Computing (2018) 100:557–595
https://doi.org/10.1007/s00607-017-0581-6

Locality-aware task scheduling for homogeneous parallel computing systems

Muhammad Khurram Bhatti[1] · Isil Oz[2] · Sarah Amin[1] · Maria Mushtaq[1] · Umer Farooq[3] · Konstantin Popov[4] · Mats Brorsson[5]

Received: 10 March 2017 / Accepted: 24 October 2017 / Published online: 1 November 2017
© Springer-Verlag GmbH Austria 2017

Abstract In systems with complex many-core cache hierarchy, exploiting data locality can significantly reduce execution time and energy consumption of parallel applications. Locality can be exploited at various hardware and software layers. For instance, by implementing private and shared caches in a multi-level fashion, recent hardware designs are already optimised for locality. However, this would all be useless if the software scheduling does not cast the execution in a manner that promotes locality available in the programs themselves. Since programs for parallel systems consist of tasks executed simultaneously, task scheduling becomes crucial for the performance in multi-level cache architectures. This paper presents a heuristic algorithm for homogeneous multi-core systems called *locality-aware task scheduling (LeTS)*. The LeTS heuristic is a work-conserving algorithm that takes into account both locality and load balancing in order to reduce the execution time of target applications. The working principle of LeTS is based on two distinctive phases, namely; *working task group formation phase (WTG-FP)* and *working task group ordering phase (WTG-OP)*. The WTG-FP forms groups of tasks in order to capture data reuse across tasks while the WTG-OP determines an optimal order of execution for task groups that minimizes the reuse distance of shared data between tasks. We have performed experiments using randomly generated task graphs by varying three major performance

✉ Muhammad Khurram Bhatti
khurram.bhatti@itu.edu.pk

1 Embedded Computing Lab, Information Technology University (ITU), 346-B Ferozpur Road, Lahore, Pakistan

2 Computer Engineering Department, Izmir Institute of Technology, Izmir, Turkey

3 Department of Electrical and Computer Engineering, Dhofar University, 211 Salalah, Oman

4 SICS, Isafjordsgatan 22, 164 29 Kista, Sweden

5 KTH Royal Institute of Technology, Isafjordsgatan 22, Box 1263, 164 29 Kista, Sweden

parameters, namely: (1) communication to computation ratio (CCR) between 0.1 and 1.0, (2) application size, i.e., task graphs comprising of 50-, 100-, and 300-tasks per graph, and (3) number of cores with 2-, 4-, 8-, and 16-cores execution scenarios. We have also performed experiments using selected real-world applications. The LeTS heuristic reduces overall execution time of applications by exploiting inter-task data locality. Results show that LeTS outperforms state-of-the-art algorithms in amortizing inter-task communication cost.

Keywords Runtime resource management · Parallel computing · Multicore scheduling · Homogeneous systems · Directed acyclic graph (DAG) · Embedded systems

Mathematics Subject Classification 68U01

1 Introduction

Recent trend in architecture design is to integrate more cores onto a single chip in order to meet the higher performance demand of computationally intensive applications [1,2]. With the increasing number of cores on a single processor die, the on-chip cache hierarchy that support these cores is also becoming more complex and larger. Consequently, the effects of non-uniform memory access are prevalent even on a single chip [3]. In such scenario, in order to reduce execution time and energy consumption of complex parallel applications, data access locality should be exploited. This is particularly important in task-based programming systems, where a computation is broken down into small code segments (i.e., *tasks*) and a scheduler decides when and where on the chip these tasks should execute. Specifically, a scheduler generates a task schedule by grouping tasks to execute on the same thread, and by applying ordering across the tasks. For load balancing among various cores, the runtime system may employ stealing to redistribute tasks from loaded threads to idle threads. However, to capture locality in a task-based system, the scheduling algorithm should be made *locality-aware*.

Programming of parallel systems for executing single application is more challenging than programming a single processor for a single application due to multiple reasons. One such reason is the larger degree of freedom in scheduling tasks over multiple computing resources that increases algorithmic complexity. Having many degrees of freedom implies many grouping and ordering choices for tasks. Another reason is that various code segments of application have precedence constraints and data dependencies among them and the complexity of many-core cache hierarchies makes the process all the more complicated. Thus, task grouping and ordering decisions taken by the scheduler must optimise locality across all cache levels, whether the hierarchy being shared or private. Due to arbitrary sizes, precedence constraints, and data dependencies among tasks, scheduling on multicore systems is considered as an NP-hard problem, i.e., it is not possible to find an optimal schedule in polynomial time (unless $NP = P$) [2,4–7]. Therefore, proposed scheduling algorithms are based on *heuristics* that try to reduce execution time on bounded computing resources [5,6].

While the heuristics usually provide fair results, there is no guarantee that solutions are always close to optimal [8,9].

With scheduling already an NP-hard problem in parallel systems, avoiding the high latency to access remote caches and main memory is increasingly critical for performance. The same holds for energy efficiency too: Moving data from a remote cache or from an off-chip memory requires 10 and 20 times more energy than an arithmetic operation [10], respectively. Thus, consensus exists [3,11,12] that memory access locality should be exploited to reduce execution time and energy to improve the performance of parallel systems.

In this paper, we propose a scheduling heuristic, called the *LeTS* (***L****ocality-awar****e*** ***T****ask* ***S****cheduling)* heuristic, for structured parallel programming systems, i.e., systems with explicit data and control dependencies across tasks. The contributions of this paper are as follows: (1) We develop a locality analysis framework and an offline list scheduler that takes the target application's profile information as input in the form of a directed acyclic graph (DAG) and generates schedules that are optimised for data access locality. (2) We then evaluate the effectiveness of the proposed scheduler using applications with 50-, 100-, and 300-tasks per graph benchmarks of Standard Task Graph (STG) [2,13]. We have also evaluated LeTS heuristic using selected real world application graphs. We analyze results for 2-, 4-, 8-, and 16-cores system execution scenarios, with variable degree of parallelism (DoP), and variable number of edges in the task graphs.

Rest of this paper is organized as follows: We discuss related work on task scheduling in Sect. 2. In Sect. 3, we present our system model and related definitions. Section 4 presents our proposed locality-aware task scheduling heuristic in detail. Experimental setup and results are discussed in Sect. 5. We conclude this paper in Sect. 6.

2 Background and related work

Task scheduling techniques are broadly classified into two main categories, i.e., *List scheduling* and *Clustering* [5,14,15]. Most of the algorithms that have been proposed for task scheduling fall into one of these two classes. Therefore, task scheduling algorithms can only be compared within their respective class. Both list scheduling and clustering can be considered as *heuristic skeletons*. An algorithm applying either of these techniques has the freedom to define the two, so far unspecified, criteria: the *priority* scheme for the nodes and the *choice criterion* for the processor [14]. Algorithms proposed under both list scheduling as well as clustering generate their schedule before the execution of applications, i.e., at compile time.

The list scheduling structure comprises of two distinct parts; in the first part, certain priority order/scheme is used to sort tasks while respecting their precedence constraints. Such sorting creates a topological order containing all *ready* tasks. In the second part, tasks are successively scheduled onto a processing unit (a core) that allows their earliest start time [2,14,16,17]. There are significantly large number of algorithms proposed in this category, such as [5,16–22]. The heterogeneous earliest finish time (HEFT) [18] is one of the well-known scheduling algorithms that uses recursive approach in the bottom-up direction in order to determine the order of nodes. Such node ordering is based on computation costs. This node order is then used to

process tasks. Critical path nodes are being preferred by HEFT, which leads it to Depth-First Search (DFS) based node ordering and execution subsequently. In a similar work proposed in [19], critical path nodes are scheduled first by the proposed algorithm and non-critical path nodes, their bottom level is considered for ordering.

Critical path/most immediate successors first (CP/MISF) algorithm proposed in [16] also uses the method of bottom level ordering. CP/MISF breaks ties in favour of the task that has higher number of successors. The constrained earliest finish time (CEFT) has been proposed in [17]. CEFT heuristic uses the constrained critical path (CCP) notion that refers to a task window representing only ready nodes at any given time instance. Critical paths are being calculated by CEFT at first and then the tasks in precomputed CCPs are scheduled using combined finish time of entire CCP, subsequently. Authors in [5] analyze priority schemes in which the node orders are decided based on the bottom level, which is the same as LeTS heuristic but at different granularity. Other metrics such as; communication of nodes and critical path are also important in determining such node ordering as discussed in [19,20]. Another heuristic proposed in [21] is called the dynamic critical path (DCP). DCP is based on a critical pattern traversal approach. Basic idea in this work is to minimize overall schedule length at each execution step by using *remaining* critical path. DCP produces the final schedule only after processing all nodes. The use of static graph analysis and parameters like node levels (bottom level and top level) stretched back to work proposed in [22] where authors have proposed the modified critical path (MCP) algorithm. The nodes under MCP are ordered by their bottom level and ties are broken in favour of successor nodes with larger bottom levels.

In rather recent work related to static task scheduling, authors in [23] have proposed *multi-objective list scheduling (MOLS)*, a general framework and heuristic algorithm for multi-objective static scheduling of scientific workflows in heterogeneous computing environments. Although the MOLS algorithm considers 04 different objectives (makespan, economic cost, energy consumption, and reliability), their results show that makespan has the largest impact on all other objectives as it is the only structure-dependent objective that preserves the precedence constraints of target application in the ordered list. This result supports LeTS heuristic's focus on optimizing mainly the makespan through static scheduling. MOLS uses similar workflow model of DAGs and *bottom-level (B-level)* criteria for ranking nodes as LeTS, which we have elaborated in Sect. 3. MOLS heuristic works in three distinctive phases namely; the constraint vector partitioning, the task ordering, and the task mapping phases, compared to two phase solution of LeTS. Authors in [24] present a list-based scheduling algorithm called predict earliest finish time (PEFT) for heterogeneous computing systems. PEFT claims to offer makespan improvements by introducing a look-ahead feature. PEFT heuristic relies on a so-called optimistic cost table for task prioritization and allocation, however, it does so without considering the availability of processors while computing cost table. Similar to LeTS, PEFT heuristic also uses randomly generated graphs with various characteristics and selected graphs of real-world applications to analyze results in terms of schedule length ratio, efficiency, and frequency of best results.

Clustering algorithms consider *collections or sets* of tasks to be mapped to appropriate processing resources [4–6,15,25–27]. Such a collection is termed as *cluster*. The clusters are then processed further to adapt for a bounded number of processing

resources. Among the earlier works on clustering, Kim and Browne [27] proposed a path clustering algorithm that select a longest path p that consists of previously unvisited edges in each iteration of clustering and merge every nodes of p into a single cluster and mark every edge in p as visited. Similarly, Sarkar [28] proposed a heuristic based on single edge clustering that sorts the edges in descending order of their communication cost. Then for each sorted edge, computes the schedule length considering the value of selected edge as zero. If the schedule length improves, then it merges the two nodes on which the selected edge is incident. The dominant sequence clustering (DSC) algorithm proposed in [15] takes an unbounded number of processors and creates clusters of tasks to schedule. DSC algorithms is proposed for homogeneous processing systems. The DSC algorithm merges the the clusters in order to adapt schedule to available computing resources. Similarly, authors in [25] propose an algorithm that perform level sorting, i.e., at any particular depth level in a task graph, the tasks are arranged in an order such that they are independent of one another. The algorithm then allocates processing resources using the earliest finish time of any given level. At any given level, the tasks offering smaller execution times are merged together in order to adhere to the number of available processors. The resource allocation is thus based on minimising the total computation and communication costs.

In recently published work, authors in [29] propose a clustering-based task scheduling algorithm called *clustering for minimizing the worst schedule length (CMWSL)* to minimize the schedule length in a large number of heterogeneous processors. CMWSL considers DAG-based application model and consists of four phases to statically schedule a DAG. It first derives the lower bound of the total execution time for each processor using the worst schedule length (WSL) and then the processor that minimizes the WSL is chosen for the cluster assignment target. In the next two phases, task clustering and scheduling is performed, respectively. CMWSL, compared to LeTS, takes more iterations to effectively decide on the clustering and task ordering.

Most of the proposed scheduling techniques work with a fine granularity, i.e., by computing priorities at *node-level* in the task graph. The LeTS heuristic assigns an execution order between topologically arranged nodes at a coarse-grain level by grouping nodes into *working task groups (WTGs)*, which favour intra-group data locality. Moreover, earlier proposed techniques base their prioritization mechanisms on the computation costs mainly and do not explicitly prioritise nodes in order to minimize inter-task communication. LeTS heuristic gives equal weight-age to both computation and communication costs while computing priorities. Section 4 gives details on the priority mechanism used by the LeTS heuristic.

3 Definitions and system model

In this section, we provide our system model, definitions, and properties/assumptions that are taken in this work.

3.1 Application model

A given program/application can be represented in the form of a *DAG*. In such representation, code segments are represented via nodes and inter-node dependencies are

represented via edges. A function $G = (V, E, w, c)$ represents such task graphs, where V is a set of nodes that represent a non-divisible sequential task of the program, i.e., $n \in V$. An edge $e_{i,j} \in E$ represents the precedence constraint between task n_i and n_j. Both control-flow and data-flow dependency can be represented through edges. The positive weight $w(n_i)$ of task $n_i \in V$ represents its computation time cost. Explicit communication cost between tasks n_i and n_j is represented by a non-negative weight $c(e_{i,j})$ on the edge $e_{i,j} \in E$ as shown in [2,6,30]. In this system model, the architecture and the application task graph is fully known (i.e., topology, computation cost, communication costs, data and precedence constraints) at compile-time. We do not impose any restrictions on the input–task graphs can have arbitrary structure, computation, and communication costs. Please note that, like all other static scheduling techniques, the LeTS heuristic works with pre-built/known task graphs only, i.e., it does not deal with the dynamic data.

