Contention-Aware Scheduling with Task Duplication

Oliver Sinnen, Andrea To, Manpreet Kaur

Department of Electrical and Computer Engineering, University of Auckland
Private Bag 92019, Auckland 1142, New Zealand
o.sinnen@auckland.ac.nz

Abstract Scheduling a task graph onto several processors is a trade-off between maximising concurrency and minimising interprocessor communication. A technique to reduce or avoid interprocessor communication is task duplication. Certain tasks are duplicated on several processors to produce the data locally and avoid the communication among processors. Most algorithms using task duplication are for the classic model, which allows concurrent communication and ignores contention for communication resources. The recently proposed, more realistic contention model introduces contention awareness into task scheduling by assigning the edges of the task graph to the links of the communication network. It is intuitive that scheduling under such a model benefits even more from task duplication. This paper proposes a contention-aware task duplication scheduling algorithm, after investigating how to use task duplication in the contention model. An extensive experimental evaluation demonstrates the significant improvements to the speedup of the produced schedules.

1 Introduction

In the task scheduling area, a program is represented as a directed acyclic graph, called task graph, where the nodes represent the tasks and the edges represent the communications between the tasks. Scheduling such a task graph on a set of processors for fastest execution is a well known NP-hard optimisation problem [10] and many heuristics have been proposed [3,7,10,17].

Task duplication is a well known technique to reduce the necessary communication between processors. In this technique certain crucial tasks are executed on more than one processor. The data they procedure is then locally available on different processors and less communication has to be sent between the processors. Again, many algorithms have been proposed that incorporate this technique into scheduling [4,7,8,9].

The classic model used by most scheduling algorithms heavily idealises the target parallel system. It is assumed that all communication can happen at the same time and that all processors are fully connected, in other words there is no contention for communication resources. It is now more and more recognised that this classic model is not realistic and does not suffice for accurate and efficient

task scheduling [1,5,15,16]. Contention aware scheduling algorithms depart from the classic model and schedule not only the tasks, but also the edges on the communication resources.

It is intuitive that avoiding or reducing interprocessor communication becomes more important under the contention model. Consequently, task duplication should be more beneficial under this model. To the authors' best knowledge however, no task duplication algorithm to be used under a contention model has been proposed. In this paper we propose a contention-aware task duplication scheduling algorithm. It works under the general contention model and its algorithmic components are based on state-of-the-art techniques used in task duplication and contention-aware algorithms. We investigate the changes to the scheduling model (Section 3) and discuss the proposed algorithm (Section 4). An extensive experimental evaluation shows that our algorithm is far superior to contention-aware algorithms that do not use task duplication and to task duplication algorithms under the classic model (Section 5). The next section gives a background on task scheduling, including the different models and basic algorithmic techniques.

2 Task scheduling

The program to be scheduled is represented by a directed acyclic graph (DAG), called task graph, $G = (\mathbf{V}, \mathbf{E}, w, c)$. The nodes $\mathbf{V}$ represent the program's tasks and the edges $\mathbf{E}$ the communications between them. An edge $e_{ij} \in \mathbf{E}$ represents the communication from node n_i to node n_j. The positive weight $w(n)$ of node $n \in \mathbf{V}$ represents its computation cost and the non-negative weight $c(e_{ij})$ of edge $e_{ij} \in \mathbf{E}$ represents its communication cost.

The set $\{n_x \in \mathbf{V} : e_{xi} \in \mathbf{E}\}$ of all direct **predecessors** of n_i is denoted by $\mathbf{pred}(n_i)$ and the set $\{n_x \in \mathbf{V} : e_{ix} \in \mathbf{E}\}$ of all direct **successors** of n_i, is denoted by $\mathbf{succ}(n_i)$.

