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

Finding an efficient schedule for a task graph on several processors is a trade-off between maximising concurrency and minimising interprocessor communication. Task duplication is a technique that has been employed to reduce or avoid interprocessor communication. Certain tasks are duplicated on several processors to produce the data locally and avoid the communication among processors. Most of the algorithms using task duplication have been proposed for the classic scheduling model, which allows concurrent communication and ignores contention for communication resources. It is more and more recognised that this classic model is unrealistic and does not permit creating accurate and efficient schedules. The recently proposed 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, yet no such algorithm

[*] A shorter preliminary version of the manuscript appeared in the proceedings of JSSPP'09.

has been proposed as it is not trivial to duplicate tasks under the contention model. This paper proposes a contention-aware task duplication scheduling algorithm. We investigate the fundamentals for task duplication in the contention model and propose an algorithm that is based on state-of-the-art techniques found in task duplication and contention-aware algorithms. An extensive experimental evaluation demonstrates the significant improvements to the speedup of the produced schedules.

Keywords: task scheduling; contention aware algorithm; edge scheduling; task duplication

1 Introduction

Parallelising a program is a challenging task. One crucial aspect in this parallelisation of a program is the scheduling of the (sub)tasks on the processors of the parallel system. 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 [22, 30] and many heuristics have been proposed [7, 9, 14, 20, 31, 32, 12]. The heuristics have to find the trade-off between high concurrency, i.e. tasks are distributed across the available processors as much as possible, and low interprocessor communication as this is time consuming and negates the benefit of parallelisation.

Task duplication is a well known technique to reduce the necessary communication between the 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 [3, 8, 14, 15, 18, 19, 21].

Communication is not only a problem on the algorithmic level, but also for the scheduling model itself. 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 [2, 4, 10, 16, 17, 27, 28, 29]. Contention aware scheduling algorithms depart from the classic model and schedule not only the tasks, but also the edges on the communication resources. The one-port model is a simple model that considers end-point contention [2], which is generalised in the contention model that allows to consider end-point and network contention in arbitrary types of networks [25, 27].

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 fundamental changes to the scheduling model and discuss the proposed algorithm. 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.

We continue in the next section with a background on task scheduling, including the different models and basic algorithmic techniques. Section 3 then investigates the general consequences of task duplication under the contention model. In Section 4 we propose our novel algorithm, which is evaluated in Section 5 in comprehensive experiments. Section 6 concludes the paper.

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 node $n \in \mathbf{V}$ without predecessors, $\mathbf{pred}(n) = \emptyset$, is named **source** node and if it is without successors, $\mathbf{succ}(n) = \emptyset$, it is named **sink** node.

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 $n \in \mathbf{V}$ on processor $P \in \mathbf{P}$. The node's **finish time** is given by $t_f(n, P) = t_s(n, P) + w(n)$, i.e. the node's start time plus its computation costs, as homogeneous processors are assumed in this work. 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 at any point in time, which means for any two nodes n_i,$n_j \in \mathbf{V}$

$$proc(n_i) = proc(n_j) = P \Rightarrow \begin{cases} & t_f(n_i, P) \leq t_s(n_j, P) \\ \text{or} & t_f(n_j, P) \leq t_s(n_i, P) \end{cases} \tag{1}$$

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

$$t_s(n_j, P) \geq t_f(e_{ij}, proc(n_i), P). \tag{2}$$

$t_f(e_{ij}, P_{src}, P_{dst})$ is the edge finish time of e_{ij} communicated from P_{src} to P_{dst}, which is defined later, depending on the scheduling model. From a task perspective, the earliest time a node n_j can start execution on processor P is called the **Data Ready Time (DRT)** t_{dr} with

$$t_{dr}(n_j, P) = \max_{e_{ij} \in \mathbf{E}, n_i \in \mathbf{pred}(n_j)} \{t_f(e_{ij}, proc(n_i), P)\} \tag{3}$$

and hence

$$t_s(n, P) \geq t_{dr}(n, P) \tag{4}$$

for all $n \in \mathbf{V}$. If $\mathbf{pred}(n) = \emptyset$, i.e. n is a source node, $t_{dr}(n) = t_{dr}(n, P) = 0$, for all $P \in \mathbf{P}$.

2.1 Classic scheduling

Traditionally, most scheduling algorithms have employed a strongly idealised model of the target parallel system [7, 9, 14, 20, 31, 32], called classic model .

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.