3.2 Architecture model

We consider a set of homogeneous multicores, with their associated caches, connected by a communication network to run application DAGs. Considered system possesses the following properties/assumptions.

1. The parallel system does not have any workload other than the scheduled application task graph.
2. Execution is non-preemptive and allows one task at a time per core.
3. *Local* communication (i.e., between tasks executed on the same core) is negligible and therefore considered as *zero*. This is because, for parallel systems, remote communication is more expensive than local communication by one or more orders of magnitude [14]. Therefore, we consider local communication cost as negligible or zero.
4. Computing resources are not involved in communication, i.e., communication subsystem is dedicated.
5. Inter-task communication is performed concurrently; there is no contention taken into account for communication resources.
6. The communication network is fully connected. Every core can communicate directly with every other core via a dedicated identical communication link.

Given the identical processing units and the fully connected network of identical communication links, the system is completely homogeneous. Note that earlier research work has used such system model as well in order to analyze the performance of scheduling algorithms such as [2,14,18,24,31]. We consider this model to permit a fair comparison with state-of-the-art algorithms.

3.3 Definitions

In this section, we introduce some relevant definitions, which will be used throughout this paper.

Task Graph Paths: Multiple paths of arbitrary length exist in an application's task graph G, starting from source node to sink node. The total length of a path in a graph can be represented as cumulative weight of nodes and edges, starting from *source* node to the *sink* node, as shown in Eq. 1.

$$pl(p) = \sum_{n \in p, V} w(n) + \sum_{e \epsilon p, E} c(e) \tag{1}$$

The computational length of a path in the graph, i.e., without including communication, is the cumulative weight of nodes only as shown in Eq. 2.

$$(pl_w(p)) = \sum_{n \in p, V} w(n) \tag{2}$$

Nodes that belong to a single path p possess an inherently sequential order due to precedence among them, which prevents their concurrent execution. This precedence helps in interpreting the total path length $pl_w(p)$ to be the time a path takes for sequential execution of all its nodes. Moreover, when the communication cost between these nodes is also taken into account as inter-processor communication, the path length can be referred as *pl(p)*. Communication cost can no more be neglected when each node of p is executed on a different processor than its predecessor.

Critical Path (cp): The *longest* path in task graph in terms of execution time, starting from the *source* to the *sink* node, is referred as *Critical Path (cp)* as shown in Eq. 3.

$$(pl(cp)) = \max_{p \in G}\{(pl(p))\} \tag{3}$$

Length of critical path, based on the computational cost only, serves as a yard-stick or a lower bound on the minimum achievable execution time for the whole program, i.e., any scheduler cannot achieve a schedule length shorter than critical path length [2,5,14].

Node Levels: For any node $n \epsilon V$, there exist paths in the graph for which node n serves as the last node, whereas for some other paths it serves as the start node. All such paths can have arbitrary lengths. A *node level* is defined as the *length of the longest path* containing the concerned node. Two distinct levels can be defined for each node.

Top Level (tl(n)): It is the length of the longest path that ends at node n, while excluding its own computation cost *w(n)*. Top level can be expressed by Eq. 4. Here, *ance(n)* refers to the set of ancestor nodes of n and *source(G)* represents the root node of the graph. With no ancestor nodes, $tl(n) = 0$.

$$tl(n) = \max_{n_i \epsilon ance(n) \cap source(G)} \{pl(p(n_i \rightarrow n))\} - w(n) \tag{4}$$

Bottom Level (bl(n)): It is the length of the longest path that starts at node *n*, while including its own computation cost *w(n)*. Bottom level can be expressed by Eq. 5. Here, *desc(n)* refers to the set of descendent nodes of *n* whereas *(sink(G)* represents the exit node of the graph. With no descendant nodes, $bl(n) = w(n)$.

$$bl(n) = \max_{n_i \epsilon desc(n) \cap sink(G)} \{pl(p(n \rightarrow n_i))\} \tag{5}$$

Schedule Length (SL): Let S be a schedule for task graph $G = (V, E, w, c)$ on system P. The schedule length (SL) of S is given by Eq. 6

$$SL(S) = \max_{n \epsilon V}\{t_f(n)\} - \min_{n \epsilon V}\{t_s(n)\} \tag{6}$$

Here, $t_s(n)$ and $t_f(n)$ are the start and finish time for task node n, respectively. All schedules considered in this paper start at time unit 0; thus, $\min_{n \epsilon V}\{t_s(n)\} = 0$ and expression in Eq. 7 suffices as the definition of the schedule length. Alternative designations for schedule length that are commonly used in the literature are *makespan* and *execution time*.

$$SL(S) = \max_{n \epsilon V}\{t_f(n)\} \tag{7}$$

4 Locality-AwarE Task Scheduling (LeTS)

While mapping tasks to cores, a locality-aware scheduler should take into account both locality and load balancing in order to reduce execution time. Two approaches to construct such a scheduler are: (1) task grouping and (2) task ordering [3]. In the former approach, executing a group of tasks on cores that share one or more levels of cache captures data reuse across tasks. In the later approach, executing tasks in an optimal order minimizes the reuse distance of shared data between tasks, which makes it easier for caches to capture the temporal locality. Thus, constructing a locality-aware scheduler depends on understanding *how* task groups should be formed, and *when* the task ordering will matter.

The LeTS heuristic, being a work-conserving algorithm, combines these two approaches in its two distinct phases: a *working task group formation phase (WTG-FP)* and a *working task group ordering phase (WTG-OP)*. Both phases take a DAG as input with nodes (tasks) and edges with computation cost on nodes and communication cost across nodes on edges, respectively. In WTG-FP, the former phase, multiple working task groups (WTGs) are formed based on the parent-child relationship information available through the input task graph. Once the WTGs are formed using appropriate criterion to favour locality, in the later phase of WTG-OP, an inter-group ordering is defined using criterion that optimises resource utilisation (load balance). Note that WTG-OP follows the WTG-FP and intra-group ordering of tasks is not explicitly defined. Intra-group task ordering is captured in WTG-FP. In the following, we elaborate both these phases of LeTS heuristic in detail.

4.1 Working task group formation phase (WTG-FP)

This phase of LeTS heuristic ensures that an arbitrary number of tasks should be grouped together for execution such that the data reuse across those tasks is maximized, hence the locality. In order to form such WTGs, we statically analyze the task graph of target parallel application that is obtained using representative input data.

As stated in Sect. 3.3, intuitively, scheduling nodes from critical path first produces an effective schedule. However, at runtime, critical path nodes may not be ready while the resources are available to run them. A task with all its predecessor tasks completed is referred as *ready* task [17]. Hence, precedence constraints play an important role is creating a partial order of execution across tasks. In our static analysis of task graphs, we use this inherently present partial ordering to form WTGs. In the following, we explain how WTGs are formed.

The principle criterion to form WTGs within a task graph is: To identify tasks that possess a partial order due to precedence and, as soon as they become ready, could be executed *in-line* on the *same core* without getting *blocked on data*. That is, all data being shared across tasks within a WTG must be available in the cache hierarchy for reuse, thus reducing the communication cost to *local* or *zero*. Task graph is initially traversed in order to identify tasks that can potentially form a WTG. All paths, including critical paths, leading from start node to exit node are identified along with their lengths using Eqs. 1 and 3. Moreover, during the same traversal, the parent-child relationship of each node is learned in order to determine precedence. Nodes with single and multiple (more than one) parent tasks (aka *join* nodes) are also identified. This information is used by the WTG-FP in order to form working task groups.

Once all paths within a task graph are identified, each path is analyzed independently. On each path, nodes between the start node and the *first encountered join node* along the path forms a *temporary* WTG or *t-WTG* (excluding the join node itself). Such *t-WTGs* are formed on each path in the same fashion. Note that these *t-WTGs* may share start or *fork* node at this stage. All *t-WTGs* are compared for their respective lengths (which includes computation and communication costs). The graph is then pruned of the longest *t-WTG* that constitutes a final WTG. The process repeats itself unless all nodes become part of a WTG. While nodes are being removed from the task graph by becoming part of WTGs, some pseudo-edges are 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 left outside WTGs. Note that a join node cannot become part of any WTG that has any of its parent tasks as member. However, a join node can always become part of any WTG with its children tasks, if any. This is due to the fact that join nodes share data across parent tasks that belong to multiple paths, which leads to a situation where a WTG cannot complete its execution on the same core without getting blocked on data. The idea behind LeTS heuristic is to execute a WTG on the same core to reduce communication cost. For a join node, however, its parent tasks might be executed apart from each other either temporally (on the same core but at a different time) or spatially (on different cores), which requires a join node to wait for the data ready. Hence, the WTG formation phase is repeated between start node to join node in the first step and then between join node

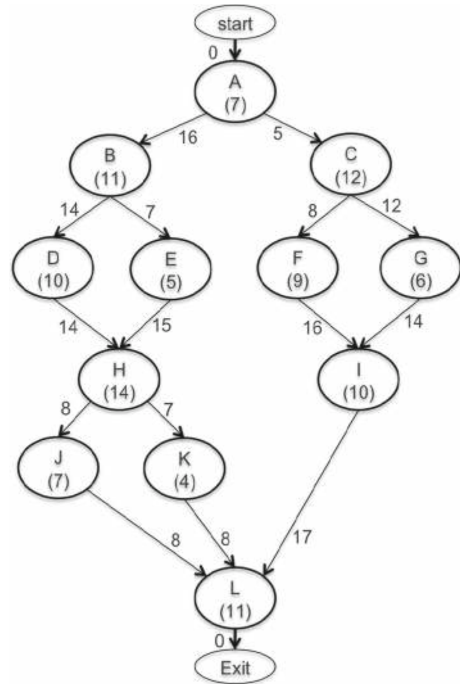


Fig. 1 Illustrative task graph to demonstrate working task group formation phase (WTG-FP) of LeTS heuristic

to join node until the whole graph is traversed and all nodes belong to any WTG. A WTG may contain any arbitrary number of nodes with a minimum of one member.

The task group formation criterion used by LeTS heuristic favours the execution of those tasks on the same core that (1) already have a partial order due to precedence and, in addition, (2) they either have longest communication cost across them or largest computation time requirements. In either case, grouping such tasks together improves data access locality in the cache hierarchy due to the fact that most recent data produced by the formerly executed tasks is readily available for the later tasks scheduled to execute on the same core in the order of precedence. Consider the graph shown in Fig. 1 with WTG-FP applied. In the first step, following paths are identified:

$$p_1 = \{A, B, D, H, J, L\},\ p_2 = \{A, B, D, H, K, L\}$$
$$p_3 = \{A, B, E, H, J, L\},\ p_4 = \{A, B, E, H, K, L\}$$
$$p_5 = \{A, C, F, I, L\},\ p_6 = \{A, C, G, I, L\}.$$

Lengths of paths are computed using Eq. 1: $pl(p_1) = 120$, $pl(p_2) = 116$, $pl(p_3) = 109$, $pl(p_4) = 104$, $pl(p_5) = 95$, and $pl(p_6) = 94$, respectively. In Fig. 1, nodes *H, I, and L* could be easily identified as join nodes in the graph. While analysing each path individually, following *t-WTGs* could be formed as shown in Fig. 2 (regions with broken lines): $t - WTG_1 = \{A, B, D\}$ from p_1 and p_2 (Fig. 2-1), $t - WTG_2 = \{A, B, E\}$ from p_3 and p_4 (Fig. 2-2), $t - WTG_3 = \{A, C, F\}$ from p_5 (Fig. 2-3), and $t - WTG_4 = \{A, C, G\}$ from p_6 (Fig. 2-4). Note that the *t-WTGs* do not include any

join node as a member. Starting from the *start* node, the WTG-FP completes its first iteration with all *t-WTGs* formed on each path *up to the first join node encountered*. At this stage, the length of *t-WTGs* is compared among themselves in order to form final *WTGs*. The longest *t-WTGs* as well as *mutually exclusive t-WTGs* form the final *WTGs*. Any two WTGs are said to be mutually exclusive if they do not have any member node in common. At this stage, however, *t-WTGs* may contain common nodes, such as node A is shared between all $t-WTGs$, node B is shared among $t-WTG_1$ *and* $t-WTG_2$ only, and node C is shared between $t-WTG_3$ *and* $t-WTG_4$ only. Mutually exclusive *t-WTGs*, if any, can form final *WTGs* in the same iteration.

