Computing (2019) 101:1015–1039
https://doi.org/10.1007/s00607-018-0636-3

ECP: a novel clustering-based technique to schedule precedence constrained tasks on multiprocessor computing systems

Ashish Kumar Maurya[1] · **Anil Kumar Tripathi**[1]

Received: 8 November 2017 / Accepted: 15 June 2018 / Published online: 21 June 2018

Abstract
Efficient scheduling is critical for achieving improved performance of distributed applications where an application is to be considered as a group of interrelated tasks and represented by a task graph. In this work, we present a clustering-based scheduling algorithm called effective critical path (ECP) to schedule precedence constrained tasks on multiprocessor computing systems. The main aim of the algorithm is to minimize the schedule length of the given application. It uses the concept of edge zeroing on the critical path of the task graph for clustering the tasks of an application. An experimental analysis is performed using random task graphs and the task graphs derived from the real-world applications such as Gaussian Elimination, fast Fourier transform and systolic array. The results illustrate that the ECP algorithm gives better performance than the previous algorithms, considered herein, in terms of the average normalized schedule length and average speedup.

Keywords DAG scheduling · Clustering · Task graphs · Multiprocessor computing systems · Static scheduling

Mathematics Subject Classification 68Q85 · 68W15 · 68W40

1 Introduction

Over the years, the high-speed multiprocessor computing systems are developed that facilitate computation of large commercial, scientific, and mathematical applications

✉ Ashish Kumar Maurya
akmaurya.rs.cse14@iitbhu.ac.in

Anil Kumar Tripathi
aktripathi.cse@iitbhu.ac.in

1 Department of Computer Science and Engineering, Indian Institute of Technology (BHU), Varanasi 221005, India

by dividing the applications into various tasks and executing them on different processing units. The scheduling of tasks in such systems is a significant problem that have been widely studied by the researchers. A multiprocessor computing system refers to a distributed suite of several processing units interconnected with each other via fast communication links. It is exploited to compute big parallel and distributed applications. Such an application is usually consists of interrelated tasks. Possible interrelations amongst the tasks are depicted by a task graph that is modeled by a Directed Acyclic Graph (DAG) [22]. The competence of computing parallel and distributed applications on multiprocessors is extremely dependent on their features, for example, size of data to transfer between tasks, computation time of tasks, precedence constraints among tasks, etc. and platform features such as number of processor, execution power of the processors, link capacity, etc. For the effective exploitation of a multiprocessor computing system, each task of the application is allocated to a best-suited processing unit such that dependency constraints among tasks are satisfied, and minimum schedule length is achieved. The minimization of makespan helps to enhance the processor utilization and system throughput. The problem of obtaining schedules with minimum length has been shown to be NP-complete [31,32]. In an attempt to give polynomial time solutions, the known algorithms end up providing at times suboptimal solutions.

To solve the problem of task scheduling, the known algorithms are grouped into static and dynamic scheduling algorithms. The static scheduling algorithms use all information regarding tasks in advance, such as computation time of tasks on processing units and communication time between tasks, etc. before starting the execution of the application while dynamic scheduling algorithms utilize the required information for scheduling decisions at run-time. There are many methods to assess such information [7]. This paper focuses on static scheduling algorithms as it generates optimal schedules without considering run-time overheads. Static scheduling algorithms may be grouped into guided heuristic-based algorithms or random search-based algorithms. The second group of the static algorithm takes more time to find the required solution even though the makespan is minimized at the cost of spare time. It gives different makespan for the same problem size and the same inputs based on the various scheduling techniques are used to allocate parallel tasks onto the suitable processors. Alternatively, the first group of static algorithm focuses on generating schedules with the minimum scheduling overhead, but the obtained makespan is not necessarily the shortest. Hence, both heuristic-based and guided random search-based algorithms can be used as per circumstances. The heuristic-based algorithms are further divided into three subgroups that are list scheduling, duplication-based scheduling, and clustering-based scheduling. The list scheduling algorithms [1,7,10,11,15,20,33,35,36], typically schedule tasks in 2 phases: the primary phase is task prioritization in which tasks are assigned priorities based on their associated execution and communication times; processor selection being the second phase in which suitable processors are selected and task assignment to processors is done. The duplication-based algorithms [2,3,12,13,19,31,34] attempt to minimize the communication time between tasks through duplication of tasks onto different processors. The clustering-based algorithms [16,18,20,23,27,32,36,38] are mainly applicable for an unbounded number of processors and generate schedules by grouping heavily communicating tasks of a given application into clusters and assigning these clusters onto appropriate processors so that the schedule length of an application can be minimized.

Clustering-based algorithms are more advantageous as they embrace a more global view, whereas list scheduling algorithms, in contrast, tend only to make local optimization decisions [4]. The aim, in this work, is to look for some fresh approach to develop a clustering-based technique for scheduling precedence constrained tasks of an application in multiprocessor computing systems. We propose an algorithm called Effective Critical Path (ECP) that schedules tasks of the application onto an unbounded number of fully connected processors. The time complexity of the ECP algorithm is $O(|V|^2(|V|+|E|))$, where $|E|$ is the number of edges and $|V|$ is the number of tasks in the task graph. The contributions here may be indicated as follows:

- Our idea of applying edge zeroing concept on the critical path leads to reduction in the communication time among the tasks of a given task graph and finally provides a meaningful clustering that can consequently help in improving the execution characteristics.
- Makespan being a significant measure for scheduling applications in a multiprocessor system, the proposed algorithm provides an alternative approach towards its minimization.
- This work demonstrates inprovement in performance of the proposed ECP over four well-known algorithms such as EZ [32], LC [18], CPPS [27], and LOCAL [26].
- The results of the simulation carried out herein show distinctively better normalized schedule length and speedup for randomly generated benchmark task graphs [8] and task graphs generated from real-world applications such as Gaussian Elimination, fast Fourier transform (FFT) and systolic array.

The rest of the article is organized as follows. Section 2 formalizes the task scheduling problem. Section 3 describes the related work. Section 4 discusses the proposed ECP algorithm with an execution trace of the algorithm using an example. The complexity analysis for the proposed algorithm is also presented in this section. The experimental results for random task graphs and task graphs generated from real-world applications are presented in Sect. 5. A summary of the findings and future work are given in Sect. 6.

2 Task scheduling problem

This work focuses on the static scheduling of a single application in a multiprocessor computing system. Assuming that the multiprocessor computing system is composed of an unbounded number of homogeneous processors that are fully interconnected and there is non-zero communication overhead between any two processors. The execution of tasks on processors and communication between tasks can be performed simultaneously, and task execution is assumed to be non-preemptive. Also assuming that no duplication of the task is allowed and all tasks in a cluster will have to execute on the same processor. Table 1 gives some symbols and their meanings that will be used in the subsequent discussion.

Table 1 Symbols and their meaning

Symbols	Meaning
$\|V\|$	Number of nodes in a task graph
$\|E\|$	Number of edges in a task graph
T_i	ith task in a task graph
$e_{i,j}$	A directed edge with precedence constraint from task T_i to T_j
T_{entry}	Task without any predecessor
T_{exit}	Task without any successor
$pred(T_i)$	Set of immediate predecessors of task T_i
$succ(T_i)$	Set of immediate successors of task T_i
$AFT(T_i)$	Actual finish time of task T_i
$ET(T_i)$	Execution time of task T_i
$CT(e_{i,j})$	Communication time from task T_i to T_j
$BL(T_i)$	Bottom level of task T_i
CP	Critical path of the task graph
ECP	Effective critical path of the task graph
$EST(T_i)$	Earliest start time of task T_i
$EFT(T_i)$	Earliest finish time of task T_i
CCR	Communication-to-computation ratio

An application is, a set of tasks, represented as a task graph which is a Directed Acyclic Graph (DAG), $G = (V, E)$, where V represents the set of nodes, and each node denotes a task, and E is the set of communication edges between tasks. In the task graph, each task $T_i \in V$ is associated with its execution time or computation time, denoted by $ET(T_i)$ and each edge $e_{i,j} \in E$ from $T_i \in V$ to $T_j \in V$ is associated with its communication time, denoted by $CT(e_{i,j})$. Each edge $e_{i,j}$ denotes the dependency constraint between tasks T_i and T_j such that task T_j cannot begin its execution until task T_i completes its execution [24]. When two tasks belong to the same cluster, the communication time between them becomes zero. An instance of a task graph containing 15 tasks for a homogeneous system is shown in Fig. 1 and is taken from literature [16]. In Fig. 1, a node denotes a task, and the value inside the node represents the execution time of that task. The value written with a directed edge between tasks T_i and T_j denotes the communication time from T_i to T_j. For example, the execution time of T_7 is 4, and the communication time from T_7 to T_{11} is 5. T_1 is immediate predecessor of T_3, T_4, and T_5. T_{11} is immediate successor of T_7 and T_{10}. T_0 is an entry task, and T_4, T_8, and T_{14} are exit tasks.

