A High Performance, Low Complexity Algorithm for Compile-Time Task Scheduling in Heterogeneous Systems

Tarek Hagras, Jan Janeček
Czech Technical University in Prague
Dept. of Computer Science and Engineering
tarek@felk.cvut.cz, janecek@cs.felk.cvut.cz

Abstract

The heterogeneous computing environment is an interesting computing platform due to the fact that a single parallel architecture may not be adequate for exploiting all of a program's available parallelism. In some cases, heterogeneous systems have been shown to produce higher performance for lower cost than a single large machine. Task scheduling is the key issue when aiming at high performance in this kind of environment. A large number of scheduling heuristics have been presented in the literature, most of them target only homogeneous computing systems. In this paper we present a simple scheduling algorithm based on list-scheduling and task-duplication on a bounded number of heterogeneous machines called Heterogeneous Critical Parents with Fast Duplicator (HCPFD). The analysis and experiments have shown that HCPFD outperforms on average all other higher complexity algorithms.

Keywords:

list scheduling, compile time scheduling, task graph scheduling, heterogeneous computing.

1. Introduction

A heterogeneous computing system is defined as a distributed suite of computing machines with diverse capabilities interconnected by diverse high speed links utilized to execute parallel programs[1]. However, the performance of the parallel program execution on such platforms is highly dependent on the scheduling of the parallel program tasks onto these machines [1], [2]. The main objective of the scheduling mechanism is to map the multiple interacting program tasks onto machines and order their execution so that precedence requirements are satisfied and minimum overall completion time is achieved [3], [4]. When the characteristics of the parallel program in terms of task execution times, task dependencies and amount of communicated data are known a priori, scheduling can be accomplished at the compile-time and the parallel program can be represented by the static model [3-5]. In the general form of a static task scheduling problem, the application is represented by a directed acyclic graph (DAG), in which nodes represent application tasks and edges represent inter-task data dependencies. Each node is labeled by the computation cost (the expected computation time) of the task and each edge is labeled by the communication cost (the expected communication time) between tasks [3-5].

Task scheduling problem in general is NP-complete [2-6]. Therefore, heuristics can be used to obtain a sub-optimal scheduling rather than parsing all possible schedules. Task scheduling has been extensively studied, and various heuristics have been proposed in the literature [6-18]. In static scheduling, these heuristics are classified into a variety of categories (such as list-based, clustering and duplication-based).

List-scheduling basically consists of two phases: a *task prioritization* phase, where a certain priority is computed and is assigned to each node of the DAG, and a *machine assignment* phase, where each task (in order of its priority) is assigned to machine that minimizes a suitable cost function. List-scheduling is generally accepted as an attractive approach since it pairs low complexity with good results [3-5], [7-13]. The basic idea of task-duplication is to try to duplicate the parent nodes of the current selected task onto the selected machine or onto another machine(s), aiming to reduce or optimize the task starting or finishing time [14-18]. The main weakness of duplication-based algorithms is their high complexity and that they mainly target an unbounded number of computing machines.

In this paper we propose a compile-time task-scheduling algorithm based on list-scheduling and task-duplication. The algorithm is called *Heterogeneous Critical Parents with Fast Duplicator* (*HCPFD*). The algorithm works for a bounded number of fully connected heterogeneous machines and aims to introduce a simple list-scheduling heuristic for the prioritization phase and a low complexity

duplication-based mechanism for the machine assignment phase. The remainder of this paper is organized as follows: the next section introduces the static scheduling problem in the heterogeneous environment and provides definitions of some parameters utilized in the algorithm. The HCPFD scheduling algorithm is presented in the third section. The fourth section contains a brief overview of other frequently used heterogeneous scheduling algorithms that we apply for performance comparison. A performance evaluation based on a large number of randomly generated application graphs and two real-world applications, is presented in the fifth section.

2. Problem Definition

This section presents: the model of the application used for static scheduling, the model of the heterogeneous computing environments, and the scheduling objective.

The ***application*** can be represented by a *directed acyclic graph* $G(V, E)$ where:

- V is the set of v nodes, each node $v_i \in V$ represents an application task, which is a sequence of instructions that must be executed serially on the same machine,
- E is the set of communication edges. The directed edge $e_{i,j}$ joins nodes v_i and v_j, where node v_i is called the parent node and node v_j is called the child node. This also implies that v_j cannot start until v_i finishes and sends its data to v_j.