Since the longest *t-WTG* is the $t-WTG_1$ (with a length of 58), therefore, it forms the final *WTG* and all its nodes are pruned of the graph and replaced by a pseudo-edge as shown in Fig. 3 (edges with broken lines). As there are no more mutually exclusive *t-WTGs*, the WTG-FP performs next iteration to identify remaining *t-WTGs*. In the second iteration, $t-WTG_5 = \{E\}$, $t-WTG_6 = \{C, F\}$, and $t-WTG_7 = \{C, G\}$ are formed. Since $t-WTG_5$ is mutually exclusive with $t-WTG_6$ and $t-WTG_7$, it can directly form a final *t-WTGs*. Between $t-WTG_6$ and $t-WTG_7$, $t-WTG_7$ happens to be the longest one, hence it forms another final *WTG* and both $t-WTG_5$ and $t-WTG_7$ are pruned of the graph as shown in Fig. 3-2. WTG-FP in its next iteration prunes of $t-WTG_8 = \{F\}$, which is a mutually exclusive *t-WTG*. As shown in Fig. 2-5, 2-6, starting from the join node *H*, two mutually inclusive *t-WTGs* are formed namely; $t-WTG_9 = \{H, J\}$ and $t-WTG_{10} = \{H, K\}$. Out of these two, $t-WTG_9$ forms the final WTG as compared to $t-WTG_{10}$ as shown in Fig. 3-4. Note that the join node *H* forms a *t-WTG* with one of its children nodes. In its final iteration, WTG-FP forms two more final WTGs with node *K* and *I* as single members, respectively, and prunes them of the graph as shown in Fig. 3-5. Sink node *L* forms the final WTG directly.

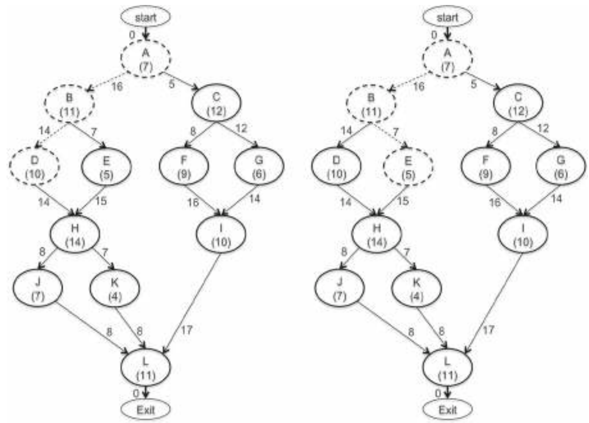
Please note that the WTG-FP is independent from the available computing resources (cores and their associated caches) and it entirely depends on the graph structure and precedence among the graph nodes. It is pertinent to highlight here that the scope of LeTS heuristic is limited to static tasking scheduling, therefore, the graph structure is considered as known.

4.1.1 Algorithms used by LeTS heuristic

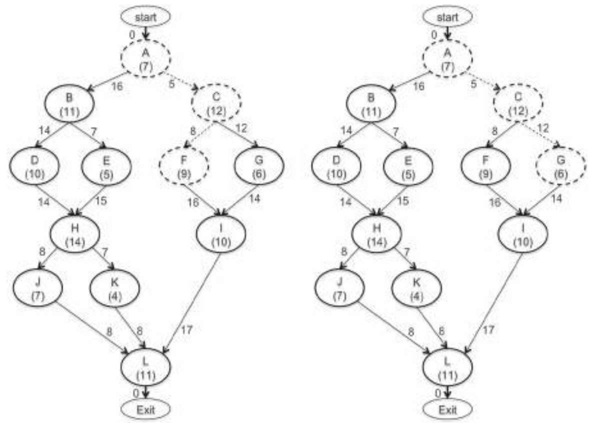
Figure 4 shows pseudocode of algorithm that is being used in identification of paths in given task graph. The algorithm implements a function named, $AddToPathList$ (v, p, P) for each node v that belongs to task graph G. In lines 6–14, the algorithm initialises a path starting with its first node and subsequently, each of its children nodes is analyzed. If a node is the last child of parent node v, then it becomes part of the created path p, otherwise, each such node creates a new path. Once all paths are created, LeTS heuristic can use these paths in WTG-FP and WTG-OP phases (Fig. 5).

The pseudocode for second algorithm is shown in Fig. 6, which creates working task groups by traversing the task graph. The algorithm takes input the set of paths created using algorithm shown in Fig. 4. After creating empty sets for temporary and final WTGs and initialising variables, the algorithm analyzes each path individually (lines 6–18). For each node belonging to each path, the algorithm verifies the number

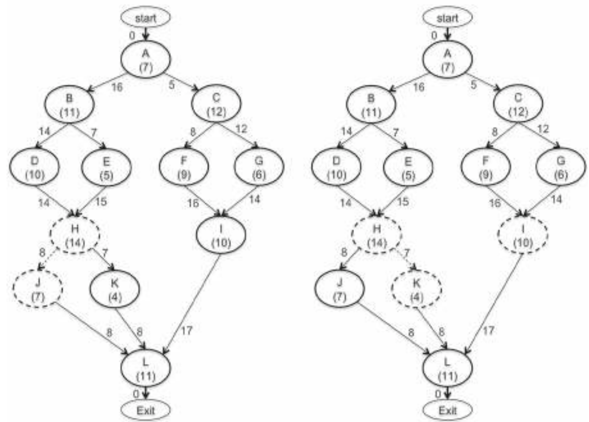
Fig. 2 Illustrative task graph to demonstrate Working Task Group Formation Phase (WTG-FP) of LeTS



1 - Graph after $t - WTG_1$ 2 - Graph after $t - WTG_2$



3 - Graph after $t - WTG_3$ 4 - Graph after $t - WTG_4$



5 - Graph after $t - WTG_5$ 6 - Graph after $t - WTG_6$

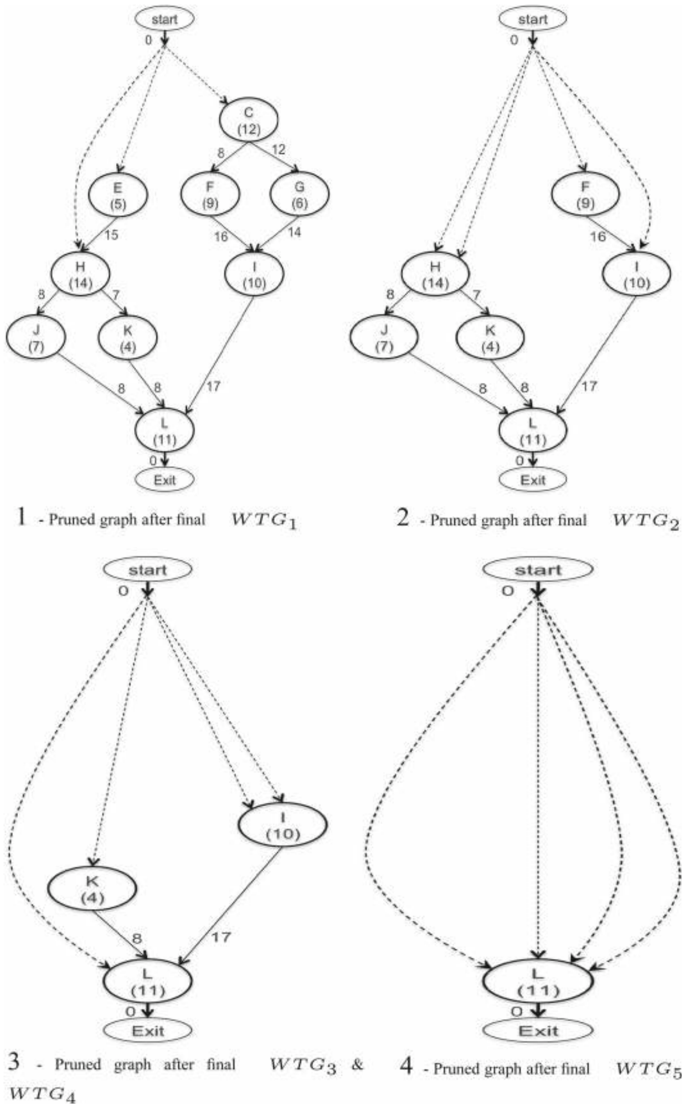


Fig. 3 Working Task Group (WTG) formation and graph pruning by LeTS

of its parent tasks. In case it has multiple parent tasks, a temporary working task group (*t-WTG*) is created and next path is selected for exploration. Otherwise, the algorithm continues to explore further nodes on the same path (lines 7–16). Once all paths are traversed, the algorithm sorts all *t-WTGs* present in *t-WTG-S* in descending order of their size and identifies mutually exclusive *t-WTGs*. All mutually exclusive *t-WTGs* form the final *WTGs* and the algorithm replaces them with pseudo-edges in the graph (lines 19–23). The process continues until all nodes in the graph become part of any WTG (lines 24–28).

Fig. 4 Pseudocode of algorithm identifying paths in the task graph

```
1:  v ← root node in G
2:  p ← ∅, current path
3:  P ← ∅, set of all paths in G
4:  function AddToPathList(v, p, P)
5:  p ← p ∪ v
6:  for each child node of v, n_i do
7:     if n_i is the last child then
8:        AddToPathList(n_i, p, P)
9:     else
10:       px ← all nodes in p, create a new path
11:       P ← P ∪ px
12:       AddToPathList(n_i, px, P)
13:    end if
14: end for
15: end function
```

Fig. 5 Pseudocode of execution time calculation algorithm used in LeTS

```
1:  n_1 ← Node being executed
2:  Execution-Time ← w(n_1)
3:  Parent ← parent nodes of n_1
4:  Processor ← processor assigned to n_1
5:  for each node n_i belonging to set Parent do
6:     P ← processor assigned to n_i;
7:     if Processor ≠ P then
8:        Execution-Time = Execution-Time + c(n_i,1);
9:     end if
10: end for
```

The working task group formation phase is based on traversing task graph using a *sliding window* with arbitrary window size. Since the size of window is caped between any two consecutive join nodes (or source node and first encountered join node for the start of graph), therefore, the window is very small and the iterations performed by WTG-FP are very fast too. Figure 7 shows the pseudocode representation of the algorithm used for scheduling WTGs obtained from WTG-FP. In order to amortize the communication cost within a WTG, it is essential to execute a given WTG on a single core. The present algorithm (lines 5–14) ensures that nodes within a WTG are sequentially executed on the same core without getting blocked on data. It traverses the set of ready nodes for each executed node to determine the next ready node in the WTG that is currently being executed. If a ready and executed node belong to the same WTG, then the core previously allocated to the executed node is assigned to that particular ready node. The algorithm further reduces the communication cost between a child node, that is also the first node of a ready WTG, and one of its parent nodes by assigning them the same core on the condition that the core is free (lines 15–33). Lastly, processors are assigned to remaining ready nodes. Pseudocode in Fig. 7 suggests that the complexity of LeTS heuristic is $O(N^3)$.

Fig. 6 Pseudocode of algorithm used to form WTGs in WTG-FP

```
1:  P ← PathSet of all paths in G
2:  WTG-S ← ∅, set of final working task groups
3:  t-WTG-S ← ∅, set of temporary working task groups
4:  WTGSize ← ∅, Size of each t − WTG in t-WTG-S
5:  ParentCount ← ∅, Number of parents of any Node
6:  for each path p_j in PathSet P do
7:     Create a t-WTG;
8:     for each node n_i belonging to path p_j do
9:        ParentCount ← number of parents of n_i;
10:       if n_i has ParentCount ≤ 1 then
11:          WTGSize_j = WTGSize_j + w(n_i) + c(n_{i−1,i});
12:       else if n_i has ParentCount > 1 then
13:          t-WTG ← {n_1 to n_{i−1}} ∈ p_j;
14:          t-WTG-S ← t-WTG-S ∪ t-WTG;
15:          break;
16:       end if
17:    end for
18: end for
19: Sort t-WTG-S in descending order of WTGSize
20: Search for mutually exclusive t-WTGs
21: Create final WTGs of the longest t-WTG and all
    mutually exclusive t-WTGs
22: WTG-S ← WTG-S ∪ final WTGs
23: Remove nodes belonging to final WTGs from graph
    and replace with pseudo-edge between source and
    sink nodes
24: if t-WTG-S ≠ ∅ then
25:    Goto step 4 with remaining nodes
26: else
27:    Traverse remaining graph till the Exit node
28: end if
```

$$c(n) = CCR \cdot w(n) \tag{8}$$

Figure 5 shows the pseudocode representation of the algorithm used to calculate time required by a node for its completion. LeTS is communication aware heuristic and works on the fact that communication cost between two nodes is minimal if both of them are executed on the same core. The algorithm adds communication cost to the execution time of the process only if both parent and child nodes get executed on different cores. If the communication cost is not incorporated in the input graph, it is systematically produced through Eq. 8. Here, CCR refers to *Communication to Computation Ratio* and is the sum of all edge weights (communication costs) divided by the sum of all node weights (computation costs) [5]. This is further elaborated in the experimental section where we study the impact of CCR, when it is varied from low to medium values, on scheduling length.

Fig. 7 Pseudocode of algorithm used for LeTS schedular

```
1: Ready ← set of all ready nodes
2: Comp ← set of all executed nodes
3: Temp ← ∅, empty set of nodes
4: assigned ← false, check if processor is assigned
5: for each node n_j belonging to set Comp do
6:    for each node n_i belonging to set Ready do
7:       if n_i and n_j belong to same WTG then
8:          Assign processor of n_j to n_i, break;
9:       else
10:         Add n_i to Temp;
11:      end if
12:   end for
13:   Ready ← Temp, Temp ← ∅;
14: end for
15: for each node n_k belonging to set Comp do
16:   P ← Processor assigned to n_k;
17:   if P is free then
18:      Children ← child nodes of n_k;
19:      for each node n_j belonging to set Children do
20:         for each node n_i belonging to set Ready do
21:            if n_i == n_j then
22:               Assign P to n_i, assigned ← true, break;
23:            else
24:               Add n_i to Temp;
25:            end if
26:            if assigned then
27:               break;
28:            end if
29:         end for
30:      end for
31:   end if
32:   Ready ← Temp, Temp ← ∅;
33: end for
34: Assign processors to remaining nodes in set Ready
```

4.2 Working task group ordering phase (WTG-OP)

The order in which nodes of task graph are considered for scheduling has a significant influence on the resulting schedule length. Gauging the importance of nodes with a priority scheme is therefore a fundamental part of scheduling schemes. Many existing scheduling techniques [5,8,18,32,33] evaluate the importance of nodes in different ways. The earlier a node is considered for scheduling, the earlier it can acquire a computing resource for its execution. The challenge, however, is to find priorities that well reflect the importance of the node. The provision of *relative importance* to nodes (rather than absolute importance w.r.t. each other) results in smaller *schedule length (SL)* of the application.

