Cluster Comput (2006) 9:417–431
DOI 10.1007/s10586-006-0010-7

A task scheduling algorithm for arbitrarily-connected processors with awareness of link contention

Ali Fuat Alkaya · Haluk Rahmi Topcuoglu

Abstract In this paper, we present a new task scheduling algorithm, called Contention-Aware Scheduling (CAS) algorithm, with the objective of delivering good quality of schedules in low running-time by considering contention on links of arbitrarily-connected, heterogeneous processors. The CAS algorithm schedules tasks on processors and messages on links by considering the earliest finish time attribute with the virtual cut-through (VCT) or the store-and-forward (SAF) switching. There are three types of CAS algorithm presented in this paper, which differ in ordering the messages from immediate predecessor tasks. As part of the experimental study, the performance of the CAS algorithm is compared with two well-known APN (arbitrary processor network) scheduling algorithms. Experiments on the results of the synthetic benchmarks and the task graphs of the well-known problems clearly show that our CAS algorithm outperforms the related work with respect to performance (given in normalized schedule length) and cost (given in running time) to generate output schedules.

Keywords DAG scheduling · Task graphs · Link contention · Heterogeneous systems

1 Introduction

A heterogeneous computing system is a platform for executing computationally intensive parallel and distributed applications, which is provided by interconnecting diverse set of resources with high-speed networks. Efficient scheduling tasks of an application on available processors is a critical issue for achieving high performance in a given heterogeneous computing system. In the general form of task scheduling problem, an application is represented by a directed acyclic graph (DAG), where the nodes are tasks of the application and edges are the inter-task data dependencies. The objective of the problem is to map tasks onto processors and order their executions so that a minimum overall completion-time is satisfied by preserving the task-precedence constraints.

The task scheduling problem is NP-complete in its general case [1] as well as some restricted cases [2]; there are only a few known polynomial-time scheduling algorithms [3, 4] that were generated by placing restrictions on the input task graph. The general form of task scheduling problem for fully-connected homogeneous set of processors has been extensively studied and various heuristics were proposed in the literature [5–16]. A number of studies extend the general task scheduling problem by considering fully-connected heterogeneous processors [17–23]. Some of the recent studies for heterogeneous systems consider multiple phases of hybridizing different techniques or strategies [24–26].

Only a few of the proposed heuristics are for the systems with arbitrarily-connected processors, which are called the APN (arbitrary processor network) scheduling algorithms [27–31]. A scheduling algorithm for APNs should consider both scheduling tasks as well as communication traffic with equal importance. In this paper, we present a new algorithm, called Contention-Aware Scheduling (CAS) algorithm, with the objective of delivering good quality of schedules in low running time by considering link contention as well as heterogeneous processors connected with APNs. The CAS algorithm supports both the virtual cut-through

A. F. Alkaya · H. R. Topcuoglu (✉)
Computer Engineering Department, Marmara University, 34722 Istanbul, Turkey
e-mail: haluk@eng.marmara.edu.tr

(VCT) switching and the store-and-forward (SAF) switching for message transfer.

In this study, we consider three versions of the CAS algorithm, with respect to ordering the messages from immediate predecessor tasks. To test the effectiveness of the proposed algorithm, we conduct experiments with both synthetically-generated task graphs and task graphs of real applications. The comparison study shows that our algorithm significantly surpass the related work in terms of both performance (given in normalized schedule length) and cost (scheduling time to deliver an output schedule).

The rest of this paper is organized as follows. In the next section, we present assumptions and related terminology of the task scheduling problem. Section 3 presents a short summary of related work that are considered in the experimental study. In Section 4, we present the details of the CAS algorithm. Section 5 gives the details of how benchmark graphs are generated. In Section 6, we present the results of computational experiments based on randomly-generated task graphs and task graphs of well-known applications. The summary of the research is presented in Section 7.

2 Problem formulation

In this section, we define the application scheduling problem for arbitrarily-connected processors. Application and system models are presented, which is followed by a section for presenting the attributes used in contention-aware scheduling.

2.1 Application model

An *application* is represented by a directed acyclic graph, $G_1 = (V, E)$, where V is the set $\{T_1, T_2, \ldots, T_v\}$ consisting of v tasks, and E is the set of e edges between the tasks. Each edge $e_{ij} \in E$ represents the precedence constraint such that task T_i should complete its execution before task T_j starts. Each task T_i is associated with a base computation cost w_i^B, which is the execution time of task T_i on a dedicated reference processor in a heterogeneous computing system. Similarly, a base communication cost c_{ij}^B is associated with each edge e_{ij} for transferring data from task T_i to task T_j. Analytical benchmarking and profiling are the two techniques used for setting the estimated values of computation and commutation costs, which is out of scope of our paper.

2.2 System model

A heterogeneous computing system is represented by an undirected graph $G_2 = (P, L)$ where P is the set of p processors $\{P_1, P_2, \ldots, P_p\}$ which are connected through an arbitrary network. Each edge L_{mn} in L represents the full duplex link between processors P_m and P_n. Two heterogeneity factors (the processors heterogeneity factor and the link heterogeneity factor) are used to model the heterogeneity of resources in the system.

The heterogeneity factor of processor P_k for task T_i, hp_i^k, is determined by computing the difference in processing capabilities of processor P_k and the reference processor with respect to T_i. The average execution time for a given set of benchmarks is the main metric for measuring the processing capabilities. The computation cost of task T_i on processor P_k is given by the equation $w_i^k = w_i^B \times hp_i^k$, where w_i^B is the computation cost of task T_i on the reference machine.

If an edge e_{ij} for data transfer between tasks T_i and T_j is scheduled to the communication link between two neighbor processors P_x and P_y, the communication cost of the edge is given by $c_{ij}^B \times hl_{xy}$, where hl_{xy} is the link heterogeneity factor for the link L_{xy} that connects processors P_x and P_y. When both T_i and T_j are scheduled on the same processor, then the communication cost of the edge is zero, since we assume that the intra-processor communication cost is negligible when it is compared with the inter-processor communication cost.

2.3 Attributes used in contention-aware scheduling algorithm

In this section, we define the *EST* and *EFT* attributes, which are derived from a given partial schedule. $EST(T_j, P_k)$ and $EFT(T_j, P_k)$ are the earliest execution start time and the earliest execution finish time of task T_i on processor P_k, respectively. For the entry task T_{entry},

$$EST(T_{entry}, P_k) = 0. \tag{1}$$

For the other tasks in the graph, the *EFT* and *EST* values are computed recursively starting from the entry task, as shown in Eqs. (2) and (3), respectively. In order to compute the *EFT* of a task T_j, all immediate predecessor tasks of T_j must have been scheduled.

$$EST(T_j, P_k) = \max\{PRT(T_j, P_k), \max_{T_i \in pred(T_j)} DRT(e_{ij}, P_k, route_a)\} \tag{2}$$

$$EFT(T_j, P_k) = EST(T_j, P_k) + w_j^B \times hp_j^k, \tag{3}$$

where $pred(T_j)$ is the set of immediate predecessor tasks of task T_j, *PRT* is the processor ready time and *DRT* is the data ready time. More formally, $PRT(T_j, P_k)$ is the earliest available time of processor P_k to start the execution of task T_j; and $DRT(e_{ij}, P_k, route_a)$ is the earliest arrival time of data for the communication edge e_{ij}, which is sent from task T_i (scheduled on processor P_m) to the task T_j (scheduled on processor P_k) through the given route, $route_a$. The *DRT* value for the given route depends on whether the SAF switching

or the VCT switching is considered for message transfers, which are explained in detail in the following paragraphs.