A task without any parent is called an *entry task* and a task without any child is called an *exit task*. If there is more than one exit (entry) task, they may be connected to a zero-cost pseudo exit (entry) task with zero-cost edges, which do not affect the schedule.

The ***heterogeneous computing environment*** model is a set P of p heterogeneous machines (processors) connected in a fully connected topology. It is also assumed that:

- any machine (processor) can execute the task and communicate with other machines at the same time.
- once a machine (processor) has started task execution, it continues without interruption, and after completing the execution it immediately sends the output data to all children tasks in parallel.

W is a $v \times p$ computation costs matrix in which each $w_{i,j}$ gives the estimated time to execute task v_i on machine p_j. The communication costs per transferred byte between any two machines are stored in matrix R of size $p \times p$. The communication startup costs of the machines are given in a p-dimensional vector S. The communication cost of edge $e_{i,j}$ for transferring μ bytes of data from task v_i (scheduled on p_m) to task v_j (scheduled on p_n), is defined as:

$$c_{i,j} = S_m + R_{m,n} \cdot \mu_{i,j}.$$

where:

- S_m is p_m communication startup cost (in secs),
- $\mu_{i,j}$ is the amount of data transmitted from task v_i to task v_j (in bytes),
- $R_{m,n}$ is the communication cost per transferred byte from p_m to p_n (in sec/byte).

Before scheduling, each task is labeled with the average execution cost, which is defined as follows:

$$\overline{w_i} = \sum_{j=1}^{p} \frac{w_{i,j}}{p},$$

and each edge $e_{i,j}$ is labeled with the average communication cost, which is defined as follows:

$$\overline{c_{i,j}} = \overline{S} + \overline{R} \cdot \mu_{i,j}.$$

where: $\overline{S}$ is the average machines communication startup cost and $\overline{R}$ is the average machines communication cost per transferred byte.

The *average earliest start time* $AEST(v_i)$ of node v_i can be computed recursively by traversing the DAG downward, starting from the entry node v_{entry}

$$AEST(v_i) = \max_{v_m \in pred(v_i)} \left\{ AEST(v_m) + \overline{w_m} + \overline{c_{m,i}} \right\},$$

where: $pred(v_i)$ is the set of immediate predecessors of v_i and $AEST(v_{entry}) = 0$.

The *average latest start time* $ALST(v_i)$ of node v_i can be computed recursively by traversing the DAG upward, starting from the exit node v_{exit}

$$ALST(v_i) = \min_{v_m \in succ(v_i)} \left\{ ALST(v_m) - \overline{c_{i,m}} \right\} - \overline{w_i},$$

where: $succ(v_i)$ is the set of immediate successors of v_i and $ALST(v_{exit}) = AEST(v_{exit})$.

The main ***objective*** of the scheduling process is to determine the assignment of tasks of a given application to a given machine (processor) set P such that the scheduling length (*makespan*) is minimized satisfying all precedence constraints.

3. Proposed Algorithm

Recently, a list-scheduling heuristic for homogeneous computing environments, called CNPT, has been presented [10]. This heuristic achieved the lower bound complexity of list-scheduling heuristics and has a performance comparable to or even better than the higher complexity heuristics. This section presents an algorithm based on list-scheduling

and task-duplication on a bounded number of fully connected heterogeneous machines called *Heterogeneous Critical Parents with Fast Duplicator* (HCPFD) which aims to achieve high performance and low complexity. The algorithm consists of two phases, a *listing phase* which is a simplified version of the CNPT heuristic for heterogeneous environments and a suggested low complexity duplication mechanism as a *machine assignment phase*.

3.1. Listing Phase

In the **listing phase** (Figure 1), the simplified heuristic divides the task graph into a set of unlisted parent-trees. The root of each parent-tree is a critical-node (CN). A CN is defined as the node that has zero difference between its $AEST$ and $ALST$. The algorithm starts with an empty queue L and an auxiliary stack S that contains the CNs pushed in decreasing order of their $ALST$s, i.e. the entry node is on the top of S ($top(S)$). Consequently, $top(S)$ is examined. If $top(S)$ has an unlisted parent (i.e. has a parent not in L), then this parent is pushed on the stack. Otherwise, $top(S)$ is popped and enqueued into L. For the DAG in Figure 3a, the critical nodes are shown by the thick edges connecting them, which are (v_1, v_2, v_9, and v_{10}). Figure 3b indicates each CN parent tree and the order of all nodes in L.

traverse the graph downward and compute $AEST$ for each node,
traverse the graph upward and compute $ALST$ for each node,
push CNs on the stack S in reverse order of their $ALST$,
while S is not empty **do**
  if there is an unlisted parent of $top(S)$
  then
    push the parent node on S
  else
    pop the $top(S)$ and enqueue it to L

Figure 1. The Listing Heuristic.

