Arab J Sci Eng (2016) 41:3279–3295
DOI 10.1007/s13369-016-2180-9

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

Scheduling of Parallel Tasks with Proportionate Priorities

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

Received: 30 October 2015 / Accepted: 19 April 2016 / Published online: 21 May 2016

Abstract Parallel computing systems promise higher performance for computationally intensive applications. Since programmes for parallel systems consist of tasks that can be executed simultaneously, task scheduling becomes crucial for the performance of these applications. Given dependence constraints between tasks, their arbitrary sizes, and bounded resources available for execution, optimal task scheduling is considered as an NP-hard problem. Therefore, proposed scheduling algorithms are based on *heuristics*. This paper presents a novel list scheduling heuristic, called the *Noodle* heuristic. Noodle is a simple yet effective scheduling heuristic that differs from the existing list scheduling techniques in the way it assigns task priorities. The priority mechanism of Noodle maintains a *proportionate fairness* among all ready tasks belonging to all paths within a task graph. We conduct an extensive experimental evaluation of Noodle heuristic with task graphs taken from Standard Task Graph. Our experimental study includes results for task graphs comprising of 50, 100, and 300 tasks per graph and execution scenarios with 2-, 4-, 8-, and 16-core systems. We report results for average Schedule Length Ratio (SLR) obtained by producing variations in Communication to Computation cost Ratio. We also analyse results for different degree of parallelism and number of edges in the task graphs. Our results demonstrate that Noodle produces schedules that are within a maximum of 12 % (in worst-case) of the optimal schedule for 2-, 4-, and 8-core systems. We also compare Noodle with existing scheduling heuristics and perform comparative analysis of its performance. Noodle outperforms existing heuristics for average SLR values.

Keywords List scheduling · Static task scheduling · Directed acyclic graph (DAG) · Multiprocessor · Multicore · Parallel computing

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

[1] Information Technology University, Lahore, Pakistan

[2] Marmara University, 34722 Istanbul, Turkey

[3] SICS ICT, Isafjordsgatan 22, Box 1263, 164 29 Kista, Sweden

[4] KTH Royal Institute of Technology, 100 44 Stockholm, Sweden

[5] COMSATS Institute of Information Technology (CIIT), Lahore, Pakistan

1 Introduction

Over the years, the trend in architecture design is to integrate more processing elements onto a single chip to meet the higher performance demand of computationally intensive applications [1]. Programming such parallel systems for the execution of a single application becomes more challenging than programming a single processor. The target application must be divided into subtasks to allow the distribution of the application's computational load among the processing units. This has led to extensive work on software parallelisation techniques that can better exploit multicore and multiprocessor systems for higher application performance.

Parallelisation of a programme involves three steps, namely task decomposition, inter-task dependence analysis, and scheduling [2]. The process of assigning tasks to the processing units (i.e. spatial assignment) and defining their execution order (i.e. temporal assignment) is referred to as task scheduling [3,4]. Generally, there are dependences between the tasks that impose a partial order on their execution. Scheduling algorithms must also ensure that tasks adhere to this partial order, which is essential for the correct execution of the application [3].

Due to arbitrary task sizes, precedence constraints, and bounded resources for execution, task scheduling is considered as an NP-hard problem, i.e. it is not possible to find an optimal schedule in polynomial time (unless $NP=P$) [2–6]. Therefore, scheduling algorithms have been proposed based on *heuristics* that try to produce near-optimal schedules for bounded computing resources [5,6]. While the heuristics usually provide fair results, there is no guarantee that the solutions are always close to optimal [3,4,7]. Therefore, heuristic algorithms should be consistent in their performance and provide good solutions within acceptable bounds with respect to the optimal solution.

Most of the proposed task scheduling algorithms fall into one of the two broad categories, namely List Scheduling and Clustering [3,5]. The most common category, however, is the traditional list scheduling [4,5,8–10]. The basic structure of list scheduling is rather simple and consists of two parts. The first part sorts the tasks of the application according to certain priority scheme while respecting their precedence constraints such that the resulting task list is in a topological order containing all *ready* tasks. A task is said to be ready, if all its precedence constraints are met. In the second part, each task on the list is successively scheduled onto a processing unit (a core) chosen for the task that allows its earliest start time. Note that the priorities for tasks are determined *a priori* and they do not depend on the status of the list or a partial schedule. The computed priorities remain *static* for tasks, i.e. they do not change at runtime.

In this paper, we present a novel list scheduling heuristic, called the *Noodle (No Node is Left Behind)* heuristic, which provides an alternate and effective way of attributing task priorities compared to the existing priority attribution schemes. The priority mechanism of Noodle heuristic maintains a *proportionate fairness* among all ready tasks belonging to *all* paths within a task graph. Thus, no particular path in a task graph is either completely stranded or unnecessarily allowed the execution of deeply spawned tasks. The Noodle heuristic uses properties of task graph, which are detailed in subsequent sections, to attribute task priorities such that, on one hand, Noodle's resulting schedule minimises the overall execution time for the target application and on the other hand, it maximises the utilisation of constrained resources (cores) available to the application. We summarise the main contributions of this work as follows:

– We propose a novel list scheduling heuristic for parallel applications represented as directed acyclic task graphs. The proposed heuristic combines the benefits of both depth-first and breadth-first approaches while assigning priorities to ready tasks.
– We conduct experimental evaluation of Noodle heuristic with task graphs with 50, 100, and 300 tasks per graph taken from Standard Task Graph (STG). We analyse results for execution scenarios with 2-, 4-, 8-, and 16-core systems and different degree of parallelism and number of edges in the task graphs. We report results demonstrating that Noodle produces schedules that are within a maximum of 12 % (in worst-case) of the optimal schedule for 2-, 4-, and 8-core systems. We also compare Noodle with existing scheduling heuristics and perform comparative analysis of its performance. Noodle outperforms existing heuristics for average value of Schedule Length Ratios (SLRs).

Rest of this paper is organised as follows: Section 2 provides related research work on task scheduling heuristics. Section 3 presents our system model and related definitions. Section 4 presents the Noodle heuristic in detail. Section 5 provides experimental support and explanation of results and Sect. 6 concludes this paper.

2 Background and Related Work

While there are many comparisons of scheduling algorithms, heuristics are compared in classes. List scheduling [3] is traditionally the most studied heuristic class of algorithms. For instance, The Heterogeneous Earliest Finish Time (HEFT) scheduling algorithm [9] uses a recursive approach in the bottom-up direction to determine the nodes ordering, which are based on computation costs. The tasks are then processed in accordance with their node order. HEFT is basically built on the notion of preferring the critical path nodes which leads it to DFS-based ordering of nodes and subsequent execution. Authors in [11] propose an algorithm, where critical path nodes are scheduled first and non-CP nodes are scheduled according to their bottom level first. Another algorithm, the Critical Path/Most Immediate Successors First (CP/MISF) algorithm by authors in [12], is also based on bottom-level node ordering with ties being broken by giving precedence to the node with the higher number of successors. Authors in [13] propose a scheduling heuristic for heterogeneous systems named Constrained Earliest Finish Time (CEFT). Their heuristic is based on the notion of a constrained critical path (CCP), which is a small task window representing ready tasks at one instance. CEFT finds critical paths in the DAG, and subsequently, the tasks in the CCPs are scheduled using the finish time of the entire CCP. Authors in [5] analyse various priority schemes that are based on node orders according to the (computation) bottom level augmented with metrics based on the entering communication of a node and critical path-based orders as proposed by authors in [11,14].

The Dynamic Critical Path (DCP) scheduling algorithm presented in [15] is based on a critical pattern traversal approach. It attempts to minimise the schedule length at each step by using remaining critical path. A final schedule is not produced until all the nodes have been processed. The DCP uses the Absolute Earliest Start Time (AEST) and the Absolute Latest Start Time (ALST) that represent, respectively, the possible execution of a task at the earliest or the latest time. These values are computed through breadth-first traversal of the task graph. Another algorithm, called the Modified Critical Path (MCP) heuristic, is proposed by authors in [16], where the nodes are ordered by their bottom level and for nodes with equal bottom levels, the bottom level of the successor nodes are considered and so on.

Clustering algorithms [17] on the other hand, consider collections of tasks termed as *clusters* to be mapped to appropriate processing resources. In general, these algorithms work for homogeneous type of systems with unbounded number of processors. The clusters are then processed further to adapt for a bounded number of processors. For instance, the Dominant Sequence Clustering (DSC) algorithm in [17] schedules tasks for unbounded number of processors by creating clusters of tasks. It works for homogeneous processing systems and also requires the clusters to be merged so as to adapt schedule to the existing number of processors. Another algorithm proposed in [18] works by performing level sorting. The level sorting arranges the tasks in an order so that the tasks at one particular level of the task graph are independent of each other. This algorithm allocates the processor using the minimum value of the finish time of a level. At one level, the tasks with low execution costs are merged to conform to the number of processors. The processor assignment then uses the criteria based on minimising the sum of execution costs and the communication costs.

3 System Model and Definitions

3.1 Application Model

Given an application programme, we can represent the programme as a *Directed Acyclic Graph (DAG)* in which the nodes represent code segments, and edges represent dependencies among the segments as shown in Fig. 1. Such a task graph can be represented by $G = (V, E, w, c)$, where each node $n \in V$ represents a nondivisible sequential task of the programme. An edge $e_{i,j} \in E$ in Fig. 1 represents the precedence constraint between task n_i and n_j. Such precedence can be due to both data or control-flow dependency. The positive weight $w(n_i)$ of task $n_i \in V$ represents its computation workload in terms of time. A nonnegative weight $c(e_{i,j})$ on the edge $e_{i,j} \in E$ is used to represent an explicit communication cost between tasks n_i and n_j, such as used in [6,19].