A schedule of a task graph on a target system consisting of a set $\mathbf{P}$ of **dedicated** processors is the association of a start time and a processor with each of its nodes: $t_s(n, P)$ denotes the **start time** of node $n \in \mathbf{V}$. Thus, the node's **finish time** is given by $t_f(n, P) = t_s(n, P) + w(n)$. The processor to which n is allocated is denoted by $proc(n)$. Further, let $t_f(P) = \max_{n \in \mathbf{V}: proc(n)=P}\{t_f(n, P)\}$ be the **processor finish time** of P and let $sl(\mathcal{S}) = \max_{n \in \mathbf{V}}\{t_f(n, proc(n))\}$ be the **schedule length** (or makespan) of $\mathcal{S}$, assuming $\min_{n \in \mathbf{V}}\{t_s(n, proc(n))\} = 0$. For such a schedule to be feasible, the following two conditions must be fulfilled for all nodes in G.

The **Processor Constraint** enforces that only one task is executed by a processor P at any point in time, which means for any two nodes n_i,$n_j \in \mathbf{V}$ that either $t_f(n_i, P) \leq t_s(n_j, P)$ or $t_f(n_j, P) \leq t_s(n_i, P)$ must be true.

The **Precedence Constraint** enforces that for every edge $e_{ij} \in \mathbf{E}$, $n_i, n_j \in \mathbf{V}$, the destination node n_j can only start after the communication associated with e_{ij} has arrived at n_j's processor P

$$t_s(n_j, P) \geq t_f(e_{ij}, proc(n_i), P). \tag{1}$$

$t_f(e_{ij}, P_{src}, P_{dst})$ is the edge finish time of e_{ij} communicated from P_{src} to P_{dst}, which is defined later, depending on the scheduling model.

2.1 Classic scheduling

Traditionally, most scheduling algorithms have employed a strongly idealised model of the target parallel system [3,7,10,17].

Definition 1 (Classic System Model).
A parallel system $M_{classic} = (\mathbf{P})$ consists of a finite set of dedicated processors $\mathbf{P}$ connected by a communication network. This dedicated system has the following properties: i) local communication has zero costs; ii) communication is performed by a communication subsystem; iii) communication can be performed concurrently; iv) the communication network is fully connected.

Based on this system model, the edge finish time only depends on the finish time of the origin node and the communication time. The **edge finish time** of $e_{ij} \in \mathbf{E}$ is given by

$$t_f(e_{ij}, P_{src}, P_{dst}) = t_f(n_i, P_{src}) + \begin{cases} 0 & \text{if } P_{src} = P_{dst} \\ c(e_{ij}) & \text{otherwise} \end{cases} \tag{2}$$

Thus, communication can overlap with the computation of other nodes, an unlimited number of communications can be performed at the same time, and communication has the same cost $c(e_{ij})$, regardless of the origin and the destination processor, unless the communication is local.

2.2 List scheduling

The scheduling problem is to find a schedule with minimal length. As this problem is NP-hard [10], many heuristics have been proposed for its solution. A heuristic must schedule a node on a processor so that it fulfils all resource and precedence constraints.

The best known scheduling heuristic is list scheduling as given in Algorithm 1. In this simple, but common, variant of list scheduling the nodes are ordered according to a priority in the first part of the algorithm. The schedule order of the nodes is important for the schedule length and many different priority schemes have been proposed [6,13,17]. A common and usually good priority is the node's **bottom level** bl, which is the length of the longest path leaving the node. Recursively defined it is

$$bl(n_i) = w(n_i) + \max_{n_j \in \mathbf{succ}(n_i)} \{c(e_{ij}) + bl(n_j)\} \tag{3}$$

Algorithm 1 List scheduling

1: Sort nodes $n \in \mathbf{V}$ into list L, according to priority scheme and precedence constraints.
2: **for** each $n \in L$ **do**
3: Find processor $P \in \mathbf{P}$ that allows earliest finish time of n.
4: Schedule n on P.

2.3 Contention aware scheduling

The classic scheduling model (Definition 1) does not consider any kind of contention for communication resources. To make task scheduling contention aware, and thereby more realistic, the communication network is modelled by a graph, where processors are represented by vertices and the edges reflect the communication links. The awareness for contention is achieved by edge scheduling [11], i.e. the scheduling of the edges of the DAG onto the links of the network graph, in a very similar manner to how the nodes are scheduled on the processors.