3.2. Machine Assignment Phase

In the **machine assignment phase** (Figure 2), the following definitions should be given to clarify the mechanism:

while not the end of L **do**
  dequeue v_i from L
  for each machine p_q in the machine set P **do**
    compute $TFT(v_i, p_q)$
  select the machine p_m that minimizes TFT of v_i
  select v_{cp} and v_{cp2} of v_i
  if the duplication condition is satisfied
    if $TST(v_{cp}, p_m) \leq RT(p_m)$
      duplicate v_{cp} on p_m at $RT(p_m)$
      $RT(p_m) = RT(p_m) + w_{cp,m}$
    else
      duplicate v_{cp} on p_m at $TST(v_{cp}, p_m)$
      $RT(p_m) = TFT(v_{cp}, p_m)$
    if $DAT(v_{cp2}, p_m) > RT(p_m)$
      assign v_i to p_m at $DAT(v_{cp2}, p_m)$
      $RT(p_m) = DAT(v_{cp2}, p_m) + w_{i,m}$
    else
      assign v_i to p_m at $RT(p_m)$
      $RT(p_m) = RT(p_m) + w_{i,m}$
  else
    assign task v_i to p_m at $TST(v_i, p_m)$
    $RT(p_m) = TFT(v_i, p_m)$

Figure 2. The Duplication Mechanism.

Definition 1: The *Task Finish Time on a machine* (TFT) is defined as:

$$TFT(v_i, p_q) = \max_{v_n \in pred(v_i)} \{RT(p_q), FT(v_n) + k.c_{n,i}\} + w_{i,q},$$

where:
- $RT(p_q)$ is the time when p_q is available,
- $FT(v_n)$ is the finishing time of the scheduled parent v_n, and
- $k = 1$ if the machine assigned to parent v_n is not p_q and, $k = 0$ otherwise.

Definition 2: The *Task Start Time on a machine* (TST) is defined as:

$$TST(v_i, p_q) = TFT(v_i, p_q) - w_{i,q}$$

Definition 3: The Duplication Time Slot:

$$DTS(v_i, p_m) = TST(v_i, p_m) - RT(p_m).$$

Definition 4: *The Critical Parent* (CP) is the parent v_{cp} (scheduled on p_q) of node v_i (tentatively scheduled on p_m) whose data arrival time at v_i is the latest.

Definition 5: $DAT(v_{cp2}, p_m)$ is the data arrival time of the *second* critical parent v_{cp2} on p_m.

Definition 6: the duplication condition is :

$$DTS(v_i, p_m) > w_{cp,m}$$

$$and$$

$$TFT(v_{cp}, p_m) < TST(v_i, p_m).$$

The mechanism simply selects the machine p_m that minimizes the TFT of v_i and duplicates its critical parent at the idle time between v_i and the previous task on p_m, if this time slot is enough and this duplication will reduce the TST of v_i on p_m. For the nodes list generated by the algorithm in Figure 3b for the the task graph in Figure 3a, the schedules produced using the non-duplication and duplication mechanisms are shown in Figure 4a and 4b, respectively where the gray tasks in Figure 4b are the duplicated tasks. As an explanation example of the duplication mechanism, the TST (v_4, p_2) using the non-duplication mechanism is 18. The duplication mechanism duplicated the critical parent of v_4 which is v_1 at the idle time slot before v_4. This duplication reduced the TST (v_4, p_2) to be 16.

4. Task Scheduling Heuristics for Heterogeneous Environments

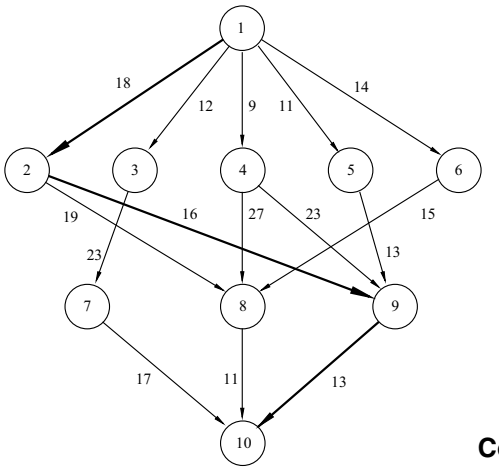
This section briefly overviews the list scheduling algorithms that we will use for the comparison with HCPT. These algorithms are: *Fast Load Balancing* (FLB-f) [7], *Heterogeneous Earliest Finish Time* (HEFT) [8], and *Critical Path on a Processor* (CPOP) [8].

