Parallel Computing 35 (2009) 83–108

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems

Anne Benoit, Mourad Hakem [*], Yves Robert

ENS Lyon, Université de Lyon, LIP Laboratory, UMR 5668, ENS Lyon – CNRS – INRIA – UCBL, Lyon, France

A R T I C L E   I N F O

Article history:
Received 28 May 2008
Received in revised form 29 October 2008
Accepted 6 November 2008
Available online 17 November 2008

Keywords:
Communication contention
Fault-tolerant scheduling
Heterogeneous systems

A B S T R A C T

Heterogeneous distributed systems are widely deployed for executing computationally intensive parallel applications with diverse computing needs. Such environments require effective scheduling strategies that take into account both algorithmic and architectural characteristics. Unfortunately, most of the scheduling algorithms developed for such systems rely on a simple platform model where communication contention is not taken into account. In addition, it is generally assumed that processors are completely safe. To schedule precedence graphs in a more realistic framework, we introduce first an efficient fault-tolerant scheduling algorithm that is both contention-aware and capable of supporting an arbitrary number of fail-silent (fail-stop) processor failures. Next, we derive a more complex heuristic that departs from the main principle of the first algorithm. Instead of considering a single task (one with highest priority) and assigning all its replicas to the currently best available resources, we consider a chunk of ready tasks, and assign all their replicas in the same decision making procedure. This leads to a better load balance of processors and communication links. We focus on a bi-criteria approach, where we aim at minimizing the total execution time, or latency, given a fixed number of failures supported in the system. Our algorithms have a low time complexity, and drastically reduce the number of additional communications induced by the replication mechanism. Experimental results fully demonstrate the usefulness of the proposed algorithms, which lead to efficient execution schemes while guaranteeing a prescribed level of fault-tolerance.

1. Introduction

The efficient scheduling of application tasks is critical in order to achieve high performance in parallel and distributed systems. The objective of scheduling is to find a mapping of the tasks onto the processors, and to order the execution of the tasks so that: (i) task precedence constraints are satisfied and (ii) a minimum schedule length is provided.

Task graph scheduling is usually studied using the so-called *macro-dataflow* model, which is widely used in the scheduling literature: see the survey papers [11,13,27,32] and the references therein. This model was introduced for homogeneous processors, and has been (straightforwardly) extended for heterogeneous computing resources. In this model, there is a limited number of computing resources, or processors, to execute the tasks. Communication delays are taken into account as follows: let task t be a predecessor of task t' in the task graph; if both tasks are assigned to the same processor, no communication overhead is paid, the execution of t' can start right at the end of the execution of t; on the contrary, if t and t' are assigned to two different processors P and P', a communication delay is paid. More precisely, if P finishes the execution of t at time-step x, then P' cannot start the execution of t' before time-step $x + \mathsf{comm}(t, t', P, P')$, where $\mathsf{comm}(t, t', P, P')$ is the

[*] Corresponding author. Tel.: +33 3 84 58 77 38.
E-mail addresses: Anne.Benoit@ens-lyon.fr (A. Benoit), Mourad.Hakem@ens-lyon.fr (M. Hakem), Yves.Robert@ens-lyon.fr (Y. Robert).

0167-8191/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2008.11.001

communication delay (which depends upon both tasks t and t' and both processors P and P'). Because memory accesses are typically one order of magnitude cheaper than inter-processor communications, it makes good sense to neglect them when t and t' are assigned to the same processor.

However, the major flaw of the macro-dataflow model is that communication resources are not limited. First, a processor can send (or receive) an arbitrary number of messages in parallel, hence an unlimited number of communication ports is assumed (this explains the name *macro-dataflow* for the model). Second, the number of messages that can simultaneously circulate between processors is not bounded, hence an unlimited number of communications can simultaneously take place on a given link. In other words, the communication network is assumed to be contention-free, which of course is not realistic as soon as the processor number exceeds a few units.

We strongly believe that the macro-dataflow task graph scheduling model should be modified to take communication resources into account. Recent papers [20,22,33,34] made a similar statement and introduced variants of the model (see the discussion of related work in Section 3). In this paper, we suggest to use the bi-directional one-port architectural model, where each processor can communicate (send and/or receive) with at most one other processor at a given time-step. In other words, a given processor can simultaneously send a message, receive another message, and perform some (independent) computation.

There is yet another reason to revisit traditional list scheduling techniques. With the advent of large-scale heterogeneous platforms such as clusters and grids, resource failures (processors/links) are more likely to occur and have an adverse effect on the applications. Consequently, there is an increasing need for developing techniques to achieve fault-tolerance, i.e., to tolerate an arbitrary number of failures during execution. Scheduling for heterogeneous platforms and fault-tolerance are difficult problems in their own, and aiming at solving them together makes the problem even harder. For instance, the latency of the application will increase if we want to tolerate several failures, even if no actual failure happens during execution.

The bi-criteria approach tackled in this paper enables to provide robust solutions while fulfilling user demands (minimizing latency under some reliability threshold, or the converse). We believe that this new approach has a great significance in practice. Not only (as already mentioned) every user is quite likely to face unrecoverable hardware failures when deploying applications on large-scale distributed platforms such as clusters or grids [14,15,1,12]. But unrecoverable interruptions can also take place in other important frameworks, such as loaned/rented computers being suddenly reclaimed by their owners, as during an episode of *cycle-stealing* [4,10,30].

In this paper, first we introduce a contention-aware fault-tolerant (CAFT) scheduling algorithm that aims at tolerating multiple processor failures without sacrificing the latency. CAFT is based on an active replication scheme to mask failures, so that there is no need for detecting and handling such failures. Next, we derive Iso-Level CAFT, a new more complex heuristic that departs from the main principle of the first algorithm. Instead of considering a single task (the one with highest priority) and assigning all its replicas to the currently best available resources, we consider a chunk of ready tasks, and assign all their replicas in the same decision making procedure. Major achievements include a low complexity, and a drastic reduction of the number of additional communications induced by the replication mechanism. Experimental results demonstrate that our algorithms outperform other algorithms that were initially designed for the macro-dataflow model, such as FTSA [7], a fault-tolerant extension of HEFT [36], and FTBAR [17].

The rest of the paper is organized as follows. Section 2 presents basic definitions and assumptions. We overview related work in Section 3. Then we outline the principle of FTSA [7] and FTBAR [17] as well as their adaptation to the one-port model in Section 4. Section 5 describes the new bi-criteria CAFT and Iso-Level CAFT algorithms. We experimentally compare both CAFT and Iso-Level CAFT with FTSA and FTBAR in Section 6; the results assess the very good behavior of our algorithms. Finally, we conclude in Section 7.

2. Framework

The reader can find a glossary which summarizes all notations in Fig. 6 at the end of the paper.

The execution model for a task graph is represented as a weighted directed acyclic graph (DAG) $G = (V, E)$, where V is the set of nodes corresponding to the tasks, and E is the set of edges corresponding to the precedence relations between the tasks. In the following we use the term node or task indifferently; $v = |V|$ is the number of nodes, and $e = |E|$ is the number of edges. In a DAG, a node without any predecessor is called an *entry* node, while a node without any successor is an *exit* node. For a task t in G, $\Gamma^{-}(t)$ is the set of immediate predecessors and $\Gamma^{+}(t)$ denotes its immediate successors. Task t cannot start being executed before it has received the output from all predecessors in $\Gamma^{-}(t)$, and the result of task t can be sent to successor tasks in $\Gamma^{+}(t)$ starting from the time at which the execution of t is finished. We let $\mathscr{V}$ be the edge cost function: $\mathscr{V}(t_i, t_j)$ represents the volume of data that task t_i needs to send to task t_j.

A target parallel heterogeneous system consists of a finite number of processors $\mathscr{P} = \{P_1, P_2, \ldots, P_m\}$ connected by a dedicated communication network. The processors are fully connected, i.e., every processor can communicate with every other processor in the system. The communication link between processors P_k and P_h is denoted by l_{kh}. As stated above, in the traditional macro-dataflow model, there is no contention for communication resources, and an unlimited number of communications can occur concurrently. The bi-directional one-port model considered in this work deals with communication contention as follows:

– At a given time-step, any processor can send a message to, and receive a message from, at most one other processor. Network interfaces are assumed full-duplex in this bi-directional model.
– Communication and (independent) computation may fully overlap. As in the traditional model, a processor can execute at most one task at each time-step.
– Communications that involve disjoint pairs of sending/receiving processors can occur in parallel.

Several variants could be considered, such as uni-directional communications, or no communication/computation overlap. But the bi-directional one-port model seems closer to the actual capabilities of modern networks (see the discussion in Section 3).

The computational heterogeneity of tasks is modeled by a function $\mathscr{E} : V \times \mathscr{P} \to R^+$, which represents the execution time of each task on each processor in the system: $\mathscr{E}(t_i, P_k)$ denotes the execution time of t_i on P_k, $1 \leqslant k \leqslant m$. The communication between tasks t_i and t_j is denoted c_{ij}, and heterogeneity in terms of communication costs is expressed by the cost of each communication on each link: $W(c_{ij}, l_{kh}) = \mathscr{V}(t_i, t_j).d(P_k, P_h)$, where task t_i is mapped on processor P_k, task t_j is mapped on processor P_h, and $d(P_k, P_h)$ is the time required to send a unit length data from P_k to P_h. The communication has no cost if two tasks in precedence are mapped onto the same processor: $d(P_k, P_k) = 0$.

Our goal is to find a task mapping of the DAG G on the platform $\mathscr{P}$ obeying the one-port model. The objective is to minimize the latency $\mathscr{L}(G)$, while tolerating an arbitrary number ε of processor failures. Our approach is based on an active replication scheme, capable of supporting ε arbitrary fail-silent (fail-stop) processor failures, hence valid results will be provided even if ε processors fail.

3. Related work

Contention-aware task scheduling is considered only in a few papers from the literature [20,22,33,34,37]. In particular, Sinnen and Sousa [33,34] show through simulations that taking contention into account is essential for the generation of accurate schedules. They investigate both end-point and network contention. Here end-point contention refers to the bounded multi-port model [21]: the volume of messages that can be sent/received by a given processor is bounded by the limited capacity of its network card. Network contention refers to the one-port model, which has been advocated by Bhatt et al. [8,9] because "current hardware and software do not easily enable multiple messages to be transmitted simultaneously". Even if non-blocking multi-threaded communication libraries allow for initiating multiple send and receive operations, all these operations "are eventually serialized by the single hardware port to the network". Experimental evidence of this fact has recently been reported by Saif and Parashar [31], who report that asynchronous sends become serialized as soon as message sizes exceed a few megabytes. Their results hold for two popular implementations of the MPI message-passing standard, MPICH on Linux clusters and IBM MPI on the SP2. The one-port model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It generalizes a simpler model studied by Banikazemi [5], Liu [24], and Khuller and Kim [23]. In this simpler model, the communication time only depends on the sender, not on the receiver: in other words, the communication speed from a processor to all its neighbors is the same.

All previous scheduling heuristics [20,22,33,34,37] were developed for minimizing latency on realistic parallel platform models, assuming that processors in the system are completely safe, i.e., they do not deal with fault-tolerance. In multiprocessor systems, fault tolerance can be provided by scheduling multiples copies (replicas) of tasks on different processors. A large number of techniques for supporting fault-tolerant systems have been proposed [3,16–19,25,26,28,29]. There are two main approaches, as described below.

(i) *Primary/backup* (*passive replication*): this is the traditional fault-tolerant approach where both time and space exclusions are used. The main idea of this technique is that the backup task is activated only if the fault occurs while executing the primary task [26,28]. To achieve high schedulability while providing fault–tolerance, the heuristics presented in [3,16,25,29] apply two techniques while scheduling the primary and backup copies of the tasks:
   – backup overloading: schedule backups for multiple primary tasks during the same time slot in order to make efficient utilization of available processor time, and
   – de-allocation of resources reserved for backup tasks when the corresponding primaries complete successfully. Note that these techniques can be applied only under the assumption that only one processor may fail at a time. All algorithms belonging to this class [3,16,25,26,28,29] share two common points:
   – only one processor can fail at any time, and a second processor cannot fail before the system recovers from the first failure;
   – they are designed for the macro-dataflow model.

(ii) *Active replication* (*N-modular redundancy*): this technique is based on space redundancy, i.e., multiple copies of each task are mapped on different processors, which are run in parallel to tolerate a fixed number of failures. For instance, Hashimoto et al. [18,19] propose an algorithm that tolerates one processor failure on homogeneous system. This algorithm exploits implicit redundancy (originally introduced by task duplication in order to minimize the schedule length) and assumes that some processors are reserved only for fault-tolerance, i.e., the reserved processors are not used in the original schedule. Girault et al. present FTBAR, a static real-time and fault-tolerant scheduling algorithm where multiple processor failures are considered [17]. Recently, we have proposed the FTSA algorithm [7], a fault-tolerant extension of HEFT [36]. We showed that FTSA outperforms FTBAR in terms of time complexity and solution quality. Here again, FTSA and FTBAR have been designed for the macro-dataflow model. A brief description of FTSA and FTBAR, as well as their adaptation to the one-port model, is given in Section 4.

To summarize, all previous fault-tolerant algorithms assume no restriction on communication resources, which makes them impractical for real life applications. To the best of our knowledge, the work presented in this paper is the first to tackle the combination of contention awareness and fault-tolerant scheduling.

4. Fault-tolerant heuristics

In this section, we briefly outline the main features of FTBAR [17] and FTSA [7], that both were originally designed for the macro-dataflow model. Next we show how to modify them for an execution under the one-port model.