Definition 2 (Edge Finish Time) *The edge finish time of $e_{ij} \in \mathbf{E}$ is given by*

$$t_f(e_{ij}, P_{src}, P_{dst}) = t_f(n_i, P_{src}) + \begin{cases} 0 & \textit{if } P_{src} = P_{dst} \\ c(e_{ij}) & \textit{otherwise} \end{cases} \tag{5}$$

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 Heuristics

The scheduling problem is to find a schedule with minimal length. As this problem is NP-hard [22, 30], many heuristics have been proposed for its solution. A heuristic must schedule a node on a processor so that it fulfils all resource (1) and precedence constraints (2). A free node $n \in \mathbf{V}$ can be scheduled on processor P within the idle time interval $[A, B]$, $A, B \in [0, \infty]$, i.e. an interval in which no task is executed, if

$$\max\{A, t_{dr}(n, P)\} + w(n) \leq B. \tag{6}$$

Algorithm 1 List scheduling

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

A free node is a node whose predecessors have already been scheduled, which is a requisite for the calculation of the DRT. So, this condition allows node n to be scheduled between already scheduled nodes (**insertion technique**) [13], i.e. $[A, B] = [t_f(n_{P_l}, P), t_s(n_{P_{l+1}}, P)]$, or after the finish time of processor P (**end technique**) [1], i.e. $[A, B] = [t_f(P), \infty]$.

2.2.1 List scheduling

The best known scheduling heuristic is list scheduling, e.g. [1], 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 [1, 11, 25, 31]. A common and usually good priority is the node's bottom level bl, which will be presented in Section 4.

To determine the start time of a node, the earliest interval $[A, B]$ is searched on each processor that complies with (6), either using the insertion or the end technique. For the found interval $[A, B]$, the start time of node n is determined as

$$t_s(n, P) = \max\{A, t_{dr}(n, P)\}. \tag{7}$$

Node n is scheduled on the processor that allows the earliest finish time

$t_f(n, P) = t_s(n, P) + w(n)$.

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 [23], i.e. the scheduling of the edges of the task graph 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 [27] captures network [23, 25] as well as end-point contention [2, 10, 17] and can represent heterogeneous communication links. 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 3 (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 [27].

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 (Definition 2).

Definition 4 (Edge Finish Time – Contention Model) *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 finish time of* e_{ij} *is*

$$t_f(e_{ij}, P_{src}, P_{dst}) = \begin{cases} t_f(n_i, P_{src}) & \text{if } P_{src} = P_{dst} \\ t_f(e_{ij}, L_l) & \text{otherwise} \end{cases} \tag{8}$$

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 [25].

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 $\mathbf{proc}(n)$, $\mathbf{proc}(n) \subseteq \mathbf{P}$ and $|\mathbf{proc}(n)| \geq 1$. The communication from these duplicated nodes then becomes local on their allocated processors, avoiding costly interprocessor communication. Figure 1 displays an example, where a simple task graph is scheduled on four processors using task duplication. Both task A and task B are scheduled more than once, i.e. duplicated, which renders all the expensive communications (e_{AC},e_{AD},e_{BE},e_{BF}) local.

Many algorithms have been proposed using task duplication [3, 8, 14, 15, 18, 19, 21]. 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 much 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.

3.1 Under classic model

Task duplication has an impact on the formulation of the Precedence Constraint, eq. (2). For a schedule $\mathcal{S}_{dup}$ with node duplication, (2) becomes

$$t_s(n_j, P) \geq \min_{P_x \in \mathbf{proc}(n_i)} \{t_f(e_{ij}, P_x, P)\}. \tag{9}$$

Given the communication e_{ij}, the node n_j cannot start until *at least one*

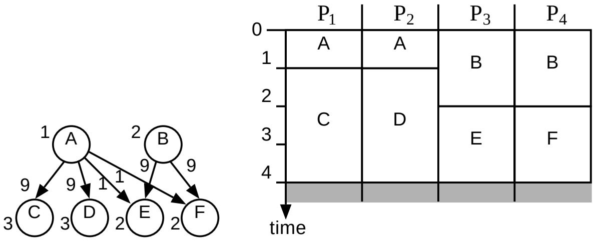

Figure 1: Task duplication example under classic model; task graph (left) and task schedule (right)

instance of the duplicated nodes of n_i has provided the communication e_{ij}. In the example of Figure 1, tasks E and F receive communication from task A of which two instances exists (on P_1 and on P_2). Here, the communication of either one arrives at the same time at P_3 and P_4, due to A's identical finish time on both.

Following from the altered Precedence Constraint condition, the definition of the data ready time (eq. (3)) must be adapted. For a schedule $\mathcal{S}_{dup}$ with node duplication, (3) becomes

$$t_{dr}(n_j, P) = \max_{n_i \in \mathbf{pred}(n_j)} \{ \min_{P_x \in \mathbf{proc}(n_i)} \{ t_f(e_{ij}, P_x, P) \} \}. \tag{10}$$

The rest of the definitions and conditions of task scheduling remains unmodified.