4.1. Fast Load Balancing (FLB-f) Algorithm

The FLB-f algorithm [7] utilizes a list called the ready-list that contains all ready-nodes to be scheduled at each step. The ready-node is defined as the node that has all its parents scheduled. In each step, the execution finish time for each ready-node at the ready-list is computed in all machines and the node-machine pair that minimizes the earliest execution finish time is selected. The complexity of the FLB-f algorithm is $O(vlogv + e)$.

4.2. Heterogeneous Earliest Finish Time (HEFT) Algorithm

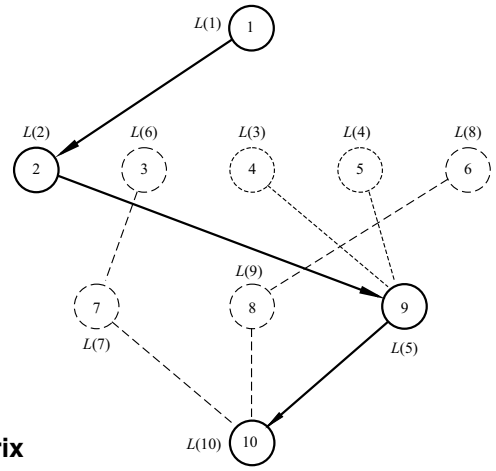
The HEFT algorithm [8] has two phases: the task prioritizing phase, where the upward rank attribute is computed for each task and a task list is generated by sorting the tasks in decreasing order of the upward rank. The second phase is the machine assignment phase, in which, tasks are selected in order of their priorities and are scheduled to the best machine that minimizes the finish time of the task in an insertion based manner. The complexity of the HEFT algorithm is (pv^3)



(a)

Task	P1	P2	P3
1	14	16	9
2	13	19	18
3	11	13	19
4	13	8	17
5	12	13	10
6	13	16	9
7	7	15	11
8	5	11	14
9	18	12	20
10	21	7	16

Computation Cost Matrix



(b)

Figure 3. Algorithm Heuristic Explanatory Example.

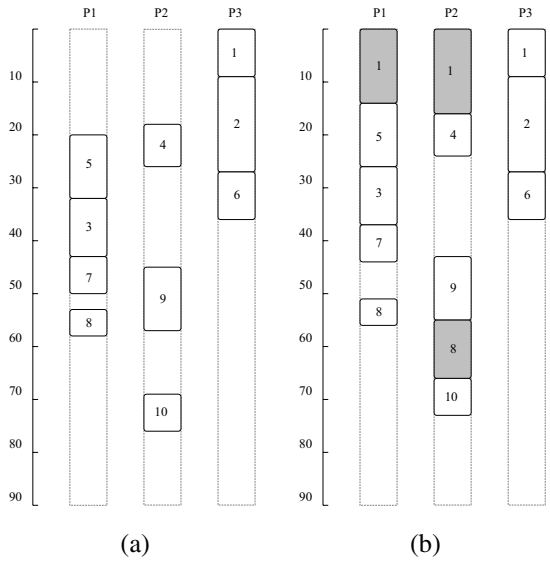


Figure 4. Scheduling of the Task Graph in Figure 2.a with: (a) The Non-Duplication Mechanism (makespan = 76), and (b) The Duplication Mechanism (makespan = 73).

4.3. Critical Path on a Processor (CPOP) Algorithm

The CPOP algorithm [8] has two phases: task prioritizing and machine assignment. In the task prioritizing phase, two attributes are used which are the upward and downward ranks. Each task is labeled with the sum of its upward rank and downward ranks. A priority queue is used to maintain the ready tasks at a given instant (initially, it contains the entry node). At any instant, the ready task with the highest priority is selected for machine assignment. In the machine assignment phase, it defines the critical path machine as the machine that minimizes the cumulative computation cost of the tasks on the critical path. If the selected task is on the critical path, it is assigned to the critical path machine; otherwise, it is assigned to a machine that minimizes the finishing execution time of the task. The complexity of the CPOP algorithm is $O(pv^2)$.