4.1. FTBAR

FTBAR [17] (fault-tolerance based active replication) is based on an existing list scheduling algorithm presented in [35]. Using the original notations of [17], at each step n in the scheduling process, one free task (a task is free if it is unscheduled and if all of its predecessors are scheduled) is selected from the list based on the cost function $\sigma^{(n)}(t_i, p_j)$, called *schedule pressure*. It is computed as follows:

$$\sigma^{(n)}(t_i, p_j) = S^{(n)}(t_i, p_j) + \bar{s}(t_i) - R^{(n-1)}.$$

The superscript number in parentheses refers to the step of the heuristic. $S^{(n)}(t_i, p_j)$ is the earliest start time (top–down) of t_i on p_j, similarly, $\bar{s}(t_i)$ is the latest start time (bottom–up) of t_i and $R^{(n-1)}$ is the schedule length (latency) at step $n-1$, *i.e.*, the total time required to execute all the tasks that have already been scheduled (before task i is scheduled). The selected task-processor pair is obtained as follows:

(i) select for each free task t_i, the $\varepsilon + 1$ processors having the minimum *schedule pressure*, where ε is the number of processor failures tolerated in the system;
(ii) select the most urgent pair (task, processor) among the previous set, *i.e.*, the one having the maximum *schedule pressure* (for all free tasks and all corresponding processors selected at step (i)).

The task t selected at step (ii) is then scheduled on the $\varepsilon + 1$ processors computed at the first step. Ties are broken randomly. A recursive *minimize-start time* procedure proposed by Ahmad and Kwok [2] is used in attempting to reduce the start time of the selected task t. The time complexity of the algorithm is $O(PN^3)$, where P is the number of processors in the system and N the number of tasks in G.

4.2. FTSA

FTSA (fault-tolerant scheduling algorithm) has been introduced in [7] as a fault-tolerant extension of HEFT [36]. At each step of the mapping process, FTSA selects a free task t (a task is free if it is unscheduled and if all of its predecessors are scheduled) with the highest priority and simulates its mapping on all processors. The first $\varepsilon + 1$ processors that allow the *minimum finish time* of t are kept. The finish time of a task t on processor P depends on the time when at least one replica of each predecessor of task t has sent its results to P (and, of course, processor P must be ready). Once this set of processors is determined, the task t is scheduled on these $\varepsilon + 1$ distinct processors (replicas). The latency of the schedule is the latest time at which at least one replica of each task has been computed, thus it represents a **lower bound** (this latency can be achieved if no processor permanently fails during execution). The **upper bound**, always achieved even with ε failures, is computed using as a finish time the completion time of the last replica of a task (instead of the first one for the lower bound). A formal definition can be found in [7].

The time complexity of the algorithm is $O(em^2 + v \log \omega)$, where ω is the *width* of the task graph (the maximum number of tasks that are independent in G). Recall that m is the number of processors, v is the number of tasks, and e is the number of edges in G.

Note that with FTSA, each task of the task graph G is replicated $\varepsilon + 1$ times, because this replication is an absolute requirement to resist to ε failures. Therefore each communication between two tasks in precedence is replicated at most $(\varepsilon + 1)^2$ times. Since there are e edges in G, the total number of messages in the fault-tolerant schedule is at most $e(\varepsilon + 1)^2$. In some cases, we may have an intra-processor communication, when two tasks in precedence are mapped on the same processor, so the latter quantity is in fact an upper bound. Still, the total number of communications induced by the fault-tolerant mechanism is very large. The same comment applies to FTBAR, where each replica of a task communicates data to each replica of its successors.

4.3. Adaptation to the one-port model

In order to adapt both FTSA and FTBAR algorithms to the one-port model, we have to take constraints related to communication resources into account. The idea consists in serializing incoming and outgoing communications on the links. Note that we do not exploit idle time slots that may occur on some communication links, as has been done for instance in [17,2,37]. Rather, once a communication has been scheduled, the ready time of the link is the time at which this last communication terminates. This drastically reduces the time complexity of the algorithm. Otherwise we would need to scan through the whole time span of each link to find the earliest time slot that is large enough to accommodate the communication, provided that precedence constraints are not violated.

A communication c on link l is characterized by its start time $\mathscr{S}(c,l)$ and its finish time $\mathscr{F}(c,l)$. Also, we define $\mathscr{R}(l)$ as the ready time of a communication link l: if C_l is the set of communications scheduled on link l, then $\mathscr{R}(l) = \max_{c \in C_l}(\mathscr{F}(c,l))$. In the following, we formalize all the one-port constraints.

(i) *Link constraint*: For any two communications c, c' scheduled on link l,

$$\mathscr{F}(c,l) \leqslant \mathscr{S}(c',l) \vee \mathscr{F}(c',l) \leqslant \mathscr{S}(c,l). \tag{1}$$

Inequality (1) states that any two communications c and c' do not overlap on a link.

(ii) *Sending constraint*: For any two communications c, c' sent from a given processor P_k, respectively, to two processors $\mathscr{P}_h$, $P_{h'}$,

$$\mathscr{F}(c,l_{kh}) \leqslant \mathscr{S}(c',l_{kh'}) \vee \mathscr{F}(c',l_{kh'}) \leqslant \mathscr{S}(c,l_{kh}). \tag{2}$$

(iii) *Receiving constraint*: For any two communications c, c' sent from processors P_k and $P_{k'}$ to the same processor P_h,

$$\mathscr{S}(c,l_{kh}) \geqslant \mathscr{F}(c',l_{k'h}) \ \vee \ \mathscr{S}(c',l_{k'h}) \geqslant \mathscr{F}(c,l_{kh}). \tag{3}$$

Inequalities (2) and (3) ensure that any two incoming/outgoing communications c and c' must be serialized at their reception/emission site.

Let t_i be a task scheduled on processor P. $\mathscr{F}(t_i,P)$ is the time at which processor P has finished the processing of task t_i. Let $\mathscr{S}_F(P)$ be the sending free time of P, *i.e.*, the time at which all sending communications already scheduled on P are finished. The earliest start time and finish time of the communication c_{ij}, with $1 \leqslant j \leqslant |\Gamma^+(t_i)|$, scheduled on link l, are defined by the following equations:

$$\begin{aligned} &\mathscr{S}(c_{ij},l) = \max(\mathscr{F}(t_i,P), \mathscr{S}_F(P), \mathscr{R}(l)), \\ &\mathscr{F}(c_{ij},l) = \mathscr{S}(c_{ij},l) + W(c_{ij},l). \end{aligned} \tag{4}$$

Thus, communication c_{ij} is constrained by both $\mathscr{S}_F(P)$, $\mathscr{R}(l)$ and the finish time of its source task t_i on P. It can start as soon as the processing of the task is finished only if we have $\mathscr{F}(t_i,P) \geqslant \mathscr{S}_F(P)$ and $\mathscr{F}(t_i,P) \geqslant \mathscr{R}(l)$.

The start time of task t_i on processor P is constrained by communications incoming from its predecessors that are assigned on other processors. Thus, $\mathscr{S}(t_i,P)$ satisfies the following conditions: it is later than the time when all messages from t_i's predecessors arrive on processor P, and it is later than the ready time of processor P, defined as the maximum of finish time of all tasks already scheduled on P. Let $\mathscr{A}(c,P)$ be the time when communication c arrives on processor $\mathscr{P}$, and $r(P)$ be the ready time of P. The start time of t_i on P is defined as follows:

$$\mathscr{S}(t_i,P) = \max\left(\max_{t_j \in \Gamma^-(t_i)}\{\mathscr{A}(c_{ji},P)\}, r(P)\right), \tag{5}$$

where c_{ji} is the communication from t_j to t_i.

The arrival time $\mathscr{A}(c_{ji},P)$ is computed for each predecessor as follows. Let l_j be the communication link that connects the processor on which t_j is mapped to P. Let $\mathscr{R}_F(P)$ be the receiving free time of P, i.e., the time when P is ready to receive data. We sort predecessors $t_j \in \Gamma^-(t_i)$ by non-decreasing order of their communication finish time $\mathscr{F}(c_{ji},l_j)$, and renumber them from 1 to $|\Gamma^-(t_i)|$. Then we have:

- $\mathscr{F}(c_{0i},l_0) = 0$.
- For $j = 1$ to $|\Gamma^-(t_i)|$

$$\mathscr{A}(c_{ji},P) \leftarrow W(c_{ji},l_j) + \max(\mathscr{R}_F(P), \mathscr{F}(c_{(j-1)i},l_{j-1}), \mathscr{F}(c_{ji},l_j) - W(c_{ji},l_j)), \tag{6}$$

where $\mathscr{F}(c_{ji},l_j) - W(c_{ji},l_j)$ reflects the start time of the communication c_{ji} scheduled to the link l_j. Eq. (6) shows that the arrival time $\mathscr{A}(c,P)$ is constrained by the receiving free time $\mathscr{R}_F(P)$ of P, since other tasks may have been scheduled on processor P, thus using its communication resources. In addition, it complies with the inequality (3), i.e., concurrent communications are serialized at the reception site.

5. Bi-criteria algorithms

In this section, two efficient bi-criteria algorithms, CAFT and Iso-Level CAFT, are developed to tackle the combination of contention awareness and fault-tolerant scheduling.

5.1. The CAFT scheduling algorithm

The one-port model enforces to serialize communications. But as pointed out above, the duplication mechanism induces a large number of additional communications. Therefore, we expect execution time to drastically increase together with the number of supported failures. This calls for the design of a variant of FTSA in which the number of communications induced by the replication scheme is reduced as much as possible. The main idea of the new CAFT (contention-aware fault-tolerant) scheduling algorithm is to have each replica of a task communicate to a unique replica of its successors whenever possible, while preserving the fault-tolerance capability (guaranteeing success if at most ε processors fail during execution). Communicating to a single replica is only possible in special cases, typically for tasks having a unique predecessor, or when every replica of all predecessors are mapped onto distinct processors. When these constraints are not satisfied, we greedily add extra communications to guarantee failure tolerance, as illustrated below through a small example.

In the following, we denote by $\mathscr{B}(t)$ the set of $\varepsilon + 1$ replicas of a task t. Also, we denote by $t^{(k)}$ those replicas, for $1 \leqslant k \leqslant \varepsilon + 1$. Thus, $\mathscr{B}(t) = \{t^{(1)}, \ldots, t^{(\varepsilon+1)}\}$. $P(t^{(k)})$ is the processor on which replica $t^{(k)}$ is scheduled.

During the scheduling process, the graph consists of two parts, the already examined (scheduled) tasks S and the unscheduled tasks U. Initially $U = V$. Let t be the current task to be scheduled by CAFT. Consider a predecessor t_j of t, $j \in \Gamma^-(t)$, that has been replicated on $\varepsilon + 1$ distinct processors. We aim at orchestrating communications incoming from predecessors t_j to t so that each replica in $\mathscr{B}(t_j)$ communicates to only one replica in $\mathscr{B}(t)$ when possible, rather than communicating to all of them as in the FTSA and FTBAR algorithms.

If t has only one predecessor t_1, then a one-to-one communication scheme resists to ε failures, as it was proved in [7]. Indeed, we can find the best mapping in which each replica of task t_1 sends data to exactly one of the replicas of t. The problem becomes more complex when t has more than one predecessor, for instance t_1, t_2 and t_3, and when replicas of different instances are mapped on a same processor. For instance, let us have $\varepsilon = 1$, $t_1^{(1)}$ and $t_2^{(1)}$ mapped on P_1, $t_1^{(2)}$ and $t_3^{(1)}$ mapped on P_2, $t_2^{(2)}$ and $t_3^{(2)}$ mapped on P_3 as shown in Fig. 1. In all possible schedules for t, both $t^{(1)}$ and $t^{(2)}$ need to receive data from two distinct processors. Hence one processor among P_1, P_2 and P_3 must communicate with both replicas. If this particular processor crashes, both replicas will miss data to continue execution, and thus the application cannot tolerate this single failure as shown in Fig. 1a. In such cases in which a processor is processing several replicas of predecessors and communicating with different replicas of t, we need to add extra communications to ensure fault-tolerance as shown in Fig. 1b.

Algorithm 1 is the main CAFT algorithm. Tasks are scheduled in an order defined by the priority of the task: the priority of a free task t is determined by $t\ell(t) + b\ell(t)$, where $t\ell(t)$ and $b\ell(t)$ are, respectively, the top level and the bottom level of task t. The top level $t\ell(t)$ is the length of the longest path from an entry (top) node to t (excluding the execution time of t) in the current partially clustered DAG. The top level of an entry node is zero. Top levels are computed according to a traversal of the graph in topological order. The bottom level $b\ell(t)$ is the length of the longest path starting at t to an exit node in the graph. The bottom level of an exit node is equal to its execution time. Bottom levels are computed according to a traversal of the graph in reverse topological order. For bottom levels, since we do not know on which processor each task will be assigned, we use the average execution time of t, defined as $\frac{1}{m}\sum_{j=1}^{m}\mathscr{E}(t, \mathscr{P}_j)$, and the average communication cost of the edge (t_i, t_j), defined as $\mathscr{V}(t_i, t_j).\bar{d}$, where $\bar{d}$ is the average delay to send a unit length data between two processors in the system. A critical task is defined as one of the free tasks with the highest priority. The definition of criticalness provides a good measure of the task importance: the greater the criticalness, the more work is to be performed along the path containing that task. $\mathscr{H}(\ell)$ is the head function which returns the first replica/task from a sorted list ℓ, where the list is sorted according to replicas/tasks priorities (ties are

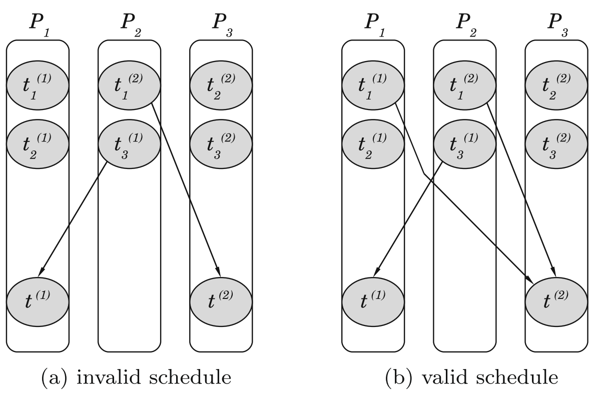


Fig. 1. Communication overhead reduction.

broken randomly). The difficult point consists in deciding where to place current task t in order to minimize the amount of communications. Also, communications should be orchestrated to avoid useless data transfer between replicas.