The LeTS heuristic uses the concept of *node levels* introduced in Sect. 3.3 and used in [6,14,30] to gauge the importance of nodes in an offline analysis. Although the bottom level bl(n), as defined, is a node-specific parameter and gives its relative distance from the sink node (or exit node), it is used for an *inter-WTG* ordering in a post WTG formation phase. Note that WTG-FP forms WTGs on individual paths by selecting nodes that can execute sequentially on the same core without getting

blocked on data. Therefore, *intra-WTG* node ordering is already captured in WTG-FP phase. Hence, it is the *inter-WTG* ordering that matters in order to optimise resource utilisation or load balance across cores and further reduce schedule length.

The LeTS heuristic allocates one WTG to one core at a time with no pre-emption and migration allowed, i.e., a WTG runs to its completion on the same core. As soon as the *first node* within a WTG gets *ready*, the whole WTG can be considered as *ready* and *non-blocking* (on data). Thus, *the WTG-OP compares the bottom levels of the first nodes of all ready WTGs and prioritises the allocation of WTG(s) having larger bottom levels.* A larger value for bottom level indicates the WTG belongs to a longer path (i.e., remaining critical path) in the graph. Such an inter-WTG priority mechanism ensures that WTGs are always selected from remaining critical paths in the graph and all paths advance in execution in a proportionate manner. Ties in WTG ordering are broken in favour of larger WTGs in size.

5 Experimental evaluation

In this section, we provide simulation results of LeTS heuristic and analyze its performance against the variation in application parameters. The simulations were performed on workstation with an Intel Xeon E5-2643 processor, operating at a fixed clock of 3.40 GHz, and 256 GB of RAM. The operating system used was Ubuntu 16.04 LTS with kernel Linux 4.2.0-42-generic. We have performed experiments and analysis of results using two types of target applications. We have demonstrated performance of LeTS heuristic using randomly generated benchmark task graphs from Standard Task Graph Set (STG) [13] that allow parameter variations as well as with selected real-world application task graphs.

STG is extensively used by [2,34,35] in their evaluations for scheduling algorithms. It provides randomly generated task graphs with variable application sizes and critical path lengths, allowing to experiment with diverse application behaviours. STGs also provide, along with their graphs, a pre-computed optimal computational schedule length (i.e., without incorporating communication costs) using exhaustive search method. We perform experiments for LeTS heuristic using application graphs that consist of 50-, 100-, and 300-tasks per graph. To validate our results, we have used 315 task graphs in total: 150 task graphs from 50-tasks per graph category, 150 task graphs from 100-tasks per graph category, and 15 task graphs from 300-tasks per graph category. Please note that in Figs. 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 and 19, we have shown results of only 40 *randomly selected* task graph instances out of 150 instances being tested in each case. This is for the sole purpose of improving readability of results. Experiments have been performed for all 315 task graphs.

Table 1 shows characteristics of STG graphs. For instance, in the category of 50 tasks per graph, the number of edges are minimum 46, maximum 953, and on average 262.02 per graph. The results are being evaluated on homogeneous multicore systems with hierarchical caches and connected by a contention-free communication network. We have considered 2-, 4-, 8-, and 16-cores execution scenarios for the execution of variable size application DAGs. Moreover, in each of these execution scenarios, we induce systematic variation in CCR in each task graph from 0.1 (i.e., communica-

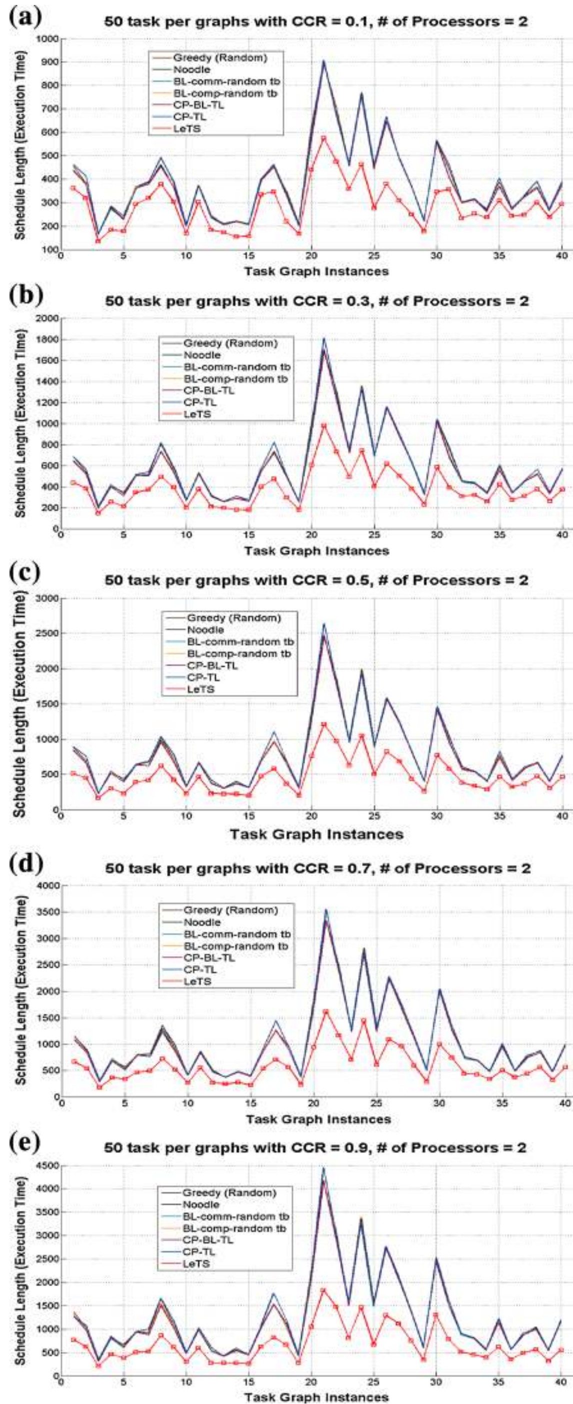


Fig. 8 Comparison of schedule length between LeTS and existing heuristics: 50-tasks/graph, 2-cores. **a** 50 Task/graph with 0.1 CCR, **b** 50 task/graph with 0.3 CCR, **c** 50 task/graph with 0.5 CCR, **d** 50 task/graph with 0.7 CCR, **e** 50 task/graph with 0.9 CCR

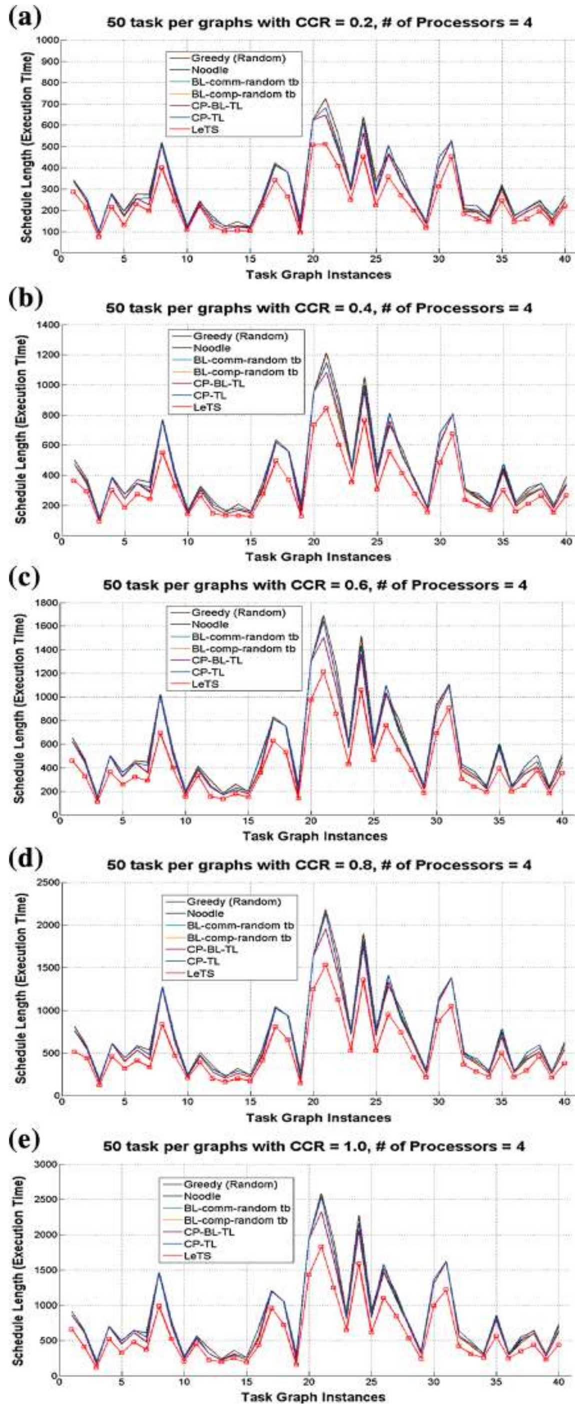


Fig. 9 Comparison of schedule length between LeTS and existing heuristics: 50-tasks/graph, 4-cores. **a** 50 Task/graph with 0.2 CCR, **b** 50 task/graph with 0.4 CCR, **c** 50 task/graph with 0.6 CCR, **d** 50 task/graph with 0.8 CCR, **e** 50 task/graph with 1.0 CCR

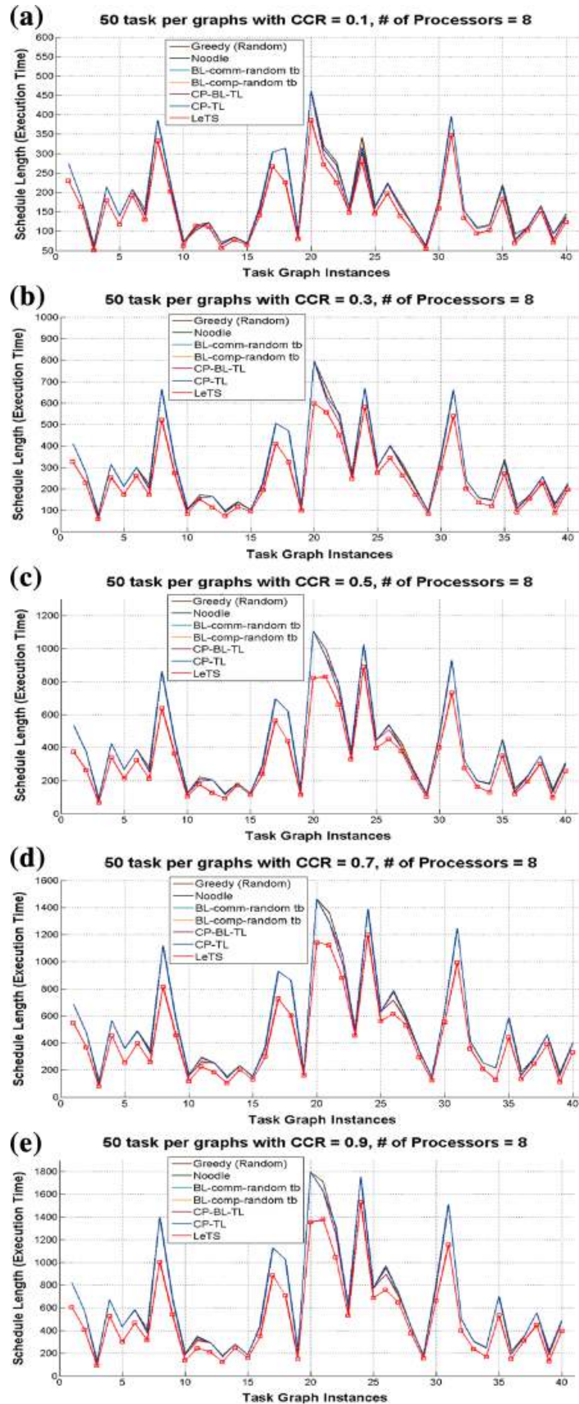


Fig. 10 Comparison of schedule length between LeTS and existing heuristics: 50-tasks/graph, 8-cores. **a** 50 Task/graph with 0.1 CCR, **b** 50 task/graph with 0.3 CCR, **c** 50 task/graph with 0.5 CCR, **d** 50 task/graph with 0.7 CCR, **e** 50 task/graph with 0.9 CCR