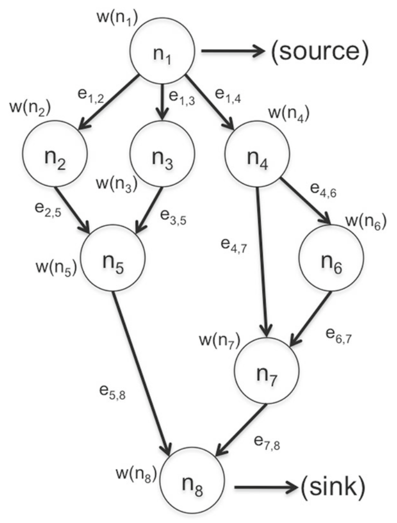


Fig. 1 Illustration of a synthetic application represented as Directed Acyclic Task Graph (DAG)

In our system model, we presume that the computation and communication structure of the programme and the target parallel system are completely known at compile time. No restrictions are imposed on the input task graph—it may have arbitrary computation and communication costs as well as an arbitrary structure.

3.2 Architecture Model

Our model for target parallel system consists of a set of identical processors connected by a communication network. The system has the following properties:

- The parallel system is dedicated to the execution of the scheduled task graph. No other programme or task is executed on the system while the scheduled task graph is executed.
- A processor can execute only one task at a time and the execution is not preemptive.
- The cost of communication between tasks executed on the same processor, hence called *local* communication, is negligible and therefore considered as zero. This assumption is based on the observation that for many parallel systems, remote communication (i.e. interprocessor communication) is one or more orders of magnitude more expensive than local communication (i.e. intra-processor communication) [3].
- Interprocessor communication is performed by a dedicated communication subsystem. The processors are not involved in communication.
- Interprocessor communication in the system is performed concurrently; there is no contention for communication resources.

– The communication network is fully connected. Every processor can communicate directly with every other processor via a dedicated identical communication link.

Given the identical processors 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 analyse the performance of scheduling algorithms such as [3,9,20,21]. We consider this model to permit a fair comparison with state-of-the-art algorithms.

3.3 Definitions

In the following, we discuss some important definitions that are frequently used in the rest of this paper related to the task graph representation of target applications.

Paths and Path Lengths In a given task graph G as shown in Fig. 1, there exist multiple paths of arbitrary length leading to the *sink* node (such as n_8) from the *source* node (such as n_1). The total length of a path p in G is the sum of the weights of its nodes and edges from the *source* node to the *sink* node and can be expressed by Eq. 1.

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

The computational length of a path p in G, i.e. without including communication, is the sum of the weights of its nodes only from the *source* node to the *sink* node and can be expressed by Eq. 2.

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

Due to the sequential order inherent in a path p, none of the nodes belonging to p is executed concurrently with any other node of p. Thus, the path length $pl_w(p)$ can be interpreted as the amount of time a path p takes for the sequential execution of all nodes (only) belonging to p and the path length $pl(p)$ can be interpreted as the time the path p takes for the execution of all its nodes if all communications between its nodes are interprocessor communications. For instance, when each node of p is allocated to a different processor, the communication cost between nodes is *nonzero*.

Another important concept related to task graphs is the *Critical Path (cp)*. A critical path in G refers to the *longest* path from the *source* node to the *sink* node. The length of critical path can be expressed by Eq. 3.

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

From the scheduling perspective, constrained sequential execution (due to precedence) of nodes that belong to the critical path takes at least the execution time of any other path in the task graph [3,5]. Thus, the computational length of critical path (i.e., without including communication cost) serves as a lower bound on the minimum execution time achievable for the application on a parallel machine. Note that this lower bound does not necessarily specify the optimal schedule length under constrained resources. Since a late execution start of any node on critical path directly results in an extended schedule length, these nodes are important for the scheduler to minimise overall execution time.

4 The Noodle Heuristic

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 list scheduling schemes.

Many existing list scheduling techniques [4,5,8–10] evaluate the importance of nodes in different ways. Earlier a node is considered for scheduling, and sooner 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 results in smaller SL (Schedule Length) of the application.

Authors in [5] show that attribution of higher priority to the nodes belonging to the critical path in a task graph can yield a minimum SL (not necessarily the optimal length). Such heuristics, however, are based on a Depth-First Search (DFS) execution that has its own merits and demerits. For instance, on the one hand, such heuristics prioritise execution of nodes on critical path that helps avoiding any direct extension in the SL. On the other hand, a DFS-based execution of nodes causes a delay in execution of relatively less important nodes belonging to the non-critical paths. This results in the underutilisation of constrained resources, i.e. some cores become idle due to insufficient number of ready nodes available in the *ready queue* at any point during execution.

Contrary to the DFS-based approach, Breadth-First Search (BFS) approach can better exploit the inherent parallelism of the target application as shown in [15]. BFS-based scheduling heuristics schedule all ready nodes before proceeding to the next spawn level, which may result in better resource utilisation. Such heuristics, however, are not always able to select nodes from critical paths, which may directly extend the SL.

Both DFS-based and BFS-based heuristics cannot simultaneously address the twofold objective of improved utilisation and minimisation of SL for the target application. A possible solution would be to assign priorities by providing *proportionate fairness* to ready nodes belonging to *all paths*. The Noodle heuristic strives to maintain this proportionate fairness in priority attribution to ready nodes. In the following, we present an example hypothetical task graph to motivate the Noodle heuristic by demonstrating the impact of both DFS- and BFS-based prioritisation approaches on the resource utilisation and SL.

4.1 Example Task Graph

Figure 2 presents our example task graph for illustration purpose. In this example, we assign the same execution time for all nodes (i.e. 1−unit) and do not include communication cost at the edges for simplicity. Precedence constraints, however, are preserved. The task graph includes four separate paths *A={1, 2, 4, 7, 10, 14, 15, 16}*, *B={1, 2, 4, 7, 11, 14, 15, 16}*, *C={1, 2, 5, 8, 12, 15, 16}*, and *D={1, 3, 6, 9, 13, 16}*, with path lengths $pl_w(A)$=$pl_w(B)$=8, $pl_w(C)$=7, and $pl_w(D)$=6, respectively. Paths *A* and *B* are the critical paths in Fig. 2 (calculated by Eq. 3).

At first, we attribute priorities to the nodes according to the depth-first approach as used in [5]. Such priority attribution would result in a schedule for a 2-core execution case as shown in Fig. 3. In Fig. 3, first column represents the time flow (one time unit for each row) and other two columns indicate tasks assigned to the cores ($C0$ and $C1$) with respect to time. Additionally, grey boxes refer to the unused time units. The resulting schedule in Fig. 3 shows sequential execution of nodes 9, 13, and 16 belonging to path D towards the end (i.e. between time units 8 and 10). Note that the prioritisation mechanism in this case leaves relatively less important path D

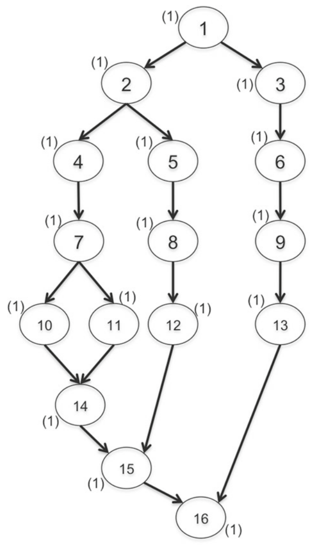


Fig. 2 Example task graph used to illustrate differences in priority attribution mechanism of Noodle, DFS-, and BFS-based scheduling heuristics

Fig. 3 Schedule for DFS-based prioritisation

time	C0	C1
1	1	▨
2	2	3
3	4	5
4	7	8
5	10	11
6	14	12
7	15	6
8	9	▨
9	13	▨
10	16	▨

Fig. 4 Schedule for BFS-based prioritisation

time	C0	C1
1	1	▨
2	2	3
3	4	5
4	6	7
5	8	9
6	10	11
7	12	13
8	14	▨
9	15	▨
10	16	▨

stranded. Stranded paths might not have enough parallelism and therefore can claim less resources that results in extended SL even when sufficient resources are available. Similarly, the attribution of priorities to nodes using the breadth-first approach results in the schedule shown in Fig. 4. Again, this schedule shows that towards the end task graph does not have enough parallelism at certain depth that could be exploited by the scheduler. Therefore, nodes 14, 15, and 16 had to execute sequentially. The resulting schedule length (*SL*=10) happens to be the same in both cases.

The Noodle heuristic attributes priorities to ready nodes belonging to *all* paths in such a way that the *priority decays along the path whose nodes are being executed*. Therefore, it prevents the execution of nodes too deep into the critical path(s) before executing nodes belonging to non-critical paths in a proportionately fair manner. Thus, no path will be stranded and *no node is left behind (noodle)* as the schedule proceeds towards the sink node. In the following, we elaborate this priority attribution principle.

4.2 Priority Attribution Mechanism of the Noodle Heuristic

The Noodle heuristic, in its priority attribution phase, uses the concept of *paths* and *path lengths* presented in [3,6,19] to gauge the importance of nodes in an offline analysis. Noodle assigns priorities to ready nodes such that the priority level *decreases* in an exponential fashion at every spawn level on those paths whose (ready) nodes are being executed. Relative priority of other paths, however, does not decay and remains the same.

This relative priority is determined for each particular path p based on its path length $pl(p)$, which is reflective of the cumulative weight of all nodes and edges belonging to p (see Eq. 1), compared to other paths in the graph. Greater the length of a path p, lower the (exponential) decay in the priority level of nodes belonging to p. We formalise this priority attribution in the following.