In *SAF switching*, when a message traverses on a route with multiple links, each intermediate processor on the route forwards the message to the next processor after it receives and stores the entire message. Assume that the communication edge e_{ij} from task T_i (scheduled on P_{k_1}) to task T_j (scheduled on P_{k_n}) is mapped on a given route $route_a = \langle P_{k_1}, P_{k_2}, \ldots, P_{k_n} \rangle$. In the following equations, three new terms are introduced: CSTL (communication start time on the link), CFTL (communication finish time on the link) and LRT (link ready time). More formally, $CSTL(e_{ij}, L_{k_x k_{x+1}})$ and $CFTL(e_{ij}, L_{k_x k_{x+1}})$ is the earliest start time and finish time of the communication for the edge e_{ij} on the link $L_{k_x k_{x+1}}$, respectively; and $LRT(e_{ij}, L_{k_x k_{x+1}})$ is the earliest time when the link $L_{k_x k_{x+1}}$ is ready for the communication edge, e_{ij}. Then, DRT term for SAF switching is determined by the following set of equations:

$$CSTL(e_{ij}, L_{k_1 k_2}) = \max\{EFT(T_i, P_{k_1}), LRT(e_{ij}, L_{k_1 k_2})\} \quad (4)$$

$$CFTL(e_{ij}, L_{k_1 k_2}) = CSTL(e_{ij}, L_{k_1 k_2}) + c_{ij}^B \times hl_{k_1 k_2} \quad (5)$$

$$CSTL(e_{ij}, L_{k_x k_{x+1}}) = \max\{CFTL(e_{ij}, L_{k_{x-1} k_x}), LRT(e_{ij}, L_{k_x k_{x+1}}))\} \quad (6)$$

$$CFTL(e_{ij}, L_{k_x k_{x+1}}) = CSTL(e_{ij}, L_{k_x k_{x+1}}) + c_{ij}^B \times hl_{k_x k_{x+1}} \quad (7)$$

$$DRT(e_{ij}, P_k, route_a) = CFTL(e_{ij}, L_{k_{n-1} k_n}) \quad (8)$$

Equations (4) and (5) give the earliest start time and finish time of the communication edge e_{ij} on the first link of the selected route, respectively; and Eqs. (6) and (7) give the earliest start time and finish time of e_{ij} for subsequent links where $2 \leq x \leq n-1$. By using the (hl) term, the underlying system's link heterogeneity is modeled.

In *VCT switching*, the packet header is examined as soon as it is received, instead of waiting the entire packet to be received as in the SAF switching. Therefore, when a message traverses on a route with multiple links, the header and the following data bytes are started forwarding by each intermediate node as soon as routing decisions have done and the output buffer is free [32]. In case of busy output channel, the complete message is buffered at the node as in the SAF switching. Consequently, at high network loads VCT switching turns out to be SAF switching. Assume that the communication edge e_{ij} from task T_i to task T_j is mapped on a given route $route_a = \langle P_{k_1}, P_{k_2}, \ldots, P_{k_n} \rangle$, where T_i is scheduled on P_{k_1} and T_j is scheduled on P_{k_n}. Then, DRT term for VCT switching is determined by the following set of equations:

$$CSTL(e_{ij}, L_{k_1 k_2}) = \max\{EFT(T_i, P_{k_1}), LRT(e_{ij}, L_{k_1 k_2})\} \quad (9)$$

$$CFTL(e_{ij}, L_{k_1 k_2}) = CSTL(e_{ij}, L_{k_1 k_2}) + c_{ij}^B \times hl_{k_1 k_2} \quad (10)$$

$$CSTL(e_{ij}, L_{k_x k_{x+1}}) = \max\{CSTL(e_{ij}, L_{k_{x-1} k_x}) + c_{header} \times hl_{k_{x-1} k_x}, LRT(e_{ij}, L_{k_x k_{x+1}})\} \quad (11)$$

$$CFTL(e_{ij}, L_{k_x k_{x+1}}) = \max\{CSTL(e_{ij}, L_{k_x k_{x+1}}) + c_{ij}^B \times hl_{k_x k_{x+1}}, CFTL(e_{ij}, L_{k_{x-1} k_x}) + c_{header} \times hl_{k_x k_{x+1}}\} \quad (12)$$

$$DRT(e_{ij}, P_k, route_a) = CFTL(e_{ij}, L_{k_{n-1} k_n}) \quad (13)$$

As in the SAF switching, Eqs. (9) and (10) give the earliest start time and finish time of e_{ij} on the first link of the selected route, respectively. For the subsequent links, the equations represent the fact that forwarding starts right after the message header is received. The c_{header} term is added in order to include the propogation time of the message header, which is a constant value. In our experimental study, c_{header} term is set with the 5% of average message size of a given application graph.

After task T_m is scheduled on a processor P_j, the earliest finish time of task T_m on P_j is the actual finish time ($AFT(n_m)$) of the task. After all tasks in a graph are scheduled, the schedule length will be the actual finish time of the exit task T_{exit}. If there are multiple exit tasks and the convention of inserting a pseudo exit task is not applied, the schedule length (which is also called *makespan*) is defined as

$$makespan = \max\{AFT(T_{exit})\}. \quad (14)$$

The *objective function* of the task-scheduling problem is to determine the assignment of tasks of a given application to processors so that its schedule length is minimized.

3 Related work

There are only a few scheduling algorithms in the literature which target for arbitrarily-connected processors by considering contention on network links. This group of algorithms are called the APN (arbitrary processor network) scheduling algorithms [33]. Two well-known scheduling algorithms for APNs, the Dynamic-Level Scheduling (DLS) algorithm [27] and the Bubble Scheduling and Allocation (BSA) algorithm [29, 30], are considered in our comparison

study due to their significantly better performance than other APN scheduling algorithms [33].

3.1 Dynamic-level scheduling (DLS) algorithm

The DLS Algorithm [27] is a list scheduling heuristic that assigns the node priorities by using an attribute called *dynamic level* (DL). The dynamic level of a task T_i on a processor P_j is equal to

$$DL(T_i, P_j) = blevel^S(T_i) - EST(T_i, P_j), \tag{15}$$

which reflects how well task T_i and processor P_j are matched. The *blevel* value of a task T_i, is the length of the longest path from T_i to the exit task including all computation and communication costs on the path. The DLS algorithm uses static blevel value, $blevel^S$, which is computed by considering only the computation costs. At each scheduling step, the algorithm selects (ready node, available processor) pair that maximizes the value of the dynamic level. The computation costs of tasks are set with the median values. A new term, $\Delta(T_i, P_j)$, is added to Eq. (15) for heterogeneous processors, which is equal to the difference between the median execution time of task T_i and its execution time on processor P_j.

The DLS algorithm requires a message routing method that is supplied by the user; and no specific routing algorithm is presented in their paper. It should be noted that it does not consider the insertion-based approach for both scheduling tasks onto processors and scheduling messages on the links. The general DLS algorithm has an $O(v^3 \times p \times f(p))$ time complexity, where v is the number of tasks, p is the number of processors and $f(p)$ is the time-complexity of the message routing algorithm.

3.2 Bubble-scheduling and allocation (BSA) algorithm

The BSA algorithm [29, 30] is an incremental strategy that targets to improve the schedule length by migrating tasks from one processor to one of its neighbor. The first phase of the algorithm establishes a serial order among the nodes that is based on the order of the *critical path* nodes. Each task can be either a critical path (CP) task, or an in-branch (IB) task (i.e., the task that has a path to a CP task), or an out-branch (OB) task (i.e., the task that is not in one of the previous classes). Tasks of a graph are ordered according to the order of CP tasks. The IB tasks of each CP task are inserted right before the corresponding CP task, and the OB tasks are appended to this order according to their *blevel* values.

Then, tasks are scheduled to the pivot processor according to the serialization order. For the heterogeneous processors, pivot processor is the one which minimizes the critical path length [30]. Then, processors are listed in a breadth-first order starting from the pivot processor. At each step, a task on the pivot processor is considered for a possible migration to a neighbor processor, if this move decreases the finish time of the task. After all tasks are considered, the next processor in the list is selected as pivot, and the same procedure is applied. It is repeated until the processor list is completely examined.

How to update the start time for tasks and messages in case of a migration was not explained in their papers. Although there is no routing table used in BSA algorithm, it may be required to keep routing information during the scheduling phase. The BSA algorithm has a total complexity of $O(p^2 \times e \times v)$, for p processors, v tasks and e number of edges between tasks.

4 Contention-aware task scheduling algorithm

In this section, we present the details of our algorithm, the Contention-Aware Scheduling (CAS) algorithm, which provides efficient task scheduling for bounded-number of heterogeneous processors that are connected with arbitrary processor topologies. The first step (given in Fig. 1) specifies the links to be used for inter-task data communication according to the switching technique (the VCT or the SAF switching) by using the Dijkstra's shortest path algorithm.