The following known definitions ascribe some common properties for task scheduling, and these definitions will be useful in the forthcoming sections.

Definition 1 ($pred(T_i)$) It represents the set of immediate predecessors of task T_i in a given task graph. A task is called an entry task, T_{entry}, if it doesn′t have any predecessor task. If a task graph consists of many entry tasks, a dummy entry task with zero execution time and edges with zero communication time can be added to the task graph.

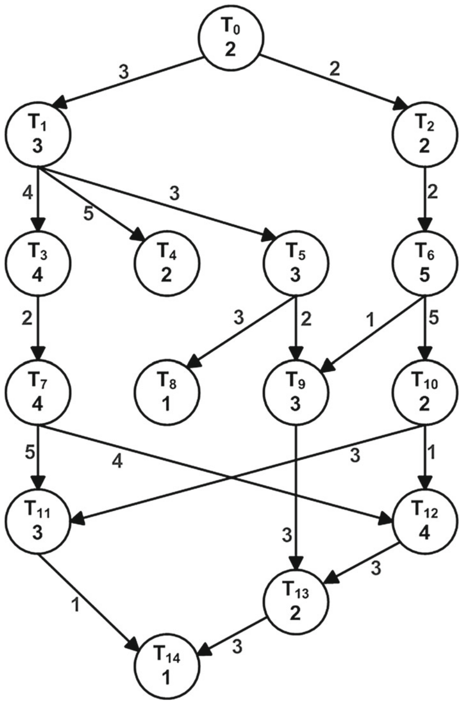

Fig. 1 Task graph consisting of fifteen tasks with their execution times and edges labelled with communication times

Definition 2 $(succ(T_i))$ It represents the set of immediate successors of task T_i in a given task graph. A task is called an exit task, T_{exit}, if it doesn′t have any successor task. Similar to the entry task, if a task graph consists of many exit tasks, a dummy exit task with zero execution time and edges with zero communication time from current many exit tasks to this dummy task can be added to the task graph.

Definition 3 $(AFT(T_i))$ It is the Actual Finish Time of a task T_i and is defined as the point of time where the execution of task T_i is completed by some assigned processor. This time is different from the estimated finishing time of tasks.

Definition 4 (*makespan*) makespan or schedule length represents the completion time of the exit task in the scheduled task graph and is defined by

$$makespan = max\{AFT(T_{exit})\} \tag{1}$$

where $AFT(T_{exit})$ denotes the Actual Finish Time of the exit task. If a task graph has many exit tasks and no redundant task is added, the makespan is computed as the maximum AFT of all exit tasks. Schedule length, makespan and parallel execution time are used interchangeably in this work.

Definition 5 (*Critical path*) Critical Path (CP) of a task graph is defined as the longest path from the entry task to the exit task in the graph where the length of a path in the task graph is the sum of the execution times of its tasks and communication times of its edges.

Definition 6 ($BL(T_i)$) It denotes the Bottom Level of a task T_i and is defined as the length of the longest path from the task T_i to the exit task in the task graph. It is obtained as given in Eq. (2):

$$BL(T_i) = ET(T_i) + \max_{T_j \in succ(T_i)} \left\{ BL(T_j) + CT(e_{i,j}) \right\} \tag{2}$$

$CT(e_{i,j})$ becomes zero, when T_i and T_j are in a same cluster. For the exit task,

$$BL(T_i) = ET(T_i). \tag{3}$$

Definition 7 ($EST(T_i)$) It denotes the Earliest Start Time of a task T_i and is defined as the time at which T_i can start its execution, after all its predecessor tasks are finished their execution and associated dependencies have been transferred to T_i. It is obtained as given in Eq. (4):

$$EST(T_i) = \max_{T_j \in pred(T_i)} \left\{ EST(T_j) + ET(T_j) + CT(e_{j,i}) \right\} \tag{4}$$

$CT(e_{j,i})$ becomes zero, when T_j and T_i are in a same cluster. For the entry task,

$$EST(T_{entry}) = 0. \tag{5}$$

Definition 8 ($EFT(T_i)$) It denotes the Earliest Finish Time of a task T_i and is defined as the time where the computation of task T_i is completed. It is obtained as given in Eq. (6):

$$EFT(T_i) = EST(T_i) + ET(T_i) \tag{6}$$

We assume that all processors are homogeneous and initially available. Thus, the AFT and EFT of a task will be equal.

The aim of the scheduling algorithm is to schedule tasks of a given task graph onto processing units such that the precedence constraints are satisfied and makespan is minimized. When all tasks of a task graph are scheduled, the makespan will be the maximum of the AFT of all the exit tasks, as expressed by Eq. (1).

3 Related work

Many algorithms for task scheduling for multiprocessor computing systems have been proposed. In this section, we give a brief explanation of some existing clustering based task scheduling algorithms. One of the most famous clustering based algorithms is Sarkar′s algorithm [32] that utilizes the concept of edge zeroing for clustering the tasks. In this concept, the tasks which involve large communications are grouped and executed on the same processor to minimize makespan of the task graph. This algorithm initially kept each task in a separate cluster and sorted the edges by their weights in non-increasing order, then scrutinizes edges one-by-one and zeroes them if the makespan does not increase. After edge zeroing concept, Kim and Browne [18] came with the idea of linear clustering that finds the critical path in the task graph and merges all the nodes of a critical path in the same cluster.

Yang and Gerasoulis [37,38] presented an algorithm, called Dominant Sequence Clustering (DSC) that groups the nodes of the task graph by their priorities which is determined by using the idea of the bottom level and top level. The top level of a node is the length of the longest path from an entry node up to that node (excluding the weight of the node) in the task graph. The greedy version of DSC algorithm is given by Dikaiakos et al. [9].

Wu and Gajski [36] proposed MCP (Modified Critical Path) algorithm for a limited number of processors that can also perform as an edge-zeroing heuristic when used for an unlimited number of processors. It exploits the idea of As-Late-As-Possible (ALAP) binding which assigns latest possible execution time to the task and determines the ALAP time for each task by moving downward through the DAG. The MCP maintains an increasing lexicographical sorted list of tasks according to their ALAP time and schedules the first task from the list at each step. Like MCP, the DCP (Dynamic Critical Path) algorithm [20] is given for a bounded number of processors and becomes an edge-zeroing heuristic when it is utilized for an unbounded number of processors. At each iteration, it determines the difference between the absolute EST and absolute latest start time for each task of the CP and schedules the task which has the smallest difference among all tasks. The authors called the intermediate CP as the dynamic CP to differentiate it from the initial CP of the DAG.

Kadamuddi and Tsai [16] presented a Clustering Algorithm for Synchronous Communication (CASC) that efficiently parallelizes the input application on multiprocessors. It includes 4 steps, that is, Initialize, ForwardMerge, BackwardMerge, and EarlyReceive. It increases the performance that occurred because of synchronous communications. The CASC prevents deadlocks and handles the problems of blocking synchronous sends at the clustering phase.

Mishra and Tripathi [27] defined a priority function for cluster pairs in the task graph and proposed an algorithm, named Cluster Pair Priority Scheduling (CPPS). The CPPS initially determines the priorities for all cluster pairs and sorts them in decreasing order, then inspects each cluster pair one-by-one and groups them if makespan decreases. The priorities of cluster pairs change whenever a grouping of tasks take place; the CPPS performs above steps until makespan decreases.