5. Experimental Results and Discussion

This section presents a performance comparison of the HCPFD algorithm with the algorithms presented above. For this purpose, we consider two sets of graphs as the workload: random generated application graphs and the graphs that represent some of the numerical real world problems. Several metrics were used for the performance evaluation. As an example, the scheduling produced by each algorithm for the application graph in Figure 3a is given in Figure 5, where the gray tasks in the HCPFD schedule are the duplicated tasks.

5.1. Comparison Metrics

The comparisons of the algorithms are based on the following metrics:

Makespan
The makespan, or scheduling length, is defined as:

$$makespan = FT(v_{exit}) \ ,$$

where: $FT(v_{exit})$ is the finishing time of the scheduled exit node.

Scheduling Length Ratio (SLR)
The main performance measure is the makespan. Since a large set of application graphs with different properties is used, it is necessary to normalize the schedule length to the lower bound, which is called the Schedule Length Ratio (SLR). The SLR is defined as:

$$SLR = \frac{makespan}{\sum_{v_i \in CP_{MIN}} \min_{p_j \in P}\{w_{i,j}\}} \ .$$

The denominator is the sum of the minimum computation costs of the tasks on a critical path CP_{MIN}. The SLR of a graph cannot be less than one, since the denominator is the lower bound. We utilized average SLR values over the number of generated task graphs in our experiments.

Speedup
The speedup value is defined as the ratio of the sequential execution time (i.e., cumulative computation costs of all tasks) to the parallel execution time (i.e., the makespan). The sequential execution time is computed by assigning all tasks to a single machine, which minimizes the cumulation of the computation costs.

$$speedup = \frac{\min_{p_j \in P}\{\sum_{v_i \in V} w_{i,j}\}}{makespan}$$

Quality Results of Schedules
The percentage number of times that an algorithm produced better, worse, and equal quality of schedules compared to every other algorithm is counted in the experiments.

5.2. Random Graph Generator

The random graph generator was implemented to generate application graphs with various characteristics. The generator requires the following input parameters:

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

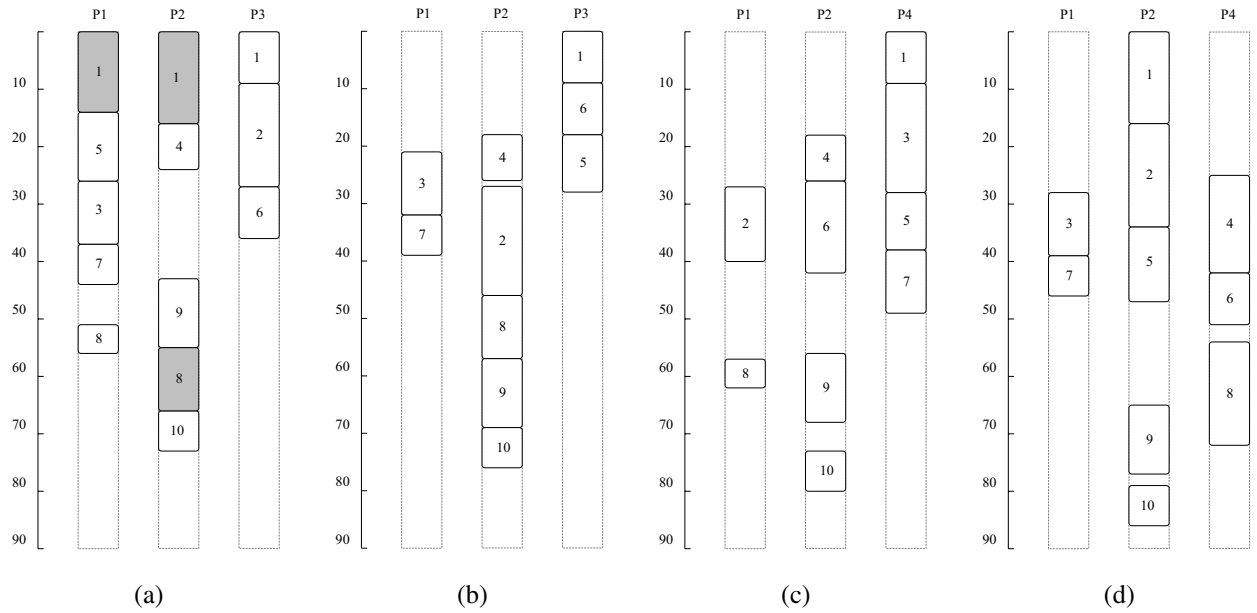


Figure 5. Scheduling of the Task Graph in Figure 2.a with: (a) HCPFD (makespan = 73), (b) FLB-f (makespan = 76), (c) HEFT (makespan = 80), and CPOP (makespan = 86).