3.2 Under contention model

Task duplication under the contention model changes significantly in regard to the Precedence Constraint. 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 same task graph as in Figure 1 is scheduled, now under the contention model, on four processors connected to a

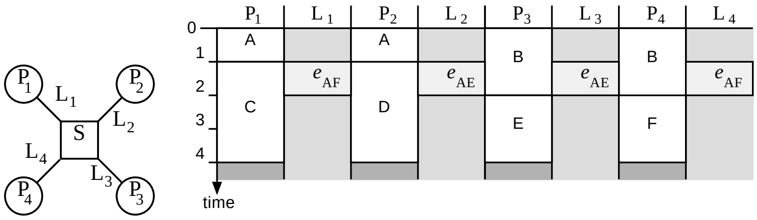

Figure 2: Task duplication under contention model; four processor system (left), task and edge schedule (right)

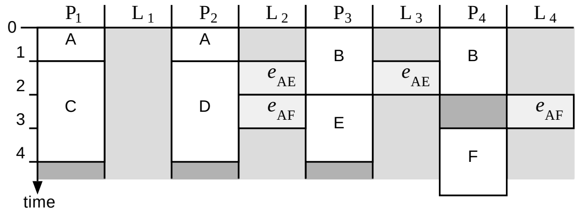

Figure 3: Contention on L_2 delays communication e_{AF}, increases schedule length

central ideal switch (left). Ideal means there is no contention within the switch [27]. As in Figure 1, 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.

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.

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. The formulation of the Precedence Constraint for task duplication under the contention model then looks the same as without duplication (eq. (2))

$$t_s(n_j, P) \geq t_f(e_{ij}, P_{src}, P) \tag{11}$$

with $P_{src} \in \mathbf{proc}(n_i)$. P_{src} is the processor from where e_{ij} is routed and scheduled to P.As this duplication is done under the contention model, the finish time of the edge remains as defined in Definition 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 $t_f(e_{AF}, P_2, P_4) = t_f(e_{AF}, L_4) = 3$.

For contention-aware scheduling, not only the above discussed duplication of the origin node of an edge is relevant, but also the duplication of the destination node. Several instances of a node n_j might exist, thus an incoming edge e_{ij} might be sent several times to *different* processors, possibly from the same source processor on which n_i is executed. It follows that an incoming edge e_{ij} of a duplicated node n_j might be scheduled on a single link more than once. To illustrate this consider the example of Figure 4, where the task graph on the left has been scheduled on the four processor system of Figure 2. Node C is duplicated on P_2 and P_3, so edge e_{AC} must be sent twice, once to P_2 and once to P_3. Observe that C is a duplicated destination node of e_{AC}. In the figure the two instances of C and e_{AC} are distinguished with superscripts 1 and 2, respectively[1]. As all communications from P_1 go through link L_1, e_{AC} has been scheduled twice on L_1. This kind of scheduling corresponds to point-to-point communication used in most parallel communication networks [5]. An intelligent network could of course avoid the duplication of e_{AC} on L_1 by using a multi-cast primitive, however many parallel computer networks do not support multi-casts. Apart from this, the duplicated edges are not necessarily sent at

[1] Also duplicating A would of course avoid all communication (Figure 5). However, this might not be beneficial in general, e.g. P_2 and P_3 are busy with other, independent, nodes. For illustration purposes a simple task graph was used.

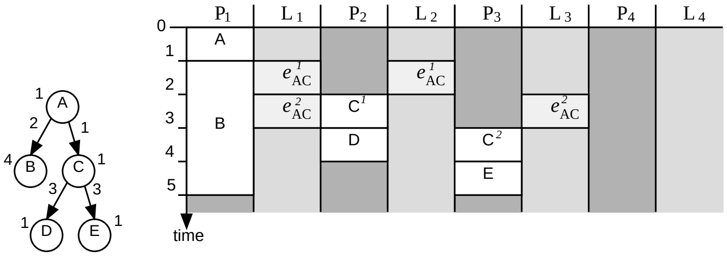

Figure 4: Edge e_{AC} is scheduled twice on link L_1: once for the communication to processor $P_2(^{\mathbf{1}})$ and once for the communication to $P_3(^{\mathbf{2}})$.

the same time, in which case multi-cast could not be employed.

Finally, note that while an edge might be sent more than once, it never arrives more than *once* at the same processor.

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 made 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 [27].

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 [25]. In the first phase the nodes are ordered according to their non-increasing **bottom levels** $bl(n)$, which is the length of the longest

Algorithm 2 Contention-aware task duplication scheduling algorithm

```
▷ *1. Part:*
Sort nodes $n \in \mathbf{V}$ into list $L$, according to non-increasing $bl(n)$
▷ *2. Part:*
for each $n \in L$ do
    for each $P \in \mathbf{P}$ do
        Tentatively schedule $n$, recursively duplicating $n$'s critical parent – record best finish time $t_f(n, P)$ and ancestors to be duplicated, if any
    end for
    Let $P_{min}$ be processor where $n$ can finish earliest
    Duplicate recorded ancestors of $n$ on $P_{min}$
    Schedule $n$ on $P_{min}$
    Remove redundant tasks and their in-edges
end for
```

path leaving the node. Recursively defined it is

$$bl(n_i) = w(n_i) + \max_{n_j \in \mathbf{succ}(n_i)} \{c(e_{ij}) + bl(n_j)\} \tag{12}$$

This bottom level was shown to be the superior node priority under the contention model in an extensive experimental evaluation [25]. Algorithm 2 outlines our proposed algorithm.