The network model proposed in [15] captures network [11,13] as well as endpoint contention [1,5]. This is achieved by using different types of edges and by using switch vertices in addition to processor vertices. Here, it suffices to define the topology network graph to be $TG = (\mathbf{P}, \mathbf{L})$, where $\mathbf{P}$ is a set of vertices representing the processors and $\mathbf{L}$ is a set of edges representing the communication links. The system model is then defined as follows.

Definition 2 (Target Parallel System – Contention Model).
A target parallel system $M_{TG} = (TG)$ consists of a set of processors $\mathbf{P}$ connected by the communication network $TG = (\mathbf{P}, \mathbf{L})$. This dedicated system has the following properties: i) local communications have zero costs; ii) communication is performed by a communication subsystem.

The notions of concurrent communication and a fully connected network found in the classic model (Definition 1) are substituted by the notion of scheduling the edges $\mathbf{E}$ on the communication links $\mathbf{L}$. Corresponding to the scheduling of the nodes, $t_s(e, L)$ and $t_f(e, L)$ denote the **start** and **finish time** of edge $e \in \mathbf{E}$ on link $L \in \mathbf{L}$, respectively.

When a communication, represented by the edge e, is performed between two distinct processors P_{src} and P_{dst}, the routing algorithm of TG returns a **route** from P_{src} to P_{dst}: $R = \langle L_1, L_2, \ldots, L_l \rangle$, $L_i \in \mathbf{L}$ for $i = 1, \ldots, l$. The edge e is scheduled on each link of the route. For details on the scheduling of the edges on the links and the topology graph refer to [15].

It is important to realise that the edge scheduling only affects the scheduling of the tasks through a redefinition of the edge finish time, when compared with the classic model (eq. 2). Let $R = \langle L_1, L_2, \ldots, L_l \rangle$ be the route for the communication of $e_{ij} \in \mathbf{E}$ from P_{src} to P_{dst} if $P_{src} \neq P_{dst}$. The **edge finish time** of e_{ij} is

$$t_f(e_{ij}, P_{src}, P_{dst}) = \begin{cases} t_f(n_i, P_{src}) \text{ if } P_{src} = P_{dst} \\ t_f(e_{ij}, L_l) \quad \text{otherwise} \end{cases} \tag{4}$$

Thus, the edge finish time $t_f(e_{ij}, P_{src}, P_{dst})$ is now the finish time of e_{ij} on the last link of the route, L_l, unless the communication is local. As nothing else changes for the scheduling of the tasks, most scheduling heuristics proposed for the classic model, can also be used under the contention model, thereby making them contention aware. This is in particular true for list scheduling [13].

3 Duplication in contention aware scheduling

Scheduling a task graph is a trade-off between maximising the concurrency and minimising the interprocessor communication costs. It often happens that the advantage of executing tasks in parallel is negated by the associated interprocessor communication cost. It is intuitive that this is even more pronounced under the more realistic contention model, where contention can increase the communication delay.

Task duplication is a well known technique that tries to reduce the communication costs, by scheduling certain tasks on more than one processor. The function $proc(n)$ for the processor allocation of node n becomes a subset of P, denoted by $\mathbf{proc}(n)$. The communication from these duplicated nodes then becomes local on their allocated processors, avoiding costly interprocessor communication.

Many algorithms have been proposed using task duplication [4,7,8,9]. The irony is that most of them have been proposed for the classic model, even though avoiding interprocessor communication under the more realistic contention model can be more crucial. This paper proposes a novel task duplication algorithm for the contention model. In this section we will study the general consequence for the scheduling of the nodes and the next section proposes a contention aware task duplication algorithm. First, let us look at task duplication under the classic model.

Under the classic model, task duplication has an impact on the Precedence Constraint, eq. (1). Given the communication e_{ij}, the node n_j cannot start until *at least one* instance of the duplicated nodes of n_i has provided the communication e_{ij}. It is not necessary to define which instance of n_i is sending the data to n_j in case there is more than one instance that can provide it on time.