Algorithm 1. The CAFT Algorithm

```
1: $P = \{P_1, P_2, \dots P_m\}$; (* Set of processors *)
2: $\varepsilon \leftarrow$ maximum number of supported failures;
3: $\mathbb{P} = \emptyset$;
4: Compute $b\ell(t)$ for each task $t$ in $G$ and set $t\ell(t) = 0$ for each entry task $t$;
5: $S = \emptyset$; $U = V$; (* Mark all tasks as unscheduled *)
6: $\alpha = \emptyset$; (* List of free tasks *)
7: Put entry tasks in $\alpha$;
8: while $U \neq \emptyset$ do
9:    $t \leftarrow \mathscr{H}(\alpha)$; (* Select task with highest priority *)
10:   $\forall 1 \leqslant j \leqslant |\Gamma^-(t)|$, compute $\lambda_j$;
11:   $\theta \leftarrow \min_j(\lambda_j)$; $i = 0$;
12:   while $i < \theta$ do
13:      One-To-One Mapping($t$);
14:      $i = i + 1$;
15:   end while
16:   while $\theta < \varepsilon + 1$ do
17:      Compute $\mathscr{F}(t, P_k)$ for $1 \leqslant k \leqslant m$ and $P_k \notin \mathbb{P}$ using Eq. (6);
18:      Keep the (task, processor) pair that allows the minimum finish time of $t$;
19:      $\theta = \theta + 1$;
20:   end while
21:   Put $t$ in $S$ and update priority values of $t$'s successors;
22:   Put $t$'s free successors in $\alpha$;
23:   $U \leftarrow U \setminus \{t\}$;
24: end while
```

If t is the current task, let us define a *singleton processor* as a processor with only one instance/replica $t_j^{(k)}$, $1 \leqslant j \leqslant |\Gamma^-(t)|$, $1 \leqslant k \leqslant \varepsilon + 1$, and let $\mathscr{X} \subseteq \bigcup_{j=1}^{|\Gamma^-(t)|}\{P(\mathscr{B}(t_j))\}$ be the set of such *singleton processors*. Let $\overline{\mathscr{B}}(t_j)$ be the subset of replicas of each predecessor t_j scheduled in $\mathscr{X}$ and $\lambda_j = |\overline{\mathscr{B}}(t_j)|$. Let T be a subset of replicas selected from the set $\bigcup_{j=1}^{|\Gamma^-(t)|}\{\overline{\mathscr{B}}(t_j)\}$.

When there are enough singleton processors with replicas of predecessor tasks, we use the one-to-one mapping procedure described in Algorithm 2. This name stems from the fact that each replica in $\bigcup_{j=1}^{|\Gamma^-(t)|}\overline{\mathscr{B}}(t_j)$ should communicate to exactly one replica in $\mathscr{B}(t)$. The number of times the one-to-one mapping procedure can be called for scheduling the $\varepsilon + 1$ replicas of the current task is determined by $\theta \leftarrow \min_j(\lambda_j)$. In this procedure, we denote by $\mathbb{P} \subseteq P$ the subset of "locked" processors which are already either involved in a communication with a replica of t, or processing it (*i.e.*, the execution of a replica of $\mathscr{B}(t)$ has been scheduled on such a processor).

Algorithm 2. One-To-One Mapping(t)

```
1: $k = 0$;
2: while $k \leqslant m$ and $P_k \notin \mathbb{P}$ do
3:    $\forall 1 \leqslant j \leqslant |\Gamma^-(t)|$, sort the set $\overline{\mathscr{B}}(t_j)$ by non-decreasing order of their communication finish time $\mathscr{F}(c, l)$ on the links;
4:    $T \leftarrow \bigcup_{1 \leqslant j \leqslant |\Gamma^-(t)|}\mathscr{H}(\overline{\mathscr{B}}(t_j))$;
5:    Simulate the mapping of $t$ on processor $P_k$ as well as the communications induced by the replicas of the set $T$ to the links;
6:    $k = k + 1$;
7: end while
8: Select the (task, processor) pair that allows the earliest finish time of $t$ as computed by Eq. (6);
9: Schedule $t$ onto the corresponding processor (let's call it $P^*$) and the incoming communications to the corresponding links;
10: Update the set $\mathbb{P}$
      $$\mathbb{P} \leftarrow \mathbb{P} \bigcup P^* \bigcup \left\{ \bigcup_{j=1}^{|\Gamma^-(t)|} P(\mathscr{H}(\overline{\mathscr{B}}(t_j))) \right\}. \quad (7)$$
11: Update each sorted list $\overline{\mathscr{B}}(t)$;
      $\forall 1 \leqslant j \leqslant |\Gamma^-(t)|, \overline{\mathscr{B}}(t_j) \leftarrow \overline{\mathscr{B}}(t_j) \setminus \mathscr{H}(\overline{\mathscr{B}}(t_j))$.
```

The computation of the finish time of t is simulated m times, once for every processor. Hence the mapping of each incoming communication onto the links is also simulated m times. To obtain an accurate view of the communication finish times on their respective links, and hence of the contention, incoming communications are removed from the links before the procedure is repeated on the next processor.

In the following we give an analytical expression of the actual number of communications induced by the CAFT algorithm for special graphs.

Proposition 1. *The total number of messages generated by CAFT for Fork and Outforest graphs is at most $e(\varepsilon + 1)$.*

Proof. An outforest graph is a directed graph in which the in-degree of every task t in G is at most one $|\Gamma^-(t)| = 1$. To resist to ε failures, each task $t \in G$ should be replicated $\varepsilon + 1$ times. Therefore, at each step of the mapping process we have $\cap P(\mathscr{B}(t_* \prec t)) = \emptyset$ and $|\mathscr{X}| = \theta = \varepsilon + 1$. Thus, the *one-to-one mapping procedure* is performed $\theta = \varepsilon + 1$ times. This ensures that each replica $t_*^{(k)}$, $1 \leqslant k \leqslant \varepsilon + 1$ sends its data results to one and only one replica of each successor task. Therefore, each task $t \in G$ will receive its input data $|\Gamma^-(t)| = 1$ times. However, in some cases, we may have an intra-processor communication, when two replicas of two tasks in precedence are mapped onto the same processor. Thus, summing up for all the v tasks in G, the total number of messages is at most $\sum_{i=1}^{v} |\Gamma^-(t)|(\varepsilon + 1) = (v - 1)(\varepsilon + 1) = e(\varepsilon + 1)$. □

For general graphs, the number of communications will also be bounded by $e(\varepsilon + 1)$ if at each step replicas are assigned to different processors (same proof as above). This condition is not guaranteed to hold, and we will have to greedily add some additional communications to preserve the robustness of CAFT. However, practical experiments (see Section 6) show that CAFT always drastically reduces the total number of messages as compared to FTBAR or FTSA, thereby achieving much better performance.

Proposition 2. *The schedule generated by the CAFT algorithm is valid and resists to ε failures.*

When a current task t to be mapped has more than one predecessor and $\theta = 0$, the *one-to-one mapping procedure* is not executed and therefore CAFT algorithm performs more than $e(\varepsilon + 1)$ communications. In this case we can resist to ε failures as it was proved in [7]. Therefore, we just need to check whether the mapping of the θ replicas performed by the *one-to-one mapping procedure* resists to $\theta - 1$ failures.

Proof. The proof is composed of two parts.

(i) *Deadlock/mutual exclusion*: First we prove that we never fall into a deadlock trap as described by the example below. Consider a simple task graph composed of two tasks in precedence $t_1 \prec t_2$. Assume that $\varepsilon = 1$, $\mathscr{B}(t_1) = \{t_1^{(1)}, t_1^{(2)}\}$ with $P(\mathscr{B}(t_1)) = \{P_1, P_2\}$ and $\mathscr{B}(t_2) = \{t_2^{(1)}, t_2^{(2)}\}$ with $P(\mathscr{B}(t_2)) = \{P_1, P_3\}$. If we retain the communications $P_1(t_1^{(1)}) \rightarrow P_3(t_2^{(2)})$ and $P_2(t_1^{(2)}) \rightarrow P_1(t_2^{(1)})$, then the algorithm is blocked by the failure of P_1. But if we enforce that the only edge from P_1 goes to itself, then we resist to one failure. Mutual exclusion is guaranteed by Eq. (7). Indeed, by simulating the mapping of t_2 on P_1,P_2 and P_3, we have two possible scenarios:
  (1) The first replica $t_2^{(1)}$ is mapped either on P_1 or on P_2, and in either cases the one assigned to the replica will be locked by Eq. 7. Suppose for instance that P_1 was chosen/locked, thus, to resist to one failure, the second replica $t_2^{(2)}$ should be mapped on $P_2 \vee P_3$. If P_2 is selected, in this case we have two internal communications. If P_3 is selected, we have one internal communication $P_1(t_1^{(1)}) \rightarrow P_1(t_2^{(1)})$ and one inter-processor communication $P_2(t_1^{(2)}) \rightarrow P_3(t_2^{(2)})$.
  (2) The first replica $t_2^{(1)}$ is mapped on P_3, then both P_3 and $P_1 \vee P_2$ are locked by Eq. (7). Suppose that P_1 is locked, then second replica $t_2^{(2)}$ should be mapped on P_2. So we have an internal communication between $P_2(t_1^{(2)}) \rightarrow P_2(t_2^{(1)})$ and an inter-processor communication $P_1(t_1^{(1)}) \rightarrow P_3(t_2^{(2)})$.

In both scenarios, we resist to one failure. All processors in $\mathbb{P}$ remain locked during the mapping process of the $\varepsilon + 1$ replicas of a task. They are unlocked only before the next step, *i.e.*, before the CAFT algorithm is repeated for the next critical free task.

(ii) *Space exclusion*: The *one-to-one mapping procedure* is based on an active replication scheme with space exclusion. Thus, each task is replicated θ times onto θ distinct processors. We have at most $\theta - 1$ processor failures at the same time. So at least one copy of each task is executed on a fault free processor. □

Theorem 1. *The time complexity of CAFT is*

$$O(em(\varepsilon + 1)^2 \log(\varepsilon + 1) + v \log \omega).$$

Proof. The proof is composed of two parts.

(i) *One-to-one mapping procedure* (*Algorithm* 2): The main computational cost of this procedure is spent in the while loop (Lines 2–7). Line 5 costs $O(|\Gamma^-(t)|m)$, since all the instances/replicas in T of the immediate predecessors t_j of task t need to be examined on each processor $P_k, 1 \leqslant k \leqslant m$. Line 3 costs $O(|\Gamma^-(t)|(\varepsilon + 1)\log(\varepsilon + 1))$ for sorting the lists $\overline{\mathscr{B}}(t_j)$, $1 \leqslant j \leqslant |\Gamma^-(t)|$. Line 7 costs $O(|\Gamma^-(t)| \log(\varepsilon + 1))$ for finding the head of the lists $\overline{\mathscr{B}}(t_j)$, $1 \leqslant j \leqslant |\Gamma^-(t)|$. Thus, the cost of this procedure for the whole m loops is $O(|\Gamma^-(t)|(\varepsilon + 1)m \log(\varepsilon + 1))$.

(ii) *CAFT* (*Algorithm* 1): Computing $b\ell(t)$ (Line 4) takes $O(e + v)$. Insertion or deletion from α costs $O(\log|\alpha|)$ where $|\alpha| \leqslant \omega$, the width of the task graph, i.e., the maximum number of tasks that are independent in G. Since each task in a DAG is inserted into α once and only once and is removed once and only once during the entire execution of CAFT, the time complexity for α management is in $O(v \log \omega)$. The main computational cost of CAFT is spent in the while loop (Lines 8–24). This loop is executed v times. Line 9 costs $O(\log \omega)$ for finding the head of α. Line 10 costs $|\Gamma^-(t)|m$ to determine λ_j. The two loops (12–15) and (16–20) are executed $\varepsilon + 1$ times. Line 17 costs $O(|\Gamma^-(t)|(\varepsilon + 1)m)$, since all the instances/replicas of the immediate predecessors t_j of task t need to be examined on each processor $P_k, 1 \leqslant k \leqslant m$. Line 21 costs $O(|\Gamma^+(t)|)$ to update the priority values of the immediate successors of t. Thus, the cost for the v loops of this line is $O(v|\Gamma^+(t)|) = O(e)$. Thus, the total cost of CAFT for the v tasks in G is

$$\sum_{i=1}^{v} O(|\Gamma^-(t_i)|m(\varepsilon+1)^2\log(\varepsilon+1) + e + v\log\omega) = O(em(\varepsilon+1)^2\log(\varepsilon+1) + v\log\omega).$$

Since $\varepsilon < m$, we can derive the upper bound $O(em^3\log m + v\log\omega)$. □

5.2. The Iso-Level CAFT scheduling algorithm

In the previous version of CAFT algorithm, we consider only one ready task (the one with highest priority) at each step, and we assign all its replicas to the currently best available resources. Instead of considering a single task, we may deal with a chunk of several ready tasks, and assign all their replicas in the same decision making procedure. The intuition is that such a "global" assignment would lead to better load balance processor and link usage.