Let us associate an *exponential decay function* [22] to each path p in G. Exponential decay is a mathematical property under which a quantity is subject to decrease at a rate proportional to its current value. Equation 4 presents quantity $N(t)$ as a function of time t that decays at a rate λ from its initial value N_0. Variable λ is a positive rate called the *exponential decay constant*. We adapt Eq. 4 to introduce a *priority function (pf)* associated with each node n_i that belongs to path p_j in a task graph G as expressed by Eq. 5.

$$N(t) = N_0 e^{-(\lambda)t} \tag{4}$$

$$pf(n_{i,j}) = pf(n_{i-1,j})e^{-(1/\lambda_j)} \tag{5}$$

Using Eq. 5, the priority function $pf(n_{i,j})$ is computed for each node n_i that belongs to path p_j by considering the path p_j in *isolation*. That is, starting from the source node (n_1), the $pf(n_{i,j})$ is calculated for each subsequent node n_i along the path p_j as a function of its current value, which is $pf(n_{i-1,j})$. Here, there are two very important parameters to be assigned for the computation of $pf(n_{i,j})$, namely the initial value $pf(n_{0,j})$ for the very first node (i.e. n_1) on each path p_j and the decay constant λ for p_j. The initial value for computation of $pf(n_{1,j})$ on any path p_j is assigned as $pf(n_{0,j})$=$pl(cp)$, i.e. equals to the length of critical path in the task graph G. As mentioned in Sect. 3, the $pl(cp)$ serves as a lower bound on the minimum execution time achievable on a parallel machine. Assigning $pl(cp)$ as the initial value of priority function makes all paths to advance towards this lower bound on execution time with respect to their own decay constant. The exponential decay constant is assigned as shown in Eq. 6.

$$\lambda_j = pl(p_j) \tag{6}$$

The decay constant λ_j for path j is equal to the path length of p_j. Note that decay constant λ is a path-specific parameter and it is different for different paths in the task graph (though, multiple paths may have the same path lengths and therefore, same value for λ). Also note that Eq. 5 uses the reciprocal value of λ as decay constant (i.e. $1/\lambda$), which implies that greater the length of any path p_j, smaller the decay in the priority functions $pf(n_{i,j})$ associated to individual nodes belonging to p_j.

The priority function $pf(n_{i,j})$ does not specify the final priority level of any node n_i in the task graph. The value of $pf(n_{i,j})$ is assigned to nodes while each path of the task graph is considered in isolation. Considering paths in isolation allows to assess the relative importance of nodes with respect to that particular path. However, there can be nodes, such as forking and joining nodes, that belong to multiple paths simultaneously. For instance, in Fig. 2, node n_2 is a shared node among two paths (A and B). Therefore, for all shared nodes, the final priority level is attributed as shown in Eq. 7. The final priority level of any node i has to be equal to the maximum priority function assigned to node i among all paths going through the node. Here, m refers to the total number of paths to which the node i belongs. Since shared nodes possess different levels of importance with respect to different paths they belong, assigning the maximum of available priorities to these nodes ensures that the proportionate fairness in priority to relatively important paths is always maintained.

$$\text{priority}(n_i) = \max_{n_i \epsilon V, 1 \le j \le m} \{pf(n_{i,j})\} \tag{7}$$

Let us reconsider the priority attribution of task graph used in our illustrative example task graph of Fig. 2 under the Noodle heuristic. The resulting $pf(n_{i,j})$ for each node are listed in Table 1. Figure 5 shows the resulting schedule produced by Noodle heuristic. One can notice that the resulting SL produced by Noodle is shorter than the one produced by DFS- and BFS-based approaches in Fig. 2 while respecting all precedence constraints of the application tasks. Moreover, computing resources are fully utilised and no core remains idle except when source (n_1) and sink (n_{16}) nodes are executing alone. The schedule produced by Noodle in Fig. 5 also happens to be an *optimal* schedule. However, Noodle does not guarantee to produce always an optimal schedule for bounded computing resources.

Figure 6 shows the pseudocode representation for the algorithm used to identify and create various paths in the task graph. The algorithm implements a function called $AddToPathList(v, p, P)$ for every node v belonging to task graph G. After initialising a path with first node, each of its children nodes are analysed (lines 6 to 14). 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, Noodle can use these paths to perform priority attribution.

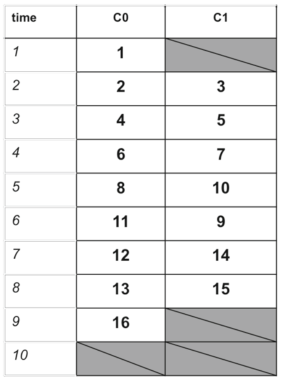
Figure 7 presents pseudocode representation of the Noodle heuristic algorithm. After computing the lengths of all paths in the task graph, the first part (lines 4 to 8) identifies the critical path in the task graph composed of a path set P=$\{p_1, p_2, p_3, \ldots, p_m\}$. Once the length of critical path is known, the second part (lines 9–15) computes the priority

Table 1 Calculated priority function for nodes of task graph in Fig. 2 while considering each path in isolation. First four sub-tables show the computed values of $pf(n_{i,j})$ for path A to D. Fifth and sixth sub-table show the final priorities for all nodes being calculated using Eq. 7

Node (n_i)	$pf(n_{i,A})$	Node (n_i)	$pf(n_{i,B})$
1	7.059	1	7.059
2	6.230	2	6.230
4	5.498	4	5.498
7	4.852	7	4.852
10	4.282	11	4.282
14	3.779	14	3.779
15	3.334	15	3.334
16	2.943	16	2.943

Node (n_i)	$pf(n_{i,C})$	Node (n_i)	$pf(n_{i,D})$
1	6.935	1	6.771
2	6.011	3	5.732
5	5.211	6	4.852
8	4.517	9	4.107
12	3.916	13	3.476
15	3.394	16	2.943
16	2.943		

Nodes	$priority(n_i)$	Nodes	$priority(n_i)$
1	7.059	9	4.107
2	6.230	10	4.282
3	5.732	11	4.282
4	5.498	12	3.916
5	5.211	13	3.476
6	4.852	14	3.779
7	4.852	15	3.394
8	4.517	16	2.943



time	C0	C1
1	1	
2	2	3
3	4	5
4	6	7
5	8	10
6	11	9
7	12	14
8	13	15
9	16	
10		

Fig. 5 Improved schedule in terms of (a) schedule length and (b) resource utilisation produced by Noodle heuristic for the task graph used in Fig. 2 on a dual-core system

```
 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. 6 Pseudocode representation of path creation algorithm in the task graph

```
 1: P ← set of all paths $\{p_1, p_2, p_3, ..., p_m\}$ in G
 2: $pl(p_j) \leftarrow$ length of each path $p_j$
 3: $pl(cp) \leftarrow \emptyset$, initial value for cp length
 4: for each path $j$ in $P$ do
 5:     if $pl(p_j) > pl(cp)$ then
 6:         $pl(cp) \leftarrow pl(p_j)$
 7:     end if
 8: end for
 9: for each path $p_j$ in $P$ do
10:     $pf(n_{0,j})=pl(cp)$
11:     $\lambda_j = pl(p_j)$
12:     for each node $n_i$ in path $p_j$ do
13:         $pf(n_{i,j})=pf(n_{i-1,j})\ e^{-(1/\lambda_j)}$
14:     end for
15: end for
16: for each node $n_i$ in the task graph $G$ do
17:     if $n_i$ belongs to multiple paths then
18:         $priority(n_i) = \max_{1 \leq j \leq size(P)} \{pf(n_{i,j})\}$
19:     end if
20: end for
```

Fig. 7 Pseudocode representation of Noodle heuristic algorithm

function for each node belonging to each path in P in isolation. In the last part (lines 16–20), priorities for all shared nodes are computed using Eq. 7.

As shown in [5], the node ordering based on decreasing bottom-level order also results in a topological order of nodes. Similarly, the Noodle heuristic achieves node ordering based on the decreasing priority function, which also results in a topological order of nodes that respects node dependences. Thus, the complexity to determine the node list is similar to the one presented in [5], i.e. $O(V+E)$ to calculate the node levels and $O(V \log V)$ for the sorting operation. The resulting overall complexity for the Noodle heuristics is $O(V \log V + E)$.

4.3 Resource Allocation Under The Noodle Heuristic

Like any other list scheduling heuristic, the second part of Noodle heuristic is to successively schedule each ready task onto a processing unit. In general, for a ready task, a processing unit (a core) is chosen directly that allows its earliest start time. Noodle, however, does this resource allocation in two phases. In the first phase, it sorts the *ready queue* in descending priority order of tasks. Out of the sorted ready queue, top m tasks are selected for allocation (here, m refers to the num-