3.1 Under contention model

Task duplication under the contention model changes significantly. Under the contention model, it must be strictly defined from where a communication is sent if there are several instances of a sending task. Regard Figure 2 where the task graph of Figure 1(left) is scheduled under the contention model on four processors connected to a central ideal switch (Figure 1(right)). Ideal means there is no contention within the switch. The tasks A and B have been duplicated and only two communications are remote. Edge e_{AE} is scheduled on links L_2 and L_3 (route from P_2 to P_3) , and e_{AF} on links L_1 and L_4 (route from P_1 to P_4). In other words, both instances of A are sending out data, but each only one edge.

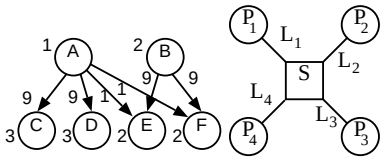

Figure 1. Example task graph (left) and topology graph of four processor system (right)

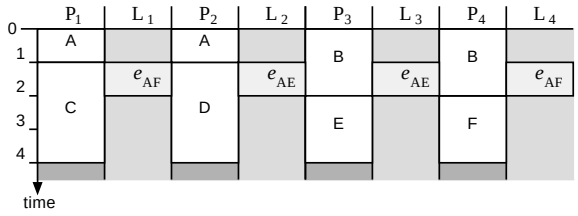

Figure 2. Task duplication under contention model

Because of the contention model, it is actually important that e_{AE} and e_{AF} are sent from different processors as can be observed in Figure 3, where both are sent from P_2. Due to contention on L_2, e_{AF} is delayed and therefore arrives one time unit later at P_4, which in turn increases the schedule length through F's later start time.

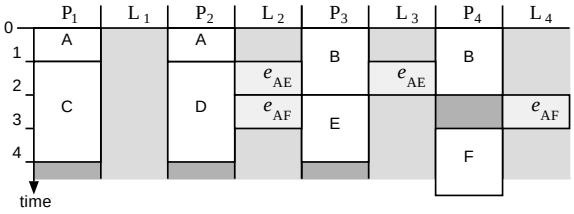

Figure 3. Contention on L_2 delays communication e_{AF}, increases schedule length

The consequence from this observation is that it must be decided during the scheduling of the tasks and edges, which instance of a duplicated task sends the communication. As several instances of a node n_i might exist, e_{ij} might be sent several times to *different* processors, possibly from the same source processor.

As this duplication is done under the contention model, the finish time of the edge remains as defined in eq. (4), that is it corresponds to the finish time of the edge on the link entering the destination processor, for example in Figure 3 the finish time of e_{AF} is $f(e_{AF}, P_2, P_4) = f(e_{AF}, L_4) = 3$.

A scheduling algorithm must carefully choose from which task a communication is sent when several instances exist so that the communication edge can be scheduled and an accurate view of the contention is gained. Under the contention model, this choice is make by tentatively scheduling the edges on the links of the different routes to see from where the communication arrives first as will be seen in the following section [15].

4 Algorithm

The contention-aware task duplication scheduling algorithm proposed in this section is based on scheduling algorithms for the contention model and task duplication techniques used under the classic model. In the following we present and discuss its elements.

List scheduling As the general algorithmic approach, list scheduling, as given in Algorithm 1, is chosen. List scheduling is easily adaptable to the contention model, as shown in [13]. In the first phase the nodes are ordered according to their bottom levels $bl(n)$, defined in (3), which was shown to be the superior node priority under the contention model in an extensive experimental evaluation [13]. Algorithm 2 outlines our proposed algorithm.

Algorithm 2 Contention-aware task duplication scheduling algorithm

```
1: ▷ 1. Part:
2: Sort nodes n ∈ V into list L, according to bl(n)
3: ▷ 2. Part:
4: for each n ∈ L do
5:    for each P ∈ P do
6:       Tentatively schedule n, recursively duplicating n's critical parent – record best
         finish time t_f(n, P) and ancestors to be duplicated, if any
7:    Let P_min be processor where n can finish earliest
8:    Duplicate recorded ancestors of n on P_min
9:    Schedule n on P_min
10:   Remove redundant tasks and their in-edges
```

Insertion technique During list scheduling, each task can be scheduled between already scheduled tasks (insertion technique) or after the finish time of processor P (end technique). The same principle applies of course to the scheduling of the edges on the links. For the necessary tentative scheduling and the redundant task/edge removal (see below) the insertion technique is more suitable and hence employed.