- number of tasks in the graph v,
- graph levels l,
- heterogeneity factor β, where w_i is generated randomly for each task and (using a randomly selected β from β set for each p_j) $w_{i,j} = \beta \cdot w_i \forall v_i$ on p_j,
- communication to computation ratio CCR, which is defined as the ratio of the average communication cost to the average computation cost.

In all experiments, only graphs with a single entry and a single exit node were considered. The input parameters were restricted to the following values:

$v \in \{20, 40, 60, 80, 100, 120\}$,
$0.2v \leq l \leq 0.8v$,
$\beta \in \{0.5, 1, 2\}$,
$CCR \in \{0.5, 1.0, 2.0\}$.

We generated sets of 1000 application graphs with randomly selected l, β and CCR for each v from the above list.

5.3. Performance Results

The performances of the algorithms were compared with respect to the graph size. We used 16 machines for all experiments. The machine computing capability was generated using randomly selected β (from the β set given above) for each machine.

The average SLR values for each set of application graphs (with a selected v and random l, β and CCR) are shown in Figure 6, and the average speedup is shown in Figure 7.

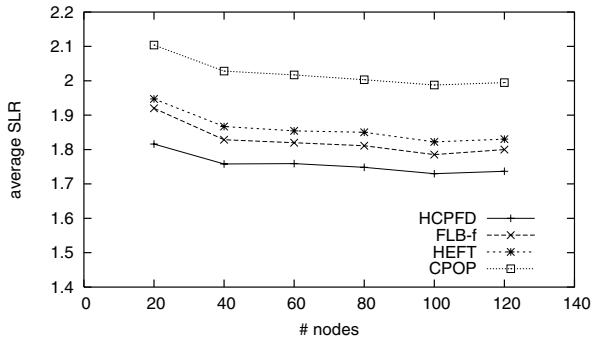


Figure 6. Average SLR (*p=16*).

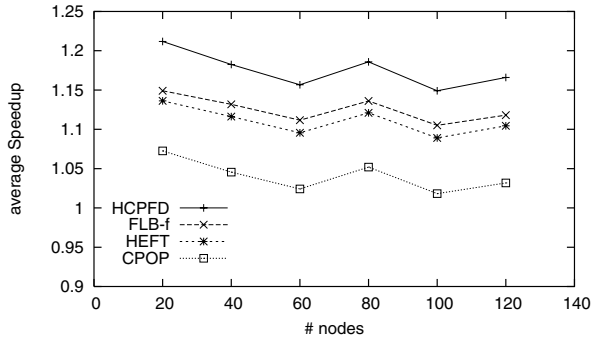


Figure 7. Average Speedup (*p=16*).

Table 1: Quality Results of the HCPFD Algorithm (*p=16*).

		FLB-f	HEFT	CPOP
HCPFD	Better	74.4%	84.08%	95.07%
	Equal	3.33%	3.08%	1.88%
	Worse	22.27%	12.84%	3.05%

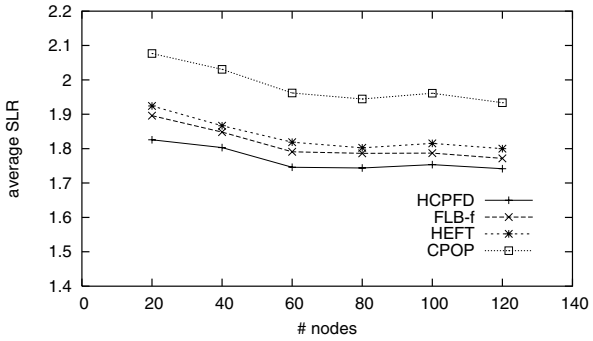


Figure 8. Average SLR (*p=8*).

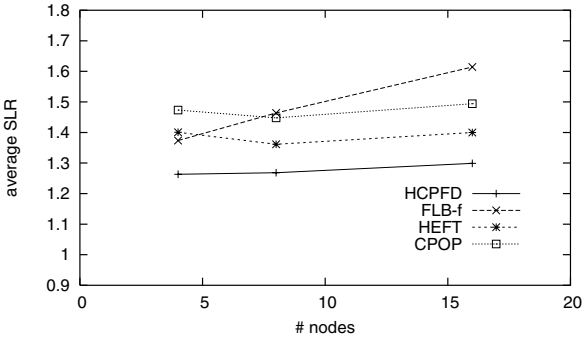


Figure 9. Average SLR for Gaussian Elimination (*p=16*).