```
1: TaskList ← all tasks $\{n_1, n_2, ..., n_i\}$ from G
2: CoreSet ← all cores $\{C0, C1, C2, ..., Cm\}$
3: IdleCoreSet ← CoresSet
4: ReadyQueue ← $\emptyset$
5: Start with the root node
6: for each SchedEvent do
7:     ReadyQueue ← all ready tasks from TaskList
8:     Sort ReadyQueue in descending order of $priority(n_i)$
9:     if Task Priorities Tie then
10:        Higher priority goes to task with larger $w_(n_i)$
11:    end if
12:    Update IdleCoreSet
13: end for
14: ExecutableTasks ← Size of IdleCoreSet
15: for Top ExecutableTasks from ReadyQueue do
16:     Assign core that offers minimum communication cost w.r.t.
        parent task
17: end for
```

Fig. 8 Pseudocode representation of runtime system for Noodle heuristic

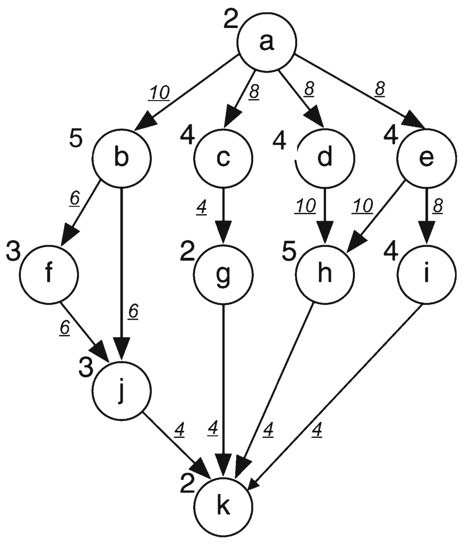


Fig. 9 An example DAG to be scheduled under Noodle Heuristic—a complete example

Table 2 Creation of paths and computation of path lengths in the DAG using Noodle heuristic

Path name	Member Nodes	Length
p1	a, b, f, j, k	41
p2	a, b, j, k	32
p3	a, c, g, k	26
p4	a, d, h, k	35
p5	a, e, h, k	35
p6	a, e, i, k	32

ber of cores available for execution). In the second phase, Noodle computes, for each of the m tasks, the total communication cost incurred per task-core combination with respect to its parent tasks (parent tasks can be more than one and may have executed on different cores). The task-core combination that offers minimum communication cost for selected task is chosen for its execution.

Figure 8 shows pseudocode of the runtime system for Noodle heuristic. It maintains two separate task queues, namely the $TaskList$ and $ReadyQueue$. Where TaskList contains all tasks from the DAG while ReadyQueue contains only ready tasks. Runtime system also maintains two lists: one list for all cores available in the system, called the $CoreSet$, and another for idle cores in the system, called $IdleCoreSet$. At every scheduling event (i.e. start and completion of tasks), both ReadyQueue and IdleCoreSet are updated (lines 6–10). Once updated, ready tasks equal to the number of cores available in IdleCoreSet are assigned to the cores such that a task is executed on the core which offers minimum communication cost with its parent task(s). Note that the ties between task priorities are broken in favour of task having larger workload, and for tasks with exactly the same workload, ties are broken at random.

4.4 DAG Scheduling Under Noodle Heuristic: A Complete Example

For better understanding of the reader and reproducibility of results, we demonstrate all steps of Noodle heuristic by producing schedule of an example directed acyclic task graph that includes communication cost. Our example DAG is shown in Fig. 9. In Fig. 9, the number mentioned around each node refers to the computation cost (w_i) whereas the number mentioned around the edge between nodes (underlined values) refer to the communication cost ($c(e_{i,j})$) as explained in Section 3.

As a first step, using algorithm shown in Fig. 6, all paths in the task graph are created. Resultant paths and their member nodes are listed in columns one and two of Table 2, respectively. Once all paths are created in the task graph, Noodle heuristic learns the lengths of each path as well as the length of critical path using algorithm presented in Fig. 7. Computations show that p1 happens to be the critical path in this DAG. After learning the lengths of all paths, Noodle applies Eq. 5 to compute priority function ($pf(n_{i,j})$) for each node while considering each path p_j in isolation. Resulting $pf(n_{i,j})$ value for each node are provided in Table 3. Using Table 3 and Eq. 7, we calculate the final priorities for each node as shown is Table 4. Based on these priorities, Noodle produces the final schedule as shown in Fig. 10. In Fig. 10, $c(e_{i,j})$ refers to the communication cost on the edge $e_{i,j}$ between nodes n_i and n_j as explained in Sect. 3. Note that Noodle breaks the ties between nodes in favour of the node incurring more computation cost. For instance, node b and d have equal priority level; therefore, in case of tie, node b would have been preferred for execution over node d.

Table 3 Calculated priority function for nodes of task graph in Fig. 9 while considering each path in isolation

Node (n_i)	$pf(n_{i,p1})$	Node (n_i)	$pf(n_{i,p2})$
a	40.012	a	39.738
b	39.047	b	38.515
f	38.107	j	37.330
j	37.188	k	36.182
k	36.292		
Node (n_i)	**$pf(n_{i,p3})$**	**Node (n_i)**	**$pf(n_{i,p4})$**
a	39.453	a	39.845
c	37.964	d	38.722
g	36.531	h	37.632
k	35.153	k	36.572
Node (n_i)	**$pf(n_{i,p5})$**	**Node (n_i)**	**$pf(n_{i,p6})$**
a	39.845	a	39.738
e	38.722	e	38.515
h	37.632	i	37.330
k	36.572	k	36.182

Table 4 Final priorities of nodes of task graph in Fig. 9 using Eq. 7

Node	Priority level
a	40.012
b	39.047
c	37.964
d	38.722
e	38.722
f	38.107
g	36.531
h	37.632
i	37.330
j	37.330
k	36.572

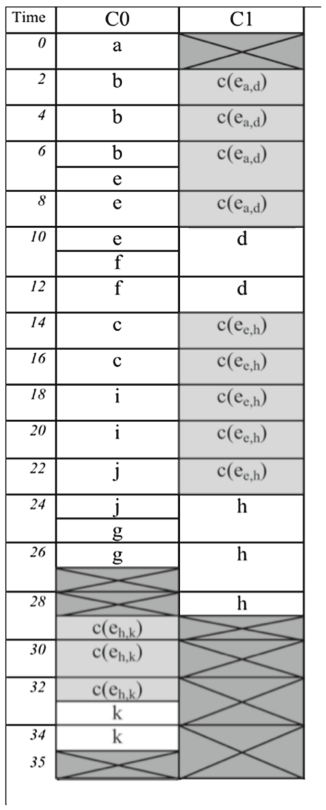


Fig. 10 Final schedule produced by Noodle for the example DAG shown in Fig. 9

Moreover, during runtime, a node is preferably executed on the resource having maximum number of its parent nodes in order to reduce data transfer cost. For instance, in Fig. 10 at time instant 14, runtime system could chose any core for the execution of nodes c and h. However, node c is allocated core $C0$ for execution due to the earlier execution of its parent node a on the same core. For node h, however, the choice of core does not make any difference as the communication cost from at least one of its parent node will incur anyway.

5 Experimental Evaluation

In this section, we present our experimental results for the evaluation of Noodle. We start with providing brief information on our experimental set-up and benchmark task graphs considered in our experimental study in Sect. 5.1.

5.1 Experimental Setup

We use task graphs from Standard Task Graph Set (STG) [23] for our performance evaluation and comparison study. STG, which is frequently used for evaluation of multiprocessor scheduling algorithms [24,25], provides randomly generated task graphs with different number of tasks. STG also includes pre-computed optimal schedule lengths (using exhaustive search) and some parameters related to task graphs. For the current version, STG does not include communication costs. We perform experiments using task graphs comprising of 50 tasks, 100 tasks, and 300 tasks. We use 308 task graphs in total: 154 task graphs with 50 tasks, 138 task graphs with 100 tasks, and 16 task graphs with 300 tasks.