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) (Section 2.2). The same principle applies of course to the scheduling of the edges on the links. While the insertion technique usually produces superior results already for scheduling without duplication, it can be even more beneficial for task duplication as duplicated task can then be inserted into the existing partial schedule. For that reason the insertion technique is 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 $\mathbf{pred}(n)$, 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 $cp(n)$ [8]. Defined with the edge finish time the critical parent is given by

$$\begin{aligned} & cp(n) = n_x \in \mathbf{pred}(n) : \\ & \qquad t_f(e_{n_x,n}, proc(n_x), P) = \max_{e_{ij} \in \mathbf{E}, n_i \in \mathbf{pred}(n)} \{t_f(e_{n_i,n}, proc(n_i), P)\} \end{aligned} \quad (13)$$

Under the contention model, $cp(n)$is the origin node of the edge $e_{cp(n),n}$ with the highest finish time $t_f(e_{cp(n),n}, L_l)$ 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. Note that with the end technique the critical parent is always the one whose corresponding edge was scheduled last in the heuristic, which is not necessarily the case with the insertion technique.

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 [6]. Figure 5 shows the schedule example of Figure 4, where now also A, i.e. is the critical parent of the critical parent of D and E, is duplicated. This approach is adopted by our algorithm as outlined in Algorithm 3. This procedure is the more detailed description of line 6 in Algorithm 2. The first step is to create the recursive list of the critical parent, its parent and so on. This is done until the first of those critical ancestors is already on the same pro-

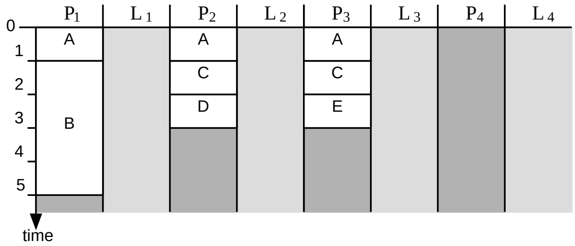

Figure 5: Not only the critical parent of D and E (C) is duplicated but also C's critical parent A (Figure 4)

cessor P on which the candidate task n is going to be scheduled. Going further is unnecessary, as the duplication of more distance ancestors would have been considered at the time this ancestor was scheduled on P. This list is now taken and all ancestors are duplicated and the finish time of the candidate task n is recorded. After that the first task of the list, i.e. the most distant ancestor, is removed and the process is repeated. In this way, the algorithm evaluates how deep the duplication of ancestors should go.

Note that with this procedure *all* critical ancestors until a certain depth are duplicated. Not duplicating consecutive ancestors does not seem to be meaningful, as communication would then go from P to another processor and back for the non-duplicated ancestor.

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 $t_{dr}(n, P)$ of a task n. 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

Algorithm 3 Finding best critical ancestor duplication for n

```
1:  Recursively prepend critical ancestors of $n$ to list $A$, until critical ancestor
    already on $P$
2:  while $A$ not empty do
3:    for each task $m \in A$ do
4:      Schedule incoming edges of $m$ on links (if there are duplicated origin
        tasks, find best origin task)
5:      Schedule $m$ on $P$
6:    end for
7:    Schedule $n$ on $P$
8:    if $t_f(n, P) < minFT$ then
9:      $minFT = t_f(n, P)$
10:     $ancestorsToDuplicate = A$
11:   end if
12:   Undo all scheduling of tasks ($n$ and tasks of $A$) on $P$ and their incoming
      edges
13:   Remove first task from $A$
14: end while
15: return $minFT$ and $ancestorsToDuplicate$
```

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 Figures 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. In Algorithm 3 the tentative scheduling is apparent in two places: at the scheduling of the incoming edges (line 4), as the best origin task needs to be identified, and at the end of trying each ancestor depth (line 12) when all scheduled tasks and edges are removed.

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. In other words, the duplicated instance on P fully substitutes other instances – they have become redundant. This is always the case when a task has only one predecessor, i.e. one out-edge [24]. In such a case, 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.

Algorithm 2 summaries our proposed contention-aware task duplication scheduling algorithm and Algorithm 3 gives details of the task duplication and its tentative scheduling.

Complexity The complexity of the first part of Algorithm 2 is $O(|\mathbf{V}| \log |\mathbf{V}| + |\mathbf{E}|)$ for the calculation and sorting of the bottom levels [24] and can be ignored in comparison to the second part.

In the second part of the proposed contention aware scheduling algorithm, the complexity is dominated by the tentative scheduling of the incoming edges of all the tasks that are considered in each step. The two outer loops iterate over all tasks (line 4 of Algorithm 2) and all processors (line 5). For each task n on each processor P, i.e. $|\mathbf{P}||\mathbf{V}|$ times, the algorithm then determines which (critical) ancestors of n should be duplicated (line 6). This procedure, as detailed in Algorithm 3, defines the total complexity. The actual scheduling of n, the duplication of the selected set of ancestors and the scheduling of all incoming edges (lines 8-10) can be neglected, as this is just a single repetition of the best solution found in Algorithm 3. Also, the removal of redundant nodes and edges (line 11) is small in terms of complexity compared to line 6.