Mishra et al. [30] proposed a randomized algorithm, called RDCC (Randomized Dynamic Computation Communication) for task scheduling. The RDCC is a randomization of the CCLC (Computation Communication Load Clustering) algorithm given in [29] and uses the concept of dynamic priority from DCCL (Dynamic Computation Communication Load) algorithm [28]. The CCLC determines the CCLoad (Computation-Communication-Load) for each task and sorts them according to their CCLoad values. The algorithm initially places all tasks in one cluster, then extracts them one-by-one at each step and places them in separate clusters if makespan decrease. The DCCL algorithm works similar to CCLC and utilizes the concept of priority which is based on the calculation of Dynamic CCLoad.

Khaldi et al. [17] presented an algorithm Bounded Dominant Sequence Clustering (BDSC) that extends the DSC algorithm for a limited number of processors. The BDSC

Table 2 A comparison of the existing clustering-based task scheduling algorithms

Algorithm	References	Concept	Complexity
EZ	[32]	Edge zeroing	$O(\|E\|(\|V\|+\|E\|))$
LC	[18]	Critical Path	$O(\|V\|(\|V\|+\|E\|))$
DSC	[37,38]	Dominant Sequence	$O((\|V\|+\|E\|)log(\|V\|))$
GDS	[9]	Greedy Dominant Sequence	$O(\|V\|(\|V\|+\|E\|))$
MCP	[36]	ALAP	$O(\|V\|^2 log(\|V\|))$
DCP	[20]	Critical Path	$O(\|V\|^3)$
CASC	[16]	Synchronous Communication	$O(\|V\|(\|E\|^2+log(\|V\|)))$
CPPS	[27]	Cluster Pair Priority	$O(\|V\|\|E\|(\|V\|+\|E\|))$
CCLC	[29]	CCLoad	$O(\|V\|^2(\|V\|+\|E\|)log(\|V\|+\|E\|))$
DCCL	[28]	Dynamic CCLoad	$O(\|V\|^2(\|V\|+\|E\|)log(\|V\|+\|E\|))$
RDCC	[30]	Randomization	$O(ab\|V\|(\|V\|+\|E\|)log(\|V\|+\|E\|))$
BDSC	[17]	Dominant Sequence	$O(\|V\|^3)$
LOCAL	[26]	Local Search	$O(\|V\|\|E\|(\|V\|+\|E\|))$

considers several major attributes like memory constraints, dependency constraints, processor availability, etc. The algorithm reduces the number of clusters to match the processor quantities and assigns each cluster to available processors which provide smallest top level. The algorithm utilizes an extra heuristic to decide the precedence among tasks by their bottom level when tasks have equal priorities.

In [26], the authors proposed a general randomized task scheduling algorithm called LOCAL(A, B) for multiprocessor environments. It uses local neighborhood search and gives a hybrid of two known task scheduling algorithms (A, and B). As an instance, the authors selected DSC as algorithm A and CPPS as algorithm B.

Table 2 shows a comparison of the existing clustering-based task scheduling algorithms in terms of the concept used and the time complexity. In Table 2, $|E|$ and $|V|$ represent the number of edges and the number of nodes in the task graph, respectively, 'a' represents the number of randomization steps, and 'b' denotes a limit on the number of clusters formed.

4 The proposed clustering-based scheduling of tasks

In this section, we introduce the proposed clustering-based scheduling algorithm named ECP for an unbounded number of processors for the task scheduling problem on the multiprocessor computing systems. Our proposed clustering based solution, to the scheduling problem, is implemented in Algorithm 3 below which makes use of Algorithms 1, and 2 and executes a chain of clustering refinement steps. The earliest step allocates each task of the task graph to a distinct cluster. At each step, the algorithm tries to improve its earlier clustering by merging suitable clusters into one. A merging operation is carried out by zeroing an edge cost linking two clusters. The main aim of the ECP algorithm is to minimize the makespan taking into consideration the precedence constraints. This algorithm uses the concept of edge zeroing on the critical path of the task graph to group the tasks. The idea of edge zeroing was initially given by Sarkar [32]. Sarkar's algorithm examines the edges one by one from sorted list and performs edge zeroing if makespan does not increase. In Sarkar's algorithm, the edge which is zeroed may not be on a path that determines the makespan. Here in the proposed algorithm, we repeatedly compute a critical path of the task graph and perform the edge zeroing steps only on the critical path edges with the aim of reducing makespan. The proposed algorithm has the following characteristics:

- it zeroes an edge of a critical path at each step, producing a new critical path, thereafter,
- it uses backtracking, after zeroing an edge at each step, when current makespan is more than the makespan in the previous step,
- it dynamically updates the schedule for each processor until makespan of the task graph does not increase, and
- it implicitly gives the feasible schedule of the clustering obtained at each step. Otherwise, current makespan would not be determined, on which clustering decisions are based.

In the following, we talk about a number of the concepts utilized in the design of our algorithm. In the first subsection, we describe the computation of the critical path and a method to select the edge for zeroing. In the second subsection, we discuss the method for merging of clusters. We describe the proposed algorithm in the third subsection. In the fourth subsection, the analysis about the complexity of the algorithms are presented, An illustrative example for proposed algorithm is given at the end of this section.

4.1 Critical path computation and edge selection

As given in Definition 5, the critical path of a task graph is the path that has the maximum sum of the execution and communication times. It determines the partial makespan of a task graph because the sum of computation time of the tasks belongs to a CP provides the lower bound on the makespan. In the proposed solution, at each step of the scheduling process, the CP of a task graph may change dynamically, because an edge on a CP in a particular step may not remain on the CP at the next step due to edge zeroing. We call the intermediate CP obtained during scheduling steps as the Effective Critical Path (ECP) to distinguish it from the initial CP of the unscheduled task graph. The ECP is computed as shown in Algorithm 1.

After the ECP computation, we need an approach to select an appropriate edge on the ECP for zeroing. To select an edge, we first sort the edges of the ECP according to their CT values in non-increasing order by using heap sort. Then choose edges from left to the right in the sorted list until schedule length does not reduce. If two edges have same CT value on the ECP, the order between edges is decided by the sum of the execution time of the associated tasks. The edge which has lower value will get higher order than the other in the sorted list. If this value is equal for both edges, the order is decided according to FCFS in the ECP. By doing this, we are giving lower priority to the computation-intensive tasks in comparison to communication-intensive.

Algorithm 1 Algorithm for the ECP computation

```
1: topo_list ← a list of all tasks $T_i \in V$ sorted in a reverse topological order
2: for each task $T_i$ in topo_list do
3:     Compute $BL(T_i)$ as Eq. (2)
4:     Store successor of $T_i$ in $ecp\_succ(T_i)$ from which $BL(T_i)$ is computed
5: end for
6: $ecp[0] \leftarrow T_i$, where $T_i \in V$ with maximum $BL(T_i)$
7: $T_{current} \leftarrow T_i$
8: $count \leftarrow 0$
9: while $ecp\_succ(T_{current}) \neq \phi$ do
10:     $ecp\_edges[count] \leftarrow e_{current,ecp\_succ(T_{current})}$
11:     $count \leftarrow count + 1$
12:     $ecp[count] \leftarrow ecp\_succ(T_{current})$
13:     $T_{current} \leftarrow ecp\_succ(T_{current})$
14: end while
```

Computation-intensive tasks are those tasks which spend more time in performing computation than communication.

4.2 Clustering

While we are able to identify an appropriate edge on an ECP for zeroing, we still require a method to merge two clusters after zeroing an edge. When two clusters are merged, their communication becomes local, and tasks of the cluster are ordered according to Algorithm 2.

Algorithm 2 Algorithm for the merging of two clusters, C_1 and C_2

```
1: Let $l_1$ is a list of tasks of $C_1$ and $l_2$ is a list of tasks of $C_2$
2: $i \leftarrow 0$, $j \leftarrow 0$
3: while $i \leq l_1.size$ and $j \leq l_2.size$ do
4:     if $l_1[i]$ is descendent of $l_2[j]$ then
5:         $j++$
6:     else if $l_2[j]$ is descendent of $l_1[i]$ then
7:         $i++$
8:     else if $BL(l_1[i]) < BL(l_2[j])$ then
9:         add pseudo edge from $l_2[j]$ to $l_1[i]$ with zero communication time
10:        $j++$
11:    else if $BL(l_2[j]) < BL(l_1[i])$ then
12:        add pseudo edge from $l_1[i]$ to $l_2[j]$ with zero communication time
13:        $i++$
14:    else if $l_2[j]$ comes earlier in topological order than $l_1[i]$ then
15:        add pseudo edge from $l_2[j]$ to $l_1[i]$ with zero communication time
16:        $j++$
17:    else
18:        add pseudo edge from $l_1[i]$ to $l_2[j]$ with zero communication time
19:        $i++$
20:    end if
21: end while
22: Make Cluster $C_2$ as Cluster $C_1$
```

The algorithm first verifies that whether the tasks of one cluster are descendent of other. If yes, there is no need to order them explicitly. Otherwise, the algorithm compares their bottom level and adds a pseudo edge with zero communication time from the task having higher BL value to the task having lower BL value. If the BL value of the tasks comes out to be equal, order the tasks according to their topological order in their clusters and add a pseudo edge with zero communication time from the higher order task to the lower order task.