We introduce a parameter B for the chunk size: B is the maximal number of ready tasks that will be considered at each step. We select the B critical tasks with the higher priority and we allocate them in the same step. Then, we update the set of ready tasks (indeed some new tasks may have become ready), and we sort them again, according to their priority values. Thus, we expect that the tasks on a critical path will be processed as soon as possible.

The difference between CAFT and the new version, which we call Iso-Level CAFT (or ILC), is sketched in Algorithm 3. With CAFT we take the ready task with highest priority and allocate all its replicas before proceeding to the next ready task. In contrast, with Iso-Level CAFT, the second replicas of tasks in the same chunk are allocated only after all first replicas have been placed. Intuitively, this more global strategy will balance best resources across all tasks in the chunk, while CAFT may assign the $\varepsilon + 1$ best resources to the current task, at the risk of sacrificing the next one, even though it may have the same bottom level.

Algorithm 3. CAFT vs Iso-Level CAFT (ILC)

```
1: initialization
2: while $U \neq \emptyset$ do
3:   $\mathcal{T} \leftarrow \emptyset$;
4:   $\mathcal{T} \leftarrow \mathcal{H}(\alpha)$; ILC: repeat B times ($^*$ CAFT: $|\mathcal{T}| = 1|$ ILC: $|\mathcal{T}| = B$ $^*$)
5:   for $1 \leqslant i \leqslant \varepsilon + 1$ do
6:     for $t \in \mathcal{T}$ do
7:       allocate task $t^{(i)}$ to processor with shortest finish time
8:     end for
9:   end for
10: end while
```

We point out that we face a difficult trade-off for choosing an appropriate value for B. On the one hand, if B is large, it will be possible to better balance the load and minimize communication costs. On the other hand, a small value of B will enable us to process the tasks on the critical path faster. In the experiments (see Section 6) we observe that choosing $B = m$, the number of processors, leads to good results.

5.3. Reducing communication overhead

When dealing with realistic model platforms, contention should be considered in order to obtain improved schedules. We account for communication overhead during the mapping process by removing some of the communications. To do so, we propose the following mapping scheme.

Let t be the current task to be scheduled. Consider a predecessor t_j of $t, j \in \Gamma^-(t)$, that has been replicated on $\varepsilon + 1$ distinct processors. We denote by $\mathcal{D}_u$ the set of replicas assigned to processor $\mathcal{P}_u$, and $\eta_u = |\mathcal{D}_u|$ its cardinality. The maximum cardinality is $\eta = \max_{1 \leqslant u \leqslant m} \eta_u$.

We would like to reduce the number of communications from all predecessors t_j to t when possible. The idea is to attempt to place each replica on the non-locked processor which currently contains the most predecessor replicas. To this purpose, we sort processors by non increasing order of number of replicas $\eta_u, 1 \leqslant u \leqslant m$, assigned to them. At each step in the mapping process, we try to take communications from replicas belonging to the non-locked processors, whenever possible. If not, we insert ε additional communications.

Fig. 2 illustrates this procedure. We set $\varepsilon = 2$ in this example. At step (0), no processor is locked. The three predecessors of the current task t, namely t_1, t_2 and t_3, are assigned. At step (1), we place the first replica $t^{(1)}$ on P_1, which becomes locked. This is represented in the figure with a superscript *, and the processor is also hatched in the figure. No communication is

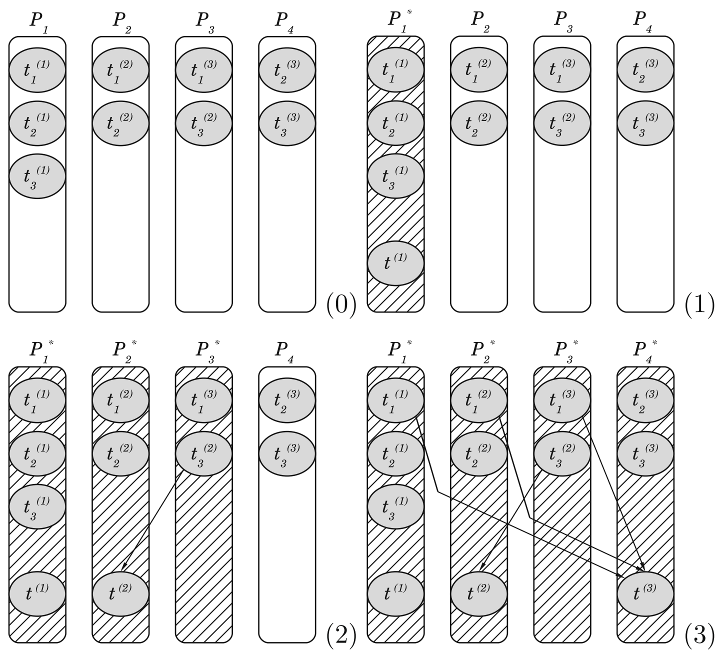


Fig. 2. CAFT scheduling steps.

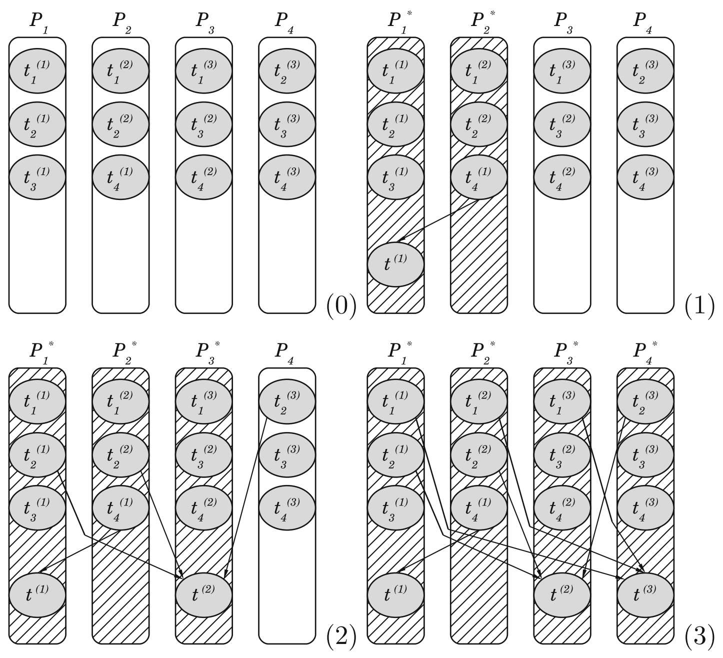


Fig. 3. CAFT scheduling steps.

added in this case. At step (2), we need to add a communication from P_3 to P_2, and thus we have three locked processors. At step (3), we place replica $t^{(3)}$ on the only non-locked processor which is P_4, and we need to add extra communication since all processors are locked.

It may happen that at some step in the scheduling process, if we lock a processor, then there is no processor left for the mapping of the remaining replicas. In such a case, we add ε additional communications and release the processors involved

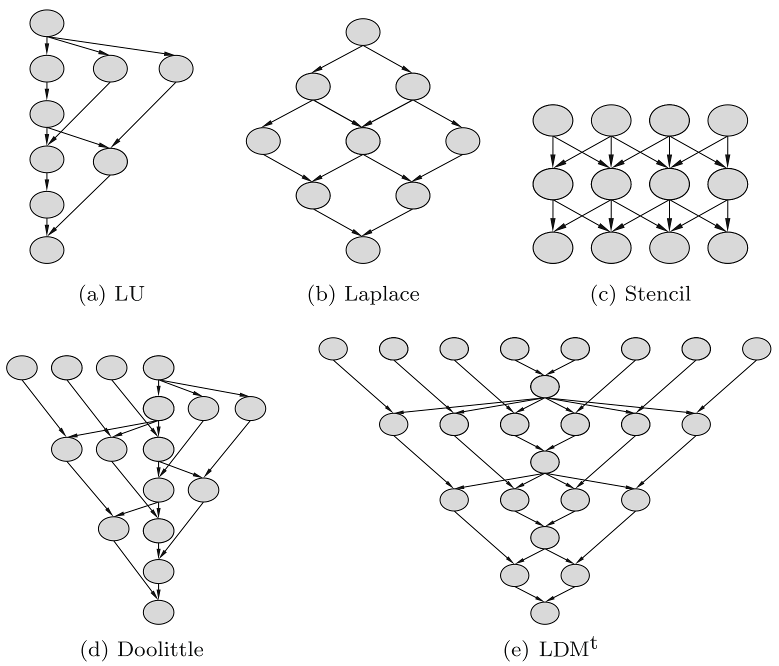


Fig. 4. Classical kernels of parallel algorithms.

in a communication with a replica of t. This special case is illustrated in Fig. 3, where current task t has four predecessors. At step (2), instead of locking P_4, we add communications from all replicas of task t_2 to $t^{(2)}$. Thus it is possible to place at step (3) replica $t^{(3)}$ on P_4.

The scheduling scheme adopted here to reduce communication overhead is similar to the CAFT pseudo code algorithm. The only difference lies in the following pseudo code which can substitute Lines 16–20 in Algorithm 1 (note that it does not change the time complexity):

```
1: Sort the remaining non-locked processors by non increasing order of number of replicas $\eta_u, 1 \leqslant u \leqslant m$, assigned to them;
2: **while** $\theta < \varepsilon + 1$ **do**
3:    for $1 \leqslant k \leqslant m$ and $P_k \notin \mathbb{P}$, try to take communications from replicas belonging to the non-locked processors, whenever possible; If not, we insert $\varepsilon$ additional communications;
4:    Compute $\mathscr{F}(t, P_k)$ using Eq. (6);
5:    Keep the (task, processor) pair that allows the minimum finish time of $t$;
6:    Update the set of locked processors $\mathbb{P}$ accordingly;
7:    $\theta = \theta + 1$
8: **end while**
```

In the following, we give an analytical expression of the actual number of communications induced by the *CAFT* algorithm. First we give an interesting upper bound for special graphs, and then we derive an upper bound for the general case.

5.3.1. *Special graphs*

First, we bound the number of communications induced by CAFT for special graphs like classical kernels representing various types of parallel algorithms [6]. The selected task graphs are:

(a) LU: LU decomposition.
(b) LAPLACE: Laplace equation solver.
(c) STENCIL: stencil algorithm.
(d) DOOLITTLE: Doolittle reduction.
(e) LDM^t: LDM^t decomposition.

Miniature versions of each task graph are given in Fig. 4.

Proposition 3. *The number of messages generated by CAFT for the above special graphs is at most*

$$V_2(\varepsilon+1)+V_3\left(\varepsilon\left\lceil\frac{(\varepsilon+2)}{2}\right\rceil+2\right),$$

where $V_2 \leqslant \lfloor\frac{\varepsilon}{2}\rfloor$ is the number of nodes of in-degree 2 and $V_3 \leqslant \lfloor\frac{\varepsilon}{3}\rfloor$ is the number of nodes of in-degree 3 in the graph.

Proof. One feature of the special graphs is that the in-degree of every task is at most 3. At each step when scheduling current task t, we have three cases to consider, depending upon its in-degree (the cardinal of $\Gamma^-(t)$). Recall that processors are ordered by non increasing η_u values, where η_u is the number of replicas already assigned to P_u, hence which do not need to be communicated again.

(1) $|\Gamma^-(t)|=1$. In this case, in order to pay no communication, we just need to place each replica of t with a replica of its predecessor.
(2) $|\Gamma^-(t)|=2$. The two predecessor tasks of t are denoted t_1 and t_2. If replicas of t_1 and t_2 are mapped on the same processor ($\mathscr{P}(t_1^{(z)})=\mathscr{P}(t_2^{(z')})=\mathscr{P}$ for some $1\leqslant z,z'\leqslant\varepsilon+1$), then there is no need for any additional communication. Other replicas of t_1 and t_2 which does not satisfy the previous property are thus mapped onto singleton processors. We perform the *one-to-one mapping* algorithm to allocate the corresponding other replicas of t. For each replica, at most one communication is added.
(3) $|\Gamma^-(t)=3$. Here we consider the number of replicas allocated to processor $\mathscr{P}_u$, denoted as η_u.
- We place a replica on each processor with $\eta_u=3$, thus no communication need to be paid for.
- Consider a processor with $\eta_u=2$. When allocating a replica of t on such a processor $\mathscr{P}_u$, we need to receive data from the third predecessor allocated to $\mathscr{P}_v\neq\mathscr{P}_u$. $\mathscr{P}_v$ may be either a singleton processor ($\eta_v=1$) or it may handle two predecessors ($\eta_v=2$).
  - If $\eta_v=1$, then we need only one communication for mapping the replica of t. In this case $\mathscr{P}_v$ communicates only to $\mathscr{P}_u$.
  - If $\eta_v=2$, then we may need to add extra communications. For the first $\lceil\frac{\varepsilon+1}{2}\rceil$ replicas of t, we add only one communication per replica, and lock processors accordingly. But for the remaining set $\lfloor\frac{\varepsilon+1}{2}\rfloor$ of replicas, we will have to generate $\varepsilon+1$ communications for each of these replicas. Overall, the number of communications is at most

$$\left\lceil\frac{\varepsilon+1}{2}\right\rceil+(\varepsilon+1)\left\lfloor\frac{\varepsilon+1}{2}\right\rfloor.$$