Algorithm 3 starts by computing the recursive sequence of critical parents (line 1), which is $O(|\mathbf{P}||\mathbf{E}|)$. As this is only done once, it has no influence on the total complexity. More important is the iteration over the created list A (line 2) which has size of $O(|\mathbf{V}|)$, hence contributes a factor $O(|\mathbf{V}|)$. In each of these iterations, the tasks of A are scheduled onto P, with the respective scheduling of their incoming edges (loop lines 3-6). The number of edges scheduled here is in total $O(|\mathbf{E}|)$ and is always greater than or equal to the number of tasks in A (remember each task has as least one incoming edge). For that reason, only the scheduling of the edges is relevant for the complexity, everything else (lines 5 and 7-13) is negligible. The $O(|\mathbf{E}|)$ edges are scheduled on the links. For each edge, the route has to be computed and the edge is scheduled on each link of this route. This is characterised by $O(routing)$, which is the complexity for finding the communication route in the network and its length. For many practically relevant systems it is $O(1)$, both in terms of finding the route and its length, e.g. fully connected networks, one-port networks, central switch networks [24]. As the origin task of the edge might be duplicated $O(|\mathbf{P}|)$ times, scheduling a single edge might be repeated $O(|\mathbf{P}|)$ times. Finally, the insertion technique is employed and there are $O(|\mathbf{E}|)$ edges scheduled on each link, so in takes $O(|\mathbf{E}|)$ time to find a slot for an edge on a link. In summary, scheduling all incoming edges of the tasks in the ancestor list is $O(|\mathbf{P}||\mathbf{E}|^{\mathbf{2}}O(routing))$. This is repeated $O(V)$ times (line 2), hence the procedure of Algorithm 3 is in total $O(|\mathbf{P}||\mathbf{V}||\mathbf{E}|^{\mathbf{2}}O(routing))$.

As stated above, this procedure of Algorithm 3 is performed $|\mathbf{P}||\mathbf{V}|$ times resulting in a total complexity of $O(|\mathbf{P}|^{\mathbf{2}}|\mathbf{V}|^{\mathbf{2}}|\mathbf{E}|^{\mathbf{2}}O(routing))$ for the proposed algorithm. Be aware that this is the worst case complexity, which should be significantly higher than the expected average case complexity in this case. For comparison, the second part of a contention-aware list scheduling with the insertion technique is $O(|\mathbf{V}|^2 + |\mathbf{P}||\mathbf{E}|^2O(routing))$ [24].

5 Experimental evaluation

This section is dedicated to the evaluation of the proposed algorithm. Two questions need to be answered: i) How do the schedules improve compared to a task duplication algorithm without contention awareness (Section 5.2.1) ? ii) How does task duplication improve upon other contention-aware scheduling algorithms (Section 5.2.2) ? 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) [25], which is essentially the same algorithm as CA-D, but without the duplication of tasks. This will give us insights about the benefit of task duplication under the contention model. To evaluate the benefit of the contention-awareness, 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.

As discussed in [26], schedules produced under the different models cannot be directly compared. 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. In other words, the tasks are assigned to the same processors in the same order, but now the edges are scheduled in the topology graph to account for contention effects. In [26], it has been shown that this is a valid and realistic method to compare schedules produced under different models. To indicate this contention simulation we named D and LS in the following D-CS and LS-CS.

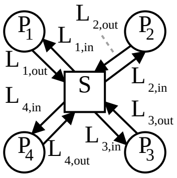

Figure 6: Target parallel system model of experiments, here example with 4 processors

5.1 Setup

For the models of the parallel target systems we have chosen sets of processors, namely 2, 8, 15, 25 and 50 processors, 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 direction can take place at the same time. Figure 6 depicts a four processor example for such a target system model. This star network with central switch corresponds to processors with full-duplex communication ports and this model is also referred to as the one-port model [2]. The employed model is realistic for modern parallel systems and allows concurrent communications to happen between distinct processor pairs. It is also a simple approximation when the network behaviour is unknown or cannot be modelled. The main motivation in using this model, however, is that if task duplication is relevant for contention elimination in such a system, it will be even more important with less ideal networks, where contention can happen in more places.

To demonstrate this effect of more restricted networks, the half-duplex variant of the network is used for comparison (Section 5.2.3), where each processor only possesses a half-duplex communication link (Figure 2). There, incoming and outgoing communication must share the same link. It will be explicitly stated in the following, if this half-duplex variant is used, otherwise the standard network variant as described above is employed.