4.3 The ECP algorithm

In this subsection, we formalize the proposed ECP algorithm in Algorithm 3. As stated earlier this algorithm makes use of Algorithms 1, and 2 for its purpose as indicated in the Algorithm 3 given below.

Algorithm 3 The ECP Algorithm

```
1: Initially, each task forms a separate cluster
2: Compute initial schedule length
3: repeat
4:     Compute the ECP via Algorithm 1
5:     Sort the edges of the ECP in non-increasing order according to their communication time
6:     for all edges of ECP in the sorted list do
7:         if both tasks of an edge belongs to same cluster then
8:             continue
9:         else
10:            Zero an edge if schedule length does not increase
11:            When two clusters are merged, use Algorithm 2
12:            Update schedule length
13:        end if
14:    end for
15: until schedule length does not increase
```

The algorithm starts with initial clustering of each task of the task graph at line 1. It then computes the initial schedule length of the unscheduled task graph. The algorithm repeats the steps from line 3 to 14, until schedule length does not increase. In these steps, the algorithm computes ECP of partially scheduled task graph via algorithm 1 and sorts the edges of the ECP according to their communication time by using heap sort. Then it selects the edges of the ECP from left to right in the sorted list and verifies whether both tasks of an edge belong to a cluster or not. Perform edge zeroing if both tasks of an edge are in different clusters and schedule length does not increase after clustering, otherwise do verification for next edges. Whenever edge zeroing is performed, clusters are merged according to Algorithm 2 and schedule length is updated.

4.4 Algorithm complexity analysis

In this section, we present an analysis of the time complexity of the algorithms discussed above.

4.4.1 Analysis of Algorithm 1

This algorithm finds the ECP of a task graph. The topological sorting in line 1 can be performed in $O(|V| + |E|)$ time. Line 2 in the for loop iterates through the edges coming out of a task T_i. Lines 3 and 4 take constant time. Therefore, the total number of iterations for lines 2 to 5 occurs $|E|$ times. Line 6 to 8 takes constant time. The while loop from lines 9 to 14 find the tasks and edges of the ECP and execute $|V_{ecp}|$ times where V_{ecp} is the number of tasks on the ECP. In the worst case, this while loop will execute $|V|$ times when all tasks on the ECP. Thus, the upper bound of this algorithm is $O(|V| + |E|)$.

4.4.2 Analysis of Algorithm 2

This algorithm finds the ECP of a task graph. Line 2 is an initialization that can execute in constant time. The while loop beginning at line 3 will execute at most $|V|$ times, and the steps from lines 4 to 20 will take constant time. Thus, the total number of iterations for lines 3 to 21 occurs $|V|$ times. Line 22 will execute $|V|$ times. Thus, the upper bound of this algorithm is $O(|V|)$.

4.4.3 Analysis of Algorithm 3

Algorithm 3 is the proposed ECP algorithm computing the final schedule for a task graph. Line 1 is an initialization that can be done in $|V|$ times. Line 2 computes the initial schedule length in $(|V| + |E|)$ times. In line 4, the ECP can be computed via Algorithm 1 in $O(|V|+|E|)$ time. The sorting in line 5 can be performed via heap sort and completed in $O(|V|lg|V|)$ time. The for loop beginning at line 6 will execute at most $|V|$ times when all tasks belong to an ECP. Line 10 can be executed in $(|V|+|E|)$ times. In line 11, when two clusters are merged, a call is issued to Algorithm 2 and can be completed in $O(|V|)$ time. Line 12 updates schedule length in constant time. The total number of iterations for lines 3 to 15 occurs $|V|$ times. Thus, the complexity of the ECP algorithm is $O(|V|^2(|V| + |E|))$.

4.5 An illustrative example

Consider the task graph given in Fig. 1. The graph consists of fifteen tasks labeled T_0 to T_{14} with their execution times. The tasks T_0 and T_{14} are the entry and exit tasks of the task graph respectively which represent the starting and ending of the application. The edges of the graph are labeled with the communication times.

At each step of the execution of the algorithm, the ECP of the task graph is determined, and edges of the ECP are examined one-by-one in non-increasing order of their communication times and corresponding tasks (clusters) of that edge are merged, if schedule length does not increase. Clustering steps of the example task graph in Fig. 1 with the ECP algorithm is shown in Fig. 2.

Initially, each task is assumed to be in a distinct cluster as shown in Fig. 2a, in which clusters are shown with different color boxes. Schedule length for the initial clustering comes out to be 39.

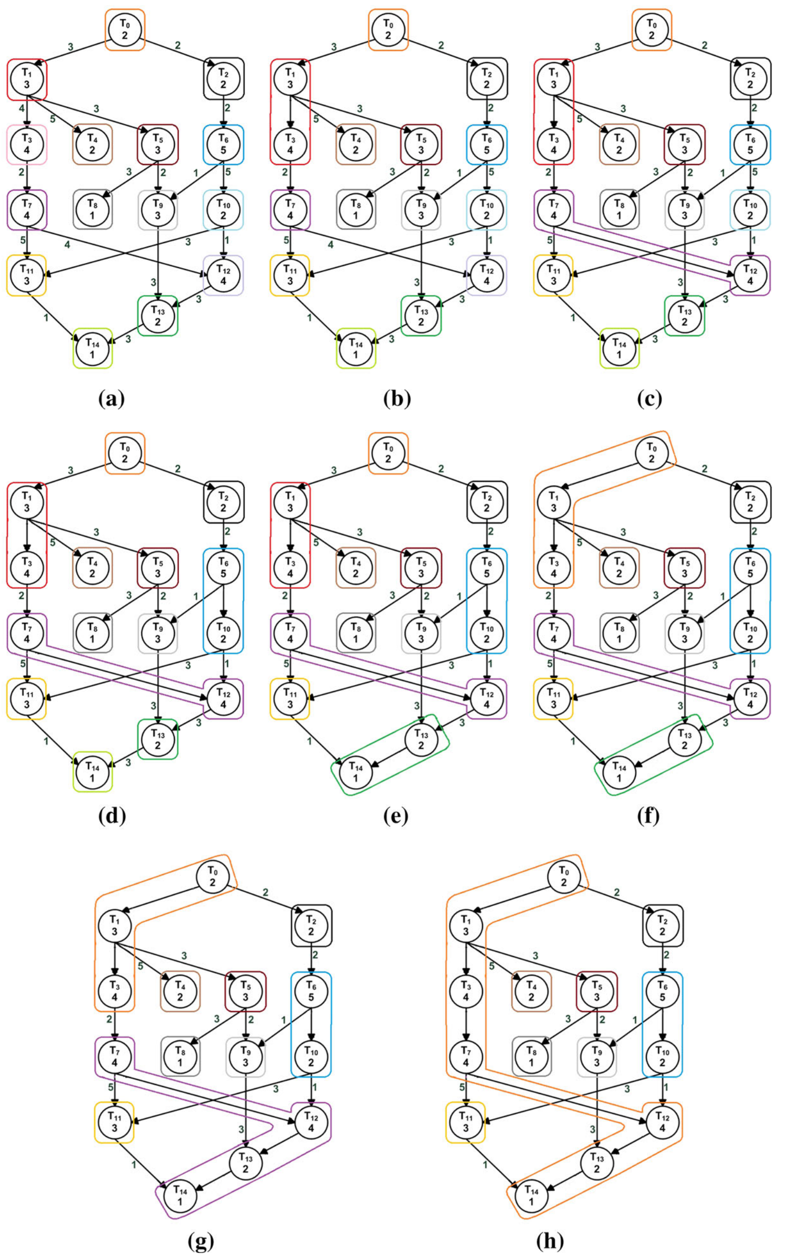

Fig. 2 Clustering steps of the example task graph in Fig. 1 with the ECP algorithm **a** initial clustering (initial schedule length = 39), **b** clustering after merging T_1 and T_3 (partial schedule length = 35), **c** clustering after merging T_7 and T_{12} (partial schedule length = 34), **d** clustering after merging T_6 and T_{10} (partial schedule length = 31), **e** clustering after merging T_{13} and T_{14} (partial schedule length = 28), **f** clustering after merging T_0 and $\{T_1, T_3\}$ (partial schedule length = 26), **g** clustering after merging $\{T_7, T_{12}\}$ and $\{T_{13}, T_{14}\}$ (partial schedule length = 25), (h) clustering after merging $\{T_0, T_1, T_3\}$ and $\{T_{12}, T_{13}, T_{14}\}$ (final schedule length = 23)