Some properties of these task graphs are given in Table 5, with minimum, maximum, and average values of listed parameters in descending order, respectively. We evaluate Noodle results for these task graphs by considering 2-core, 4-core, 8-core, and 16-core execution scenarios.

Our experimental results can be divided into two distinct parts. In the first part, we provide results on schedule length of task graphs obtained under Noodle and compare them

Table 5 Values of parameters related to STG task graphs used in our experiments

	TGs with 50 tasks	TGs with 100 tasks	TGs with 300 tasks
Number of edges	46 / 953 / 262.02	93 / 1677 / 629.81	309 / 2958 / 1835.69
Max. predec.	3 / 42 / 14.23	4 / 40 / 18.05	6 / 32 / 19.06
Max. proc. time.	7 / 70 / 22.27	7 / 83 / 24.09	8 / 76 / 27.81
Parallelism	1.66 / 11.92 / 5.62	2.85 / 19.55 / 7.06	7.16 / 27.27 / 13.55

with *optimal* schedule length. Note that in this part of evaluation, we take into account the computation cost of tasks only and we consider that there is no data transfer/communication cost between tasks. These results allow us to demonstrate that Noodle produces optimal and near-optimal schedule length in most cases. For analysis, the obtained schedule length under Noodle is compared with the optimal solution provided by the STG for their task graphs. In the second part, we provide results on the schedule length of task graphs obtained under Noodle including communication cost between tasks. Since STG includes only computation cost for its tasks, we add communication costs to the available task graphs. In order to study the impact of variation in the amount of data transfer among tasks, the communication cost is taken into account by using the *Communication to Computation cost Ratio (CCR)*. The value for communication cost is computed using Eq. 8. Here, $w(n_j)$ refers to the computation cost of the node n_j and $c(e_{i,j})$ is the communication cost on the edge between task n_j and its parent task n_i. Note that task n_j can have more than one parents (for instance, tasks representing join nodes). In this case, communication cost with all parents is calculated to be the same and data reception is considered in parallel.

$$c(e_{i,j}) = CCR * w(n_j) \tag{8}$$

Generally, scheduling strategies are evaluated in terms of a metric termed as *Schedule Length Ratio (SLR)* [13], which is the ratio of the schedule length or *makespan* to the length of critical path $(pl(cp))$ of the task graph as expressed in Eq. 9. The scheduling strategy producing the smaller SLR value therefore implies to produce better schedule.

$$\text{SLR} = \frac{SL}{pl(cp)} \tag{9}$$

In the second part, we evaluate the performance of Noodle heuristic using SLR values and compare results with other heuristics.

5.2 Part-I: Evaluation of Noodle Compared to Optimal Schedule Lengths (without communication cost)

In this section, we evaluate the resulting schedule lengths of task graphs under Noodle with different number of processors and compare them with optimal schedule lengths provided by the STG benchmark. Since STG includes task graphs with only computation costs and provides optimal schedule lengths for these graphs, we assume that there is no communication cost (CCR=0) and execute Noodle on the STG task graphs. In order to demonstrate the difference between the schedule lengths, we calculate the ratios of the changes for each case.

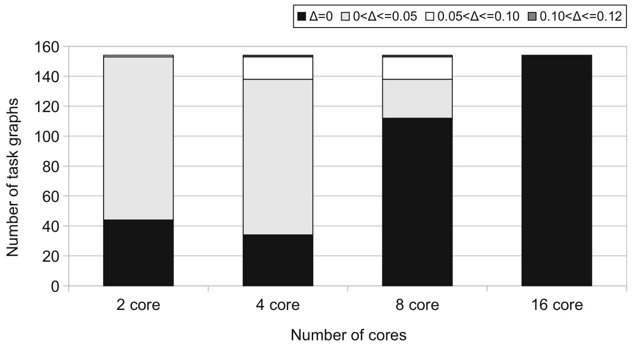


Fig. 11 Results with task graphs comprising of 50 tasks per graph

Figures 11, 12, and 13 present the distribution of differences for the task graphs with 50 tasks, 100 tasks, and 300 tasks, respectively. To better analyse the variations in produced schedule lengths (SL), we observe the number of task graphs that end up having the difference (Δ) in the same range. Therefore, we define four ranges or intervals for Δ such that:

- $\Delta = 0$: the SL produced by the Noodle and the optimal SL are the same.
- $0 < \Delta \leq 0.05$: the SL produced by the Noodle is larger than the optimal SL by at most 5 %.
- $0.05 < \Delta \leq 0.10$: the SL produced by Noodle is larger than optimal SL by more than 5 % and at most 10 %.
- $0.10 < \Delta \leq 0.12$: the SL produced by Noodle is larger than optimal SL by more than 10 % and at most 12 %.

As shown in Figs. 11, 12, and 13, the schedule length produced by the Noodle remains within a maximum difference of 12 % with respect to the optimal schedule length for all task graphs used in the experiments. For instance, the first column in Fig. 11 demonstrates that the Noodle results in the optimal

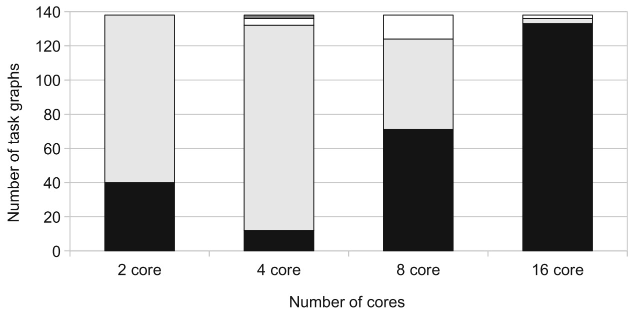


Fig. 12 Results with task graphs comprising of 100 tasks per graph

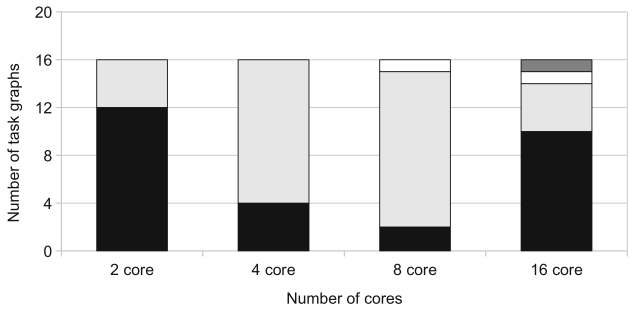


Fig. 13 Results with task graphs comprising of 300 tasks per graph

SL for 45 task graphs (with 50 tasks per graph in a 2-core execution scenario), while it ends up with an SL that is larger than optimal by at most 5 % for 112 task graphs out of 159 task graphs. There are only 2 task graphs for which the SL produced by Noodle falls in the interval $0.05 < \Delta \leq 0.10$. For 2-core system, Noodle never resulted in a SL beyond $\Delta \leq 0.10$, i.e. 10 % of the optimal length. Noodle performs similarly for 4-core and 8-core systems. With sufficiently large number of cores, such as the case with 16 cores, Noodle performs the same as optimal schedule for all task graphs. We observe similar behaviour for task graphs with 100 tasks and 300 tasks (Fig. 12, 13).

5.3 Part-II: Evaluation of Noodle Compared to Other Heuristics (with communication cost)

In this section, we present our comparison study of Noodle with existing scheduling heuristics in the literature. We select scheduling heuristics whose working principles are based on the concept of paths and path length information similar to Noodle. The heuristic algorithms used in our comparison study are as follows [5]:

- *BL*: A heuristic in which nodes are ordered by their computation bottom levels. The bottom level, *bl(n)*, of node $n \epsilon V$ is the length of the longest path *starting* from *n*, including weight of node *n* itself.