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 [24]. 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) [24]. CCR is a measure for the importance of communication and is defined as the total edge weight over the total node weight

$$CCR = \frac{\sum_{e \in \mathbf{E}} c(e)}{\sum_{n \in \mathbf{V}} w(n)}.$$

Each type of graph had parameters unique to it. In-trees and out-trees were generated with a maximum branching factor of 3, and were either balanced or unbalanced. SP graphs were generated with spread values of 2, 3, 4, and 5. Random graphs had a density of 0.5, 1 or 3. 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 7 to 8 that display the speedup over the number of processors for six 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, whereby the schedule length is simply the sum of all node weights

$$speedup(\mathcal{S}) = \frac{\sum_{n \in \mathbf{V}} w(n)}{sl(\mathcal{S})}. \tag{14}$$

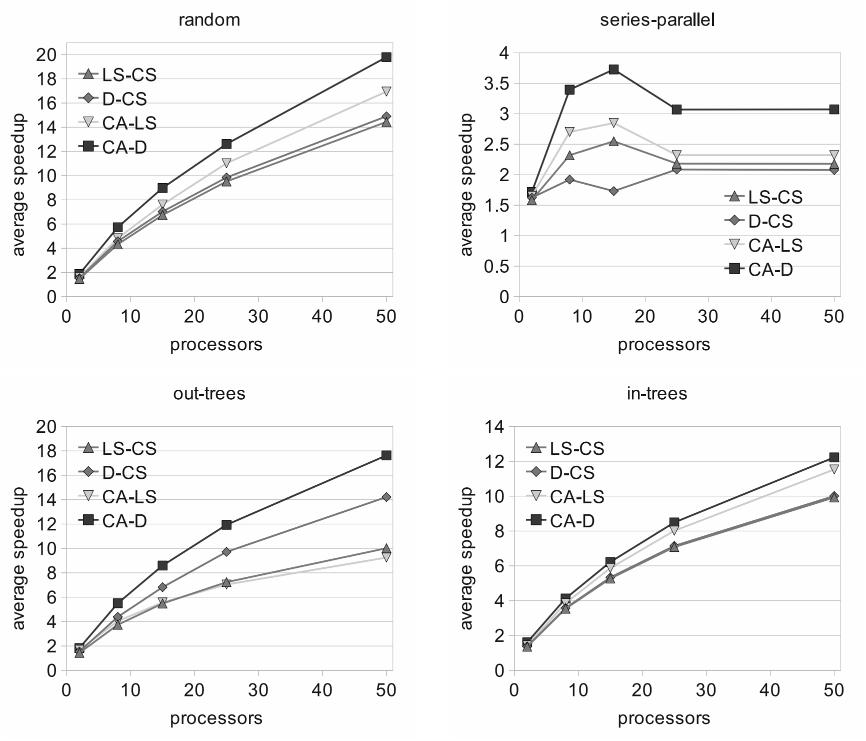

Figure 7: Speedup over processors for random graphs, SP-graphs, out-trees and in-trees

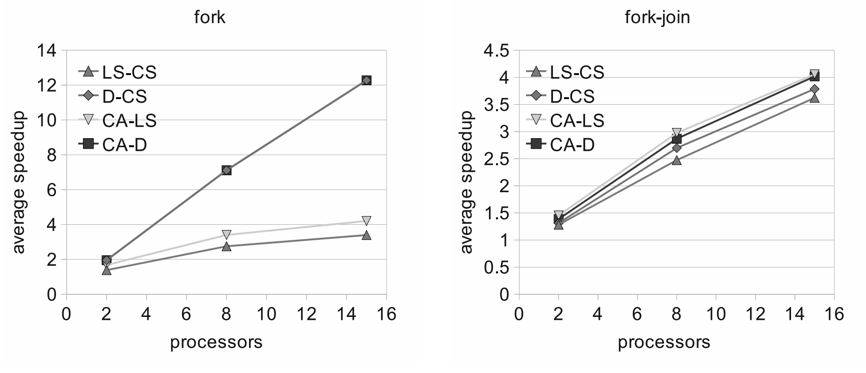

Figure 8: Speedup over processors for fork and fork-join graphs (focus on 2-15 processors); left: D-CS and CA-D are identical

5.2.1 Contention aware duplication (CA-D) vs. non-contention aware duplication (D-CS)

The results 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, where the speedup curves are the same. 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. With more processors, the scaling is limited by the SP graph structure, in particular the low branching factor, for all processors. The speedup on 50 processors for out-trees is 24 percent greater than that of non-contention aware duplication, and for random graphs, the improvement is 33 percent. Contention aware duplication also produces greater speedup for join and in-tree graphs. However, the difference is smaller than with other graphs (as can be seen in Figure 7 for in-trees) because these graphs cannot benefit from duplication due to their structure [24], thus the improvement comes from the contention awareness. Conversely, this also means that due to contention awareness, CA-D generally, i.e. for those graphs where duplication is beneficial, duplicates more tasks than D-CS, which results in less interprocessor communication and in turn less contention.

CA-D and D-CS produce the same curve for fork graphs (Figure 8). This is because both algorithms use the same duplication technique. All nodes in a fork graph, except for the root node, have the root node as their only parent. The root node is duplicated on the processors by both algorithms, which means no remote communication will take place in the schedule. Therefore, the two algorithms actually produce the same schedule.