When the Dijkstra's Algorithm is applied, the routing strategy determines how to compute the distance of a given route $route_a = \langle P_{k_1}, P_{k_2} \ldots, P_{k_n} \rangle$ between processors P_{k_1} and P_{k_n}. If the VCT switching is considered, it is equal to

$$Distance^{VCT}(route_a) = \max_{1 \le x \le n-1} (hl_{k_x k_{x+1}}), \tag{16}$$

where $hl_{k_x k_{x+1}}$ is the link heterogeneity factor for the link $L_{k_x k_{x+1}}$ that connects processors P_{k_x} and $P_{k_{x+1}}$. For the SAF switching, the summation of communication costs are considered along the path. Therefore, the "max" term is replaced with a summation:

$$Distance^{SAF}(route_a) = \sum_{x=1}^{n-1} (hl_{k_x k_{x+1}}). \tag{17}$$

After the computation and communication costs are set with the mean values, the next step (step 4) is the task prioritizing phase that computes the priorities of each task in the graph with the *blevel* values, since *blevel* based task prioritizing outperforms other alternative attributes [21]. The *blevel* value of task T_i, $blevel(i)$, is equal to the length of critical path from T_i to the exit node, including the computation of T_i. It is formally defined as:

$$blevel(i) = \overline{w_i} + \max_{T_j \in succ(i)} (\overline{c_{i,j}} + blevel(j)), \tag{18}$$

Fig. 1 The CAS algorithm

1. **If** (VCT Switching) **then** call $Dijktsra(Distance^{VCT})$
2. **else if** (SAF Switching) call $Dijktsra(Distance^{SAF})$
3. Set the computation costs of tasks and communication costs of edges with mean values.
4. Compute the priorities (given in *blevel*) of all tasks by traversing the graph upward.
5. Sort the tasks in a list, $list_T$, by non-increasing order of *blevel* values.
6. **while** there are unscheduled tasks in $list_T$ **do**
7. Select the highest priority task T_i from $list_T$.
8. Order the predecessors of T_i in a list $pred(T_i)$ based on the *predecessor-selection policy*.
9. **for** each processor P_x from the processor-list **do**
10. **for** each predecessor T_j (scheduled on P_y) from list $pred(T_i)$ (in the given order) **do**
11. **if** $(P_x \neq P_y)$ **then**
12. Schedule e_{ji} on $route\_a =< P_y ... P_x >$, according to the *switching technique*.
13. **endif**
14. **endfor**
15. Compute $EFT(T_i, P_x)$ using insertion-based scheduling.
16. **endfor**
17. Assign task T_i to the processor P_j that minimizes EFT of task T_i.
18. **endwhile**

where $succ(i)$ is the set of immediate successors of T_i, $\overline{c_{i,j}}$ is the average communication cost of edge (i, j) and $\overline{w_i}$ is the average computation cost of task T_i.

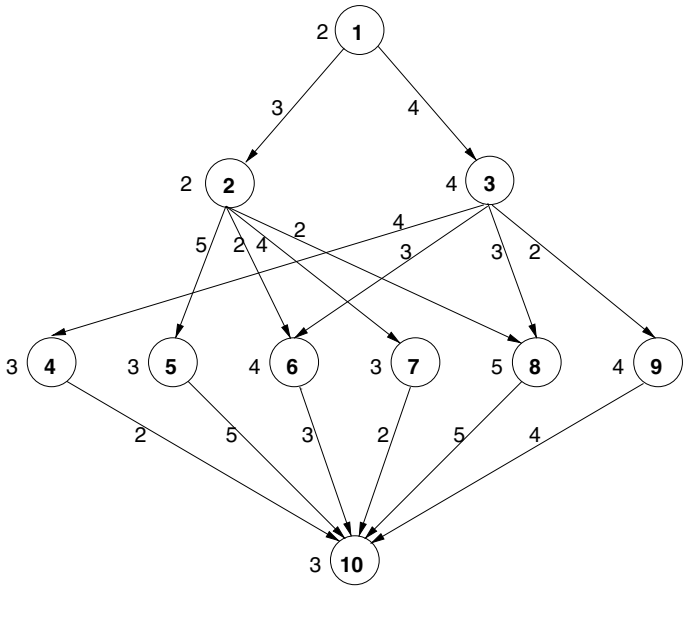
The scheduling process is performed according to the decreasing order of *blevel* values, given in the list, $list_T$. For each selected task T_i, the $EFT(T_i, P_j)$ attribute (the *earliest* finish time of T_i on P_j) is computed for each processor P_j based on Eqs. (2) and (3). Then, task T_i is scheduled to the processor which minimizes the *EFT* attribute (step 17). The *PRT* term in Eq. (2), the earliest available time of the selected processor to start the execution of the selected task, is computed by considering the possible insertion of the task in an earliest idle time slot between two already-scheduled tasks on the processor by preserving the precedence constraints, which is the applicability of insertion-based scheduling on processors.

In our CAS algorithm, the ready time of each link (i.e., the *DRT* term in the set of equations for the VCT and the SAF switchings) is determined based on insertion-based scheduling as in processor ready time. If the SAF switching is considered, the data ready time through a given route is set with the finish time of the communication edge on the last link of the given route. Although the data ready time through a given route for VCT switching is set similarly with the finish time on the last link of the given route, the earliest time of scheduling a message on intermediate links of the given route is the finish time of the header portion of the message on the previous link, not the finish time of the complete message.

As part of the scheduling process, it is also required to schedule incoming messages of each task onto links; and the order of scheduling incoming messages may affect the overall completion time of a given application. Each task in the scheduling list ($list_T$ in step 5) may have more than one incoming messages from its predecessor tasks. In our CAS algorithm, we consider three different *predecessor-selection policies* for ordering the incoming messages, which are called as CAS_1, CAS_2 and CAS_3.

- CAS_1—The list (which is named $pred(T_i)$ at line 8 of the code) is constructed in increasing order of finish times of the predecessor tasks.
- CAS_2—The list is constructed in increasing order of the summation of predecessor task's finish time and communication cost of the message with the predecessor task.
- CAS_3—The list is constructed increasing order of communication costs with the predecessor tasks.

The CAS algorithm has an order of $O(e \times p \times f(p))$, where e is the number of edges, p is the number of processors and $f(p)$ is the time-complexity of the message routing



Processor Heterogeneity Values

Task	P1	P2	P3	P4
1	3	4	3	4
2	3	4	2	5
3	3	4	4	3
4	2	2	4	5
5	2	5	5	5
6	5	4	4	5
7	5	4	2	3
8	3	3	4	4
9	2	2	3	4
10	4	4	3	2

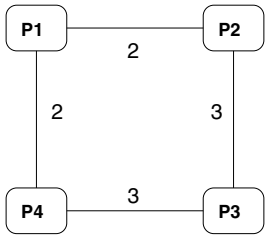
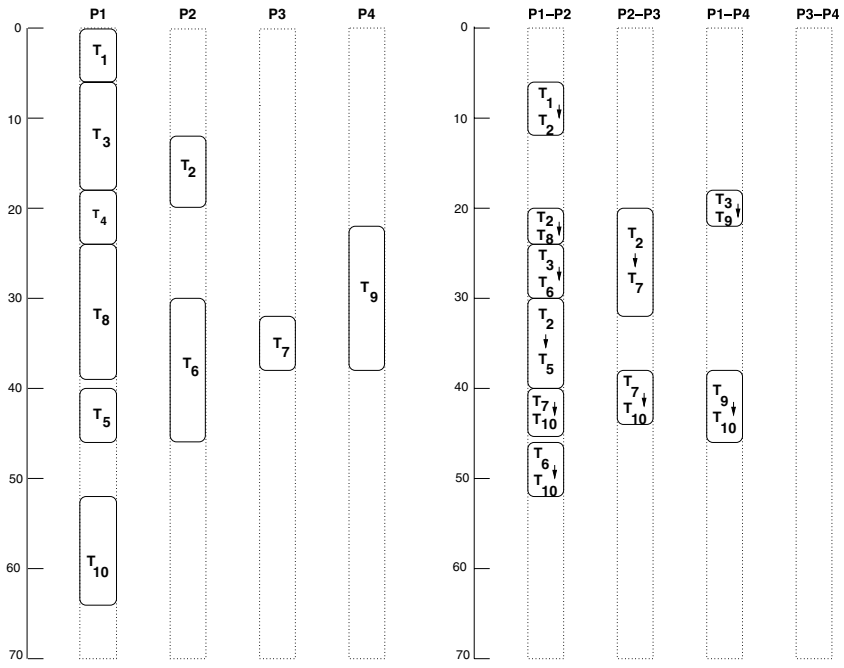