- *CP_BL_TL*: A heuristic in which nodes on the critical path are prioritised among all ready nodes, while the other nodes are ordered by their bottom levels. Ties are broken by considering top levels of nodes. The top level, *tl(n)*, of node $n \epsilon V$ is the length of the longest path *ending* in *n*, excluding weight of node *n* itself.
- *CP_TL*: A heuristic that also prioritises the nodes on the critical path, while the other nodes are ordered by their top levels.
- *Noodle*: Our proposed Noodle heuristic such that the nodes are ordered by priority function described in Sect. 4 and ties are broken by prioritising the nodes with larger task weights.

We execute these heuristics and Noodle for STG task graphs with different number of tasks. Our experimental study includes results for task graphs with 50 tasks, 100 tasks, and 300 tasks. We generate communication costs by using different CCR values. Specifically, we use 0.1, 0.5, 1, 2, 5, 10 for CCR value. In order to analyse the scheduling performance for different number of cores, we execute each task graph with different number of cores, such as 2-core, 4-core, 8-core, and 16-core executions. Finally, we have $154 \times 6 \times 4 = \mathit{3696}$, $138 \times 6 \times 4 = \mathit{3312}$, $16 \times 6 \times 4 = \mathit{384}$ cases for task graphs with 50 tasks, 100 tasks, and 300 tasks, respectively.

We report average SLR values for a set of configuration parameters. While we include results for different number of cores and CCR values, we also report results for different parallelism and number of edges provided by the STG benchmark. Since task graphs with different number of tasks have different ranges (inclusive-lower-bound) for parallelism and number of edges, we create four distinct ranges for each set of task graphs with 50 tasks, 100 tasks, and 300 tasks.

5.3.1 Task Graphs with 50 Tasks

Table 6 presents configuration parameters and values for task graphs with 50 tasks. Since parallelism differs between 1.66 and 11.92 (given in Table 5), we collect the results in the following four ranges in our analysis: 1–2, 2–4, 4–8, and 8–12. Similarly, we report average SLR values in the range of different number of edges.

Figures 14, 15, 16, and 17 present the average SLR values for different parameter values for graphs with 50 tasks. Noo-

Table 6 Configuration parameters for task graphs with 50 tasks

Parameter	Values
Number of cores	2, 4, 8, 16
CCR	0.1, 0.5, 1, 2, 5, 10
Parallelism	1–2, 2–4, 4–8, 8–12
Number of edges	<100, 100–250, 250–500, >500

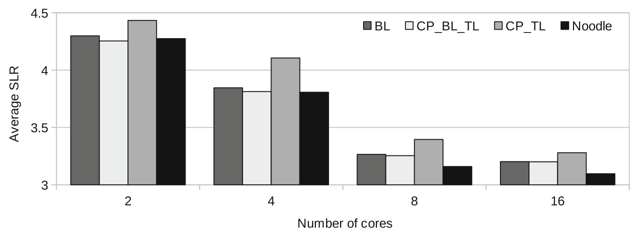


Fig. 14 Results for the given number of cores

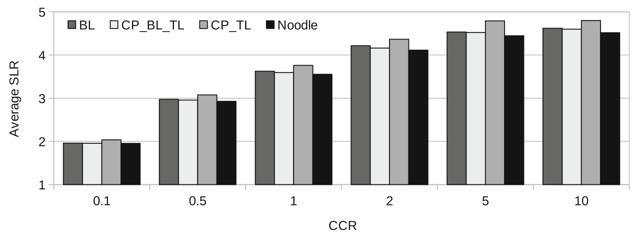


Fig. 15 Results for the given CCR values

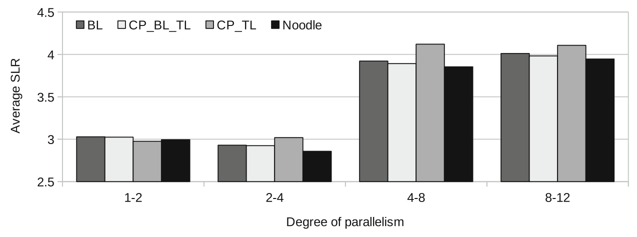


Fig. 16 Results for the given parallelism intervals

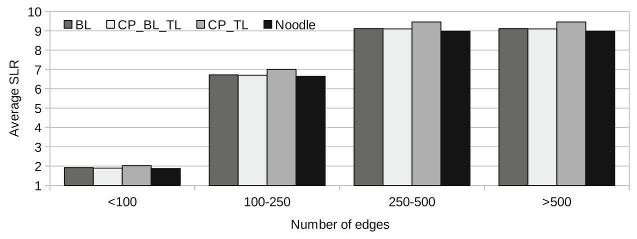


Fig. 17 Results for the given number of edges

dle performs better (i.e. has smaller average SLR values) for all the cases.

Figure 14 presents results for different number of cores and demonstrates that scheduling results improve by increasing number of cores. Since the largest degree of parallelism is less than 12 for this set of task graphs, 8-core and 16-core results do not differ significantly. Moreover, the difference between Noodle and the other heuristics increases by increasing number of cores, i.e. Noodle takes more advantage of larger number of cores than other heuristics. For instance, the difference between average SL values of BL and Noodle equals 0.02 for 2-core case, the same value for 16-core case equals 0.11.

Figure 15 demonstrates the effect of CCR, the degree of communication, on scheduling performance. Noodle performance becomes a little better for higher CCR values due to its communication-aware resource allocation strategy (explained in Sect. 4.3). While small communication cost does not lengthen the schedule, the effect becomes clearer for larger communication costs. However, it does not help much since priority attribution mechanism does not apply communication-centric rules.

Figures 16 and 17 present the average SLR values for different intervals of parallelism and number of edges parameters, respectively. When the parallelism is less than 4, all heuristics perform better due to availability of resources. On the other hand, their performance decreases when parallelism becomes larger, since they have to deal with the shortage of processors for most of the cases. Moreover, the performance of scheduling heuristics decreases as the number of edges increases in the task graph similar to CCR value. Noodle performance again becomes a little better for larger number of edges, but not very significantly. Among four comparison cases, we see the largest average SLR values for the task graphs with the largest number of edges (around 9 for the last column (>500) in Fig. 17).

5.3.2 Task Graphs with 100 Tasks

Table 7 presents configuration parameters and values for task graphs with 100 tasks. We divide the results into ranges for different parallelism and number of edges parameters. We include 2–4, 4–8, 8–16, and 16–20 intervals (among the parallelism values between 2.85 and 19.55 given in Table 5). Similarly, we report average SLR values in the range of different number of edges.

Figures 18, 19, 20, and 21 present the average SLR values for different parameter values for graphs with 100 tasks. As shown in the figures, the results are similar to the cases given for graphs with 50 tasks. Noodle performs better for all the cases. The number of cores, CCR, and the number of edges parameters affect the scheduling performance in a similar manner. While the performance worsens for larger degree of parallelism until it gets 16, there is an improvement for the cases where parallelism is between 16 and 20 (Fig. 20). When we look at these task graphs, we see that the number of edges for this set of graphs is very low (the maximum is 107, the average is 101.25 while the overall maximum is 1677 and

Table 7 Configuration parameters for task graphs with 100 tasks

Parameter	Values
Number of cores	2, 4, 8, 16
CCR	0.1, 0.5, 1, 2, 5, 10
Parallelism	2–4, 4–8, 8–16, 16–20