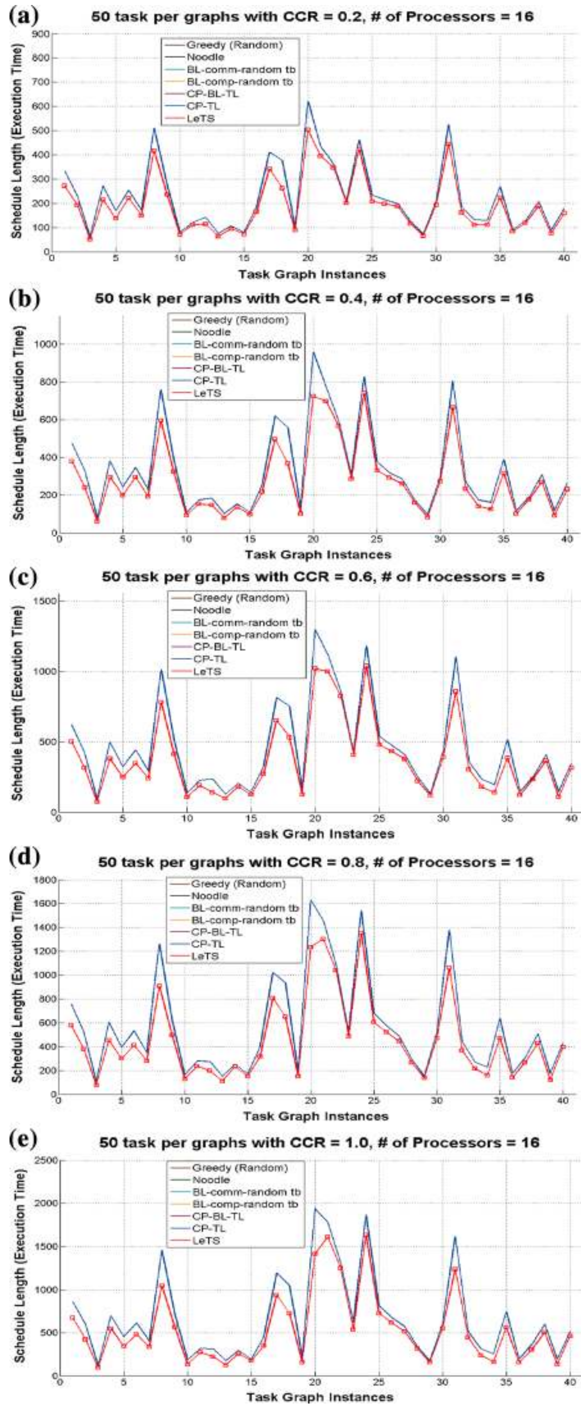


Fig. 11 Comparison of schedule length between LeTS and existing heuristics: 50-tasks/graph, 16-cores. **a** 50 Task/graph with 0.2 CCR, **b** 50 task/graph with 0.4 CCR, **c** 50 task/graph with 0.6 CCR, **d** 50 task/graph with 0.8 CCR, **e** 50 task/graph with 1.0 CCR

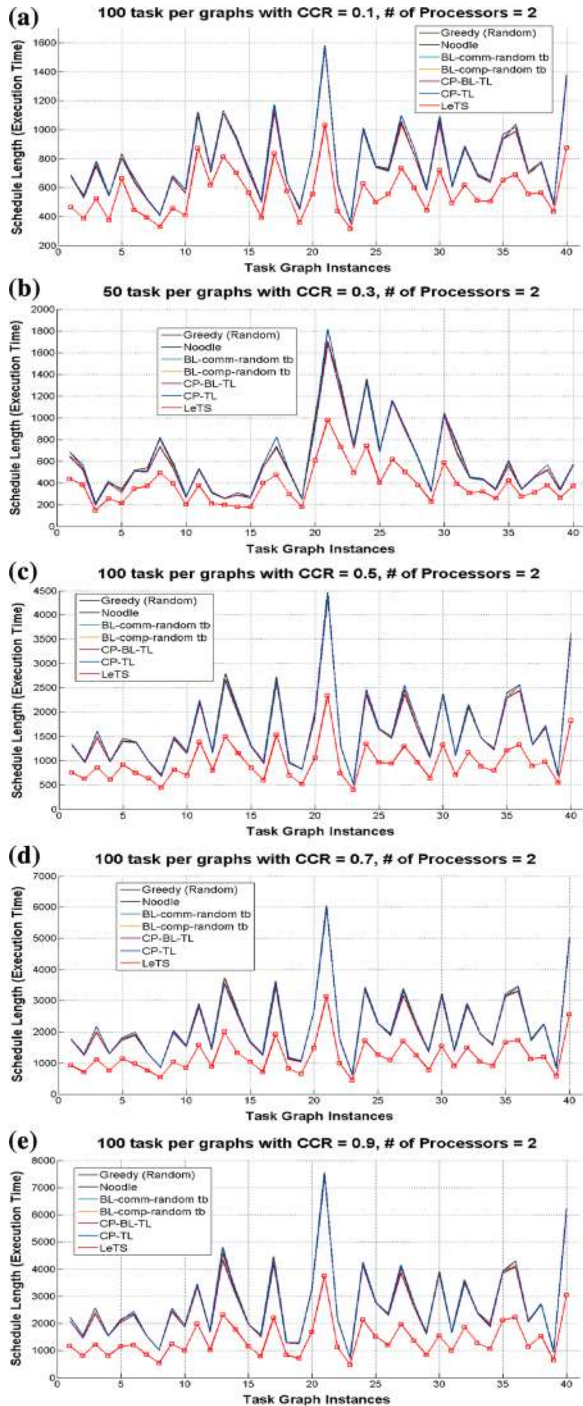


Fig. 12 Comparison of schedule length between LeTS and existing heuristics: 100-tasks/graph, 2-cores. **a** 100 Task/graph with 0.1 CCR, **b** 100 task/graph with 0.3 CCR, **c** 100 task/graph with 0.5 CCR, **d** 100 task/graph with 0.7 CCR, **e** 100 task/graph with 0.9 CCR

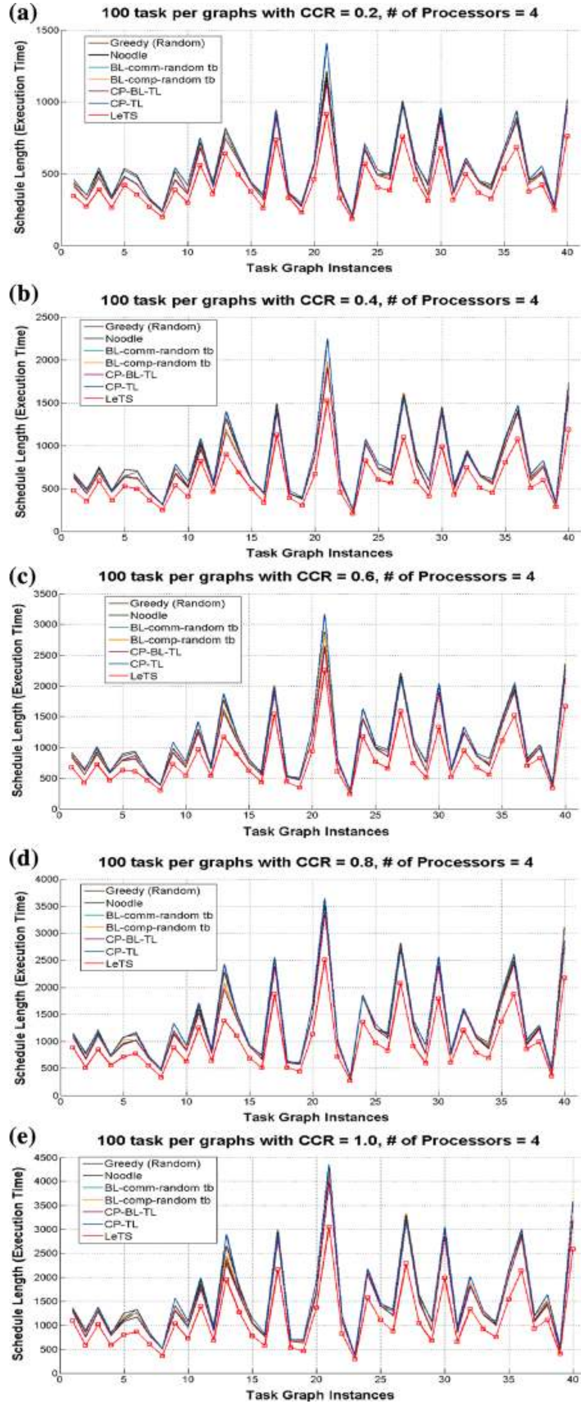


Fig. 13 Comparison of schedule length between LeTS and existing heuristics: 100-tasks/graph, 4-cores. **a** 100 Task/graph with 0.2 CCR, **b** 100 task/graph with 0.4 CCR, **c** 100 task/graph with 0.6 CCR, **d** 100 task/graph with 0.8 CCR, **e** 100 task/graph with 1.0 CCR

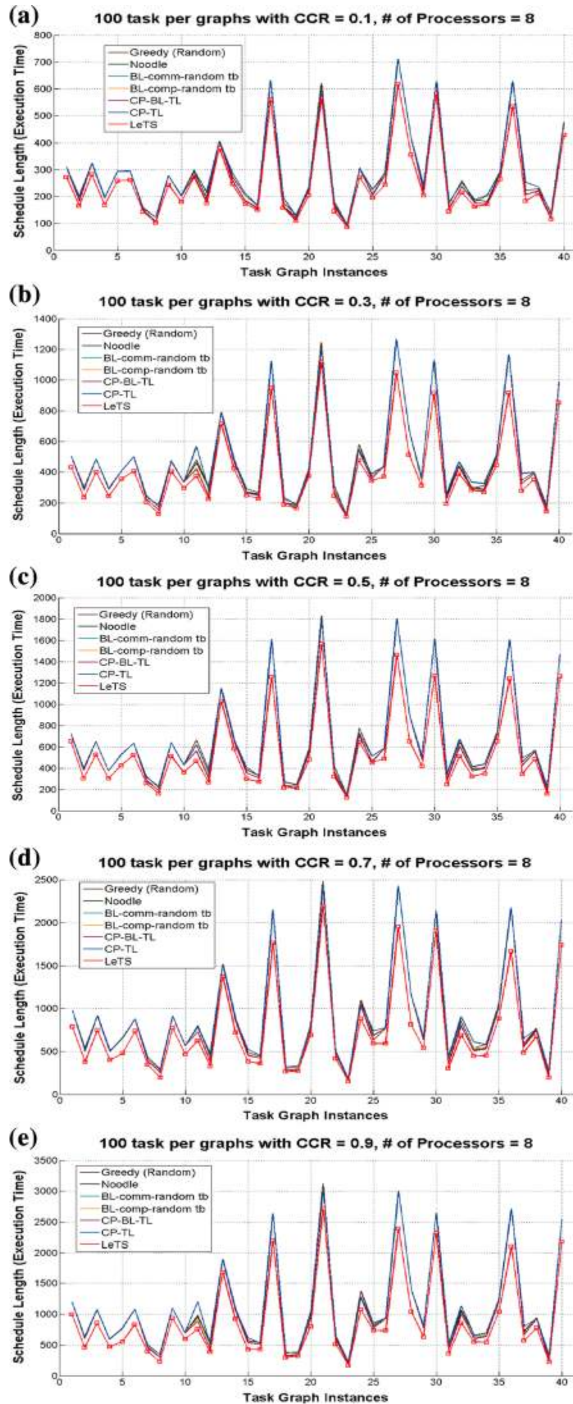


Fig. 14 Comparison of schedule length between LeTS and existing heuristics: 100-tasks/graph, 8-cores. **a** 100 Task/graph with 0.1 CCR, **b** 100 task/graph with 0.3 CCR, **c** 100 task/graph with 0.5 CCR, **d** 100 task/graph with 0.7 CCR, **e** 100 task/graph with 0.9 CCR

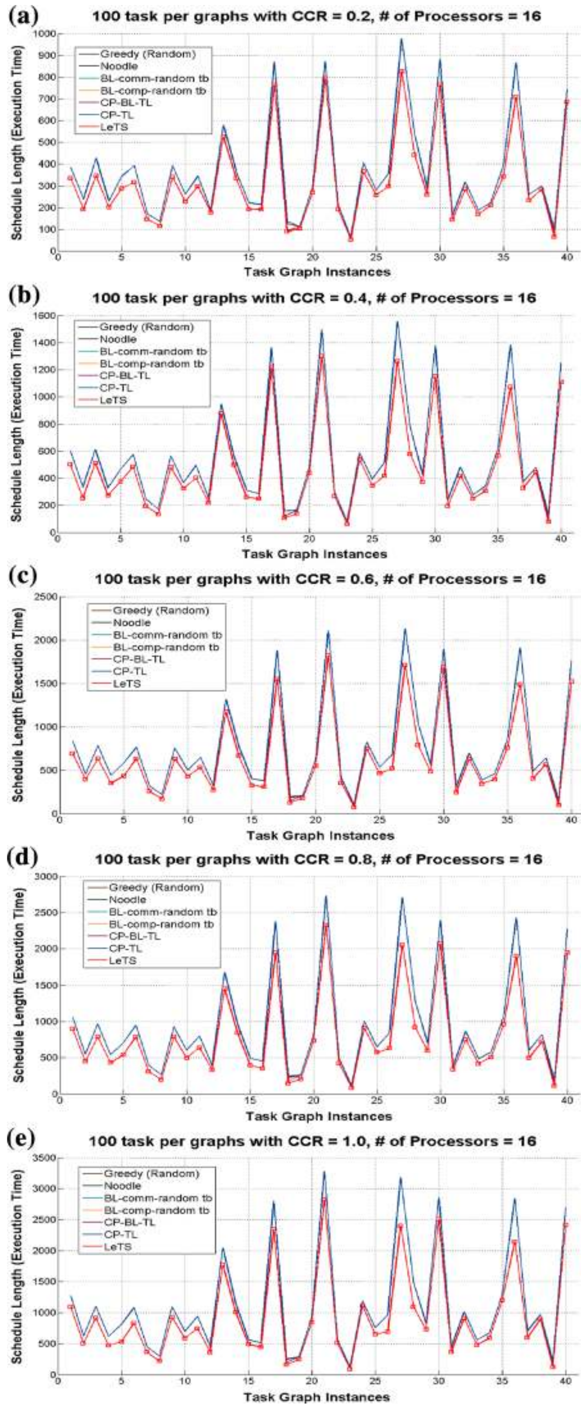


Fig. 15 Comparison of schedule length between LeTS and existing heuristics: 100-Tasks/graph, 16-cores. **a** 100 Task/graph with 0.2 CCR, **b** 100 task/graph with 0.4 CCR, **c** 100 task/graph with 0.6 CCR, **d** 100 task/graph with 0.8 CCR, **e** 100 task/graph with 1.0 CCR