Figure 9 shows on the left the average speedup across graph types produced by each algorithm for different CCR values on 15 processors. The average spee-

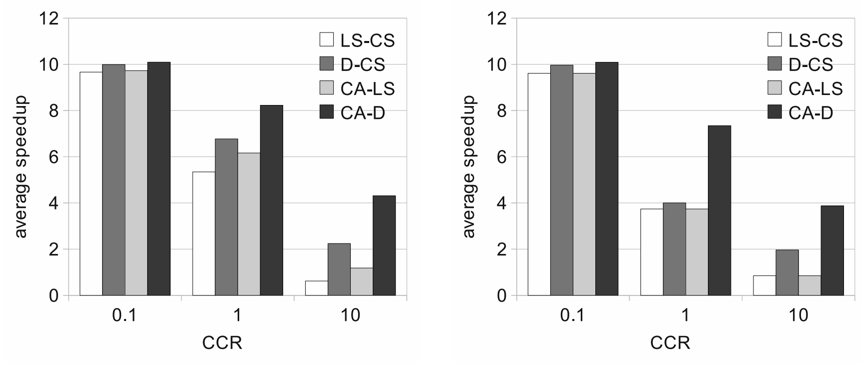

Figure 9: Speedup over CCR for different graph types on 15 processors; left: standard network variant (full-duplex); right: half-duplex network variant

dup values produced by the algorithms for high communication graphs (CCR = 10) show the greatest difference between contention aware duplication (CA-D) and non-contention aware duplication (D-CS). The speedup produced by CA-D is 95 percent greater than that of D-CS. The difference is less, but still significant for medium communication graphs (contention aware duplication is 20 percent greater). The speedup produced for low communication graphs is similar among the algorithms. 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. This complements earlier research [27] which shows that contention aware list scheduling is more efficient than non-contention aware list scheduling.

5.2.2 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 Figures 7 to 8, CA-D has greater speedup on all numbers of processors for all graph types except for fork-join graphs. The greatest difference between CA-D and CA-LS is with fork graphs, where the speedup produced by CA-D is 192 percent greater on 15 processors, since duplication removed the need for any remote communication to occur. 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 50 processors by CA-D is 90 percent greater than that of CA-LS for out-trees, 32 percent greater for SP graphs, and 17 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.

The speedup produced by CA-D for fork-join graphs is slightly worse than that of CA-LS (Figure 8). An analysis of the produced schedules revealed, that CA-D makes the incorrect decision of duplicating the root node at the start of the algorithm, because it does not foresee the significant amount of remote communication that doing this will cause when the final join node is scheduled. However, CA-D is not much worse in this situation and is still better than non-contention aware duplication (D-CS).

Regarding the CCR, the difference between CA-D and CA-LS on 15 processors (Figure 8(left)) is greatest with graphs that have a high level of communication as can be expected. The difference lessens as the communication level decreases.

To summarise, the duplication technique does significantly improve the list scheduling heuristic under the contention model for most task graphs. Using task duplication under the contention model is of crucial advantage for graphs with medium to high communication and even more beneficial than under the classic model.

5.2.3 With more contention: half-duplex network variant

The effect of different networks on contention-aware scheduling was already studied in [27]. In this subsection, the objective is to demonstrate that contention elimination through task duplication becomes more relevant when the network is less ideal. To achieve this, the half-duplex network variant (Section 5.1) is used as the target system, leaving everything else, i.e. graphs and algorithms, as before. Figure 10 repeats the experiments of Figure 7, with the difference that the task graphs were scheduled on the half-duplex network variant. As can be seen, the charts are very similar to the ones in Figure 7 with two very crucial differences: First, the absolute speedup values are lower than before. This is expected, as the network can sustain less communications at the same time, hence the produced schedules become longer. Second, and most importantly, the difference between the contention-aware duplication CA-D and all the other algorithms has significantly increased. In other words, with CA-D the impact of the weaker network on the schedule lengths is less pronounced than with the other algorithms.

The same observations can be made more easily when looking at Figure 9(right), which shows the average speedup across graph types produced by each algorithm for different CCR values on 15 processors. Again, the half-duplex variant was the employed target system. We identify the same changes as in Figure 10: i) the absolute speedup has dropped for all algorithms for CCR = 1 and CCR = 10; ii) the difference between CA-D and the other algorithms as quite dramatically increased for the medium to high communication cases