Fig. 2 An example task graph and a ring network topology

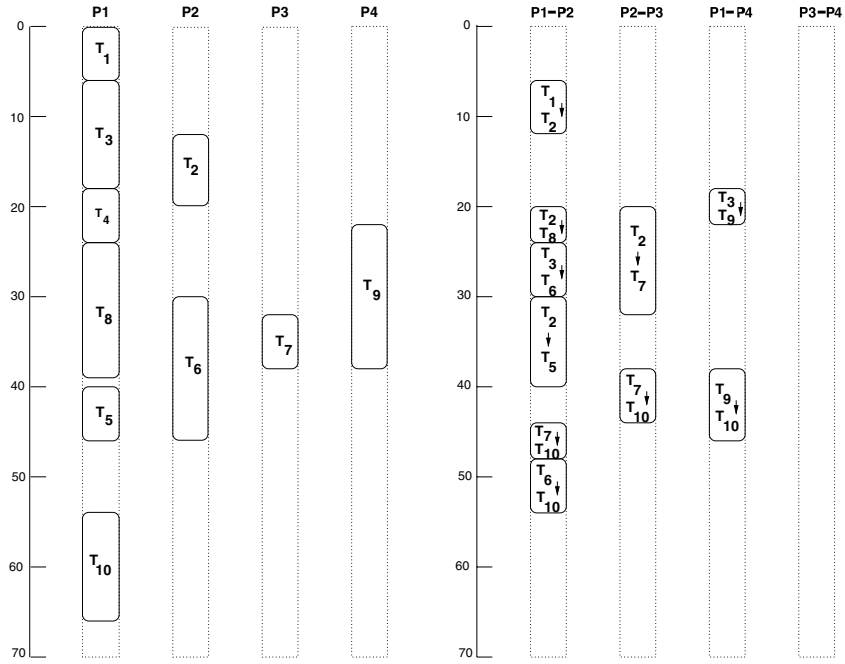
algorithm. For a dense graph when the number of edges is proportional to $O(v^2)$ (v denotes the number of tasks), then the time complexity turns out to be on the order of $O(v^2 \times p \times f(p))$.

Figure 2 gives an example task graph, processor topology and the values of processor heterogeneity parameters. Figures 3 and 4 present schedules obtained by the CAS algorithm for the task graph given in Fig. 2 by considering the VCT the SAF switchings, respectively. The schedule length is equal to 64 for the case of the VCT switching; and it is equal to 66 for the SAF switching. Message scheduling for the edge $e_{7,10}$ of the given graph is a good example to show the difference between the timings of the two switchings. It should be noted that the c_{header} term (given in Eqs. (11) and (12) is set to %20 in order to show the VCT switching more clearly.



(a) Processor Schedule (b) Link Schedule

Fig. 3 Schedule of the CAS algorithm using the VCT switching



(a) Processor Schedule (b) Link Schedule

Fig. 4 Schedule of the CAS algorithm using the SAF switching

5 Benchmark graph generation

In this section, we present the details of the benchmark graphs, which require generating random application graphs and random processor networks (i.e., topology graphs). For benchmark graph generation, we extend two models presented in the literature [21, 34]. An application graph generator is implemented to generate weighted application DAGs with various characteristics that depend on the following input parameters:

- Number of tasks in the application graph, (v).
- Shape coefficient of the graph, (α). It is the root of the ratio of average width (i.e., the number of tasks in a level) to the average height (i.e., the number of levels) of the graph. A lower-height graph with high parallelism degree generated by selecting $\alpha \gg 1.0$. Similarly, it generates a higher-depth graph with minimum parallelism capability, if $\alpha \ll 1.0$.
- Average fan-out degree of the tasks in a given graph, (out_d). To support fully-connectivity in application graphs, fan-out degree is also set with the maximum value in the experiments.
- Task heterogeneity limit, (ϕ_T). It represents the variation among the base computation costs of tasks on the base or reference processor. The base computation cost of each task T_i on a reference machine, w_i^B, is randomly assigned from the range $[1, \phi_T]$.
- Communication to computation ratio, (CCR). It is the ratio of the average communication cost to the average computation cost. By using the CCR parameter, the base communication cost of an edge (i, j) is set with $c_{ij}^B = random(CCR \times \phi_T)$.

Similarly, a processor-network generator is implemented to generate processor graphs with various network topologies and characteristics that depend on the following input parameters:

- Number of processing elements, (p).
- Topology of the network. The possible topologies in our benchmarks are ring, hypercube, arbitrarily-connected, and fully-connected networks.
- Connectivity parameter of a processor graph, (ρ). The number of communication links of a given processor in an arbitrary processor network is randomly assigned from a range $[1, \rho]$.
- Processor heterogeneity limit, (ϕ_M). It represents the variation among the computation costs of a task across all processors. The heterogeneity factor of processor P_k with respect to reference machine for the given task T_m, which is represented by hp_i^k, is set from the range $[1, \phi_M]$. For setting the heterogeneity factors, we consider both *consistent* and *inconsistent* matrices in our simulation model. In the former case, whenever a machine P_j executes a task T_i faster than machine P_k, then machine P_j executes all tasks faster than machine P_k. In the latter case, machine P_j is faster than machine P_k for some tasks and slower for others.

Table 1 Values of parameters used in computational experiments

Parameter	Set of values
Number of tasks (v)	20 40 60 80 **100** 150 200 250
Shape parameter (α)	0.1 0.2 0.5 **1.0** 2.0 5.0 10.0
Fan-Out Degree (out_d)	2, **5**, 10, fully_connected
Task heterogeneity limit (ϕ_T)	1, 10, **100**, 1000
CCR Value	0.1 0.2 0.5 **1.0** 2.0 5.0
Number of processors (p)	2, 4, 8, **16**, 32, 64
Processor heterogeneity limit (ϕ_M)	1, 5, 10, 20, 50, **100**, 500, 1000
Link heterogeneity limit (ϕ_L)	1, 2, 4, 6, 8, **10**, 50, 100

- Link heterogeneity limit, (ϕ_L). It represents the variation among the communication costs of a given edge for all inter-processor links. The link heterogeneity factor, $hl_{x,y}$, is set from the range $[1, \phi_L]$.

The value of each parameter is assigned from its set given in Table 1 for computational experiments presented in Section 6. The connectivity parameter is not included in the table, since it is only considered for arbitrarily-connected processors. In each experiment, the values for one or more number of parameters are varied from the given set, by keeping remaining parameters fixed. Bold numbers in Table 1 are the fixed values of the parameters.

6 Results and discussion

The following three comparison metrics are applied for evaluating the performance of the algorithms in the computational experiments:

- Normalized Schedule Length (NSL), which is expressed as:

$$NSL = \frac{makespan}{\sum_{T_i \in CP_{min}} w_i^B \times \min_{j \in P} \left\{ hp_i^j \right\}} \tag{19}$$

where makespan is the schedule length produced by the algorithm. For an unscheduled graph, if the computation costs of each task is set with the minimum value among the various processor alternatives, then the critical path of the graph is represented as CP_{MIN}. The summation term in the denominator is the value obtained by adding minimum computation costs of tasks that are on the critical path CP_{MIN}. The NSL value of an output schedule can not be lower than 1, since the denominator is a lower bound for the output schedule. Since a large set of task graphs with different properties is used, it is necessary to normalize each schedule length to its lower bound.

- Running time (scheduling time) of an algorithm, which is the execution time to obtain the output schedule of a given task graph.
- Total number of processors used in the output schedule.

The results of computational experiments given in the following subsections present the effectiveness of our algorithm with respect to the characteristics of both the application graphs and the topology graphs (i.e.,the target processor networks).