Figure 2b–g show the intermediate clusters and the partial schedule length of the task graph. In Fig. 2h, the schedule length comes out to be 23 that can′t be reduced after merging any edge in the ECP. Thus, it is the final schedule length of the task graph, and the clustering obtained in this step reflects the final schedule of the example task graph.

5 Experimental results and discussion

This section provides the performance evaluation of the ECP algorithm with four well-known clustering based task scheduling algorithms, the EZ, the LC, the CPPS, and the LOCAL, using various comparison metrics. For this purpose, two types of task graphs are considered: (i) randomly generated and (ii) derived from real-world applications. The experiments are carried out on a Dell PowerEdge R420 server with CentOS (version 7.3-1611), Intel(R) Xeon(R) CPU E5-2420 v2 @ 2.20 GHz processor, and 192 GB of memory.

5.1 Comparison metrics

The comparisons of the task scheduling algorithms are based on the following performance metrics:

5.1.1 Normalized schedule length

As many number of task graphs having various characteristics are utilized, it is required to normalize the schedule length to the lower bound. Hence, the Normalized Schedule Length (NSL) [21] of a schedule for a given task graph or Scheduling Length Ratio (SLR) [25] can be obtained by dividing the makespan to the total execution times on the critical path of the task graph as given in Eq. (7):

$$NSL = \frac{makespan}{\sum_{T_i \in CP} ET(T_i)} \tag{7}$$

The total execution times on the critical path provide the lower bound on the makespan. So we have the following inequality for the NSL:

$$NSL \gg 1. \tag{8}$$

5.1.2 Speedup

The speedup [25,35] for a given task graph is the improvement in speed of execution when task graph is executed on single and many processors. It can be obtained by dividing the sequential execution time (i.e. by assigning all tasks of a task graph to a single processor) to the parallel execution time (i.e. the makespan of the final schedule) as given in Eq. (9):

$$Speedup = \frac{\sum_{T_i \in V} ET(T_i)}{makespan} \tag{9}$$

It is used to show the effect of performance after processor enhancement.

5.1.3 CCR

The CCR (Communication to Computation Ratio) [21] of a given task graph is defined as the ratio of the average communication time to the average execution time as given in Eq. (10):

$$CCR = \frac{\left(\sum_{e_{i,j} \in E} CT(e_{i,j})\right)/|E|}{\left(\sum_{T_i \in V} ET(T_i)\right)/|V|} \tag{10}$$

If CCR value of a task graph is very high, it can be considered as a communication intensive task graph.

5.1.4 Percentage improvement in NSL

The percentage improvement in NSL of ECP algorithm over algorithm A [38] is defined as follows:

$$\%NSL_{improvement} = \left(1 - \frac{NSL_{ECP}}{NSL_A}\right) \times 100 \tag{11}$$

where NSL_{ECP} represents the NSL generated by the ECP algorithm and NSL_A is the NSL generated by the compared algorithms.

5.1.5 Percentage improvement in speedup

The percentage improvement in speedup of ECP algorithm over algorithm A is defined as follows:

$$\%Speedup_{improvement} = \left(1 - \frac{Speedup_A}{Speedup_{ECP}}\right) \times 100 \tag{12}$$

where $Speedup_{ECP}$ represents the speedup generated by the ECP algorithm and $Speedup_A$ is the speedup generated by the compared algorithms.

5.2 Random task graphs

David and Gabriel [8] proposed a set of 180 benchmark random task graphs for comparison and analysis of heuristic algorithms. The graphs are divided into 6 subsets each of which contains 30 graphs and having the number of nodes as 50, 100, 200, 300, 400, and 500 respectively.

The average NSL results of random graphs for the different number of nodes are shown in Fig. 3. For each set of task graphs, the ECP algorithm gives better schedules than other algorithms. Overall, the ECP algorithm provides an improvement of 19.93,

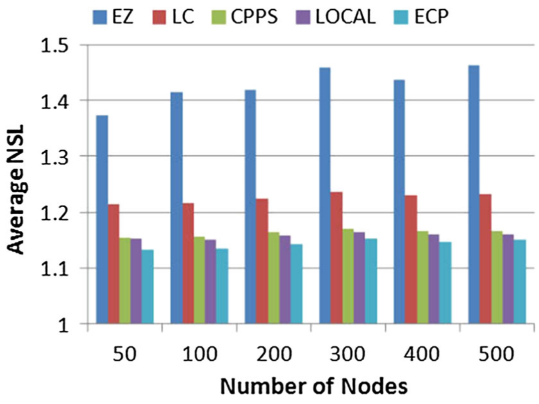

Fig. 3 Average NSL results obtained for random task graphs as a function of number of nodes

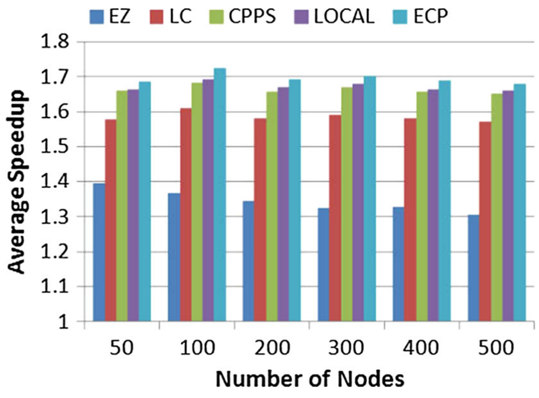

Fig. 4 Average Speedup results obtained for random task graphs as a function of number of nodes

6.72, 1.76, and 1.25 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

The average Speedup results of random graphs for the different number of nodes are shown in Fig. 4. For each set of task graphs, the ECP algorithm produces better speedup than other algorithms. Overall, the ECP algorithm provides an improvement of 20.74, 6.54, 1.92, and 1.38 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

5.3 Real-world task graphs

Besides randomly generated graphs, we evaluated and compared the performance of the algorithms concerning real-world applications, namely Gaussian Elimination [6, 35,36], fast Fourier transform [5,35], and systolic array [14]. All of these applications are well-known and used in real-world problems.

5.3.1 Gaussian Elimination task graphs

In Gaussian Elimination task graphs, the number of tasks is $(m^2 + m - 2)/2$ where m is the matrix size. A Gaussian Elimination task graph for matrix size $m = 5$ is shown

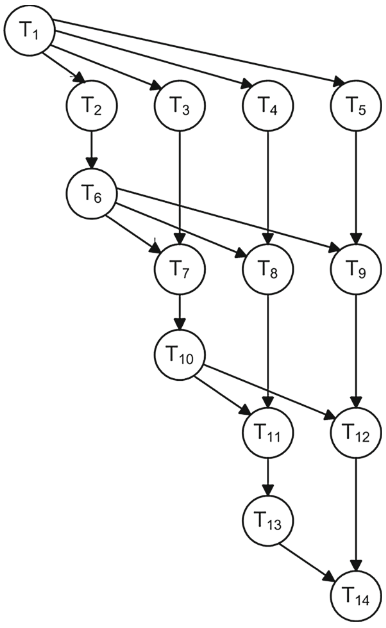

Fig. 5 A Gaussian Elimination task graph for matrix of size 5

in Fig. 5. The values of m used in this experiment are varied from 5 to 30 with an interval of 5. As the structure of this graph is known, we use different values of CCR as [0.1, 1, 10] to carry out experiments.