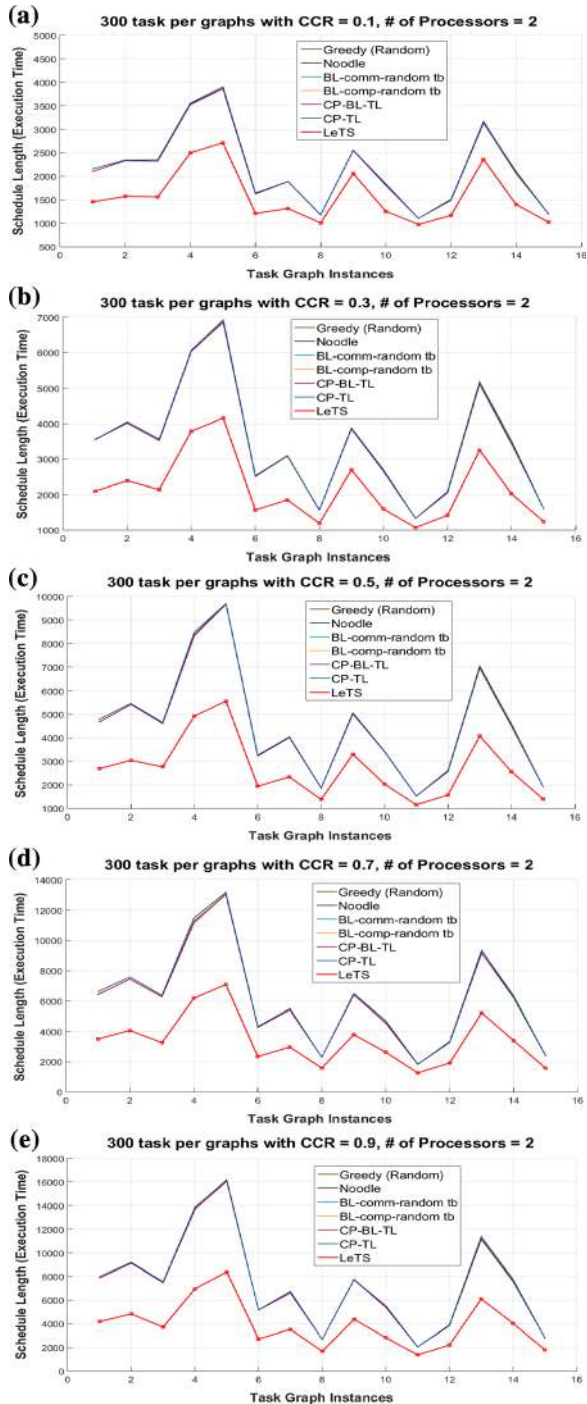


Fig. 16 Comparison of schedule length between LeTS and existing heuristics: 300-tasks/graph, 2-cores. **a** 300 Task/graph with 0.1 CCR, **b** 300 task/graph with 0.3 CCR, **c** 300 task/graph with 0.5 CCR, **d** 300 task/graph with 0.7 CCR, **e** 300 task/graph with 0.9 CCR

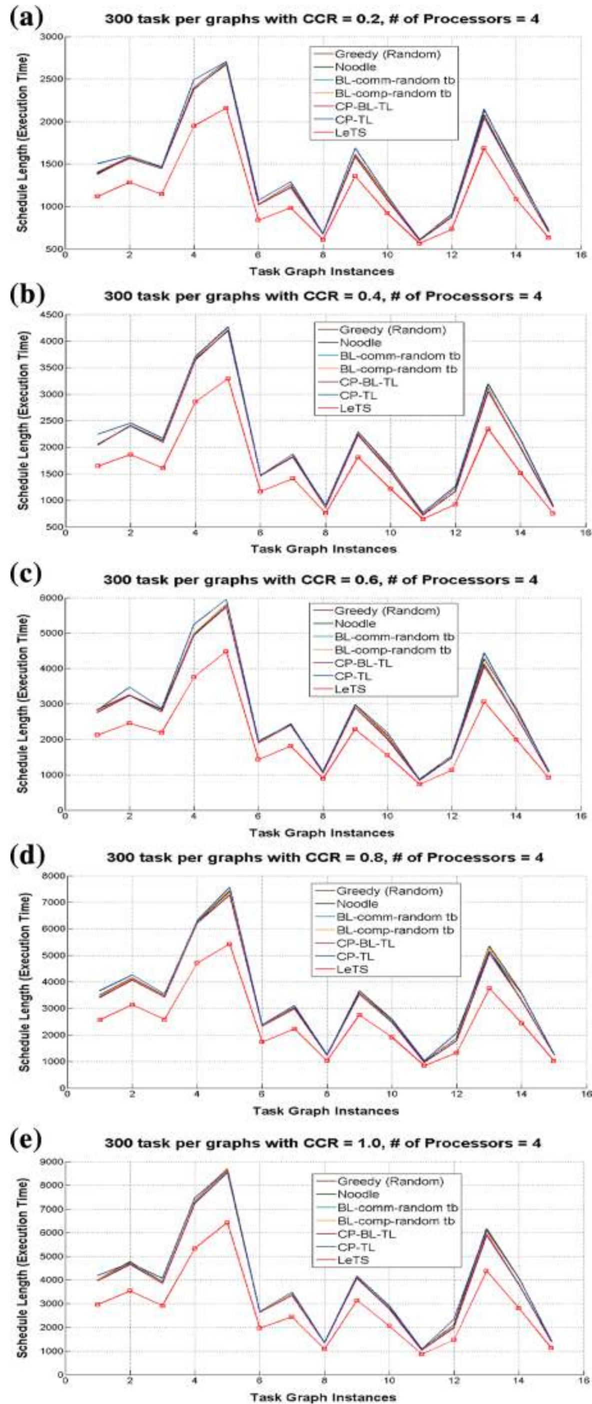


Fig. 17 Comparison of schedule length between LeTS and existing heuristics: 300-tasks/graph, 4-cores. **a** 300 task/graph with 0.2 CCR, **b** 300 task/graph with 0.4 CCR, **c** 300 task/graph with 0.6 CCR, **d** 300 task/graph with 0.8 CCR, **e** 300 task/graph with 1.0 CCR

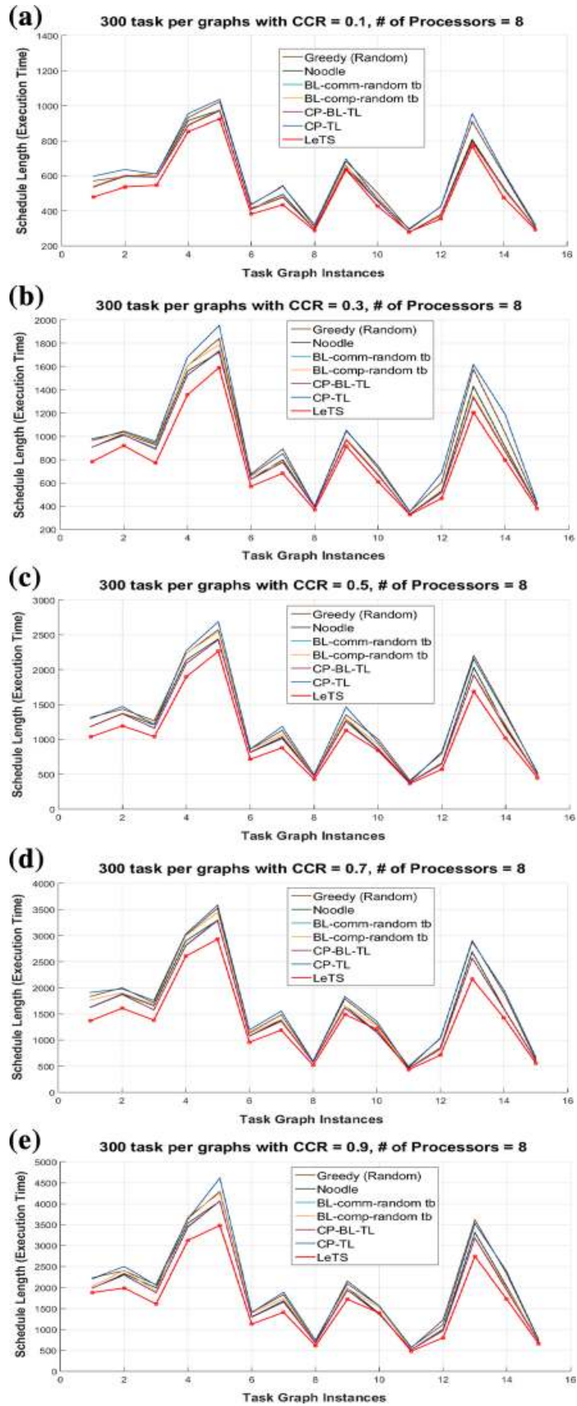


Fig. 18 Comparison of schedule length between LeTS and existing heuristics: 300-tasks/graph, 8-cores. **a** 300 Task/graph with 0.1 CCR, **b** 300 task/graph with 0.3 CCR, **c** 300 task/graph with 0.5 CCR, **d** 300 task/graph with 0.7 CCR, **e** 300 task/graph with 0.9 CCR

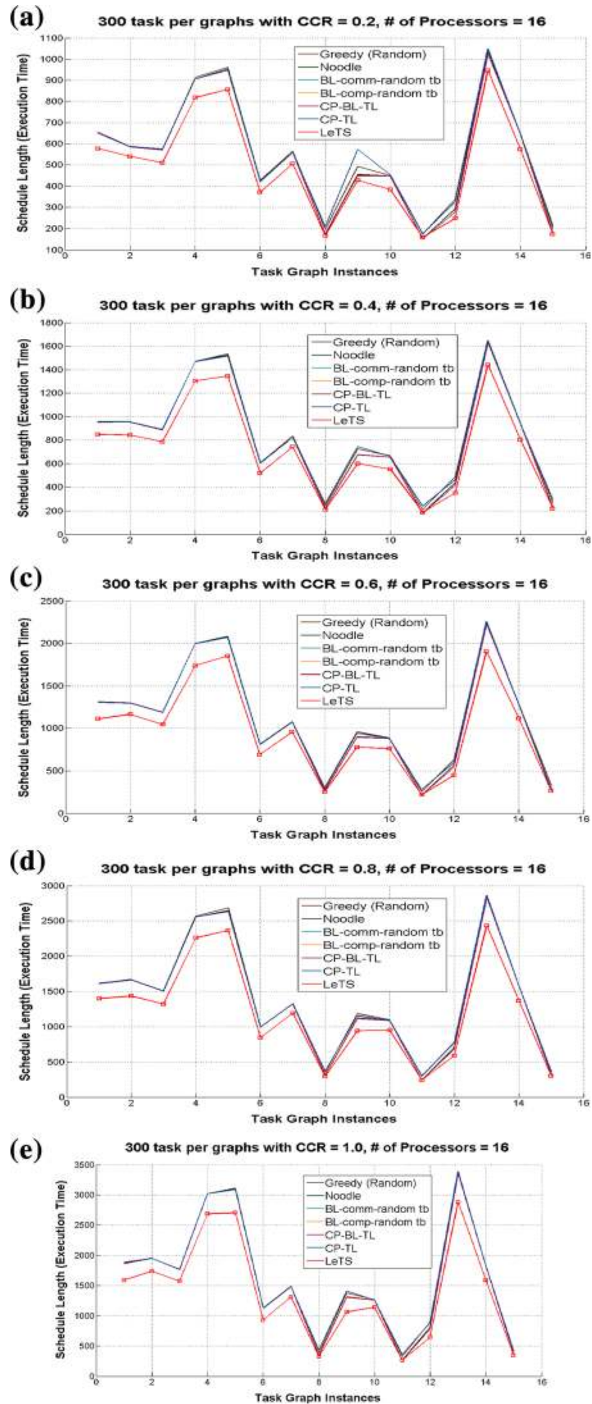


Fig. 19 Comparison of schedule length between LeTS and existing heuristics: 300-tasks/graph, 16-cores. **a** 300 Task/graph with 0.2 CCR, **b** 300 task/graph with 0.4 CCR, **c** 300 task/graph with 0.6 CCR, **d** 300 task/graph with 0.8 CCR, **e** 300 task/graph with 1.0 CCR

Table 1 Standard task graphs (STG) parameters

	50-tasks per graph	100-tasks per graph	300-tasks per graph
Number of edges	46/262.02/953	93/629.81/1677	309/1835.69/2958
Maximum predecessors	3/14.23/42	4/18.05/40	6/19.06/32
Maximum proc. time	7/22.27/70	7/24.09/83	8/27.81/76
Degree of parallelism	1.66/5.62/11.92	2.85/7.06/19.55	7.16/13.55/27.27

tion cost being 10% of computation cost of parent task) to 1.0 (i.e., communication cost being 100% of computation cost of parent task). Since STGs do not incorporate communication cost on edges in their graphs, we have calculated the values of communication cost using the method described in Sect. 4 (Eq. 8). Using Eq. 8, we calculate communication cost with the hypothesis that if a task is computing more then it has the tendency to produce more data that is shared with child/children task(s). Communication cost, however, can also be calculated as random. The variation in CCR allows to observe the impact of communication cost relative to computation cost while running the same task graph over different number of cores. We have explained the effect of each of these variations (i. e., number of cores, CCR, size of application task graphs etc) on Schedule Length (SL) and have subsequently compared the SL obtained under LeTS with all other heuristics described below.