6.1 Performance study with respect to graph size

The first experiment compares the performance of algorithms with respect to various graph sizes (see Table 2). In this experiment, four different network topologies are examined with both the VCT and the SAF switching techniques, which generates 8 different test cases. The target platform of scheduling is one of the four different network topologies with 16 processors, by considering both the VCT and the SAF switchings. For each graph size (i.e., each row in Table 2), the best results are shown in bold for two switching techniques considered. The performance ranking of the algorithms for all topologies (starting from the algorithm with best performance) is $\{CAS_1, CAS_2, CAS_3, DLS, BSA\}$. The NSL values decrease with an increase in connectivity of the network. Specifically, messages are scheduled on less number of links in a ring network because of its low-connectivity, which causes an increase in contention on the links; and it subsequently increases average NSL values. As expected, the results of SAF and VCT switching cases are same for a fully-connected network topology.

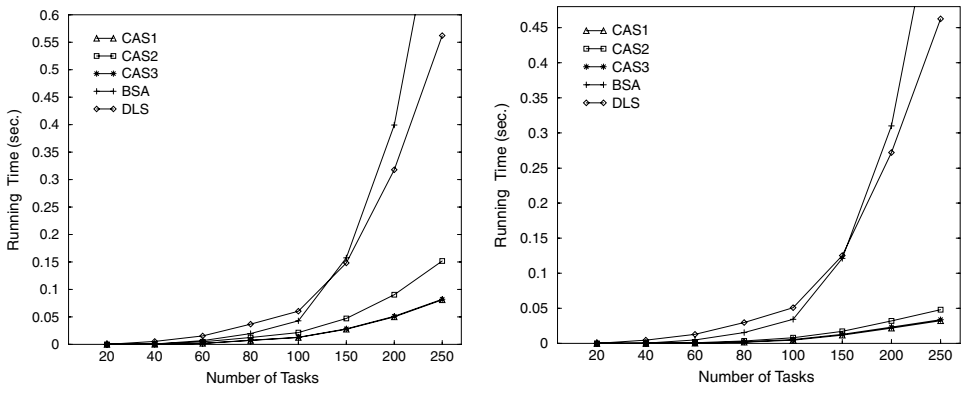
Figure 5 shows the running times of the algorithms tested on arbitrarily-connected processors. The three versions of the CAS algorithm require significantly less amount of time to generate output schedules; the BSA algorithm and the DLS algorithms requires up to 93% more time than the CAS_1 algorithm. The same ranking of the algorithms is obtained for the other network topologies.

6.2 Performance study with respect to CCR values

The quality of schedules generated by the algorithms with respect to CCR values on different topologies with the VCT switching are presented in Fig. 6. Our algorithms outperform

Table 2 Average NSL values with respect to graph size for various network topologies

Topology	Graph size	VCT switching					SAF switching				
		CAS_1	CAS_2	CAS_3	BSA	DLS	CAS_1	CAS_2	CAS_3	BSA	DLS
Ring	20	**3.80**	3.82	3.86	5.92	3.92	**4.43**	**4.43**	4.47	5.95	4.65
	40	**5.55**	5.57	5.62	9.02	5.78	**6.28**	6.29	6.32	8.94	6.55
	60	**7.63**	7.64	7.69	12.76	7.95	**8.51**	8.53	8.55	12.77	8.88
	80	**8.93**	8.95	9.02	15.35	9.46	**9.93**	9.95	9.99	15.37	10.56
	100	**10.13**	10.14	10.22	17.68	10.81	**11.16**	11.19	11.25	17.92	12.03
	150	**13.60**	13.62	13.72	23.87	14.58	**14.74**	14.77	14.87	24.21	16.10
	200	**16.18**	16.20	16.33	29.48	17.39	**17.23**	17.26	17.41	29.86	19.06
	250	**17.85**	17.87	18.01	31.31	19.31	**18.85**	18.88	19.06	32.27	21.14
Hypercube	20	**3.15**	3.20	3.27	5.05	3.31	**3.26**	3.29	3.32	4.84	3.43
	40	**4.33**	4.36	4.46	8.10	4.62	**4.21**	4.22	4.29	7.65	4.46
	60	**5.68**	5.72	5.87	10.52	6.14	**5.44**	5.45	5.51	10.35	5.76
	80	**6.29**	6.32	6.48	13.81	6.96	**6.01**	6.02	6.10	13.41	6.52
	100	**7.09**	7.11	7.30	15.45	7.94	**6.52**	6.53	6.64	14.90	7.20
	150	**9.20**	9.23	9.46	19.05	10.46	**8.44**	**8.44**	8.58	18.59	9.38
	200	**10.53**	10.56	10.87	23.83	12.24	**9.54**	**9.54**	9.70	23.41	10.78
	250	**11.54**	11.57	11.88	26.65	13.60	**10.30**	10.31	10.48	26.63	11.83
Arbitrarily-Connected	20	**2.69**	2.73	2.78	4.93	2.81	**2.79**	2.81	2.85	4.86	2.92
	40	**3.61**	3.63	3.70	7.76	3.79	**3.58**	3.59	3.65	7.48	3.77
	60	**4.35**	4.37	4.48	9.75	4.64	**4.22**	**4.22**	4.29	9.30	4.45
	80	**5.55**	5.57	5.66	11.85	5.93	**5.46**	5.47	5.54	11.35	5.81
	100	**5.61**	5.63	5.73	13.62	6.17	**5.38**	**5.38**	5.46	13.30	5.87
	150	**6.80**	6.82	6.96	17.15	7.46	**6.29**	**6.29**	6.39	16.63	6.86
	200	**8.14**	8.15	8.32	22.82	9.01	**7.73**	**7.73**	7.85	22.53	8.49
	250	**8.82**	8.84	8.95	28.44	9.65	**8.29**	8.30	8.39	28.32	9.09
Fully-Connected	20	**2.16**	2.18	2.19	4.34	2.22	**2.16**	2.18	2.19	4.02	2.22
	40	**2.41**	2.45	2.49	7.18	2.54	**2.41**	2.45	2.49	6.40	2.54
	60	**2.82**	2.85	2.90	9.11	2.96	**2.82**	2.85	2.90	8.11	2.96
	80	**2.96**	2.99	3.04	11.91	3.12	**2.96**	2.99	3.04	11.35	3.12
	100	**3.11**	3.13	3.18	13.07	3.28	**3.11**	3.13	3.18	12.48	3.28
	150	**3.69**	3.71	3.77	16.57	3.90	**3.69**	3.71	3.77	15.43	3.90
	200	**4.01**	4.03	4.09	18.60	4.25	**4.01**	4.03	4.09	18.12	4.25
	250	**4.34**	4.36	4.42	22.23	4.60	**4.34**	4.36	4.42	22.47	4.60



(a) VCT Switching (b) SAF Switching

Fig. 5 Average running times of algorithms with different graph sizes

the related work and the performance ranking of the algorithms is {CAS_1, CAS_2, CAS_3, *DLS*, *BSA*}. The algorithms provide higher NSL values for communication-intensive task graphs, i.e., the graphs with higher CCR values. An increase in average communication costs increases the makespan value, which results in higher NSL values. The same ranking is observed when the same experiments are repeated with the SAF switching.