The average NSL results of Gaussian Elimination graphs for different matrix sizes are shown in Fig. 6. For each set of task graphs, the ECP algorithm gives better schedules than other algorithms. Overall, the ECP algorithm provides an improvement of 18.90, 34.04, 51.23, 44.27, and 9.44 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

The average NSL results of Gaussian Elimination graphs for different values of CCR are shown in Fig. 7. For CCR = 0.10, the ECP algorithm provides an improvement of 3.94 percent over the EZ algorithm. For CCR = 1, the ECP algorithm provides an improvement of 19.84, 5.07, 2.92, and 2.04 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively. For CCR = 10, the ECP algorithm provides an improvement of 22.18, 45.65, 15.48, and 11.57 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

The average speedup results of Gaussian Elimination graphs for different matrix sizes are shown in Fig. 8. For each set of task graphs, the ECP algorithm gives better schedules than other algorithms. Overall, the ECP algorithm gives an improvement of 13.46, 8.47, 60.74, 49.34, and 2.78 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

The average speedup results of Gaussian Elimination graphs for different values of CCR are shown in Fig. 9. For CCR = 0.10, the ECP algorithm provides an improvement of 4.14 percent over the EZ algorithm. For CCR = 1, the ECP algorithm provides an improvement of 20.96, 5.30, 3.52, and 2.88 percent over the EZ, LC, CPPS and LOCAL

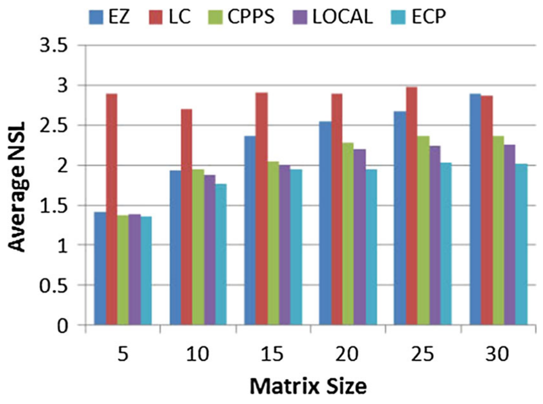

Fig. 6 Average NSL results obtained for Gaussian Elimination task graphs as a function of matrix size

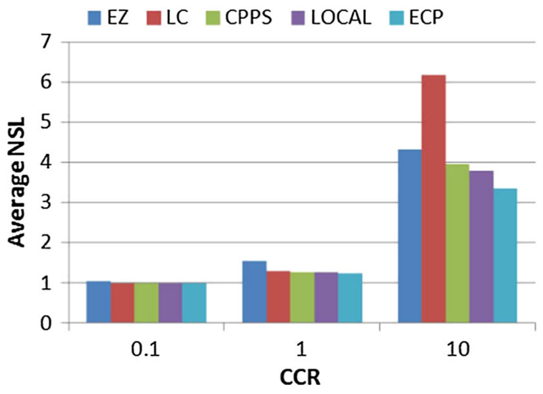

Fig. 7 Average NSL results obtained for Gaussian Elimination task graphs as a function of CCR

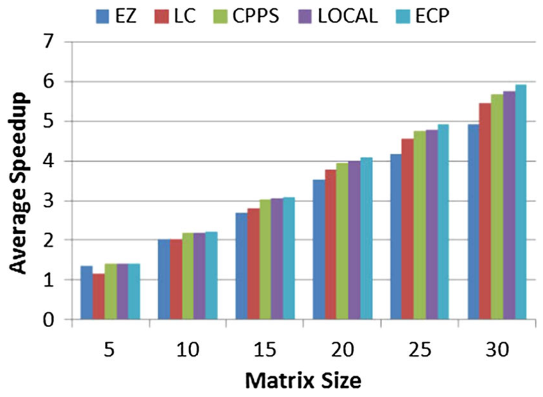

Fig. 8 Average speedup results obtained for Gaussian Elimination task graphs as a function of matrix size

scheduling algorithms respectively. For CCR = 10, the ECP algorithm provides an improvement of 21.95, 43.93, 14.04 and 9.37 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

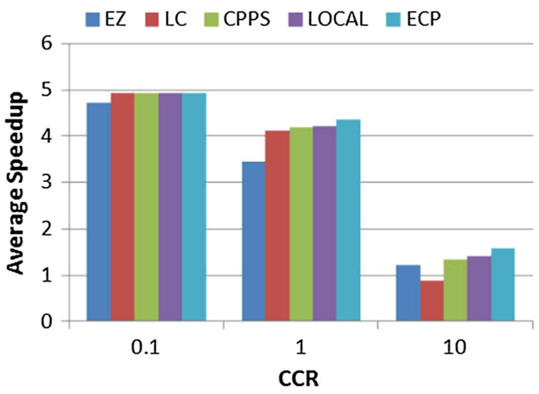

Fig. 9 Average speedup results obtained for Gaussian Elimination task graphs as a function of CCR

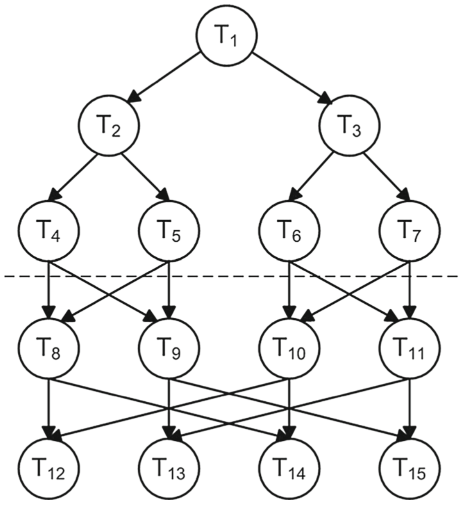

Fig. 10 A FFT task graph for input points 4

5.3.2 Fast Fourier transform task graphs

In fast Fourier transform (FFT) task graphs, the number of tasks is $(2m-1)+mlog_2 m$ where m is the input points and $m = 2^k$ for some integer k. The FFT task graph contains two types of tasks: recursive call tasks and butterfly operation tasks. An FFT task graph for four input points is shown in Fig. 10. In this figure, the tasks above dashed line are recursive call tasks, and the tasks below dashed line are butterfly tasks. The structure of FFT is known; hence, we use different values for input points as [2, 4, 8, 16, 32] and CCR as [0.1, 1, 10] to carry out experiments.

The average NSL results of FFT graphs for different number input points are shown in Fig. 11. For each set of task graphs, the ECP algorithm produces better schedules than other algorithms. Overall, the ECP algorithm gives an improvement of 12.83, 39.47, 67.30, 55.15, and 21.09 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

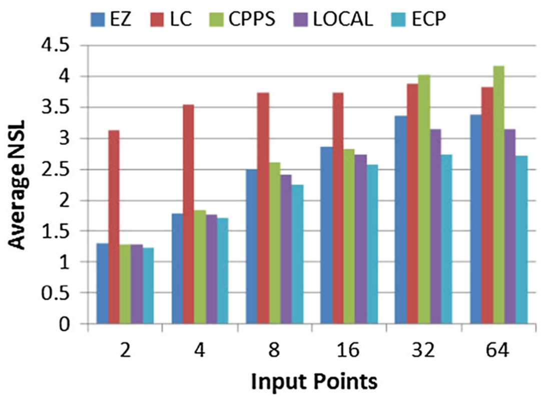

Fig. 11 Average NSL results obtained for FFT task graphs as a function of input points