Number of edges	<100, 100–500, 500–1000, >1000

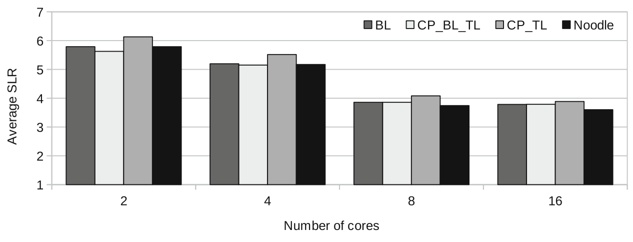


Fig. 18 Results for the given number of cores

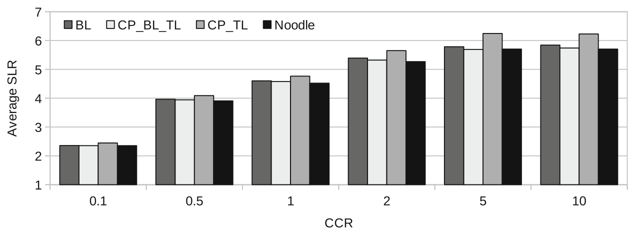


Fig. 19 Results for the given CCR values

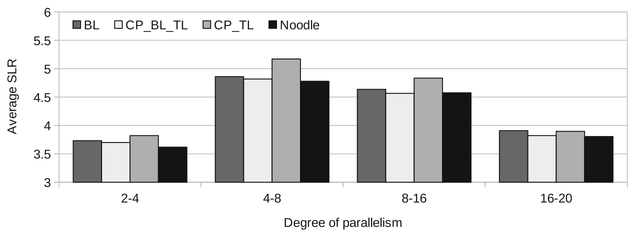


Fig. 20 Results for the given parallelism intervals

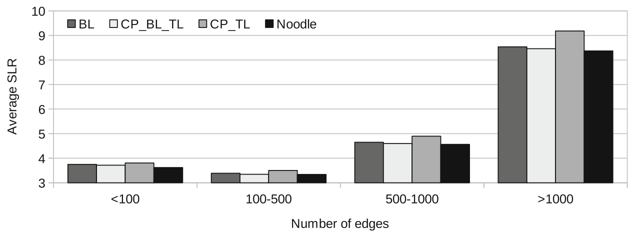


Fig. 21 Results for the given number of edges

the overall average is 629.81), which explains the unexpected behaviour. When task graphs have smaller number of edges (less communication overhead), larger amount of parallelism does not hurt performance very much in case of resource shortage.

5.3.3 Task Graphs with 300 Tasks

Table 8 presents configuration parameters and values for task graphs with 300 tasks. We again define intervals for parallelism and number of edges for our analysis. Since the range of possible values is larger for this set of task graphs (parallelism between 7.16 and 27.27, number of edges between 309 and 2958 as given in Table 5), we have coarser-grained intervals for this part of our study.

Table 8 Configuration parameters for task graphs with 300 tasks

Parameter	Values
Number of cores	2, 4, 8, 16
CCR	0.1, 0.5, 1, 2, 5, 10
Parallelism	<8, 8–12, 12–20, >20
Number of edges	<500, 500–1000, 1000–2000, >2000

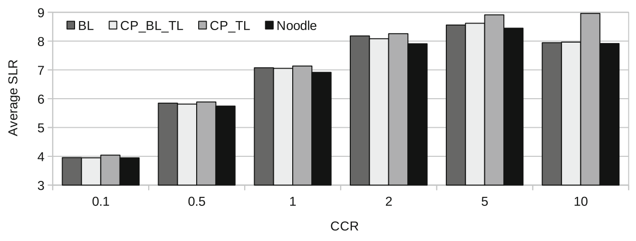


Fig. 22 Results for the given number of cores

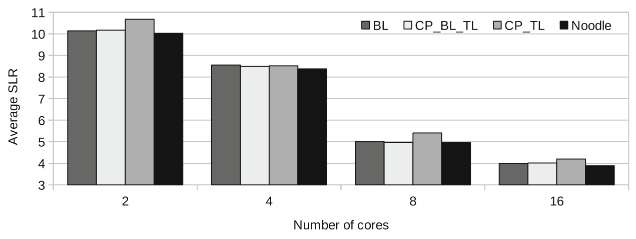


Fig. 23 Results for the given CCR values

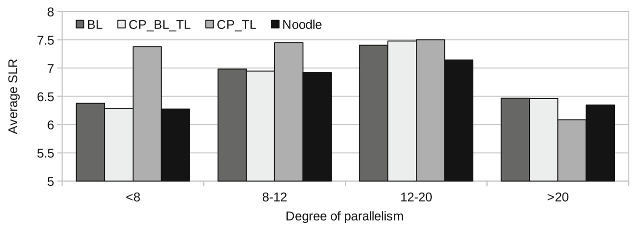


Fig. 24 Results for the given parallelism intervals

Figures 22, 23, 24, and 25 present the average SLR values for different set of configuration parameters. Among all results, we get the largest values for the cases with 2-core execution (average SLR = 11). Since the number of tasks and degree of parallelism is much larger for this set of task graphs, 2-core system does not struggle with this amount of work. As shown in Fig. 22, we do not reach the maximum benefit of number of cores in the system. Moreover, Fig. 24 demonstrates that Noodle performs much better than other heuristics for larger degree of parallelism, where there is more work needed using proportionate fairness in the task graph.

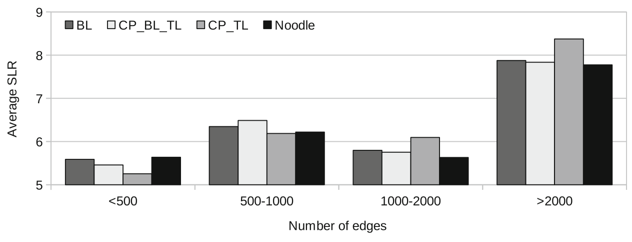


Fig. 25 Results for the given number of edges

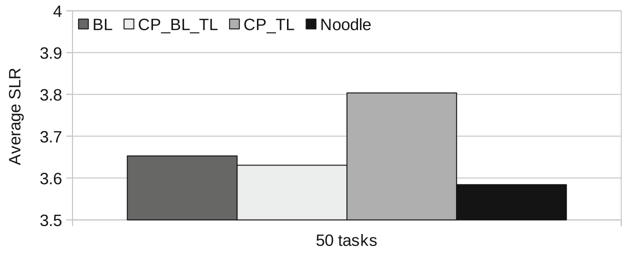


Fig. 26 Results for graphs with 50 tasks

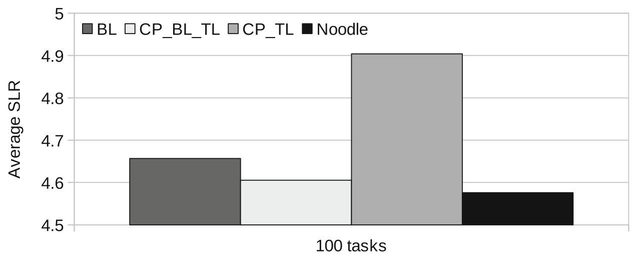


Fig. 27 Results for graphs with 100 tasks

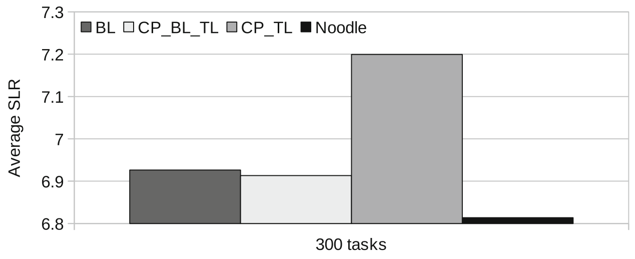


Fig. 28 Results for graphs with 300 tasks

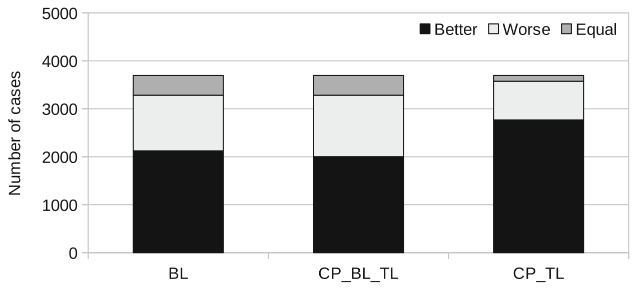


Fig. 29 Number of cases Noodle performed better, worse, and equal in comparison with other heuristics (for graphs with 50 tasks)

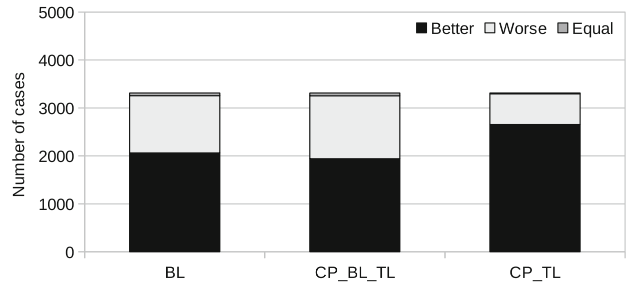


Fig. 30 Number of cases Noodle performed better, worse, and equal in comparison with other heuristics (for graphs with 100 tasks)

5.3.4 Performance Analysis

To summarise our comparison study with other scheduling techniques, we present overall average SLR values for each set of task graphs. Figures 26, 27, and 28 present average SLR values for all task graphs in our experimental study. Noodle has 2–3 % better results compared to the best performing heuristic.

Moreover, we calculate the number of cases that Noodle performs better, worse, and equal in comparison with other heuristics. Figures 29, 30, and 31 present these results for task graphs 50 tasks, 100 tasks, and 300 tasks. For task graphs with 50 tasks, Noodle performs better than BL for 2124 cases, better than CP_BL_TL for 2002 cases, and better than CP_TL for 2769 cases, among 3696 cases. The percentages of the schedules, where Noodle performs *better* or *equal* are 69, 65, 78 %, respectively.

For task graphs with 100 tasks, the number of cases that Noodle performs better are 2063, 1941, and 2653 among 3312 cases. The percentages of the schedules where Noodle performs *better* or *equal* are 64, 60, 81 %, respectively.

The proportion of the cases that Noodle performs better is larger for task graphs with 300 tasks, better than BL for 284 cases, better than CP_BL_TL for 255 cases, better than CP_TL for 298 cases, among 384 cases. The percentages of the schedules where Noodle performs *better* or *equal* are 74, 67, 78 %, respectively.

5.3.5 Sensitivity Analysis

We conduct a sensitivity analysis to better understand the relationship between different configuration parameters. Since the degree of parallelism represents a major characteristic of task graphs (and parallel applications represented by task graphs), we include a set of analysis based on parallelism values. Since we have larger amount of cases, we conduct our analysis for task graphs with 50 tasks. Figures 32, 33, 34, 35, 36, 37, 38 and 39, and Figures 40, 41, 42, and 43 present the average SLR values with pair-wise comparisons, number of cores, CCR values, and number of edges, respectively.

Figures 32, 33, 34, and 35 present the results for different number of cores with the given parallelism. When task graphs

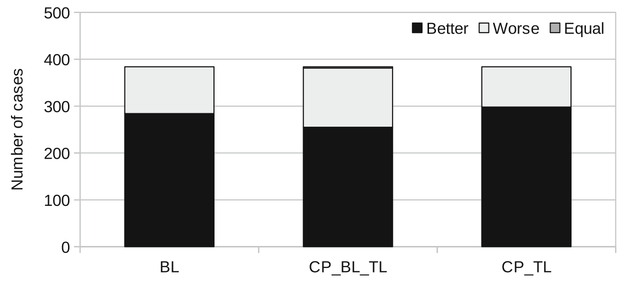


Fig. 31 Number of cases Noodle performed better, worse, and equal in comparison with other heuristics (for graphs with 300 tasks)

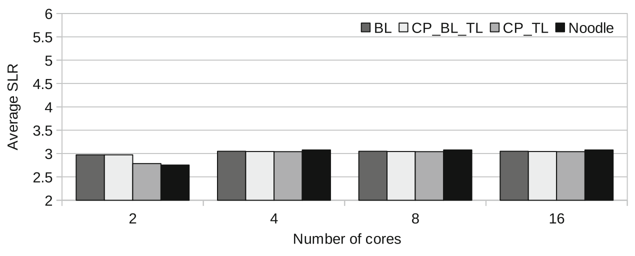


Fig. 32 Results for the given number of cores with parallelism 1–2

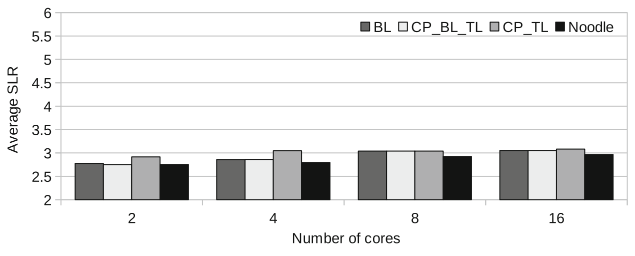


Fig. 33 Results for the given number of cores with parallelism 2–4

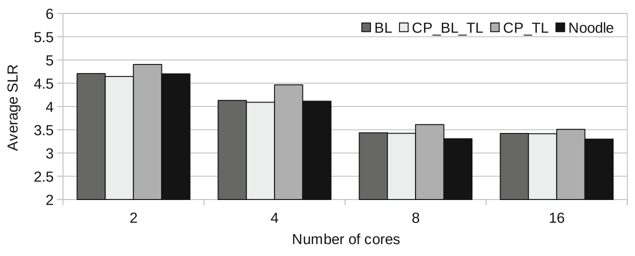


Fig. 34 Results for the given number of cores with parallelism 4–8

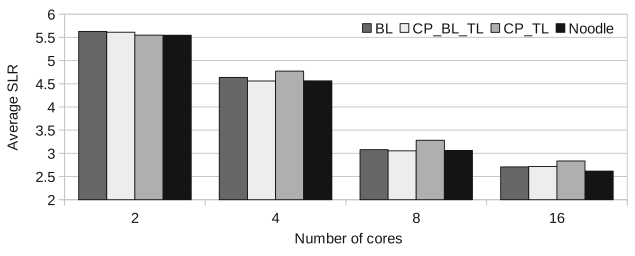


Fig. 35 Results for the given number of cores with parallelism 8–12

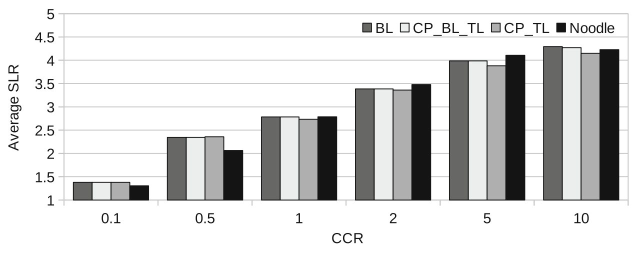


Fig. 36 Results for the given CCR values with parallelism 1–2

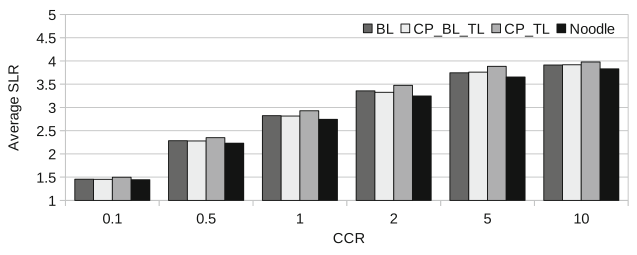


Fig. 37 Results for the given CCR values with parallelism 2–4

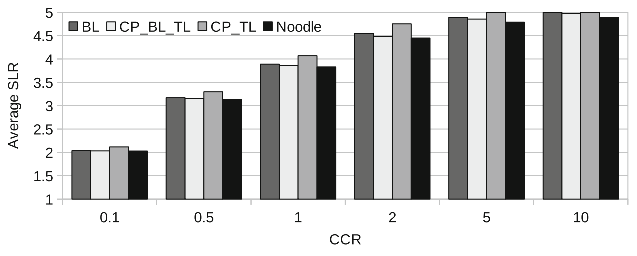


Fig. 38 Results for the given CCR values with parallelism 4–8

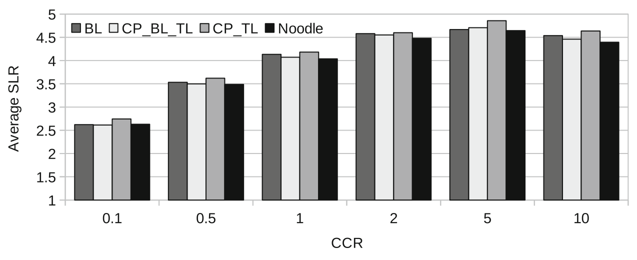


Fig. 39 Results for the given CCR values with parallelism 8–12