We have performed experiments using three real-world applications to evaluate performance consistency of LeTS heuristic. First application is Sparse Matrix Solver with n tasks and m edges. Sparse matrices often appear in scientific or engineering applications when solving partial differential equations. Second application is a Robot Control application with n tasks and m edges, and our third application is SPEC95 subroutine fpppp with n tasks and m edges.

In order to quantify improvements offered by LeTS, we have performed comparative analysis of results using the following heuristics [5,14,36]:

- *Greedy (Random)*: There is no prioritization rule for Random Greedy and nodes are executed as they get ready irrespective of any specific order.
- *Noodle*: Priority for nodes decreases exponentially at each depth level in a graph on those paths for which ready nodes are being executed. That is, nodes on those paths that have remaining length greater than other paths have higher priorities. In case of ties, preference is given to nodes with larger task weights.
- *BL-comm*: Nodes that have the highest bottom level are prioritised over all other ready nodes. Both communication and computation costs are considered while calculating bottom level of a node.
- *BL-comp*: Nodes are prioritised on the basis of their decreasing computation bottom-levels. Here, only computation is taken into account while calculating bottom level of a node.
- *CP-BL-TL*: Under this heuristic, priority is given to the nodes belonging to critical path compared to all other ready nodes. All remaining ready nodes are ordered based on their bottom-level. In case of tie, it is broken in favor of node with the smallest top-level.

Table 2 Difference between SL of LeTS and other heuristics (2-cores)

CCR	Number of cores = 2		
	50-tasks/graph min./max./avg.	100-tasks/graph min./max./avg.	300-tasks/graph min./max./avg.
0.1	19/388/89.6	33/613/217.8	128/1146/572
0.3	40/867/202	82/1463/493	258/2701/1246
0.5	50/1334/308.3	111/2429/747	367/4102/1794.5
0.7	75/2144/443.6	155/3601/1071.3	560/5968/2624.6
0.9	108/2571/572	197/4408/1364.7	670/7667/3362.6

Table 3 Difference between SL of LeTS and other heuristics (4-cores)

CCR	Number of cores = 4		
	50-tasks/graph min./max./avg.	100-tasks/graph min./max./avg.	300-tasks/graph min./max./avg.
0.2	2/314/1.5	6/261/88.8	36/511/234.7
0.4	4/546/80.1	22/557/163.8	72/894/415.1
0.6	4/629/114.3	36/789/235	114/1265/599.8
0.8	4/818/152.1	12/1009/313.1	118/1839/786
1.0	3/978/181.7	29/1206/365.7	172/2104/931

- *CP-TL*: Similar to CP-BL-TL, priority is given to the nodes belonging to critical path compared to all other ready nodes, while all other ready nodes are ordered by their top-level.
- *LeTS*: Each node is assigned to a working task group and the groups are ordered by the methods described in Sect. 4.2. In case of tie, it is broken in favor of WTGs that are larger in size.

In the Sect. 5.1, we provide our observations and comparative analysis based on the impact of variation in CCR, size of application task graphs, and number of cores. Note that we do not claim LeTS being the optimal algorithm. The results obtained demonstrate relative gains in terms of SL. Moreover, STG [13] does not provide optimal schedule lengths with communication cost being applied, therefore, the optimal SL is unknown for the obtained results.

5.1 Comparative analysis

In this section, we have provided in details the impact of variations in application parameters such as CCR ration, application size, and the number of computing resources.

Table 4 Difference between SL of LeTS and other heuristics (8-cores)

CCR	Number of Cores = 8		
	50-tasks/graph min./max./avg.	100-tasks/graph min./max./avg.	300-tasks/graph min./max./avg.
0.1	1/186/19.3	1/93/26.5	6/59/31.5
0.3	2/440/43.4	1/250/63.8	5/168/79.3
0.5	1/651/64.4	1/464/101.1	15/235/112.8
0.7	1/727/92	1/525/141.3	23/397/175.9
0.9	9/914/116.6	1/621/180.2	32/578/225.07

Table 5 Difference between SL of LeTS and other heuristics (16-cores)

CCR	Number of cores = 16		
	TGs with 50 tasks min./max./avg.	TGs with 100 tasks min./max./avg.	TGs with 300 tasks min./max./avg.
0.2	2/314/27.7	1/161/43.3	3/92/52.3
0.4	5/546/51.2	1/318/81.1	3/189/99.2
0.6	4/629/73.3	1/448/115.1	6/323/143.8
0.8	2/818/97.1	11/658/155.9	14/396/177.9
1.0	6/978/114.47	3/906/183.3	15/474/213.36

5.1.1 Impact of variation in CCR

Since LeTS heuristic aims at amortizing mainly the communication cost, therefore, the impact of variations in CCR values is the most important observation in our results. We have varied CCR from 0.1 to 1.0 (i.e., 10–100% of computation cost of parent task) with a step size of 0.1 and studied its impact on the final schedule length for each algorithm. We have obtained results for CCR variation in 2-, 4-, 8-, and 16-cores execution scenarios with applications of 50-, 100-, and 300-tasks per graph. In favour of space, we have shown results with CCR values either 0.1, 0.3, 0.5, 0.7 and 0.9 in some cases or 0.2, 0.4, 0.6, 0.8 and 1.0 in other cases. Our observations on experiments performed for all other CCR values remain valid.

The collective results obtained for 2-core execution scenario for 50-, 100-, and 300-tasks per graph are shown in Figs. 8, 12, and 16, respectively. These graphs clearly show that LeTS outperforms other algorithms specifically when CCR is increased. We observe in Figs. 8, 12, and 16 that, for the same number of cores, increasing the number of tasks per graph did not effect the difference in obtained SL under LeTS and other heuristics despite increased CCR. That is, the pattern in obtained SL remains the same as long as the number of cores remain the same. The absolute value of SL is just increased by almost the same proportion as CCR increases. This is due to the fact that scheduler finds the same choices in terms of number of cores to run the application task graph. The difference in obtained SL under LeTS and other heuristics is more

significant for smaller number of cores, thus more amortization of communication cost is obtained in this case. This observation correlates with our other observations in Sect. 5.1.3, where we have discussed the effect of running larger applications with larger number of cores.

Table 2 shows the minimum, the maximum, and the average difference in SL obtained under LeTS and other algorithms for 2 core execution scenario. The entries in table clearly show that the difference between obtained SL also becomes significantly large with increasing values of CCR. For instance, in case of 50 tasks per graph, the average difference in SL for CCR=0.1 is 89.6 units, whereas for CCR=0.9, this difference increases to 572 units. Similar kind of results have also been depicted in case of applications with 100 and 300 tasks per graph as shown in Table 2 for 2-cores execution scenario. The CCR has been varied is the same fashion for 4, 8, and 16 cores scenarios as well and results are presented through Figs. 9, 10, 11, 13, 14, 15, 17, 18, and 19. Precise difference is SL is quantified in Tables 3, 4, and 5. Our observations on these results are similar in case of 2 cores scenario.

An important observation for the reader here is that, as the number of cores increases for the same application size (i. e., task graph size) and the same pattern of variation in CCR values, the *difference* in SL obtained under LeTS and other algorithms reduces. For instance, when the same application graphs with 50 tasks per graph are run on 2-, 4-, 8-, and 16-cores, the obtained difference in average SL significantly reduces as shown in Figs. 8, 9, 10, and 11 as well as in Tables 2, 3, 4, and 5. Similar observation can be made in cases where the application task graphs have 100 and 300 tasks per graph. LeTS, however, still outperforms other algorithms in these cases. Reasons for this reduction in difference of SL with increasing number of cores have been discussed in Sect. 5.1.2.

5.1.2 Impact of number of cores

Variation in the number of cores is another significant parameter that we have analyzed. For application task graphs ranging from 50 to 300 tasks per graph, we have run them on 2-, 4-, 8-, and 16-cores to observe if LeTS can still reduce the communication cost between tasks. With larger number of cores to execute a given task graph, the choice for the scheduler to run tasks on different cores naturally increases. With this increased choice, a work-conserving scheduler would run tasks as soon as cores get *free* and therefore, the impact of unavoidable communication cost between tasks will also increase.

Figures 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 and 19 show that when the number of cores increase, there are two variations in obtained SL that can be observed: (1) The overall SL decreases as there are more cores to execute any given application task graph with same CCR value. (2) The difference in SL produced by the LeTS and other heuristics decreases for any given application task graph with the same CCR. This is due to the fact that, with increased number of cores, LeTS heuristic faces the similar difficulty of maintaining its work-conserving nature while minimizing communication cost. As a result, tasks are executed on different cores and thus the communication cost between tasks cannot be amortized. Tables 2, 3, 4, and 5 show the minimum, maximum, and average quantified difference in SL obtained under LeTS and other

Fig. 20 Comparison of schedule length between LeTS and existing heuristics: sparse matrix solver application with CCR variation from 0.1 to 0.9 on 2, 4, 8, and 16 cores. **a** Sparse matrix solver application with CCR = 0.2, **b** sparse matrix solver application with CCR = 0.4, **c** sparse matrix solver application with CCR = 0.6, **d** sparse matrix solver application with CCR = 0.8, **e** sparse matrix solver application with CCR = 1.0

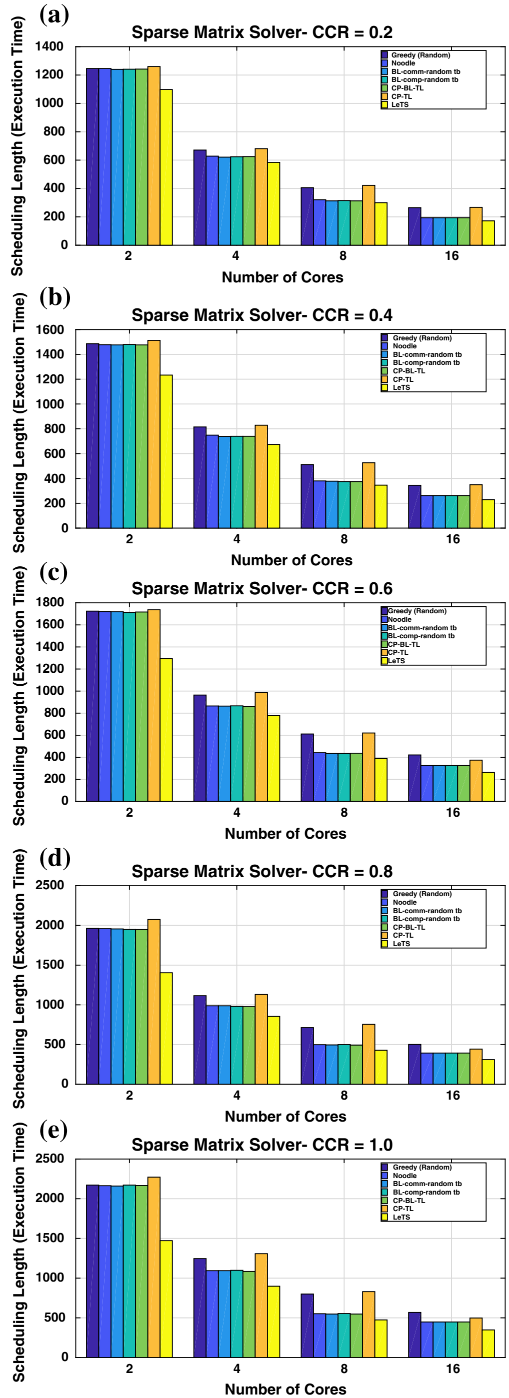
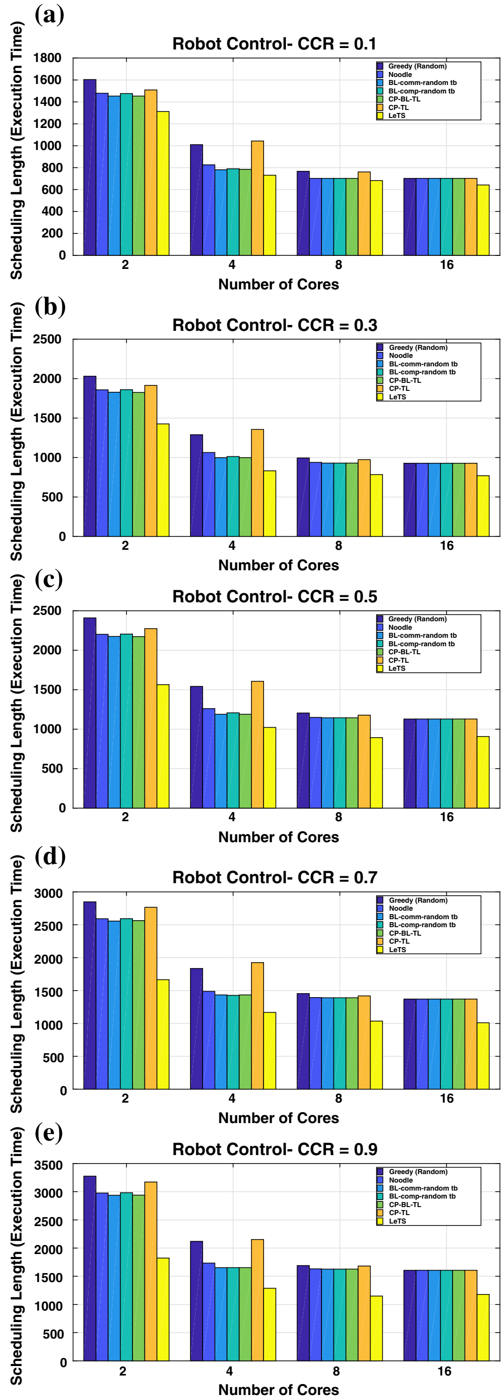