Let $X=\lceil\frac{\varepsilon+1}{2}\rceil+(\varepsilon+1)\lfloor\frac{\varepsilon+1}{2}\rfloor$. Let $Y=\varepsilon\left\lceil\frac{(\varepsilon+2)}{2}\right\rceil+1$. If $\varepsilon=2k$ is even, then $X=2k^2+k+1\leqslant 2k^2+2k+1=Y$. If $\varepsilon=2k+1$ is odd, then $X=2k^2+2k+1\leqslant 2k^2+3k+1=Y$. In all cases $X\leqslant Y$, hence the number of communications is at most Y.
- Now, all remaining processors have at most one replica ($\eta=1$). Thus task t needs its data from two other replicas. So we have to take at most two communications for each replicas mapped. Thus for the mapping of $\varepsilon+1$ replicas, we will have at most a number of communications equal to $2(\varepsilon+1)$. Note that $2(\varepsilon+1)\leqslant Y+1=\varepsilon\left\lceil\frac{(\varepsilon+2)}{2}\right\rceil+2$ for all ε, hence the result. □

5.3.2. General graphs

Proposition 4. *For general graphs, the number of messages generated by CAFT is at most*

$$e\left(\varepsilon\left\lceil\frac{(\varepsilon+2)}{2}\right\rceil+1\right).$$

Proof. At each step when scheduling current task t:

(i) For the first $\lceil\frac{\varepsilon+1}{2}\rceil$ replicas, we generate at most $\sum_{u=1}^{\lceil\frac{(\varepsilon+1)}{2}\rceil}(|\Gamma^-(t)|-\eta_u)$ communications (recall that η_u is the number of replicas already assigned to P_u, hence which do not need to be communicated again). Altogether, we have at most $\left\lceil\frac{(\varepsilon+1)}{2}\right\rceil|\Gamma^-(t)|$ communications for these replicas.
(ii) We still have to map the remaining $\lfloor\frac{\varepsilon+1}{2}\rfloor$ of t replicas. In the worst case, each replica placed will generate $\varepsilon+1$ communications (this is because processors may be locked in this case). Thus for this remaining set of replicas, the number of communications is at most

$$(\varepsilon+1)\sum_{u=\lceil\frac{(\varepsilon+1)}{2}\rceil+1}^{\varepsilon+1}(|\Gamma^-(t)|-\eta_u)\leqslant(\varepsilon+1)\left\lfloor\frac{\varepsilon+1}{2}\right\rfloor|\Gamma^-(t)|.$$

From (i) and (ii), we have a total number of communications of $|\Gamma^-(t)|X$, where $X = \lceil \frac{\varepsilon+1}{2} \rceil + (\varepsilon+1)\lfloor \frac{\varepsilon+1}{2} \rfloor$. As in the proof of Proposition 3, we know that $X \leqslant Y$, where $Y = \varepsilon \left\lceil \frac{(\varepsilon+2)}{2} \right\rceil + 1$. Hence the number of communications is at most Y.Thus, summing up for all the v tasks in G, the total number of messages is at most

$$\sum_{u=1}^{v} |\Gamma^-(t)| \left(\varepsilon \left\lceil \frac{(\varepsilon+2)}{2} \right\rceil + 1 \right) = e \left(\varepsilon \left\lceil \frac{(\varepsilon+2)}{2} \right\rceil + 1 \right). \qquad \square$$

The following proposition deals with disjoint and complementary replica sets. In fact, the number of communications can be drastically reduced in such a case.

Proposition 5. *For general graphs, if at each step when scheduling a task t, we can determine replica sets $\mathcal{D}_u$ that are both disjoint ($\mathcal{D}_u \cap \mathcal{D}_{u'} = \emptyset$ if $u \neq u'$) and complementary ($\sigma_{u=1}^{m} |\mathcal{D}_u| = |\Gamma^-(t)|$, or in other words $\cup_{1 \leqslant u \leqslant m} \mathcal{D}_u$ contains a replica of each predecessor of t), then the number of messages is at most e(ε + 1).*

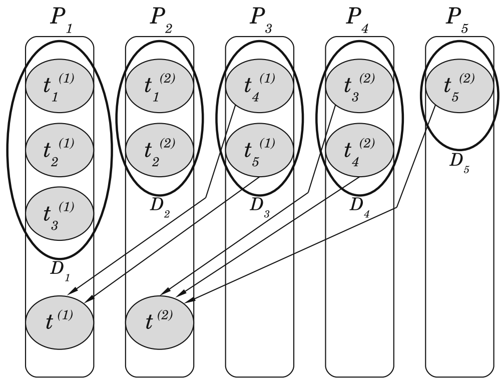


Fig. 5. Complementary/disjoint sets of replicas.

Notation	Meaning
General	
ε	Number of supported failures
Application graph	
V	Set of nodes
$v = \|V\|$	Number of nodes/tasks
E	Set of edges
$e = \|E\|$	Number of edges
$\Gamma^-(t)$	Immediate predecessors of task $t \in V$
$\Gamma^+(t)$	Immediate successors of task $t \in V$
$\mathcal{V}(t_i, t_j)$	Volume of data sent from t_i to t_j
Computing platform	
m	Number of processors
l_{kh}	Link between processors P_k and P_h
Schedule costs	
$\mathcal{E}(t_i, P_k)$	Execution time of t_i on P_k
c_{ij}	Communication between tasks t_i and t_j
$W(c_{ij}, l_{kh})$	Communication cost of c_{ij} when link l_{kh} is used

Fig. 6. Glossary of notations.

Proof. We map a replica on $\mathscr{D}_u$ and add communications from all complementary sets, which generates at most $|\Gamma^-(t)| - |\mathscr{D}_u| = |\cup_{1 \leqslant u' \leqslant m, u' \neq u} \mathscr{D}_{u'}| \leqslant |\Gamma^-(t)|$.

Thus, for the mapping of $\varepsilon + 1$ replicas, and summing up for the set V of tasks in G, the total number of messages is at most $\sum_{t \in V} |\Gamma^-(t)|(\varepsilon + 1) = e(\varepsilon + 1)$. $\square$

This proposition is illustrated on Fig. 5. For the mapping of the first replica $t^{(1)}$, we have $|\Gamma^-(t)| - |\mathscr{D}_1| = 5 - 3 = 2 = |\mathscr{D}_3|$. In addition, both $\mathscr{D}_1$ and $\mathscr{D}_3$ are mutually complementary/disjoints and they form a complete instance of all predecessors. Also, for the mapping of the second replica $t^{(2)}$, we have $|\Gamma^-(t)| - |\mathscr{D}_2| = 5 - 2 = 3 = |\mathscr{D}_4 \cup \mathscr{D}_5|$. Similarly, the condition of complementarity/disjunction of the sets $\mathscr{D}_2$, $\mathscr{D}_4$ and $\mathscr{D}_5$ holds.

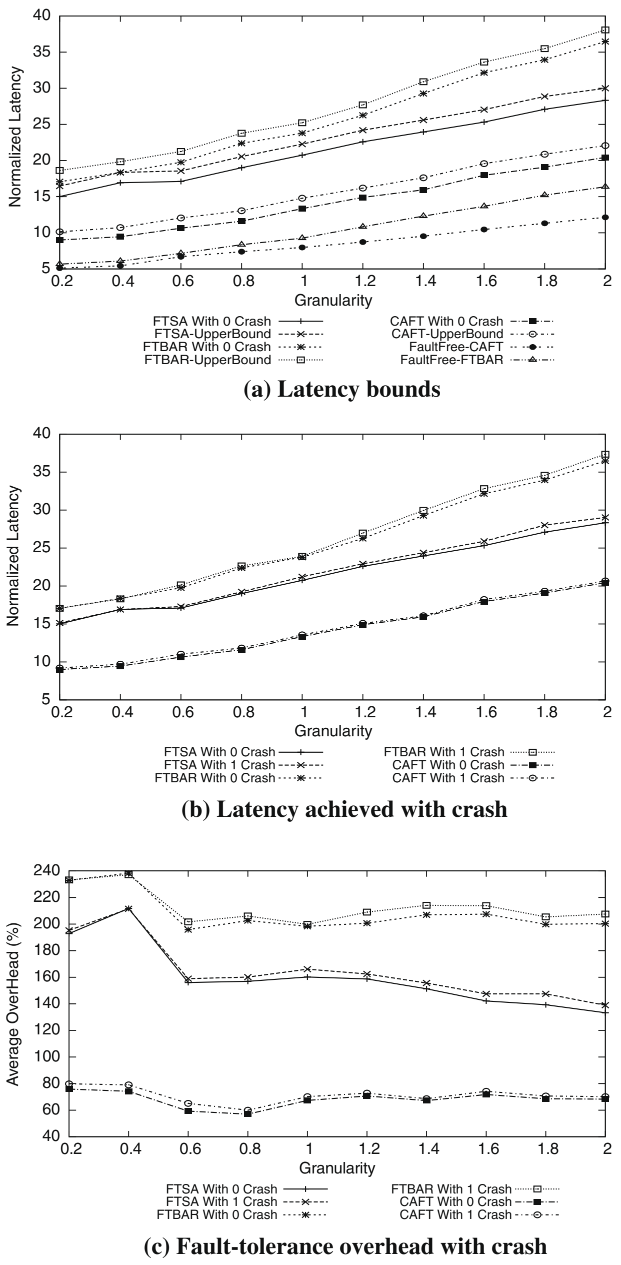


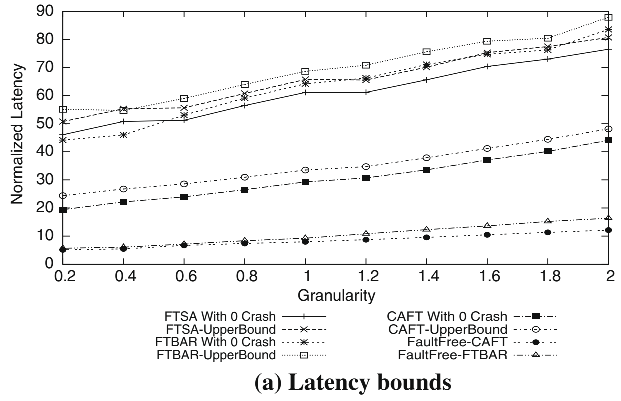
Fig. 7. Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR (bound and crash cases, $\varepsilon = 1$, $m = 10$).

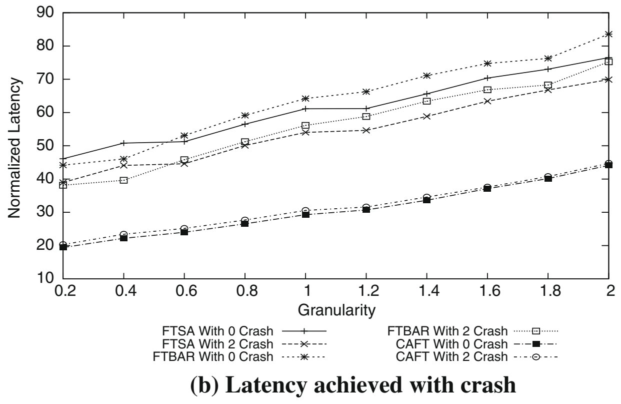
6. Experimental results

We assess the practical significance and usefulness of the Iso-Level CAFT and CAFT algorithms through simulation studies. First, we compare the performance of CAFT with the most two relevant fault-tolerant scheduling algorithms, namely FTSA and FTBAR. Then we compare CAFT against Iso-Level CAFT.

We use randomly generated graphs, whose parameters are consistent with those used in the literature [17,29]. We characterize these random graphs with three parameters: (i) the number of tasks, chosen uniformly from the range [80,120]; (ii) the number of incoming/outgoing edges per task, which is set in [1,3]; and (iii) the granularity of the task graph $g(G)$.

For a given graph G and processor set $\mathscr{P}$, the granularity $g(G,\mathscr{P})$ is the ratio of the average computation time of the tasks (on the slowest processor) to that of communication time (on the slowest link). If $g(G,\mathscr{P}) \geqslant 1$, the task graph is said to be *coarse grain*, otherwise it is *fine grain*. For *coarse grain* DAGs, each task receives or sends a small amount of communication compared to the computation load of its adjacent tasks.





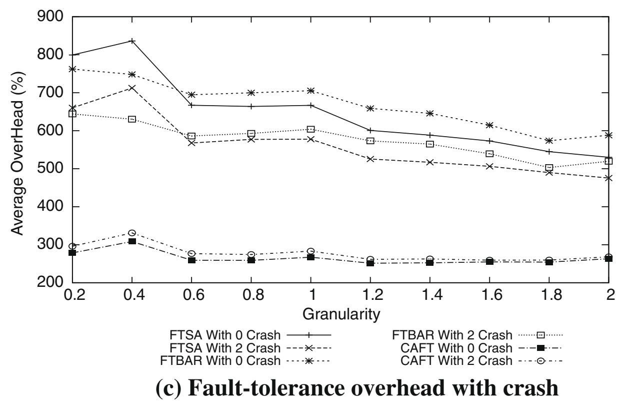
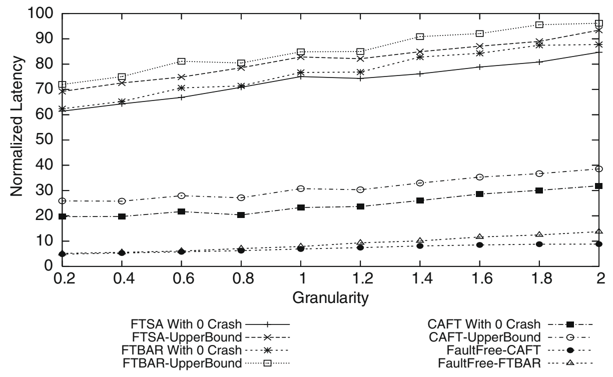


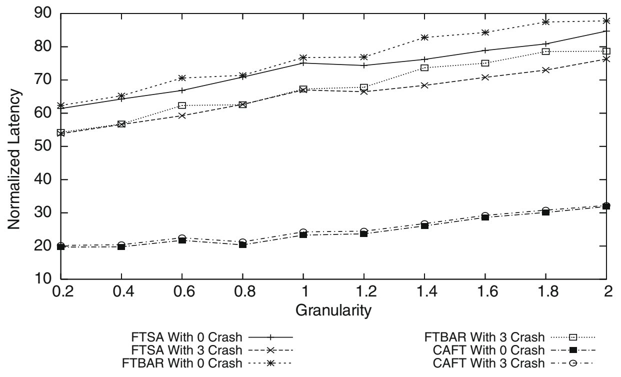
Fig. 8. Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR (bound and crash cases, $\varepsilon = 3$, $m = 10$).