Table 3: Quality Results of the HCPFD Algorithm for Gaussian Elimination (*p=16*).

		FLB-f	HEFT	CPOP
HCPFD	Better	93.33%	88%	92%
	Equal	3.33%	8.67%	6%
	Worse	3.34%	3.33%	2%

The number of occurrences of better, equal, and worse results (quality results) of HCPFD compared with the other algorithms is given in Table 1.

When the number of machines was reduced to one half ($p = 8$), the average SLR was as shown in Figure 8 and the quality results are as shown in Table 2.

The results indicate that HCPFD outperforms the other algorithms in terms of SLR, Speedup and number of better results. Even when the number of machines was reduced to one half, HCPFD still outperforms.

Table 2: Quality Results of the HCPFD Algorithm (*p=8*).

		FLB-f	HEFT	CPOP
HCPFD	Better	69.2%	78.25%	91.3%
	Equal	4.02%	2.93%	2.28%
	Worse	26.78%	18.82%	6.42%

5.4. Applications

Finally, we compared the performance of the scheduling algorithms based on two real-world applications: Gaussian Elimination [12] and Molecular Dynamics Code [19]. sixteen heterogeneous machines were used for both experiments. The machine computing capability was generated using random selected β for each machine from the β set given above.

5.4.1. Gaussian Elimination

The structure of the application graph is defined in Gaussian Elimination [12]. The number of tasks v, and the number of graph levels l depends on the matrix size m. Therefore, only CCR was selected randomly for each m. The total number of tasks v in a Gaussian Elimination graph is equal to $\frac{m^2+m-2}{2}$. Figure 9 presents the average SLR of 100 generated graphs for each matrix size $m \in \{4, 8, 16\}$, and the quality results are as shown in Table 2.

5.4.2. Molecular Dynamic Code

Molecular Dynamic Code [19] is an irregular application since it has a fixed number of tasks ($v = 41$) and a known graph structure. Hence, we varied only CCR and β in our experiments. Figure 10 presents the average SLR of 100 graphs generated for each $CCR \in \{0.5, 1, 2\}$, and Table 4 presents the quality results.

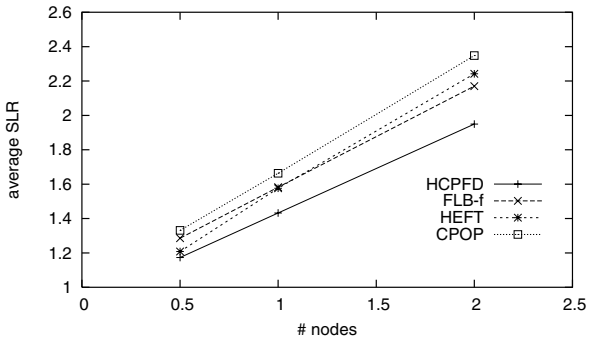


Figure 10. Average SLR for Molecular Dynamic Code (*p=16*).

Table 4: Quality Results of the HCPFD Algorithm for Molecular Dynamic Code (*p=16*).

		FLB-f	HEFT	CPOP
HCPFD	Better	93%	88.67%	96.67%
	Equal	0.33%	2%	0%
	Worse	6.67%	9.33%	3.33%

6. Conclusion

In this paper, we presented the HCPFD algorithm for scheduling tasks onto a bounded number of heterogeneous machines. The algorithm consists of two phases: a simple listing heuristic for task selection instead of the classical prioritization phase of list-scheduling and a fast duplication mechanism works for a bounded number of heterogeneous machines. Based on the experimental study using a large set of randomly generated application graphs with various characteristics and application graphs of real world problems (such as Gaussian Elimination, and Molecular Dynamic Code), HCPFD outperformed the other algorithms in terms of performance and complexity.

References

[1] J.G.Webster, "Heterogeneous Distributed Computing," *Encyclopedia of Electrical and Electronics Engineering*, Vol. 8, pp. 679-690, 1999.

[2] D.Feitelson, L.Rudolph, U.Schwiegelshohm, K.Sevcik, and P.Wong, "Theory and Practice in Parallel Job Scheduling," *JSSPP*, pp. 1-34, 1997.