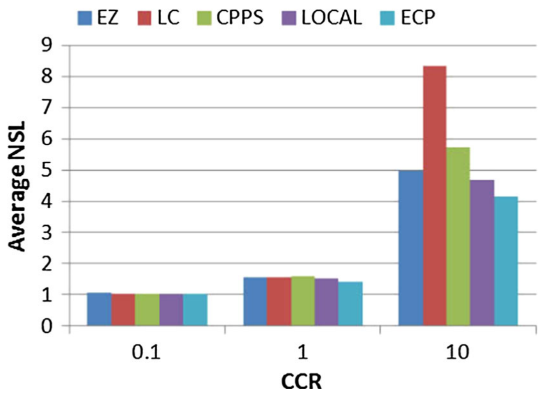

Fig. 12 Average NSL results obtained for FFT task graphs as a function of CCR

The average NSL results of FFT graphs for different values of CCR are shown in Fig. 12. For CCR = 0.10, the ECP algorithm provides an improvement of 2.23, 0.70, 1.68, and 1.13 percent over the EZ, LC, DCCL, RDCC and CPPS scheduling algorithms respectively. For CCR = 1, the ECP algorithm provides an improvement of 8.22, 7.46, 10.95, and 5.54 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively. For CCR = 10, the ECP algorithm provides an improvement of 16.51, 50.22, 27.48, and 11.14 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

The average speedup results of FFT graphs for the different number of input points are shown in Fig. 13. For each set of task graphs, the ECP algorithm produces better schedules than other algorithms. Overall, the ECP algorithm gives an improvement of 7.53, 7.63, 79.40, 72.37, and 10.00 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

The average speedup results of FFT graphs for the different values of CCR are shown in Fig. 14. For CCR = 0.10, the ECP algorithm provides an improvement of 3.18, 0.79, 2.81 and 2.09 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively. For CCR = 1, the ECP algorithm provides an improvement of 10.97, 6.87, 15.37, and 5.98 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively. For CCR = 10, the ECP algorithm provides an improvement of

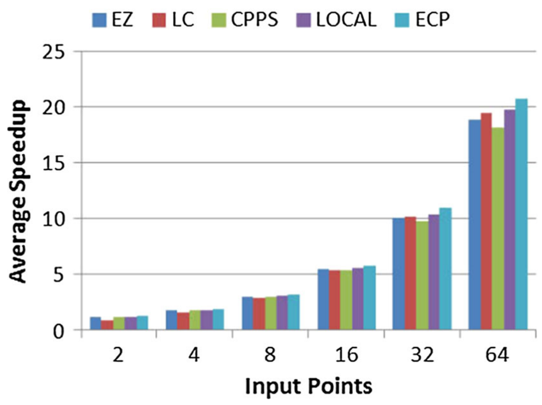

Fig. 13 Average speedup results obtained for FFT task graphs as a function of input points

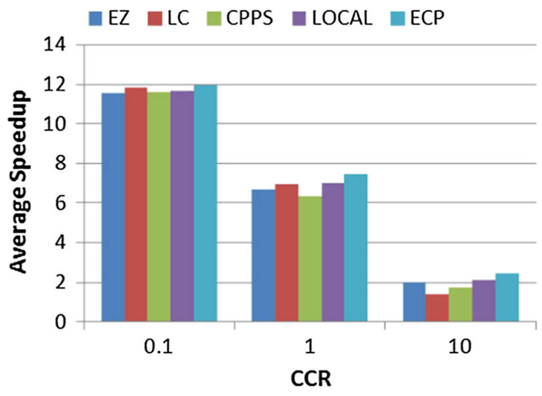

Fig. 14 Average speedup results obtained for FFT task graphs as a function of CCR

18.20, 43.35, 28.68, and 12.17 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

5.3.3 Systolic array task graphs

In systolic array task graphs, the number of nodes is n^2 and number of edges are $2n(n + 1)$ where n is the number of nodes on a path from the start node to the centre node. A systolic array task graph for n = 3 is shown in Fig. 15. The values of n used in this experiment are varied from 5 to 20 with an interval of 5. As the structure of this graph is known, we use different values of CCR as [0.1, 1, 10] to carry out experiments.

The average NSL results of systolic array graphs for the different number of nodes are shown in Fig. 16. For each set of task graphs, the ECP algorithm produces better schedules than other algorithms. Overall, the ECP algorithm gives an improvement of 9.35, 41.55, 37.08, 35.13, and 21.64 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

The average NSL results of systolic array graphs for the different values of CCR are shown in Fig. 17. For CCR = 0.10, the ECP algorithm provides an improvement of 2.40, 1.04, 2.14, and 1.38 percent over the EZ, LC, CPPS and LOCAL scheduling

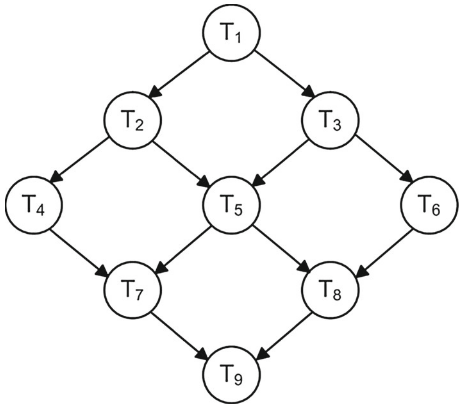

Fig. 15 A systolic array task graph for n = 3

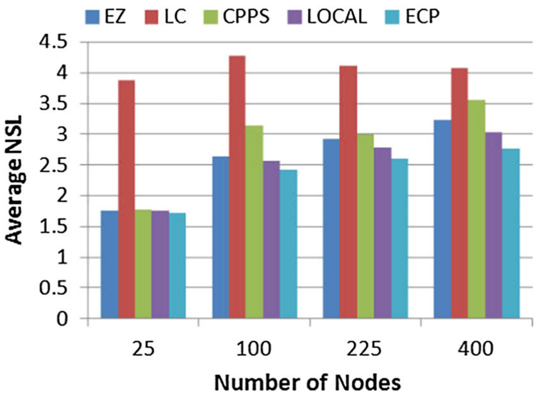

Fig. 16 Average NSL results obtained for systolic array task graphs as a function of number of nodes

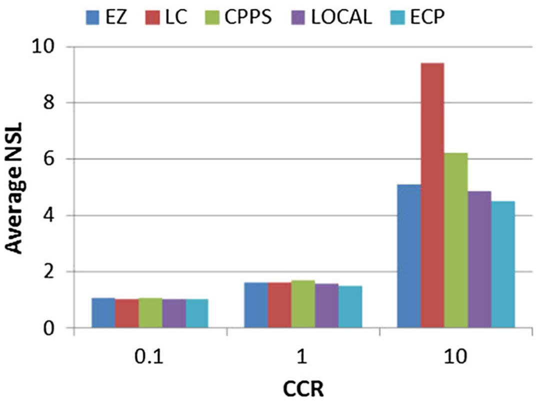

Fig. 17 Average NSL results obtained for systolic array task graphs as a function of CCR

algorithms respectively. For CCR = 1, the ECP algorithm provides an improvement of 7.58, 6.34, 12.02, and 4.55 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively. For CCR = 10, the ECP algorithm provides an improvement of 11.36, 52.06, 27.59, and 7.56 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

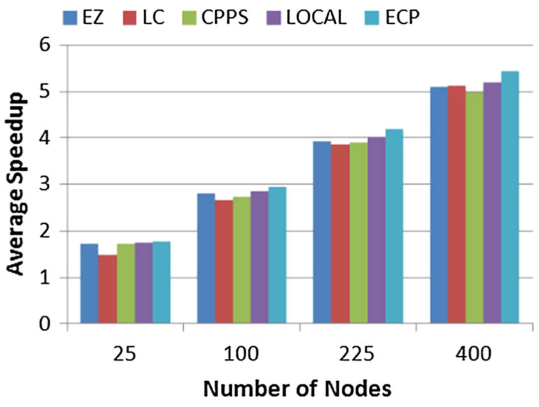

Fig. 18 Average speedup results obtained for systolic array task graphs as a function of number of nodes

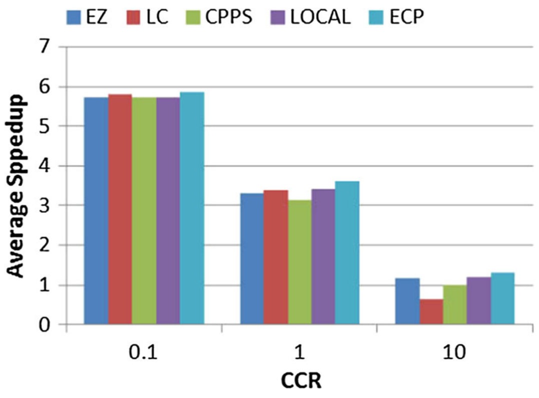

Fig. 19 Average speedup results obtained for systolic array task graphs as a function of CCR

The average Speedup results of systolic array graphs for the different number of nodes are shown in Fig. 18. For each set of task graphs, the ECP algorithm produces better schedules than other algorithms. Overall, the ECP algorithm gives an improvement of 5.39, 8.48, 58.09, 49.61, and 8.38 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

The average Speedup results of systolic array graphs for the different values of CCR are shown in Fig. 19. For CCR = 0.10, the ECP algorithm provides an improvement of 2.59, 0.92, 2.43, and 2.24 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively. For CCR = 1, the ECP algorithm provides an improvement of 8.08, 5.65, 13.15, and 5.43 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively. For CCR = 10, the ECP algorithm provides an improvement of 10.55, 50.33, 21.94, and 8.35 percent over the EZ, LC, CPPS and LOCAL scheduling algorithms respectively.