We consider two types of graphs, with a granularity (A) in [0.2,2.0] and increments of 0.2, and (B) in [1,10] and increments of 1. Two types of platforms are considered, first with 10 processors and $\varepsilon = 1$ or $\varepsilon = 3$, and then with 20 processors and $\varepsilon = 5$. To account for communication heterogeneity in the system, the unit message delay of the links and the message volume between two tasks are chosen uniformly from the ranges [0.5,1] and [50,150], respectively. Each point in the figures represents the mean of executions on 60 random graphs. The metrics which characterize the performance of the algorithms are the latency and the overhead due to the active replication scheme. The fault free schedule is defined as the schedule generated by each algorithm without replication, assuming that the system is completely safe (i.e., setting $\varepsilon = 0$). For each algorithm, we compare the fault free version (without replication) and the fault-tolerant algorithm. Note that the fault free version of CAFT reduces to an implementation of HEFT, the reference heuristic in the literature [36]. Also recall that the upper bounds of the schedules are computed as explained in Section 4.2 or [7].

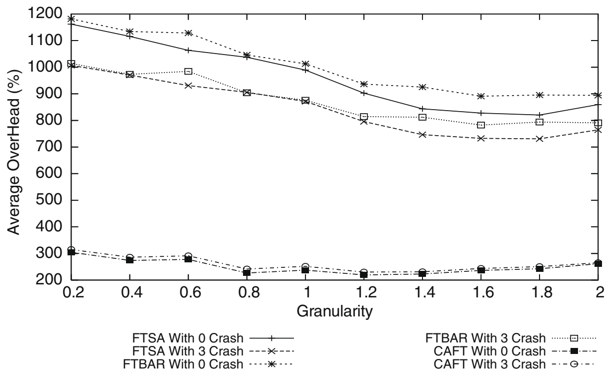
Each algorithm is evaluated in terms of achieved latency and fault-tolerance overhead. We run algorithms CAFT^0, Iso-Level CAFT^0, FTSA^0, FTBAR^0 where the superscript 0 means that the resulting latency is the one achieved during an execution



(a) Latency bounds



(b) Latency achieved with crash



(c) Fault-tolerance overhead with crash

Fig. 9. Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR (bound and crash cases, $\varepsilon = 5$, $m = 20$).

with no failure (but with a schedule which tolerates up to ε failures). We also measure the latency when c failures occur, thus obtaining CAFT^c, FTSA^c, and FTBAR^c. The overhead of each algorithm is then computed as follows:

$$\text{Overhead}_{algo} = \frac{L_{algo} - \text{CAFT}^*}{\text{CAFT}^*},$$

where L_{algo} is the latency achieved by the algorithm, and the superscript * denotes the latency achieved by the fault free schedule.

Note that for each algorithm, if a replica of task t and a replica t_*^z of its predecessor t_* are mapped on the same processor $\mathscr{P}$, then there is no need for other copies of t_* to send data to processor $\mathscr{P}$. Indeed, if $\mathscr{P}$ is operational, then the copy of t on $\mathscr{P}$ will receive the data from t_*^z (intra-processor communication). Otherwise, $\mathscr{P}$ is down and does not need to receive anything.

Figs. 7a, 8a and 9a clearly show that CAFT outperforms both FTSA and FTBAR. These results indicate that network contention has a significant impact on the latency achieved by FTSA and FTBAR. This is because allocating many copies of each

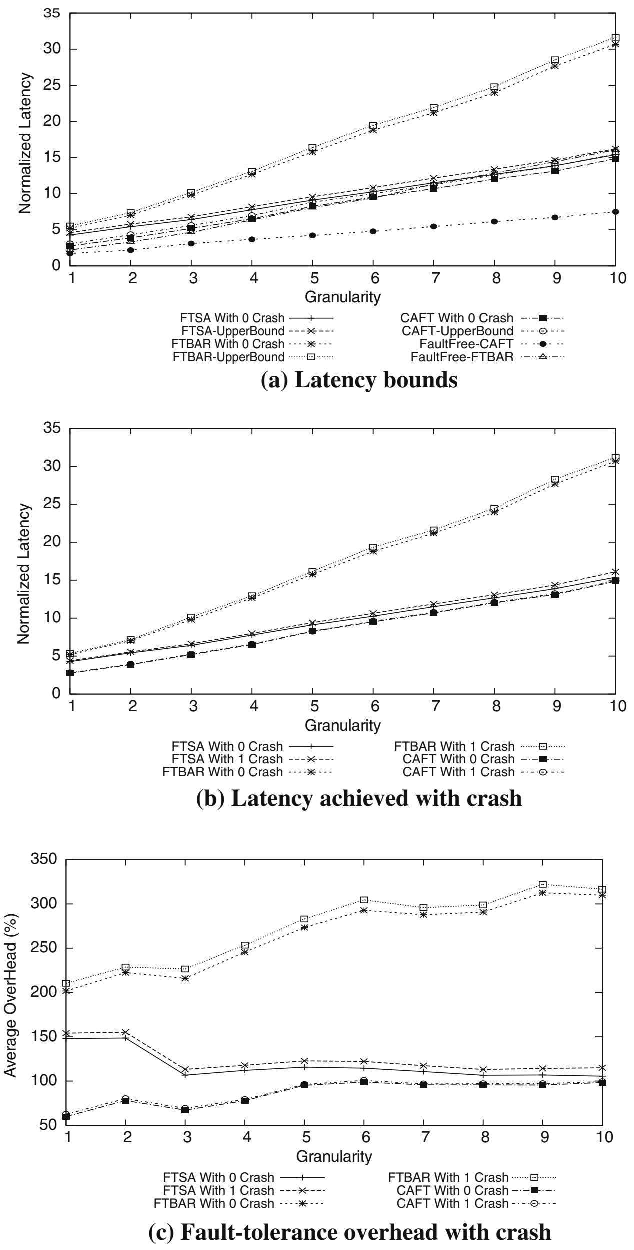


Fig. 10. Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR for coarse grain graphs $g(G) \geqslant 1$ (bound and crash cases, $\varepsilon = 1$, $m = 10$).

task will severely increase the total number of communications required by the algorithm: we move from e communications (one per edge) in a mapping with no replication, to $e(\varepsilon+1)^2$ in FTSA and FTBAR, a quadratic increase. In contrast, the CAFT algorithm is not really sensitive to contention since it uses the one-to-one procedure to reduce this overhead down to, in the most favorable cases, a linear number $e(\varepsilon+1)$ of communications. In addition, we find that CAFT achieves a really good latency (with 0 crash), which is quite close to the fault free version. As expected, its upper bound is close to the latency with 0 crash since we keep only the best communication edges in the schedule.

We have also compared the behavior of each algorithm when processors crash down by computing the real execution time for a given schedule rather than just bounds (upper bound and latency with 0 crash). Processors that fail during the schedule process are chosen uniformly from the range [1,10]. The first observation from Figs. 7b and 8b is that even when crash occurs, CAFT^c behaves always better than FTSA^c and FTBAR^c. This is because CAFT accounts for communication overhead during the mapping process by removing some of the communications. The second interesting observation is that the latency achieved by both FTSA and FTBAR compared to the schedule length generated with 0 crash sometimes increases (see

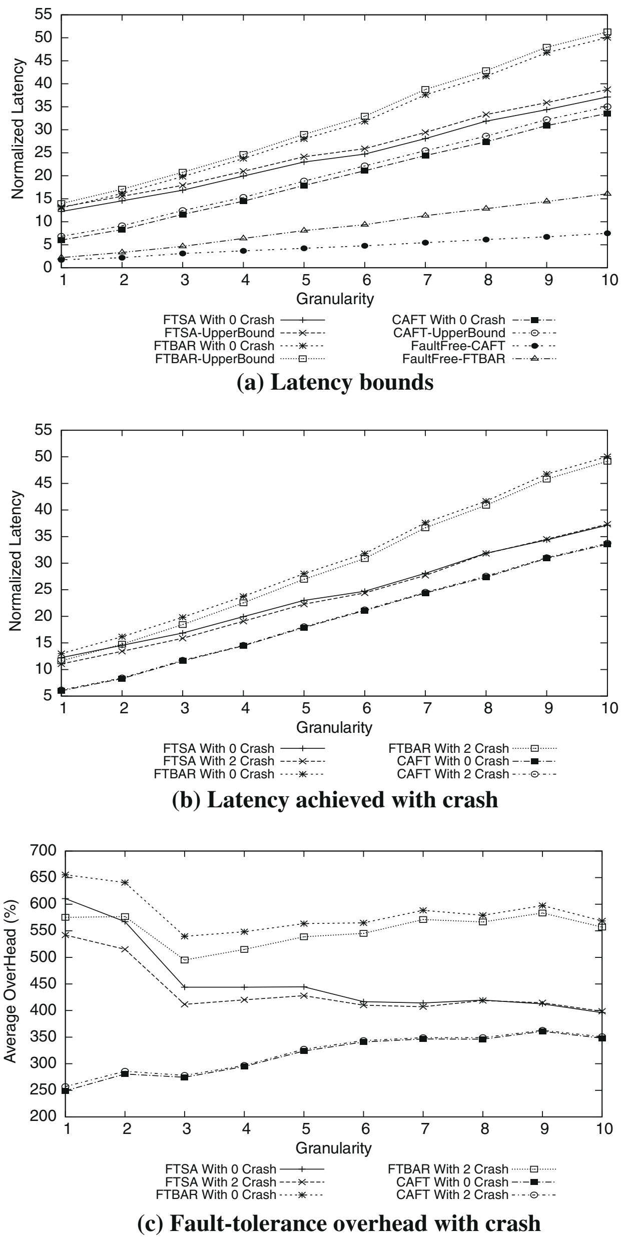


Fig. 11. Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR for coarse grain graphs $g(G) \geqslant 1$ (bound and crash cases, $\varepsilon = 3$, $m = 10$).

Fig. 7b) and other times decreases (Fig. 8b). To explain this phenomenon, consider the example of a simple task graph composed of three tasks in precedence $(t_1 \curlywedge t_3) \wedge (t_2 \curlywedge t_3)$ and $P = \{P_m, 1 \leq m \leq 6\}$. Assume that $\varepsilon = 1$, $\mathscr{B}(t_1) = \{t_1^{(1)}, t_1^{(2)}\}$, $\mathscr{B}(t_2) = \{t_2^{(1)}, t_2^{(2)}\}$ and $\mathscr{B}(t_3) = \{t_3^{(1)}, t_3^{(2)}\}$ with $\mathscr{P}(\mathscr{B}(t_1)) = \{P_1, P_2\}$, $\mathscr{P}(\mathscr{B}(t_2)) = \{P_3, P_4\}$ and $\mathscr{P}(\mathscr{B}(t_3)) = \{P_5, P_6\}$ respectively. Assume that the latency achieved with 0 crash is determined by the replica of $t_3^{(1)}$ $(t_3^{(1)} \leftarrow \min_{1 \leq k \leq 3}\{\mathscr{F}(t_3^k, P(t_3^k))\})$.