[3] Y.Kwok and I.Ahmed, "Benchmarking the Task Graph Scheduling Algorithms," *Proc IPPS/SPDP*, 1998.

[4] J.Liou and M.Palis, "A Comparison of General Approaches to Multiprocessor Scheduling," *Proc. Int'l Parallel Processing Symp.* pp.152-156, 1997.

[5] A.Khan, C. McCreary, and M. Jones, "A Comparison of Multiprocessor Scheduling Heuristics," *ICPP*, Vol.2, pp.243-250, 1994.

[6] A.Gerasoulis and T.Yang, "A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol.16, pp.276-291, 1992.

[7] A.Radulescu and A.van Gemund, "Fast and Effective Task Scheduling in Heterogeneous Systems," *9th Heterogeneous Computing Workshop*, pp.229-238, 2000.

[8] H.Topcuoglu, S.Hariri, and W.Min-You, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE trans. on Parallel and Distributed Systems*, Vol.13, No.3, pp.260-274, 2002.

[9] G.Sih, and E.Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. In Parallel and Distributed Systems*, Vol.4, pp.75-87, 1993.

[10] T.Hagras, J.Janecek, A High Performance, Low Complexity Algorithm for Compile-Time Job Scheduling in Homogeneous Computing Environments, *IEEE Proc. Int'l Conf. Parallel Processing Workshops (ICPP03 workshops)*, pp. 149-155, October 2003.

[11] K.Taura, and A.Chien, "A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources," *Heterogeneous Computing Workshop,* pp. 102-118, 2000.

[12] W.Min-You, D.Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, 1990.

[13] Y.Kwok, and I.Ahmed, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graph to Multiprocessors", *IEEE Trans. Parallel and Distributed Systems*, Vol. 7, pp.506-521, 1996.

[14] I.Ahmed, and Y.Kwork, "A New Approach to Scheduling Parallel Program Using Task Duplication," *Int'l Conf. Parallel Processing*, Vol.2, pp.47-51, 1994.

[15] S.Darbha and D.Agrawal, "A Fast and Scalable Scheduling Algorithm for Distributed Memory Systems," *Proc. of Symp. on Parallel and Distributed Processing*, pp. 60-63, 1995.

[16] I.Ahmed, and Y.Kwork, "A Comparsion of Task-Duplication-Based Algorithms for Scheduling Parallel Programs to Message-Passing Systems," *Proc. 11th Int'l Symp. of High-Performance Computing Systems*, pp. 39-50, 1997.

[17] G.Park, B.Shirazi, and J.Marquis, "DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor System," *Proc. Int'l Conf., Parallel Processing*, pp.157-166, 1997.

[18] K.Oh-Han and A.Dharma, "S3MP: A Task Duplication Based Scalable Scheduling Algorithm for Symmetric Multiprocessor," *Proc. IPDP'00*, pp.451-456, 2000.

[19] S.Kim and J.Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *Proc. Int'l Conf. Parallel Processing*, Vol.2, pp.1-8, 1988.

Biography

Tarek Hagras received his B.Sc. degree in Control and Computer Engineering and his M.Sc. degree in Computer Engineering from Assiut University, Assiut, Egypt, in 1992, and 1998, respectively. He is currently studying for his Ph.D. degree at the Department of Computer Science and Engineering at the Czech Technical University in Prague, Czech Republic. He has worked as an assistant lecturer at the High Institute of Energy, Aswan, Egypt, since 1999. His research interest is task scheduling in homogeneous and heterogeneous computing environments.

Jan Janeček is an associate professor in the Department of Computer Science and Engineering at the Czech Technical University in Prague. He received his M.Sc. degree and his Ph.D. degree in technical cybernetics from the Czech Technical University in Prague in 1973, and 1981, respectively. Currently, he lectures on Local Area Networks, Advanced Technologies of Computer Networks, Distributed Systems and Applications of Embedded Systems. His research focuses on distributed computation, middleware technologies, networking and embedded applications. He has led and participated in research teams working on projects dealing with networking technologies (X.25 PAD, ATM switch, VoIP software), software implementation tools (embedded C and Pollux compilers, efficient SOAP parser), distributed quorum algorithms, support for asynchrony in distributed applications, and effectiveness of middleware technologies. He is a member of IEEE and its Computer and Communication Societies, and serves as a vice chairman to the Czech Chapter of the IEEE Computer Society.