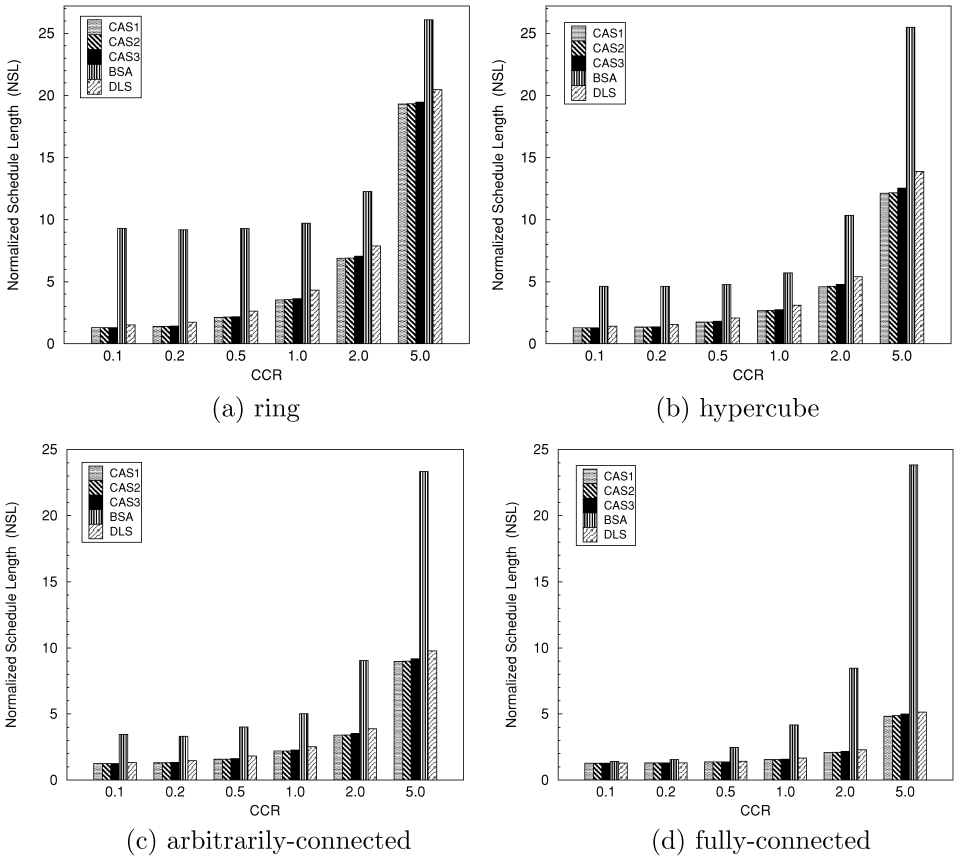


Fig. 6 Performance of algorithms with different CCR values

6.3 Performance study with respect to graph structures

Our rationale for using graphs with various structures in our experiments is to avoid any bias that an algorithm may have toward a particular graph structure. Figure 7 presents the performance of algorithms with respect to various graph structures by varying the shape coefficient and considering the graphs with 100 nodes on an arbitrary network topology of 16 processors. The three CAS algorithms outperform the related work for all cases; and the performance difference between our methods and related work become more significant at high shape coefficients.

An increase in value of the shape coefficient increases the degree of parallelism, which causes more number of tasks that are not on the critical path. When the degree of parallelism is not tolerated with the number of processors (i.e., the parallelism degree is significantly higher than the number of processors), an increase in NSL value occurs due to increase in schedule length and decrease (or not-change) in the denominator part of the NSL equation. When Fig. 7 is

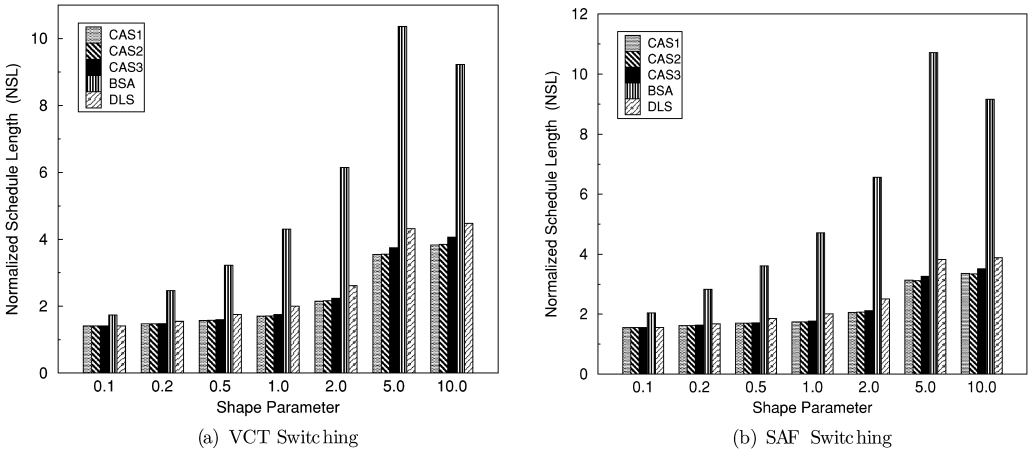
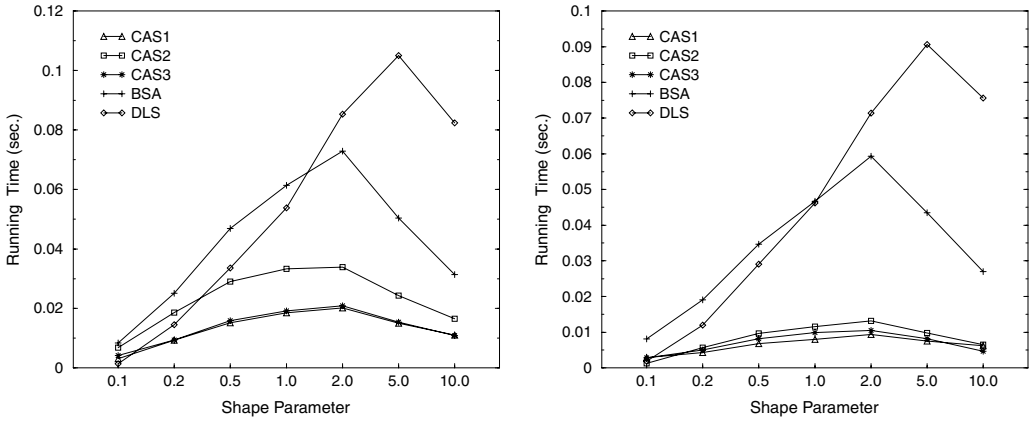


Fig. 7 Performance of algorithms with different graph structures



(a) VCT Switching (b) SAF Switching

Fig. 8 Running time of algorithms for various graph structures

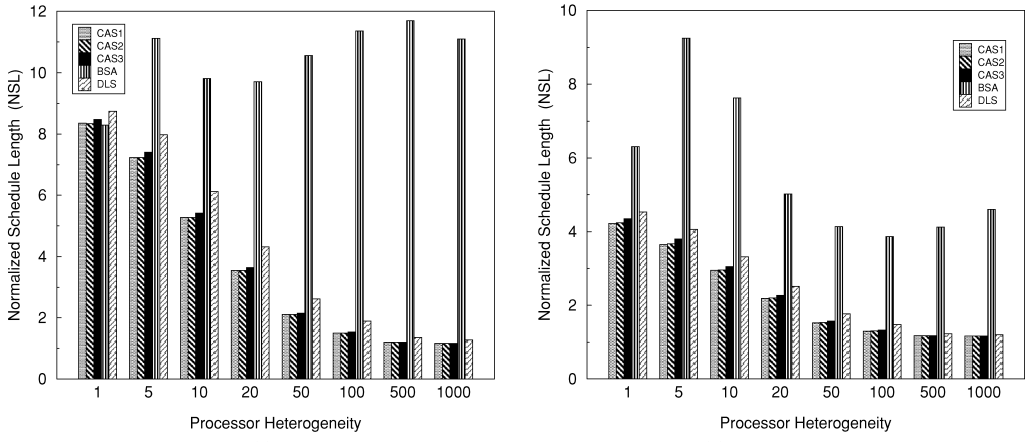
examined, the NSL values of algorithms start to increase when $\alpha \geq 2.0$. If $\alpha = 2.0$, one or more levels of the task graph will have more than 16 tasks; therefore, the degree of parallelism can not be handled with the available number of processors.

When the total number of nodes are fixed, the graphs with very low or very high α values (i.e., $\alpha < 0.5$ or $\alpha > 2.0$) require less number of edges than balanced graphs with equal height and width values. The scheduling algorithms require more time to generate output schedule for the graphs with the higher number of edges; therefore the running time of the algorithms increase up to some level and then they decrease with an increase in shape coefficient (Fig. 8). The CAS algorithm require less time than the related work, which is consistent with the results given in Fig. 5. Although, the results given in this section is for the arbitrarily-connected processors, same performance rankings are observed for the other networks.

6.4 Performance study with respect to link and processor heterogeneity limits

An increase in link heterogeneity limit causes differences in communication costs of each message over various links of the network topology. Consequently, it increases the schedule length of an application, which is true for both the VCT and the SAF switchings. The three versions of our CAS algorithm outperform the related work for both low and high link heterogeneity values.