Critical parent An essential question for task duplication algorithms is which tasks should be duplicated. When a task n is scheduled on a processor P, the

primary candidates for duplication are its predecessors **pred**(n), or parents. As task duplication algorithms have shown, it is usually not beneficial to duplicate all predecessors. The most important task to duplicate is the task from which the data transfer arrives the latest, called critical parent $cp(n)$ [4]. Under the contention model, this corresponds to the edge $e_{cp(n),n}$ with the highest finish time $f(e_{cp(n),n}, L_l)$ on the link L_l entering the processor P. If that communication $e_{cp(n),n}$ can be made local, task n might start earlier. Hence, our proposed algorithm considers the critical parent for duplication. The duplication is accepted if the task n can start earlier.

Recursive duplication In some situations it can be more beneficial to not only duplicate the critical parent, but also considering the predecessors of the critical parent for duplication. Task duplication algorithms therefore consider the recursive duplication of the critical parent $cp(n)$, its critical parent $cp(cp(n))$ and so on [2]. This approach is adopted by our algorithm, whereby the recursive duplication goes as deep as it is most beneficial, i.e. as it reduces the start time of task n most.

Tentative scheduling A characteristic aspect of scheduling under the contention model is the need to tentatively schedule edges on the communication links in order to obtain the data ready time of a task n, i.e. the time when all incoming edges have finished communication. For example, we search for the processor that allows task n_i's earliest finish time and n_i has the in-edges e_{li} and e_{ki}. Then, for each processor P, we must schedule the communication on the links of the route from $proc(n_l)$ and $proc(n_k)$ to P. That gives us an accurate data ready time of n_i on P. Before the next processor is considered, the edges must be removed from the schedule, hence tentative scheduling. With task duplication this tentative scheduling is even more involved as there might be more than one instance of n_l and n_k, as seen with task A in the example of Figure 2 and 3. Our algorithm therefore integrates tentative scheduling also on this level, i.e. the communication is tentatively scheduled from each instance of a predecessor task in order to find the best data provider.

Redundant task/edge removal When a task n is duplicated on processor P, the original and other instances of n might have become redundant. This is the case, if one or more of these instances do not provide data to any predecessor. The redundant tasks can and should be removed from the schedule. Under the contention model, the removal of a task implies that also its in-edges can be removed from the links. Especially together with the insertion technique, the freed space can be used by subsequently scheduled tasks and their edges, potentially leading to shorter schedules. Our algorithm checks for and removes redundant tasks after the scheduling of each task.

Complexity The complexity of contention-aware list scheduling with the insertion technique is $O(|\mathbf{V}|^2 + |\mathbf{P}||\mathbf{E}|^2 O(routing))$ [12]. $O(routing)$ is the complexity for finding the communication route in the network and its length, but is for

many practically relevant systems $O(1)$. With our recursive task duplication the complexity increases to $O(|\mathbf{P}|^2(|\mathbf{V}|^3 + |\mathbf{V}||\mathbf{E}|^2 O(routing)))$.

5 Experimental evaluation

Two questions need to be answered in the evaluation of the proposed algorithm: i) How do the schedules improve compared to a task duplication algorithm without contention awareness? ii) How does task duplication improve upon other contention-aware scheduling algorithms? To answer these questions, we have implemented four algorithms. The proposed contention-aware task duplication algorithm (CA-D) is compared with a contention-aware list scheduling (CA-LS) [13], which is essentially the same algorithm as CA-D, but without the duplication of tasks. Further, we implemented a task duplication (D) and a list scheduling algorithm (LS) under the classic model. Again, they are identical to CA-D and CA-LS, respectively, but without the contention awareness.