For the sake of simplicity, assume that the sorted list of the instances/replicas of both $\mathscr{B}(t_1)$ and $\mathscr{B}(t_2)$ (sorting is done by non-decreasing order of their communication finish time $\mathscr{F}(c, l)$ on the links) are in this order $\{t_1^{(1)}, t_1^{(2)}, t_2^{(1)}, t_2^{(2)}\}$. $t_3^{(1)}$ will receive its input data 4 times. But as soon as it receives its input data from $t_2^{(1)}$, the task is executed and ignores the later incoming data from $t_2^{(2)}$. So, without failures and since communications are serialized at the reception, three communications are taken into account so that $t_3^{(1)}$ can run earlier. But, in the presence of one failure, two scenarios are possible: (i) if P_2 fails, the finish time of the replica $t_3^{(1)}$ will be sooner than its estimated finish time; (ii) if P_2 and P_3 fail, the start time of the replica is

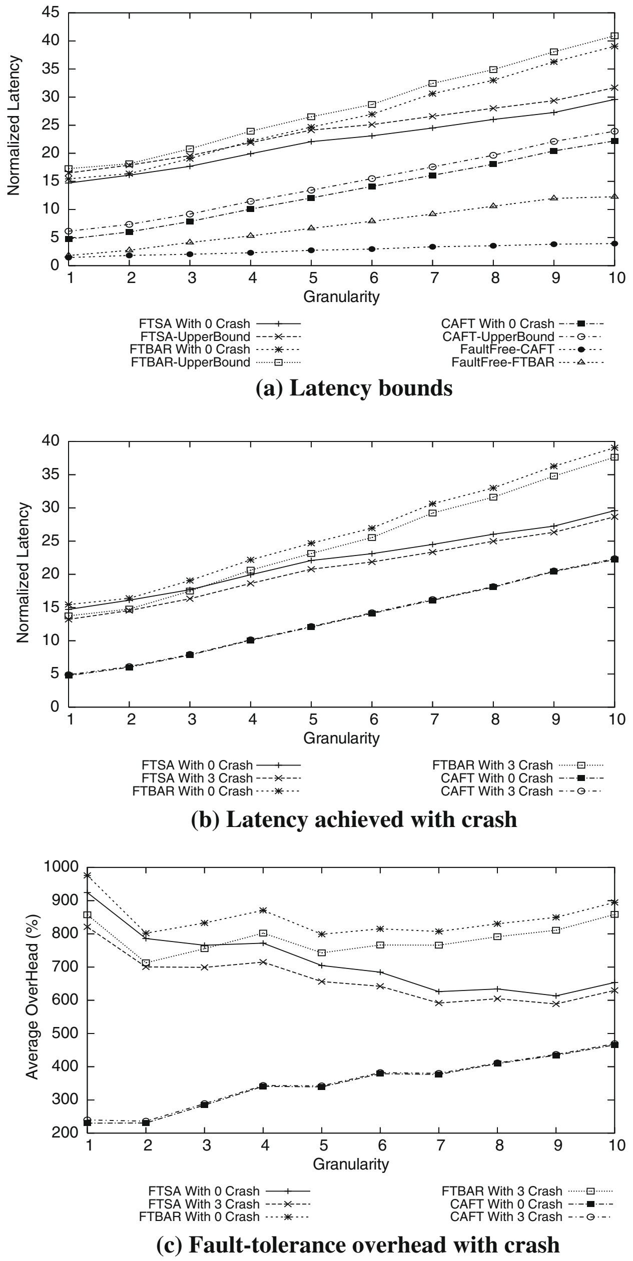
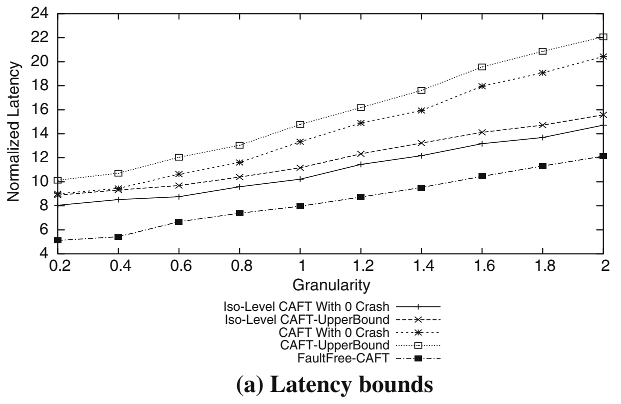


Fig. 12. Average normalized latency and overhead comparison between CAFT, FTSA and FTBAR for coarse grain graphs $g(G) \geqslant 1$ (bound and crash cases, $\varepsilon = 5$, $m = 20$).

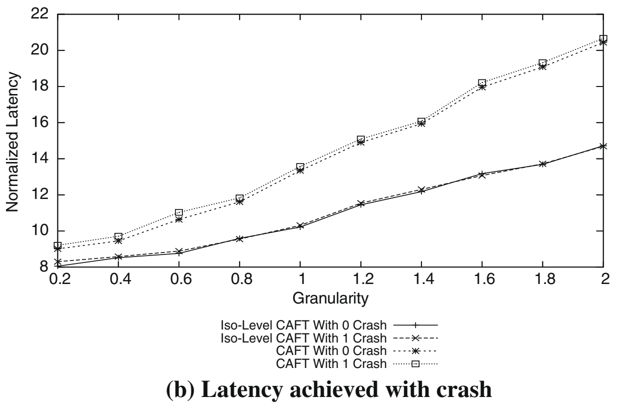
delayed until the arrival of its input data from $t_2^{(2)}$. This second case leads to an increase of its finish time and consequently to an increase of the latency achieved with crash.

Applying this reasoning to all tasks of G, the impact of processors crash are spread throughout the execution of the application, which may lead either to a reduction or to an increase of the schedule length. This behavior is identical when we consider larger platforms, as illustrated in Fig. 9.

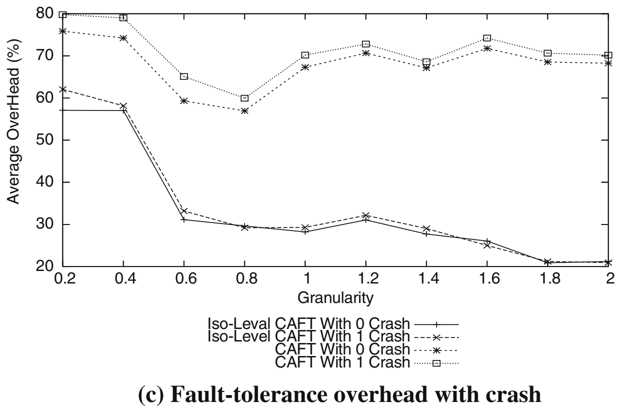
We also evaluated the impact of the granularity on the performance of each algorithm. Thus, Figs. 10–12 reveal that when the $g(G)$ value is small, the latency of CAFT is significantly better than that of FTSA and FTBAR. This is explained by the fact that for small $g(G)$ values, i.e., high communication costs, contention plays quite a significant role. However, the impact of contention becomes less important as the granularity $g(G)$ increases, since larger $g(G)$ values result in smaller communication times. Consequently, the fault tolerance overhead of FTSA diminishes gradually and becomes closer to that of CAFT as the $g(G)$ value goes up. However, the fault tolerance overhead of FTBAR increases with the increasing values of the granularity. The reason of the poorer performance of FTBAR can be explained by the inconvenience of the schedule pressure



(a) Latency bounds



(b) Latency achieved with crash



(c) Fault-tolerance overhead with crash

Fig. 13. Average normalized latency and overhead comparison between Iso-Level CAFT and CAFT (bound and crash cases, $\varepsilon = 1$, $m = 10$).

function adopted for the processor selection. Processors are selected in such a way that the schedule pressure value is minimized. Doing so, tasks are not really mapped on those processors which would allow them to finish earlier.

Comparing the results of Iso-Level CAFT to the results of CAFT, we observe in Figs. 13 and 14 that Iso-Level CAFT gives the best performance. It always improves the latency significantly in all figures. This is because the Iso-Level CAFT algorithm tries incrementally to ensure a certain degree of load balancing for processors by scheduling a chunk of ready tasks before considering their corresponding replicas. This better load balancing also decreases communications between tasks. Consequently, this leads to minimize the final latency of the schedule. Similarly, Iso-Level CAFT achieves much better results than CAFT when considering larger platforms, as shown in Fig. 15.

We also find in Figs. 16–18 that the performance difference between CAFT and Iso-Level CAFT increases when the granularity increases. This interesting result comes from the fact that larger granularity indicates that we are dealing with intensive computations applications in heterogeneous platforms. Thus, in order to reduce the latency for such applications, it is

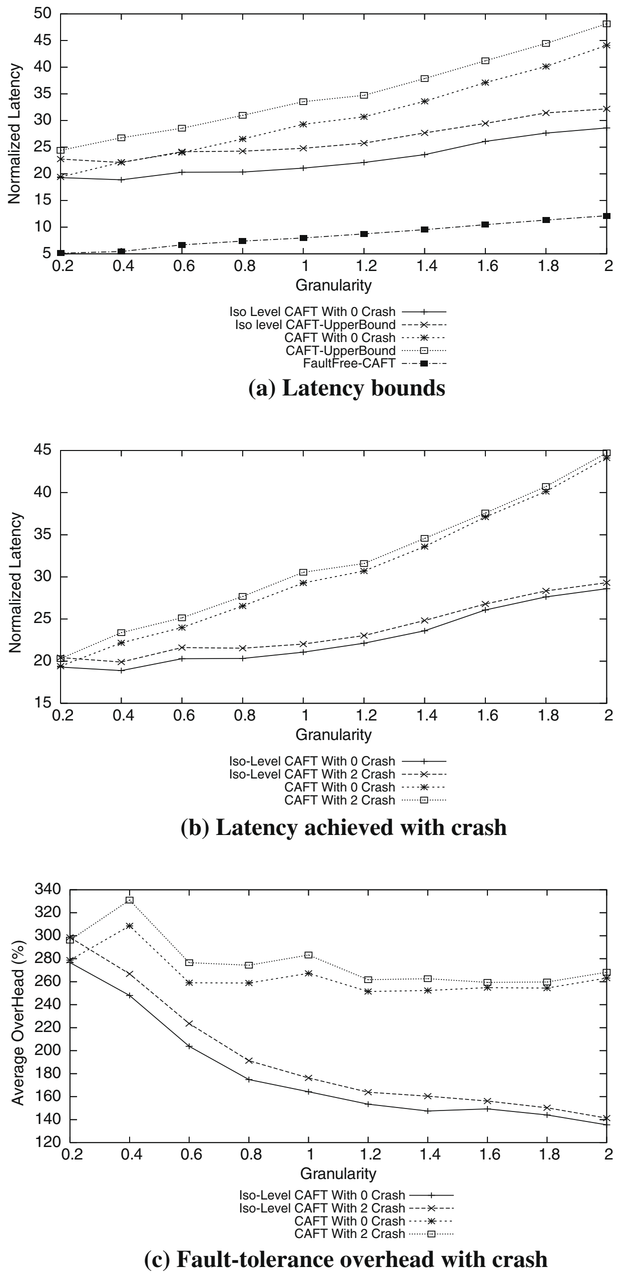


Fig. 14. Average normalized latency and overhead comparison between Iso-Level CAFT and CAFT (bound and crash cases, $\varepsilon = 3$, $m = 10$).

important to better parallelize the application. That is why we changed the backbone of CAFT to perfectly balance the load of processors at each step of the scheduling process.

Finally, we readily observe from all figures that we deal with two conflicting objectives. Indeed, the fault-tolerance overhead increases together with the number of supported failures. We also see that latency increases together with granularity, as expected. In addition, it is interesting to note that when the number of failures increases, there is not really much difference in the increase of the latency achieved by CAFT and Iso-Level CAFT, compared to the schedule length generated with 0 crash. This is explained by the fact that the increase in the schedule length is already absorbed by the replication done previously, in order to resist to eventual failures.

To summarize, the simulation results show that both CAFT and Iso-Level CAFT are considerably superior to the other two algorithms in all the cases tested ($0.2 \leqslant g(G) \leqslant 10, m = \{10, 20\}$). They also indicate that network contention has a significant impact on the latency achieved by FTSA and FTBAR. Thus, this experimental study validates the usefulness of our algorithms CAFT and Iso-Level CAFT, and confirms that when dealing with realistic model platforms, contention should

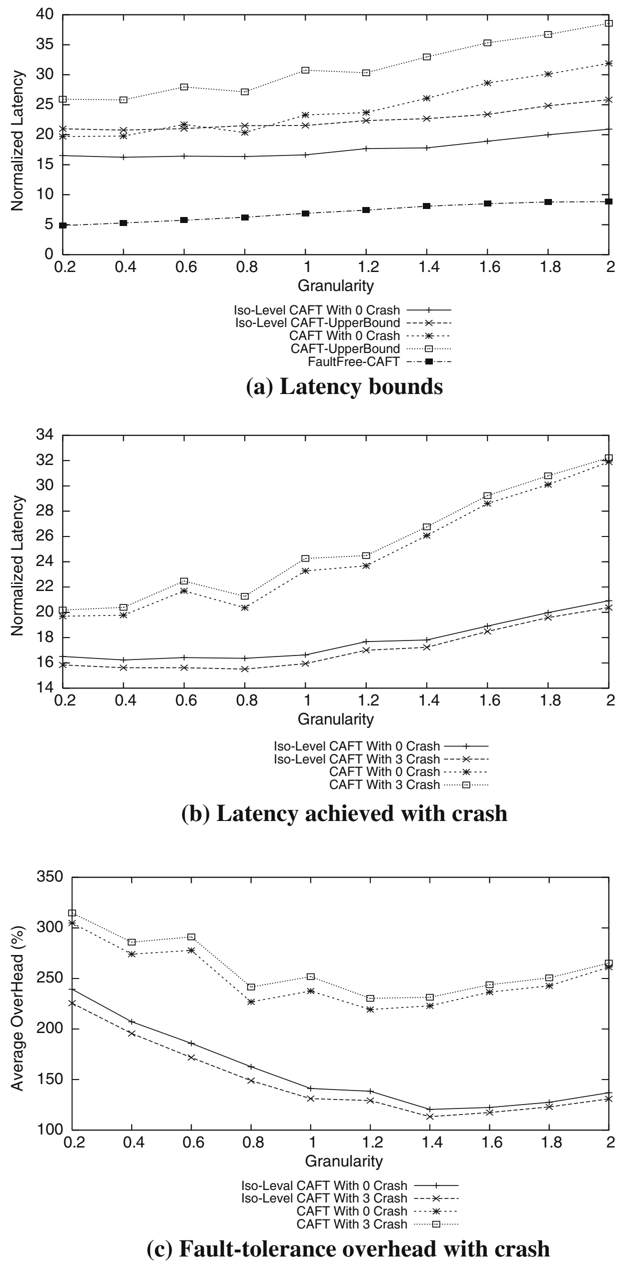
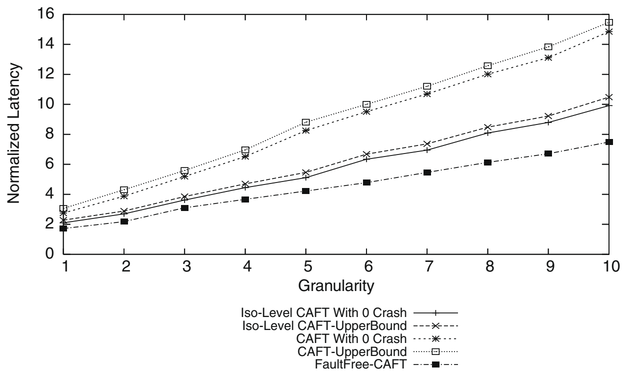
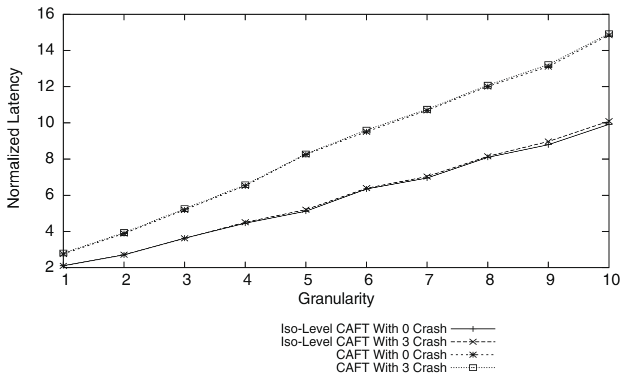


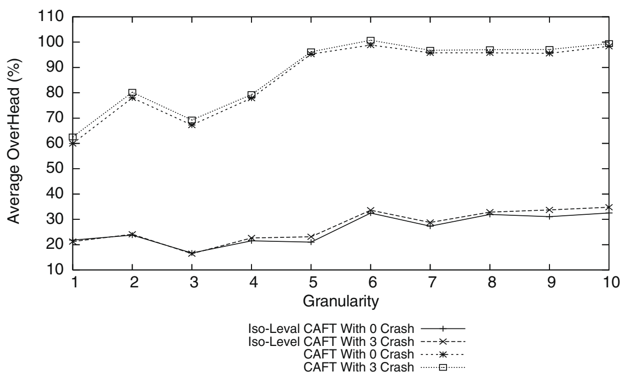
Fig. 15. Average normalized latency and overhead comparison between Iso-Level CAFT and CAFT (bound and crash cases, $\varepsilon = 5$, $m = 20$).