An increase in processor heterogeneity parameter generates variance in computation costs of each task among processors. This change increases the average computation costs of tasks, whereas the communication costs may not increase. When a significant increase occurs for processor heterogeneity limit (see Fig. 9), the numerator part (which consists of computation and communication costs) of NSL term becomes closer to the denominator part (which consists



(a) ring (b) arbitrarily-connected

Fig. 9 Performance of algorithms with different processor heterogeneity limits

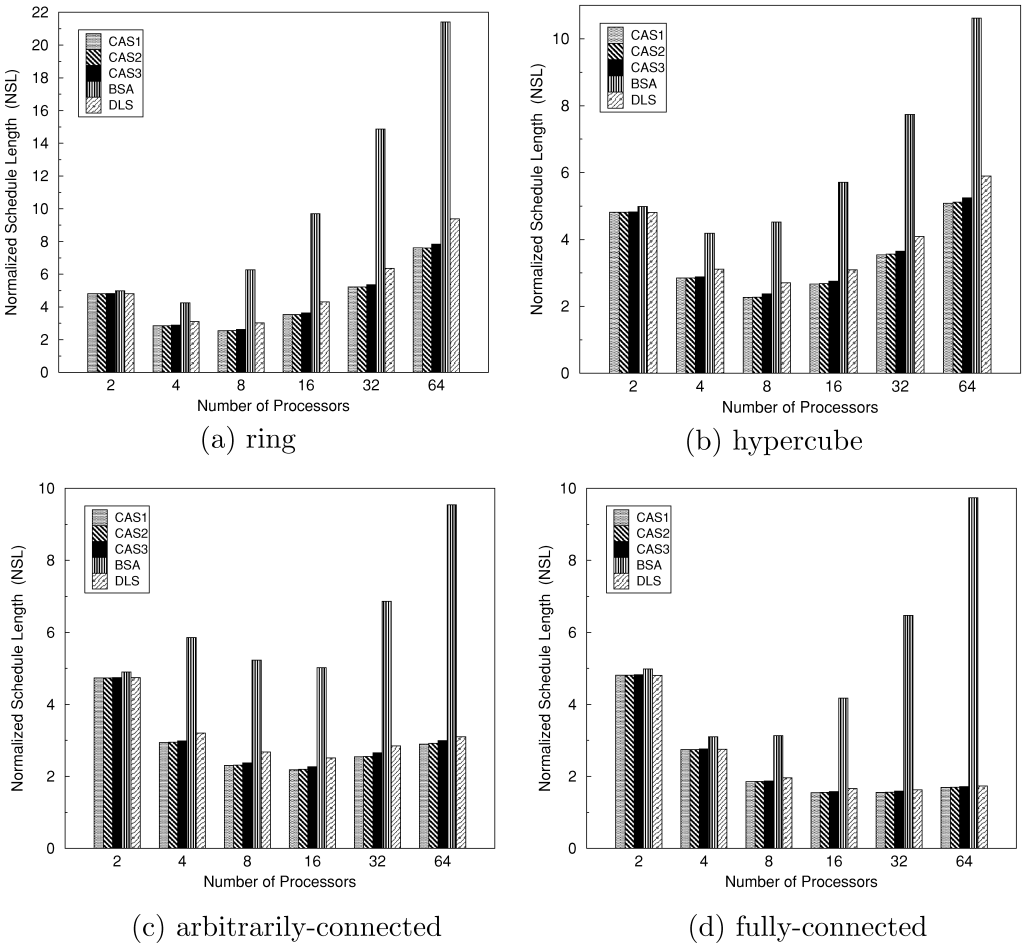


Fig. 10 Performance of algorithms with different processor sizes

of only computation costs). Average NSL values decrease with an increase in processor heterogeneity limit for hypercube and fully-connected topologies with VCT switching, which is also valid for other networks.

6.5 Performance study with respect to processor availability

If the results given in Fig. 10 are examined, the algorithms other than BSA algorithm exploit almost equal average NSL values. Additionally, the NSL values of algorithms first decrease and then increase in both ring and hypercube networks, when the number of processors is increased. The increase is not significant for arbitrarily-connected networks; and there is no increase for fully-connected networks.

The contention of links in a fully-connected topology may not increase with an increase in number of processors, due to high edge connectivity in the topology. Therefore, the NSL values do not increase, with an increase in processor availability. A similar observation can be done for the arbitrarily-connected processors. On the other hand, the connectivity degree remains unchanged in a ring network and it slightly increases in a hypercube network in case of a significant increase in processor availability. Therefore, the lengths of output schedules increase with an increase in number of processors, when ring or hypercube networks are considered (Fig. 10(a) and 10(b)).

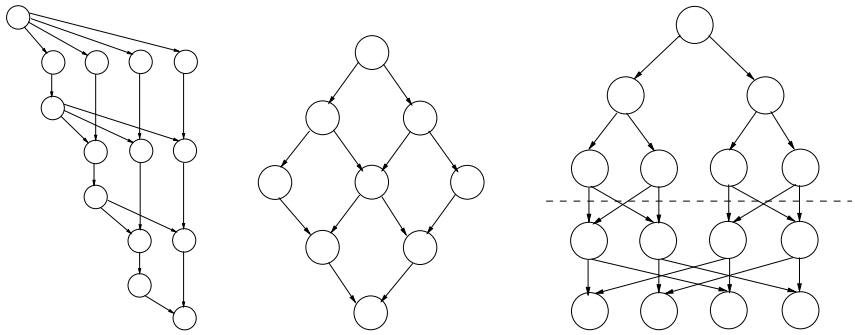
The actual processor usage of algorithms are also compared as part of this study. The BSA algorithm requires significantly less number of processors (up to 5 times less number of processors) than the other algorithms; and the CAS algorithms and the DLS algorithm require almost equal number of processors for each topology. The BSA algorithm schedule all tasks to a single processor and then targets to move the tasks to neighbor processors in the topology; therefore, it comes up with the lowest processor usage.

6.6 Experimental evaluation based on task graphs of well-known problems

In this section, we consider task graphs that capture the representations and features of the three well-known problems,

Table 3 Performance of algorithms on the application graphs given in Fig. 11

Application	NSL			Time			Processor usage		
	CAS_1	BSA	DLS	CAS_1	BSA	DLS	CAS_1	BSA	DLS
Gauss elimination [5, 10, 15, 20, 25]	3.04	4.49	3.06	0.01	0.03	0.03	6.11	5.51	5.90
Laplace equation solver [5, 10, 15, 20, 25]	3.56	6.81	3.57	0.04	0.25	0.12	5.64	5.22	5.77
FFT [2, 4, 8, 16, 32]	3.42	5.14	3.42	0.01	0.02	0.03	5.55	5.00	5.63



(a) Gauss (b) Laplace (c) FFT

Fig. 11 Task graphs of three well-known problems

which are Gauss elimination [35, 36], Laplace Equation Solver [35] and Fast Fourier Transformation [37]. Figures 11(a) gives the task graph of Gauss elimination algorithm, when the dimension of the matrix is equal to 5. Similarly, Fig. 11(b) gives the task graph of a parallel Laplace application based on the Gauss-Seidel algorithm. The task graph of a recursive, one-dimensional FFT Algorithm [37] for the case of four data points is given in Fig. 11(c). This graph can be divided into two parts—the tasks above the dashed line are the recursive call tasks and the ones below the line are the butterfly operation tasks.

Since the graph structures of the given applications are known and fixed, several graph parameters including α, number of tasks, out degree values are not considered during the graph generation phase; a new parameter, which is matrix size for Gauss elimination and Laplace Equation Solver or number of data points for FFT algorithm, is considered instead. The CCR, processor heterogeneity and task heterogeneity parameters are set from a given range. The number of available processors is set to 8 in our experiments. Each row of Table 3 presents the average results of various matrix sizes (or data points) for the outputs generated by the algorithms. The CAS algorithm and the DLS algorithm significantly outperform the BSA algorithm with respect to average NSL values. The CAS algorithms require less than or equal time with the DLS algorithm to generate the results. As in the previous subsection, the BSA algorithm allocates less number of processors than the other algorithms.