Schedules produced under the different models cannot be directly compared [14]. Usually, schedules under the contention-model are longer, but more realistic, resulting in shorter execution times. Hence to compare the schedule, we simulated contention for D and LS. This was done by rescheduling the D's and LS's schedules under the contention model [14]. To indicate this contention simulation we named D and LS in the following D-CS and LS-CS.

5.1 Setup

For the models of the parallel target systems we have chosen sets of processors (2, 8 and 15) connected to an ideal switch. Each processor has an out-going and an in-coming link connected to this switch, thus only one communication in each direct can take place at the same time. This corresponds to full-duplex communication ports and this model is also referred to the one-port model [1].

A large set of graphs was generated as the workload for the scheduling algorithms. This set comprised of graphs of seven types: In-trees, Out-trees, Series-Parallel (SP), Fork, Join, Fork-join and Random [12]. Within each type, graphs of different sizes were created (number of nodes= 20, 100, 500, 1000) with random node and edge weights, scaled to achieve different communication to computation ratios (CCR = 0.1, 1, 10) [12]. CCR is a measure for the importance of communication and is defined as the total edge weight over the total node weight $CCR = \frac{\sum_{e \in \mathbf{E}} c(e)}{\sum_{n \in \mathbf{V}} w(n)}$. In total about 2000 graphs were generated and scheduled.

5.2 Results

In this section the significant experimental results are shown and discussed. Regard Figures 4 and 5 that display the speedup over the number of processors for different graph types. The displayed values are average values across all different graphs of the same type. Speedup of a schedule $\mathcal{S}$ is defined as the sequential length of the graph over the schedule length $speedup(\mathcal{S}) = \frac{\sum_{n \in \mathbf{V}} w(n)}{sl(\mathcal{S})}$.

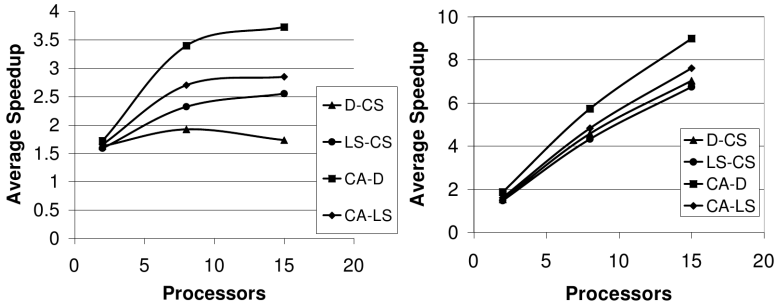

Figure 4. Speedup over processors for SP-graphs (left) and random graphs (right)

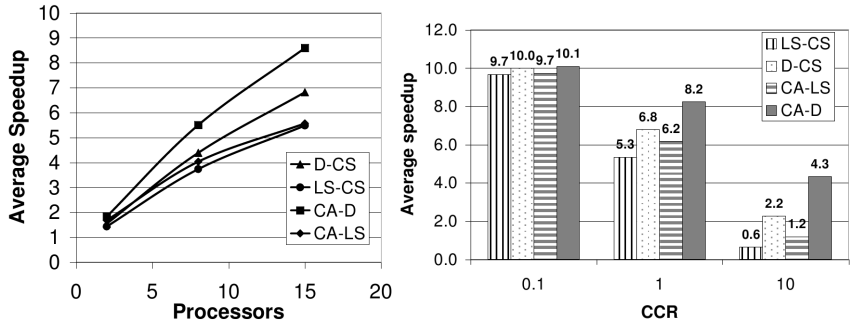

Figure 5. Speedup over processors for out-trees (left) and speedup over CCR for all graphs on 15 processors (right)

Contention aware (CA-D) vs. non-contention aware duplication (D-CS) The figures show that contention aware duplication (CA-D) is never worse than non-contention aware duplication (D-CS). In fact, CA-D produces greater speedup than D-CS for all graphs, except for fork graphs. The difference between the two algorithms is the greatest with SP graphs, where the speedup produced by CA-D on 15 processors is 120 percent greater than that of D-CS.