6 Conclusion

We have proposed and explained here a clustering-based scheduling algorithm that makes use of critical path, and we name it Effective Critical Path (ECP) algorithm, for the problem of task scheduling in multiprocessors. The ECP algorithm goes for edge zeroing on critical paths for clustering the tasks. The complexity of the ECP algorithm works out to be $O(|V|^2(|V| + |E|))$, where $|E|$ represents the number of edges and $|V|$ denotes the number of tasks in the task graph. The performance of this ECP algorithm is compared with four well-known clustering-based scheduling algorithms such as EZ, LC, CPPS, and LOCAL. The comparative study is based on two types of task graphs such as randomly generated benchmark task graphs and task graphs that correspond to real-world applications. The task graphs derived from real-world applications are Gaussian Elimination, fast Fourier transform and systolic array. The ECP algorithm proposed here significantly outperforms the said algorithms in terms of average NSL and average speedup for all types of task graphs considered in this work. For future work, the proposed task scheduling algorithm may be suitably extended for heterogeneous multiprocessors or may be integrated with the existing duplication-based task scheduling strategies for different real-world applications.

References

1. Arabnejad H, Barbosa JG (2014) List scheduling algorithm for heterogeneous systems by an optimistic cost table. IEEE Trans Parallel Distrib Syst 25(3):682–694
2. Bajaj R, Agrawal DP (2004) Improving scheduling of tasks in a heterogeneous environment. IEEE Trans Parallel Distrib Syst 15(2):107–118
3. Bansal S, Kumar P, Singh K (2003) An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. IEEE Trans Parallel Distrib Syst 14(6):533–544
4. Boeres C, Filho JV, Rebello VE (2004) A cluster-based strategy for scheduling task on heterogeneous processors. In: Proceedings of the 16th symposium on computer architecture and high performance computing (SBAC-PAD'04). IEEE, pp 214–221
5. Chung YC, Ranka S (1992) Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In: Proceedings of the 1992 ACM/IEEE conference on supercomputing. IEEE Computer Society Press, pp 512–521
6. Cosnard M, Marrakchi M, Robert Y, Trystram D (1988) Parallel Gaussian elimination on an mimd computer. Parallel Comput 6(3):275–296
7. Daoud MI, Kharma N (2008) A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. J Parallel Distrib Comput 68(4):399–409
8. Davidović T, Crainic TG (2006) Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multiprocessor systems. Comput Oper Res 33(8):2155–2177
9. Dikaiakos MD, Steiglitz K, Rogers AA (1994) Comparison of techniques used for mapping parallel algorithms to message-passing multiprocessors. In: Proceedings of the sixth IEEE symposium on parallel and distributed processing, 1994. IEEE, pp 434–442
10. El-Rewini H, Lewis TG (1990) Scheduling parallel program tasks onto arbitrary target machines. J Parallel Distrib Comput 9(2):138–153
11. Gogos C, Valouxis C, Alefragis P, Goulas G, Voros N, Housos E (2016) Scheduling independent tasks on heterogeneous processors using heuristics and column pricing. Fut Gener Comput Syst 60:48–66
12. Hagras T, Janeček J (2005) A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. Parallel Comput 31(7):653–670
13. Hu M, Luo J, Wang Y, Veeravalli B (2017) Adaptive scheduling of task graphs with dynamic resilience. IEEE Trans Comput 66(1):17–23
14. Ibarra OH, Sohn SM (1990) On mapping systolic algorithms onto the hypercube. IEEE Trans Parallel Distrib Syst 1(1):48–63
15. Ilavarasan E, Thambidurai P (2007) Low complexity performance effective task scheduling algorithm for heterogeneous computing environments. J Comput Sci 3(2):94–103
16. Kadamuddi D, Tsai JJ (2000) Clustering algorithm for parallelizing software systems in multiprocessors environment. IEEE Trans Softw Eng 26(4):340–361

17. Khaldi D, Jouvelot P, Ancourt C (2015) Parallelizing with BDSC, a resource-constrained scheduling algorithm for shared and distributed memory systems. Parallel Comput 41:66–89
18. Kim S, Browne J (1988) A general approach to mapping of parallel computation upon multiprocessor architectures. In: International conference on parallel processing, vol 3, p 8
19. Kruatrachue B, Lewis T (1988) Grain size determination for parallel processing. IEEE Softw 5(1):23–32
20. Kwok YK, Ahmad I (1996) Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. IEEE Trans Parallel Distrib Syst 7(5):506–521
21. Kwok YK, Ahmad I (1999) Benchmarking and comparison of the task graph scheduling algorithms. J Parallel Distrib Comput 59(3):381–422
22. Kwok YK, Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput Surv (CSUR) 31(4):406–471
23. Liou JC, Palis MA (1996) An efficient task clustering heuristic for scheduling dags on multiprocessors. In: Workshop on resource management, symposium on parallel and distributed processing, pp 152–156
24. Maurya AK, Tripathi AK (2017) Performance comparison of heft, lookahead, ceft and peft scheduling algorithms for heterogeneous computing systems. In: Proceedings of the 7th international conference on computer and communication technology (ICCCT'2017). ACM, pp 128–132
25. Maurya AK, Tripathi AK (2018) On benchmarking task scheduling algorithms for heterogeneous computing systems. J Supercomput. https://doi.org/10.1007/s11227-018-2355-0
26. Mishra A, Mishra PK (2016) A randomized scheduling algorithm for multiprocessor environments using local search. Parallel Process Lett 26(01):1650,002
27. Mishra A, Tripathi AK (2010) An extention of edge zeroing heuristic for scheduling precedence constrained task graphs on parallel systems using cluster dependent priority scheme. In: 2010 International conference on computer and communication technology (ICCCT). IEEE, pp 647–651
28. Mishra P, Mishra K, Mishra A (2011) A clustering algorithm for multiprocessor environments using dynamic priority of modules. Ann Math Inform 38:99–110
29. Mishra PK, Mishra KS, Mishra A (2010) A clustering heuristic for multiprocessor environments using computation and communication loads of modules. Int J Comput Sci Inf Technol 2(5):170–182
30. Mishra PK, Mishra KS, Mishra A, Tripathi AK (2012) A randomized scheduling algorithm for multiprocessor environments. Parallel Process Lett 22(04):1250,015
31. Papadimitriou CH, Yannakakis M (1990) Towards an architecture-independent analysis of parallel algorithms. SIAM J Comput 19(2):322–328
32. Sarkar V (1987) Partitioning and scheduling parallel programs for execution on multiprocessors. Tech. rep., Stanford University, CA (USA)
33. Sih GC, Lee EA (1993) A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Trans Parallel Distrib Syst 4(2):175–187
34. Tang X, Li K, Liao G, Li R (2010) List scheduling with duplication for heterogeneous computing systems. J Parallel Distrib Comput 70(4):323–329
35. Topcuoglu H, Hariri S, Wu My (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst 13(3):260–274
36. Wu MY, Gajski DD (1990) Hypertool: a programming aid for message-passing systems. IEEE Trans Parallel Distrib Syst 1(3):330–343
37. Yang T, Gerasoulis A (1991) A fast static scheduling algorithm for dags on an unbounded number of processors. In: Proceedings of the 1991 ACM/IEEE conference on Supercomputing. ACM, pp 633–642
38. Yang T, Gerasoulis A (1994) Dsc: scheduling parallel tasks on an unbounded number of processors. IEEE Trans Parallel Distrib Syst 5(9):951–967