7 Conclusions

In this paper, we presented a new scheduling algorithm for arbitrarily-connected heterogeneous processors, called Contention-Aware Scheduling (CAS) algorithm, with the objective of delivering good quality of schedules with lower cost by considering contention on links. The algorithms supports both the VCT switching and the SAF switching for message scheduling. As part of the experimental study, the performance of our algorithm was compared with the two-well known heuristics, the BSA algorithm and the DLS algorithm. From the figures and tables given in the previous section, following observations can be done:

- The CAS algorithm (by including all versions) significantly outperform the related work. The best results are observed by the CAS_1 version, which selects the incoming messages according to the increasing order of finish times of the predecessor tasks.
- The CAS algorithm requires significantly lower cost (up to 93% less running time) to generate output schedules than the related work. Additionally, all algorithms require less time to generate the output schedule for the SAF switching than the VCT switching, which is due to higher complexity in determining the available intervals in the VCT switching for scheduling an inter-task communication on a given route.
- The BSA algorithm requires less number of processors than other algorithms for each experiment presented. It is

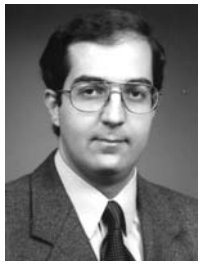
due to the fact that it first schedules all tasks on a single processor and targets to migrate them to the neighboring processors. This technique may prevent of mapping a task onto the processor which minimizes the finish time of the task, which leads to significantly higher schedule lengths. Additionally, the number of processors used by the BSA algorithm decreases dramatically when the connectivity level in a processor topology decreases.

References

1. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness* (W. H. Freeman and Co., 1979).
2. J.D. Ullman, NP-complete scheduling problems, Journal of Computer and Systems Sciences 10 (1975) 384–393.
3. T.C. Hu, Parallel sequencing and assembly line problems, Operational Research 9(6)(1961) 841–848.
4. E.G. Coffman, *Computer and Job-Shop Scheduling Theory* (John Wiley and Sons, 1976).
5. M. Wu and D. Gajski, Hypertool: A programming aid for message passing systems, IEEE Transactions on Parallel and Distributed Systems 1 (July 1990) 330–343.
6. Y. Kwok and I. Ahmad, Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors, IEEE Transactions on Parallel and Distributed Systems 7 (May 1996) 506–521.
7. E.S.H. Hou, N. Ansari, and H. Ren, A genetic algorithm for multiprocessor scheduling, IEEE Transactions on Parallel and Distributed Systems 5 (February 1994) 113–120.
8. I. Ahmad and Y. Kwok, A new approach to scheduling parallel programs using task duplication, in: *Proc. of Int'l Conference on Parallel Processing*, vol. II (1994) 47–51.
9. T. Yang and A. Gerasoulis, DSC: Scheduling parallel tasks on an unbounded number of processors, IEEE Transactions on Parallel and Distributed Systems 5 (September 1994) 951–967.
10. M. Wu, W. Shu and J. Gu, Local search for DAG scheduling and task assignment, in: *Proc. of 1997 Int'l Conference on Parallel Processing* (1997) pp. 174–180.
11. R.C. Correa, A. Ferreria, and P. Rebreyend, Integrating list heuristics into genetic algorithms for multiprocessor scheduling, in: *Proc. of Eight IEEE Symp. on Parallel and Distributed Processing (SPDP96)* (October 1996).
12. H. El-Rewini, H.H. Ali, and T. Lewis, Task scheduling in multiprocessor systems, IEEE Computer (December 1995) 27–37.
13. G. Park, B. Shirazi, and J. Marquis, DFRN: A new approach for duplication based scheduling for distributed memory multiprocessor systems, in: *Proc. of Int'l Conference on Parallel Processing* (1997) pp. 157–166.
14. Benjamin S. Macey, and Albert Y. Zomaya, A performance evaluation of CP list scheduling heuristics for communication intensive task graphs, in: *Proceedings of the Joint 12th International Parallel Processing Symposium/9th Symposium on Parallel and Distributed Processing*, March 30–April 3, Orlando, Florida, (1998) pp. 538–541.
15. T.L. Adam, K.M. Chandy, and J.R. Dickson, A comparison of list schedules for parallel processing systems, Communication of the ACM 17 (1974) 685–689.
16. B. Kruatrachue, Static task scheduling and grain packing in parallel processing systems, PhD Thesis, Oregon State University, (1987).
17. M. Iverson, F. Ozguner, and G. Follen, Parallelizing existing applications in a distributed heterogeneous environment, in: *Proc. of Heterogeneous Computing Workshop* (1995) pp. 93–100.
18. M. Maheswaran and H.J. Siegel, A dynamic matching and scheduling algorithm for heterogeneous computing systems, in: *Proc. of Heterogeneous Computing Workshop* (1998) pp. 57–69.
19. L. Wang, H.J. Siegel, and V.P. Roychowdhury, A genetic-algorithm-based approach for task matching and scheduling in heterogeneous computing environments, in: *Proc. of Heterogeneous Computing Workshop* (1996).
20. H. Singh and A. Youssef, Mapping and scheduling heterogeneous task graphs using genetic algorithms, in: *Proc. of Heterogeneous Computing Workshop* (1996) pp. 86–97.
21. H. Topcuoglu, S. Hariri, and M. Wu, Performance effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems 13(3) (March 2002) 260–274.
22. T. Hagras and J. Janecek, A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems, Parallel Computing 31 (2005) 653–670.
23. A.S. Wu, H. Yu, S. Jin, K. Lin, and G. Schiavone, An incremental genetic algorithm approach to multiprocessor scheduling, IEEE Transactions on Parallel and Distributed Systems 15 (2004) 824–834.
24. C. Boeres and A. Lima, Hybrid task scheduling: Integrating static and dynamic heuristics, in: *Proc. of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD03)*, Sao Paulo, Brazil, 10–12 November (2003) pp. 199–206.
25. R. Sakellariou and H. Zhao, A hybrid heuristic for dag scheduling on heterogeneous systems, in: *Proc. of International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico (2004).
26. G.Q. Liu, K.L. Poh and M. Xie, Iterative list scheduling for heterogeneous computing, Journal of Parallel and Distributed Computing 65 (2005) 654–665.
27. G.C. Sih and E.A. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Transactions on Parallel and Distributed Systems 4 (February 1993) 175–186.
28. H. El-Rewini and T.G. Lewis, Scheduling parallel program tasks onto arbitrary target machines, Journal of Parallel and Distributed Computing 9 (1990) 138–153.
29. Y. Kwok and I. Ahmad, Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures, in: *Proc. of Symposium on Parallel and Distributed Processing (SPDP)* (1995) pp. 36–43.
30. Y. Kwok and I. Ahmad, Link contention-constrained scheduling and mapping of tasks and messages to a network of heterogeneous processors, Cluster Computing: The Journal of Networks, Software Tools and Applications 3 (2000) 113–124.
31. J.M. Orduna, F. Silla, and J. Duato, On the development of a communication-aware task mapping technique, Journal of System Architecture 50 (2004) 207–220.
32. J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach* (Morgan Kaufmann Publishers, 2003).
33. Y. Kwok and I. Ahmad, Benchmarking and comparison of the task graph scheduling algorithms, Journal of Parallel and Distributed Computing 59 (1999) 381–422.
34. T. Braun, H.J. Siegel, N. Beck, L.L. Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hengsen, and R.F. Freund, A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems, in: *Proc. of Heterogeneous Computing Workshop* (1999) pp. 15–29.

35. M. Wu and D. Gajski, Hypertool: A programming aid for message passing systems, IEEE Transactions on Parallel and Distributed Systems 1 (July 1990) 330–343.
36. M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystram, Parallel Gaussian elimination on an MIMD computer, Parallel Computing, 6 (1988) 275–295.
37. Y. Chung and S. Ranka, Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors, in: *Proc. of Supercomputing* (November 1992) pp. 512–521.

Ali Fuat Alkaya received the B.Sc. degree in mathematics from Koc University, Istanbul, Turkey in 1998, and the M.Sc. degree in computer engineering from Marmara University, Istanbul, Turkey in 2002. He is currently a Ph.D. student inengineering management department at the same university. His research interests include task scheduling and analysis of algorithms.



Haluk Rahmi Topcuoglu received the B.Sc. and M.Sc. degrees in computer engineering from Bogazici University, Istanbul, Turkey, in 1991 and 1993, respectively. He received the Ph.D. degree in computer science from Syracuse University in 1999. He has been on the faculty at Marmara University, Istanbul, Turkey since Fall 1999, where he is currently an Associate Professor in computer engineering department. His main research interests are task scheduling and mapping in parallel and distributed systems; parallel processing; evolutionary algorithms and their applicability for stationary and dynamic environments. He is a member of the ACM, the IEEE, and the IEEE Computer Society.
e-mail: haluk@eng.marmara.edu.tr
e-mail: falkaya@eng.marmara.edu.tr