Figure 5(right) shows the average speedup across graph types produced by each algorithm for different CCR values on 15 processors. The average speedup values produced by the algorithms for high communication graphs (CCR = 10) show the greatest difference (95 percent) between contention aware duplication (CA-D) and non-contention aware duplication (D-CS). The difference is less, but still significant for medium communication graphs (contention aware duplication is 20 percent greater). As can be expected, contention aware duplication can excel most when the CCR value is medium to high, in other words when avoiding communication and contention is most important. To summarise, duplication under the contention model is significantly better than under the classic model.

Contention aware duplication (CA-D) vs. contention aware list scheduling (CA-LS) Task duplication has never been used in contention-aware algorithms. In this sub-section we are therefore evaluating if it improves the schedule length at least as much as it does under the classic model, so we compare CA-D with CA-LS, both contention aware algorithms, but only CA-D does duplication. As can be seen in the figures, CA-D has greater speedup on all numbers of processors for all graph types. Graphs with structures that benefit from task duplication (i.e., graphs where there is at least one node with more than one child) show the greatest difference in speedup. Speedup produced on 15 processors by CA-D is 54 percent greater than that of CA-LS for out-trees, 31 percent greater for SP graphs, and 18 percent greater for random graphs. Note that the difference between the non-contention aware algorithms D-CS and LS-CS is sometimes significantly less, e.g. for random-graphs. This is evidence supporting our hypothesis that task duplication is more important for scheduling under the contention model. To summarise, the duplication technique does significantly improve the list scheduling heuristic under the contention model for most task graphs, even more than under the classic model.

6 Conclusions

This paper proposed a novel contention-aware task duplication scheduling algorithm. It was studied how task duplication can be performed under the contention model. Based on this an algorithm was proposed using state-of-the-art scheduling techniques found in classic task duplication algorithms and other contention-aware algorithms.

An extensive experimental evaluation of the algorithm was performed, comparing the proposed algorithm with task duplication under the classic model and with a contention-aware algorithm without task duplication. This revealed very significant speedup gains, both compared to task duplication under the classic model and to other contention-aware scheduling algorithms without task duplication. As predicted, task duplication is even more beneficial under the contention model than under the classic model.

References

1. O. Beaumont, V. Boudet, and Y. Robert. A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In *HCW'2002, the 11th Heterogeneous Computing Workshop*. IEEE Computer Society Press, 2002.
2. S. Darbha and D. P. Agrawal. Optimal scheduling algorithm for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):87 – 95, January 1998.
3. A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, December 1992.

4. T. Hagras and J. Janeček. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Parallel Computing*, 31(7):653–670, 2005.
5. T. Kalinowski, I. Kort, and D. Trystram. List scheduling of general task graphs under LogP. *Parallel Computing*, 26:1109–1128, 2000.
6. H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33:1023–1029, November 1984.
7. B. Kruatrachue and T. G. Lewis. Grain size determination for parallel processing. *IEEE Software*, 5(1):23–32, January 1988.
8. J.-C. Liou and M. A. Palis. A new heuristic for scheduling parallel programs on multiprocessor. In *1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 358 – 365, October 1998.
9. F. E. Sandnes and G. M. Megson. An evolutionary approach to static taskgraph scheduling with task duplication for minimised interprocessor traffic. In *Proc. Int. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2001)*, pages 101–108, Taipei, Taiwan, July 2001. Tamkang University Press.
10. V. Sarkar. *Partitionning and Scheduling Parallel Programs for Execution on Multiprocessors*. MIT Press, 1989.
11. 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(2):175–186, February 1993.
12. O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley, May 2007.
13. O. Sinnen and L. Sousa. List scheduling: Extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing*, 30(1):81–101, January 2004.
14. O. Sinnen and L. Sousa. On task scheduling accuracy: Evaluation methodology and results. *The Journal of Supercomputing*, 27(2):177–194, February 2004.
15. O. Sinnen and L. Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, June 2005.
16. A. Tam and C. L. Wang. Contention-aware communication schedule for high speed communication. 6(4):339–353, 2003.
17. M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, July 1990.