have lower degree of parallelism, the number of cores does not affect the scheduling performance as shown in Figure 32 and Figure 33. Since large amount of parallel work does not exist in the workload, the system cannot take advantage of Noodle performance improvement. On the other hand, the performance of Noodle becomes significant for the cases where the degree of parallelism is larger and the system has larger amount of cores to execute parallel tasks (Fig. 35).

As shown in Figs. 36, 37, 38, and 39, the pair-wise comparison with CCR values does not yield significantly different results. Noodle performance becomes better for higher CCR values regardless of degree of parallelism in the task graphs. The similar observation is possible for the comparison between number of edges and parallelism. Noodle performance again becomes a little better for higher number of edges regardless of degree of parallelism in the task graphs as pre-

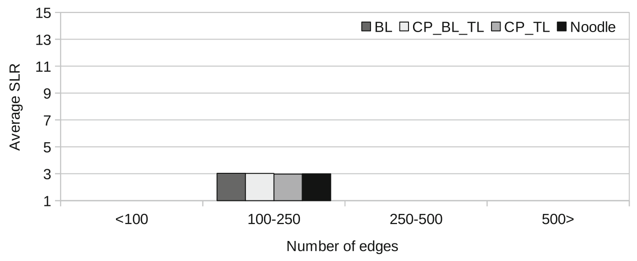


Fig. 40 Results for the given number of edges with parallelism 1–2

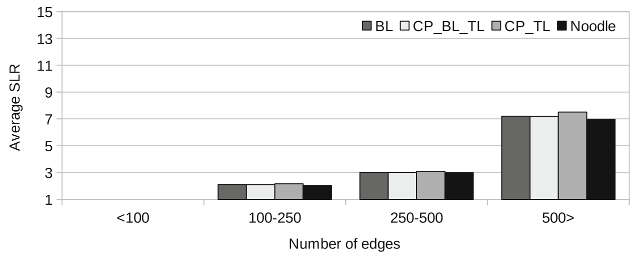


Fig. 41 Results for the given number of edges with parallelism 2–4

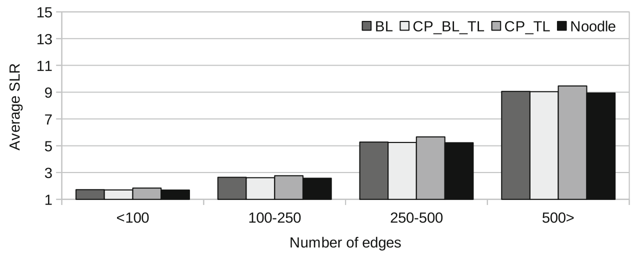


Fig. 42 Results for the given number of edges with parallelism 4–8

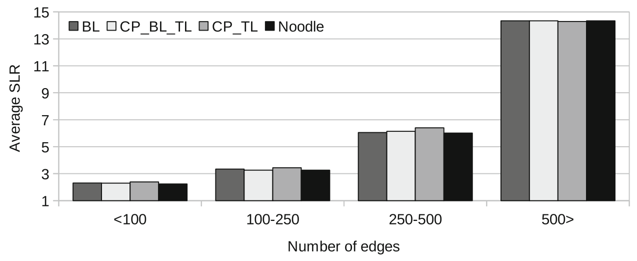


Fig. 43 Results for the given number of edges with parallelism 8–12

sented in Figures 40, 41, 42, and 43. While some ranges for number of edges are not available for some cases, e.g. there is no task graph instance that has number of edges less than 100 with degree of parallelism below 4 (Figs. 40, 41), trend does not differ for available data.

6 Conclusions and Future Work

This paper proposes a novel list scheduling heuristic, called the Noodle heuristic, for static task scheduling on parallel computing systems. The proposed heuristic combines the benefits of both depth-first and breadth-first approaches while assigning priorities to ready tasks. Noodle heuristic is based on the concept of maintaining proportionately fair attribution of priorities to ready tasks that results in improved resource utilisation and reduced schedule length. We conduct experimental evaluation of Noodle heuristic with task graphs with 50, 100, and 300 tasks per graph taken from Standard Task Graph (STG). We analyse results for execution scenarios with 2-, 4-, 8-, and 16-core systems and different degree of parallelism and number of edges in the task graphs. We report results demonstrating that Noodle produces schedules that are within a maximum of 12 % (in worst-case) of the optimal schedule for 2-, 4-, and 8-core systems. We also compare Noodle with existing scheduling heuristics and perform comparative analysis of its performance. Noodle outperforms existing heuristics for average value of Schedule Length Ratios (SLRs). As a future work, we intend to extend Noodle heuristic as an online scheduling algorithm.

References

1. Wolf, W.; Jerraya, A.A.; Martin, G.: Multiprocessor system-on-chip (mpsoc) technology. IEEE Trans. CAD ICs Syst. **27**(10), 1701–1713 (2008)
2. Grama, A.; Gupta, A.; Karypis, G.; Kumar, V.: Introduction to Parallel Computing, vol. 2nd edn. Pearson, A. Wesley, Boston (2003)
3. Sinnen, O.: Task scheduling for parallel systems. Wiley, New York. ISBN: 978-0-471-73576-2, p. 296, May (2007)
4. Semar Shahul, A.; Sinnen, O.: Scheduling task graphs optimally with a*. J. Supercomput. **51**(3), 310–332 (2010)
5. Sinnen, O.; Sousa, L.: List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. Parallel Comput. **30**(1), 81–101 (2004)
6. Sinnen, O.: Reducing the solution space of optimal task scheduling. Comput. OR. **43**, 201–214 (2014)
7. Sinnen, O.; Sousa, L.A.: Communication contention in task scheduling. IEEE Trans. Parallel Distrib. Syst. **16**(6), 503–515 (2005)
8. Darte, A.; Robert, Y. and Vivien, F.: Scheduling and automatic parallelization. BirkhŠuser, New York. ISBN 0-8176-4149-1, (2002)
9. Topcuouglu, H.; Hariri, S.; you Wu, M.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel Distrib. Syst. **13**(3), 260–274 (2002)
10. Suter, F.; Desprez, F. and Casanova, H.: From heterogeneous task scheduling to heterogeneous mixed parallel scheduling. In: Euro-Par 2004 Parallel Processing. pp. 230–237, (2004)
11. Kwok, Y.-K.; Ahmad, I.: Link contention-constrained scheduling and mapping of tasks and messages to a network of heterogeneous processors. Cluster Comput. **3**(2), 113–124 (2000)
12. Kasahara, H.; Narita, S.: Practical multiprocessor scheduling algorithms for efficient parallel processing. IEEE Trans. Comput. C **33**(11), 1023–1029 (1984)
13. Khan, M.A.: Scheduling for heterogeneous systems using constrained critical paths. Parallel Comput. **38**, 175–193 (2012)
14. Ahmad, I.; Kwok, Y.-K.: On exploiting task duplication in parallel program scheduling. IEEE Trans. Parallel Distrib. Syst. **9**(9), 872–892 (1998)

15. Kwok, Y.-K.; Ahmad, I.: Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. IEEE Trans. Parallel Distrib. Syst. **7**(5), 506–521 (1996)
16. Wu, M.-Y.; Gajski, D.: Hypertool: a programming aid for message-passing systems. IEEE Trans. Parallel Distrib. Syst. **1**(3), 330–343 (1990)
17. Yang, T.; Gerasoulis, A.: Dsc: scheduling parallel tasks on an unbounded number of processors. IEEE Trans. Parallel Distrib. Syst **5**(9), 951–967 (1994)
18. Iverson, M.A.; Ozguner, F. and Follen, G.J.: Parallelizing existing applications in a distributed heterogeneous environment. In: *HCW '95*, pp. 93–100 (1995)
19. Shahul, A.Z.; Sinnen, O.: Scheduling task graphs optimally with a*. J. Supercomput. **51**(3), 310–332 (2010)
20. Arabnejad, H.; Barbosa, J.: List scheduling algorithm for heterogeneous systems by an optimistic cost table. IEEE Trans. Parallel Distrib. Syst. **25**(3), 682–694 (2014)
21. Deelman, E.; Singh, G.; Su, M.-H.; Blythe, J.; Gil, Y.; Kesselman, C.; Mehta, G.; Vahi, K.; Berriman, G.B.; Good, J.; Laity, A.; Jacob, J.C.; Katz, D.S.: Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Sci. Prog. **13**(3), 219–237 (2005)
22. http://en.wikipedia.org/wiki/exponential_decay.
23. Set, S.T.G.: http://www.kasahara.elec.waseda.ac.jp/schedule.
24. Orsila, H.; Kangas, T.; Salminen, E.; Hamalainen, T.D.; Hannikainen, M.: Automated memory-aware application distribution for multi-processor system-on-chips. JSA **53**(11), 795–815 (2007)
25. de Langen, P.; Juurlink, B.: Leakage-aware multiprocessor scheduling. J. Signal Process. Syst. **57**(1), 73–88 (2009)