90042-N.
[17] G.C. Sih, E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Transactions on Parallel and Distributed Systems 4 (2) (1993) 175–187.
[18] B. Kruatrachue, T. Lewis, Grain size determination for parallel processing, IEEE Software 5 (1988) 23–32, doi:10.1109/52.1991.
[19] M.A. Al-Mouhamed, Lower bound on the number of processors and time for scheduling precedence graphs with communication costs, IEEE Transactions on Software Engineering 16 (1990) 1390–1401, doi:10.1109/32.62447.
[20] I. Ahmad, Y.-K. Kwok, On exploiting task duplication in parallel program scheduling, IEEE Transactions on Parallel and Distributed Systems 9 (9) (1998) 872–892, doi:10.1109/71.722221.
[21] D. Bozdağ, F. Özgüner, U.V. Catalyurek, Compaction of schedules and a two-stage approach for duplication-based dag scheduling, IEEE Transactions on Parallel and Distributed Systems 20 (2009) 857–871, doi:10.1109/TPDS.2008.260.
[22] G.-L. Park, B. Shirazi, J. Marquis, Dfrn: a new approach for duplication based scheduling for distributed memory multiprocessor systems, in: Proceedings of the 11th International Symposium on Parallel Processing, IPPS'97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 157–166.
[23] S. Darbha, D. Agrawal, Sdbs: a task duplication based optimal scheduling algorithm, in: Proceedings of the Scalable High-Performance Computing Conference, 1994, 1994, pp. 756 –763, doi:10.1109/SHPCC.1994.296717.
[24] A. Dogan, F. Özgüner, Ldbs: a duplication based scheduling algorithm for heterogeneous computing systems, in: Proceedings of the 2002 International Conference on Parallel Processing, ICPP'02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 352–359.
[25] Y.-K. Kwok, I. Ahmad, Exploiting duplication to minimize the execution times of parallel programs on message-passing systems, in: Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing, 1994 1994, pp. 426 –433. doi:10.1109/SPDP.1994.346138.
[26] H.B. Chen, B. Shirazi, K. Kavi, A.R. Hurson, Static scheduling using linear clustering with task duplication, in: Proceedings of the ISCA International Conference on Parallel and Distributed Computing and Systems, 1993, pp. 285–290.
[27] J. chiou Liou, M.A. Palis, An efficient task clustering heuristic for scheduling dags on multiprocessors, in: Multiprocessors, Workshop on Resource Management, Symposium of Parallel and Distributed Processing, 1996, pp. 152–156.
[28] B. Cirou, E. Jeannot, Triplet: a clustering scheduling algorithm for heterogeneous systems, in: IEEE Symposium on Reliable Distributed Systems, IEEE, 2001, pp. 231–236.
[29] S.J. Kim, J.C. Brown, A general approach to mapping of parallel computations upon multiprocessor architectures, in: Proceedings of the International Conference on Parallel Processing, 1988, pp. 1–8.
[30] A. Gerasoulis, T. Yang, A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors, Journal of Parallel and Distributed Computing 16 (4) (1992) 276–291, doi:10.1016/0743-7315(92)90012-C. <http://www.sciencedirect.com/science/article/B6WKJ-4BRJJ23-2S/2/fc1925064e66c33d6dd85b414435a3af>.
[31] M.A. Iverson, F. zgnner, O. Gregory, G.J. Follen, Parallelizing existing applications in a distributed heterogeneous environment, in: 4th Heterogeneous Computing Workshop (HCW'95), 1995, pp. 93–100.
[32] R.E. Lord, J.S. Kowalik, S.P. Kumar, Solving linear algebraic equations on an mimd computer, Journal of the ACM 30 (1983) 103–117, doi:10.1145/322358.322366. <http://doi.acm.org/10.1145/322358.322366>.
[33] V.A.F. Almeida, I.M.M. Vasconcelos, J.N.C. Árabe, D.A. Menascé, Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems, in: Supercomputing'92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, USA, 1992, pp. 683–691.