Fig. 21 Comparison of schedule length between LeTS and existing heuristics: robot control application with CCR variation from 0.1 to 0.9 on 2, 4, 8, and 16 cores. **a** Robot control application with CCR = 0.1, **b** robot control application with CCR = 0.3, (c) robot control application with CCR = 0.5, (d) robot control application with CCR = 0.7, (e) robot control application with CCR = 0.9



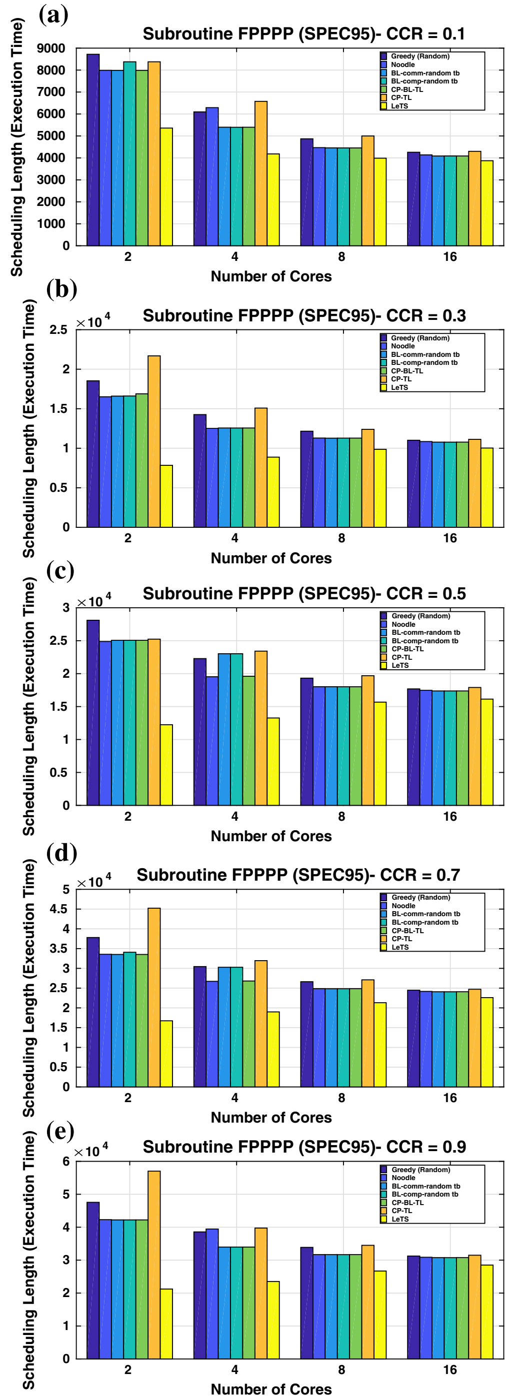


Fig. 22 Comparison of schedule length between LeTS and existing heuristics: SPEC955 Subroutine fpppp application with CCR variation from 0.1 to 0.9 on 2, 4, 8, and 16 cores. **a** SPEC95 subroutine application with CCR = 0.1, **b** SPEC95 subroutine application with CCR = 0.3, **c** SPEC95 subroutine application with CCR = 0.5, **d** SPEC95 subroutine application with CCR = 0.7, **e** SPEC95 subroutine application with CCR = 0.9

heuristics when the number of cores vary. We observe that for all three cases of different application sizes (50-, 100-, and 300-tasks per graph), when there are larger number of cores available, the difference between SL obtained under LeTS and other heuristics always reduces. Despite a decrease in the margin, LeTS still outperforms other heuristics.

5.1.3 Impact of application size

Another important aspect that we have analyzed is the impact of variation in application size, i.e., the task graph size. As stated above, we have analyzed a total of 315 task graphs out of which, we have analyzed 150 graphs with size of 50-tasks/graphs, 150 graphs with 100-tasks/graph, and 15 graphs with 300-tasks/graph.

We have observed that when larger applications (for instance, applications with 100- and 300-tasks/graph) are run on larger number of cores (for instance, 8- and 16-cores), the resultant SL produced by almost all algorithms is pretty much comparable as shown in Figs. 10, 11, 14, 15, 18, and 19. This is due to the fact that applications with larger number of tasks per graph offer sufficiently large degree of parallelism to execute tasks from multiple paths within a graph onto the available cores and therefore, makes it difficult to amortize communication cost between tasks. For smaller number of cores, however, this is not the situation as the degree of parallelism in application task graph is often larger than the cores available to run.

Primarily, we have performed experiments with randomly generated task graphs from STG [13] with self-introduced variation is parameters. However, in order to validate the performance consistency of LeTS heuristic, we have also performed experiments with 03 real world applications. We have analyzed the impact of variation in CCR and number of cores on these fixed sized applications. Figure 20, 21, and 22 show our results for Sparse Matrix Solver, Robot Control, and SPEC95 fpppp applications, respectively. The results obtained and discussed in Sect. 5.1.1, 5.1.2, and 5.1.3 were found consistent with the real applications. As Figs. 20, 21, and 22 show, LeTS heuristic performs better than other heuristics on real applications as well. As discussed in Sect. 5.1.2, the most significant difference in SL of real applications is also observed with smaller number of cores as larger number of cores to execute a given task graph increases the choice for scheduler to run tasks on different cores, thus amortizing communication cost becomes difficult, irrespective of CCR values. Variation in CCR affects magnitude of SL mainly in this case. These observations are valid for all three applications.

6 Conclusions and future work

The LeTS heuristic focuses on amortizing the communication cost between tasks by exploiting inter-task data locality and minimizes the overall schedule length (SL) of the target application. It takes into account both locality and load balancing in order to reduce the execution time of target applications in multi-level cache hierarchy. Extensive experimental evaluation, conducted using task graphs taken from Standard Task Graph (STG) shows that LeTS outperforms best known state-of-the-art algorithms

in amortizing the inter-task communication cost. We have performed experiments by varying three major performance parameters, namely: (1) CCR between 0.1 and 1.0, (2) Application size, i.e., task graphs that consist of 50-, 100-, and 300-tasks/graph, and (3) Number of cores with 2-, 4-, 8-, and 16-cores execution scenarios. Results show that conscious decision-making by the scheduler regarding data reuse across tasks and optimal task ordering to minimize reuse distance of shared data between tasks can play an important role in minimising inter-task communication cost. Our results show in depth how variations in the application size and number of cores available to run these applications impact the overall execution time. The LeTS heuristic achieves load balancing through its work-conserving nature and the WTG-OP phase of its working principle. The working principal of LeTS requires the application task graph to be known a priori. The future extensions of LeTS heuristic will work for heterogeneous computing systems and partially-known task graphs.

References

1. Wolf W, Jerraya AA, Martin G (2008) Multiprocessor system-on-chip (MPSoC) technology. IEEE Trans CAD ICs Syst 27(10):1701–1713
2. Bhatti MK, Oz I, Popov K, Brorsson M, Farooq U (2016) Scheduling of parallel tasks with proportionate priorities. Arab J Sci Eng 41(8):3279–3295. https://doi.org/10.1007/s13369-016-2180-9
3. Yoo RM, Hughes CJ, Kim C, Chen Y-K, Kozyrakis C (2013) Locality-aware task management for unstructured parallelism: a quantitative limit study. In: Proceedings of the twenty-fifth annual ACM symposium on parallelism in algorithms and architectures, ser. SPAA '13. ACM, New York, NY, pp 315–325. https://doi.org/10.1145/2486159.2486175
4. Grama A, Gupta A, Karypis G, Kumar V (2003) Introduction to parallel computing, 2nd edn. Pearson A. Wesley, Reading
5. Sinnen O, Sousa L (2004) List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. Parallel Comput 30(1):81–101
6. Sinnen O (2014) Reducing the solution space of optimal task scheduling. Comput OR 43:201–214
7. Bhatti MK, Belleudy C, Auguin M (2011) Hybrid power management in real time embedded systems: an interplay of DVFs and DPM techniques. Real-Time Syst 47(2):143–162
8. Shahul AS, Sinnen O (2010) Scheduling task graphs optimally with a*. J Supercomput 51(3):310–332
9. Sinnen O, Sousa LA (2005) Communication contention in task scheduling. IEEE Trans Parallel Distrib Syst 16(6):503–515
10. Dally W (2009) The future of GPU computing. In: The 22nd annual supercomputing conference
11. Hill M, Kozyrakis C (2012) Advancing computer systems without technology progress. In: DARPA/ISAT workshop
12. Consortium CC (2012) 21st century computer architecture. A community white paper
13. Set STG http://www.kasahara.elec.waseda.ac.jp/schedule
14. Sinnen O (2007) Task scheduling for parallel systems. Wiley, New York. ISBN 978-0-471-73576-2
15. Yang T, Gerasoulis A (1994) Dsc: scheduling parallel tasks on an unbounded number of processors. IEEE Trans Parallel Distrib Syst 5(9):951–967
16. Kasahara H, Narita S (1984) Practical multiprocessor scheduling algorithms for efficient parallel processing. IEEE Trans Comput C–33(11):1023–1029
17. Khan MA (2012) Scheduling for heterogeneous systems using constrained critical paths. Parallel Comput 38:175–193
18. Topcuouglu H, Hariri S, you Wu M (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parallel Distrib Syst 13(3):260–274
19. Kwok Y-K, Ahmad I (2000) Link contention-constrained scheduling and mapping of tasks and messages to a network of heterogeneous processors. Cluster Comput 3(2):113–124
20. Ahmad I, Kwok Y-K (1998) On exploiting task duplication in parallel program scheduling. IEEE Trans Parallel Distrib Syst 9(9):872–892

21. Kwok Y-K, 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
22. Wu M-Y, Gajski D (1990) Hypertool: a programming aid for message-passing systems. IEEE Trans Parallel Distrib Syst 1(3):330–343
23. Fard HM, Prodan R, Barrionuevo JJD, Fahringer T (2012) A multi-objective approach for workflow scheduling in heterogeneous environments. In: 2012 12th IEEE/ACM international symposium on cluster, cloud and grid computing (ccgrid 2012), pp 300–309
24. Arabnejad H, Barbosa J (2014) List scheduling algorithm for heterogeneous systems by an optimistic cost table. IEEE Trans Parallel Distrib Syst 25(3):682–694
25. Iverson MA, Ozguner F, Follen GJ (1995) Parallelizing existing applications in a distributed heterogeneous environment. In: HCW '95, pp 93–100
26. Bertrand Cirou EJ (2001) Triplet: a clustering scheduling algorithm for heterogeneous systems. New York. https://doi.org/10.1109/ICPPW.2001.951956
27. Kim S, Browne J (1988) General approach to mapping of parallel computations upon multiprocessor architectures. Unknown J 3:1–8
28. Sarkar V (1989) Partitioning and scheduling parallel programs for multiprocessors. MIT Press, Cambridge, MA
29. Kanemitsu H, Hanada M, Nakazato H (2016) Clustering-based task scheduling in a large number of heterogeneous processors. IEEE Trans Parallel Distrib Syst 27(11):3144–3157
30. Shahul AZ, Sinnen O (2010) Scheduling task graphs optimally with a*. J Supercomput 51(3):310–332
31. Deelman E, Singh G, Su M-H, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman GB, Good J, Laity A, Jacob JC, Katz DS (2005) Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Sci Program 13(3):219–237
32. Darte A, Robert Y, Vivien F (2002) Scheduling and automatic parallelization. BirkhŁuser, New York. ISBN 0-8176-4149-1
33. Suter F, Desprez F, Casanova H (2004) From heterogeneous task scheduling to heterogeneous mixed parallel scheduling. In: Euro-Par 2004 parallel processing, pp 230–237
34. Orsila H, Kangas T, Salminen E, Hamalainen TD, Hannikainen M (2007) Automated memory-aware application distribution for multi-processor system-on-chips. JSA 53(11):795–815
35. de Langen P, Juurlink B (2009) Leakage-aware multiprocessor scheduling. J Signal Process Syst 57(1):73–88
36. Bhatti MK, Oz I, Popov K, Muddukrishna A, Brorsson M (2014) Noodle: a heuristic algorithm for task scheduling in MPSoC architectures. In: 2014 17th Euromicro conference on digital system design (DSD). IEEE, pp 667–670