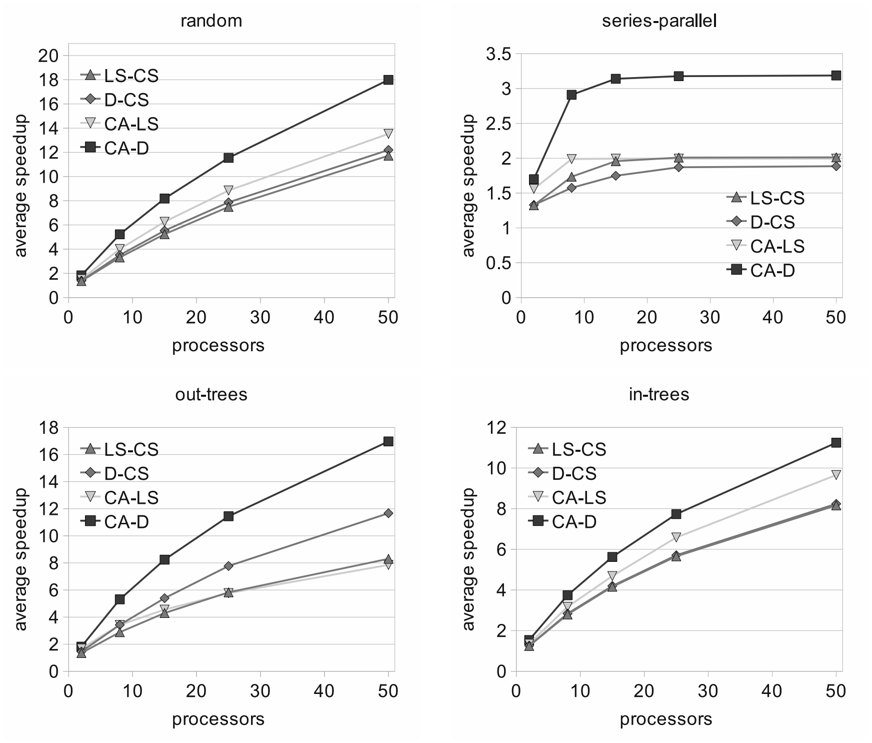

Figure 10: On half-duplex network variant: Speedup over processors for random graphs, SP-graphs, out-trees and in-trees

(CCR = 1 and CCR = 10).

To summarise, CA-D performs in relative terms better when the network is more restricted. This is the intended evidence for our hypothesis that duplication is more important for less ideal networks as stated in Section 5.1.

6 Conclusions

This paper proposed a novel contention-aware task duplication scheduling algorithm. Due to the lack of prior work, it was investigated how task duplication can be performed under the contention model and corresponding methods were devised. Based on this an algorithm was proposed based on state-of-the-art scheduling techniques found in 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 expected, task duplication is even more beneficial under the contention model than under the classic model and this effect increases for more restricted networks. The results strongly recommend task duplication as a standard technique when scheduling task graphs with medium to high communication under the contention model.

References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17:685–689, 1974.

[2] 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.

[3] D. Bozdag, F. Ozguner, and U.V. Catalyurek. Compaction of schedules and a two-stage approach for duplication-based DAG scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):857–871, 2009.

[4] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of

parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[5] D. E. Culler and J. P. Singh. *Parallel Computer Architecture*. Morgan Kaufmann Publishers, 1999.

[6] 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.

[7] 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.

[8] 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.

[9] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 18(2):244–257, April 1989.

[10] T. Kalinowski, I. Kort, and D. Trystram. List scheduling of general task graphs under LogP. *Parallel Computing*, 26:1109–1128, 2000.

[11] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33:1023–1029, November 1984.

[12] Sang Cheol Kim, Sunggu Lee, and J. Hahm. Push-pull: Deterministic search-based dag scheduling for heterogeneous cluster systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(11):1489–1502, 2007.

[13] B. Kruatrachue. *Static Task Scheduling and Grain Packing in Parallel Processing Systems.* PhD thesis, Oregon State University, USA, 1987.

[14] B. Kruatrachue and T. G. Lewis. Grain size determination for parallel processing. *IEEE Software*, 5(1):23–32, January 1988.

[15] 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.

[16] B. S. Macey and A. Y. Zomaya. A performance evaluation of CP list scheduling heuristics for communication intensive task graphs. In *Parallel Processing Symposium, 1998. Proc. of IPPS/SPDP 1998*, pages 538 –541, 1998.

[17] L. Marchal, V. Rehn, Y. Robert, and F. Vivien. Scheduling algorithms for data redistribution and load-balancing on master-slave platforms. *Parallel Processing Letters*, 17(1):61–77, 2007.

[18] M. A. Palis, J.-C. Liou, and D. S. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, January 1996.

[19] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19(2):322–328, April 1990.

[20] P. Rebreyend, F. E. Sandnes, and G. M. Megson. Static multiprocessor task graph scheduling in the genetic paradigm: A comparison of genotype representations. Research Report 98-25, Ecole Normale Superieure de Lyon, Laboratoire de Informatique du Parallelisme, Lyon, France, 1998.

[21] 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.

[22] V. Sarkar. *Partitionning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press, 1989.

[23] 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.

[24] O. Sinnen. *Task Scheduling for Parallel Systems.* Wiley, May 2007.

[25] 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.

[26] O. Sinnen and L. Sousa. On task scheduling accuracy: Evaluation methodology and results. *The Journal of Supercomputing*, 27(2):177–194, February 2004.

[27] O. Sinnen and L. Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, June 2005.

[28] O. Sinnen, L. Sousa, and F. E. Sandnes. Toward a realistic task scheduling model. *IEEE Transactions on Parallel and Distributed Systems*, 17(3):263–275, 2006.

[29] A. Tam and C. L. Wang. Contention-aware communication schedule for high speed communication. 6(4):339–353, 2003.

[30] J. D. Ullman. NP-complete scheduling problems. *Journal of Computing System Science*, 10:384–393, 1975.

[31] 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.

[32] T. Yang and A. Gerasoulis. PYRROS: static scheduling and code generation for message passing multiprocessors. In *Proc. of 6th ACM International Conference on Supercomputing*, pages 428–437, Washington D.C, August 1992.