(a) Latency bounds



(b) Latency achieved with crash



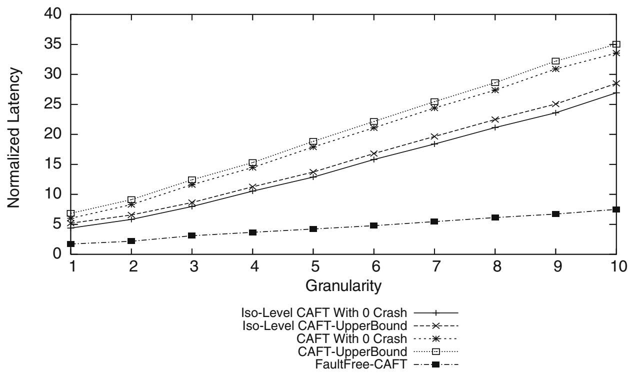
(c) Fault-tolerance overhead with crash

Fig. 16. Average normalized latency and overhead comparison between Iso-Level CAFT and CAFT for coarse grain graphs $g(G) \geqslant 1$ (bound and crash cases, $\varepsilon = 1$, $m = 10$).

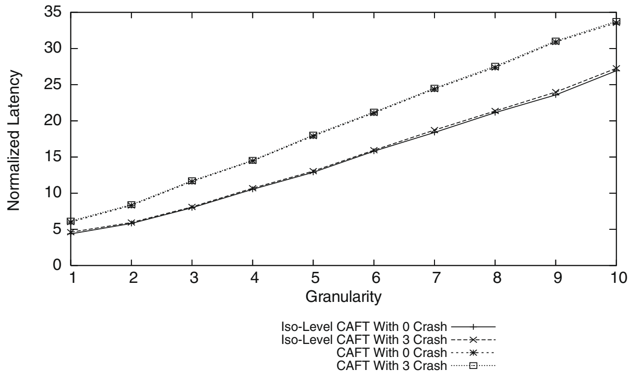
absolutely be considered in order to obtain improved schedules. To the best of our knowledge, the proposed algorithms are the first to address both problems of network contention and fault-tolerance scheduling.

7. Conclusion

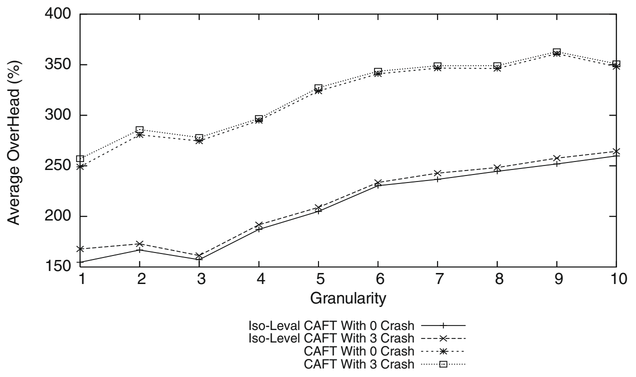
In this paper we have presented two efficient fault-tolerant algorithms for heterogeneous systems, namely CAFT and Iso-Level CAFT. CAFT is based on an active replication scheme, and is able to drastically reduce the communication overhead induced by task replication, which turns out a key factor in improving performance when dealing with realistic, communication contention-aware, platform models. To assess the performance of CAFT, simulation studies were conducted to compare it with (the one-port adaptation of) FTBAR and FTSA, which seem to be its main direct competitors from the literature. We have shown that CAFT is very efficient both in terms of computational complexity and quality of the resulting schedule.



(a) Latency bounds



(b) Latency achieved with crash



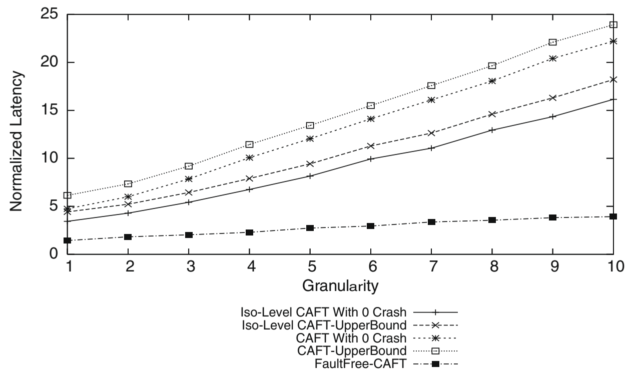
(c) Fault-tolerance overhead with crash

Fig. 17. Average normalized latency and overhead comparison between Iso-Level CAFT and CAFT for coarse grain graphs $g(G) \geqslant 1$ (bound and crash cases, $\varepsilon = 3$, $m = 10$).

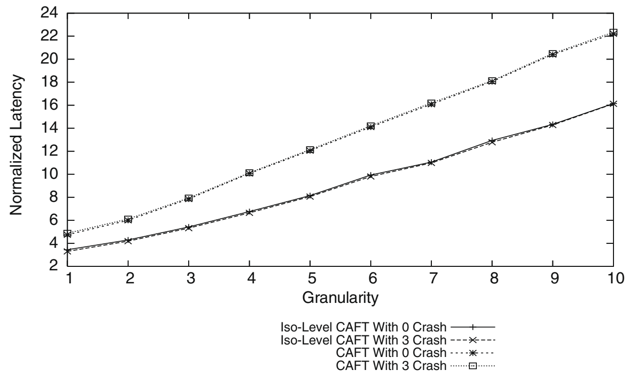
The design of Iso-Level CAFT is motivated by (i) the search for a better load balance and (ii) the generation of fewer communications. These goals are achieved by scheduling a chunk of ready tasks simultaneously, which enables for a global view of the potential communications. The experimental results fully demonstrate the usefulness of Iso-Level CAFT.

An extension of Iso-Level CAFT would be to adapt it to sparse interconnection graphs (while we had a clique in this paper). On such platforms, each processor is provided with a routing table which indicates the route to be used to communicate with another processor. To achieve contention awareness, at most one message can circulate on a given link at a given time-step, so we need to schedule long-distance communications carefully.

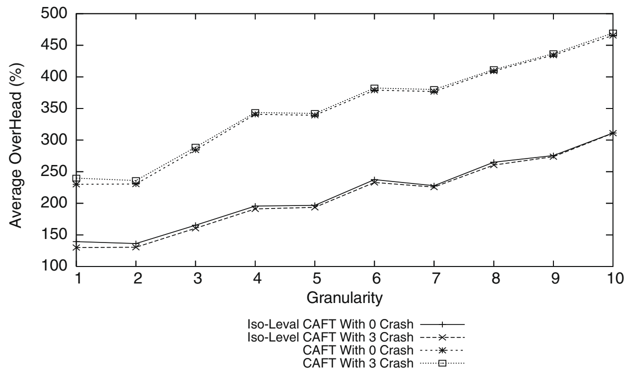
Finally, it would be very interesting to extend or modify the CAFT and Iso-Level CAFT algorithms to the context of pipelined workflows made up of collections of identical task graphs (rather than dealing with a single graph as in this paper). We would then need to solve a challenging tri-criteria optimization problem (latency, throughput and fault-tolerance).



(a) Latency bounds



(b) Latency achieved with crash



(c) Fault-tolerance overhead with crash

Fig. 18. Average normalized latency and overhead comparison between Iso-Level CAFT and CAFT for coarse grain graphs $g(G) \geqslant 1$ (bound and crash cases, $\varepsilon = 5$, $m = 20$).

Acknowledgement

We thank the reviewers for their numerous comments and suggestions, which greatly improved the final version of the paper.

References

[1] J. Abawajy, Fault-tolerant scheduling policy for grid computing systems, in: International Parallel and Distributed Processing Symposium IPDPS'2004, IEEE Computer Society Press, 2004.
[2] I. Ahmad, Y.-K. Kwok, On exploiting task duplication in parallel program scheduling, IEEE Transactions on Parallel and Distributed Systems (1998).
[3] R. Al-Omari, A.K. Somani, G. Manimaran, Efficient overloading techniques for primary-backup scheduling in real-time systems, Journal of Parallel and Distributed Computing 64 (5) (2004) 629–648.

[4] B. Awerbuch, Y. Azar, A. Fiat, F. Leighton. Making commitments in the face of uncertainty: how to pick a winner almost every time, in A. Press (Ed.), 28th ACM Symposium on Theory of Computing, 1996, pp. 519–530.
[5] M. Banikazemi, V. Moorthy, D.K. Panda, Efficient collective communication on heterogeneous networks of workstations, in: Proceedings of the 27th International Conference on Parallel Processing (ICPP'98), IEEE Computer Society Press, 1998.
[6] O. Beaumont, V. Boudet, Y. Robert, A realistic model and an efficient heuristic for scheduling with heterogeneous processors, in: Proceedings of the 11th Heterogeneous Computing Workshop, 2002.
[7] A. Benoit, M. Hakem, Y. Robert, Fault-tolerant scheduling of precedence task graphs on heterogeneous platforms, in: Proceedings of APDCM'08, the 10th Workshop on Advances in Parallel and Distributed Computational Models, Miami, USA, April 2008. Also available as Research Report RR2008-03, at graal.ens-lyon.fr/abenoit/.
[8] P. Bhat, C. Raghavendra, V. Prasanna, Efficient collective communication in distributed heterogeneous systems, in: ICDCS'99 19th International Conference on Distributed Computing Systems, IEEE Computer Society Press, 1999, pp. 15–24.
[9] P. Bhat, C. Raghavendra, V. Prasanna, Efficient collective communication in distributed heterogeneous systems, Journal of Parallel and Distributed Computing 63 (2003) 251–263.
[10] S. Bhatt, F. Chung, F. Leighton, A. Rosenberg, On optimal strategies for cycle-stealing in networks of workstations, Transactions on Computers 46 (5) (1997) 545–557.
[11] P. Chrétienne, E.G. Coffman Jr., J.K. Lenstra, Z. Liu (Eds.), Scheduling Theory and its Applications, John Wiley and Sons, 1995.
[12] A. Duarte, D. Rexachs, E. Luque, A distributed scheme for fault-tolerance in large clusters of workstations, NIC Series, vol. 33, John von Neumann Institute for Computing, Julich, 2006, pp. 473–480.
[13] H. El-Rewini, H.H. Ali, T.G. Lewis, Task scheduling in multiprocessing systems, Computer 28 (12) (1995) 27–37.
[14] A.H. Frey, G. Fox, Problems and approaches for a teraflop processor, in: Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, ACM Press, 1988, pp. 21–25.
[15] A. Geist, C. Engelmann, Development of naturally fault-tolerant algorithms for computing on 100,000 processors, 2002. <http://www.csm.ornl.gov/geist/Lyon2002-geist.pdf>.
[16] S. Ghosh, R. Melhem, D. Mosse, Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems, IEEE Transactions on Parallel and Distributed Systems 8 (3) (1997) 272–284.
[17] A. Girault, H. Kalla, M. Sighireanu, Y. Sorel, An algorithm for automatically obtaining distributed and fault-tolerant static schedules, in: International Conference on Dependable Systems and Networks, DSN'03, 2003.
[18] K. Hashimito, T. Tsuchiya, T. Kikuno, A new approach to realizing fault-tolerant multiprocessor scheduling by exploiting implicit redundancy, in: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS'97), 1997, p. 174.
[19] K. Hashimito, T. Tsuchiya, T. Kikuno, Effective scheduling of duplicated tasks for fault-tolerance in multiprocessor systems, IEICE Transactions on Information and Systems E85-D (3) (2002) 525–534.
[20] L. Hollermann, T.S. Hsu, D.R. Lopez, K. Vertanen, Scheduling problems in a practical allocation model, Journal of Combinatorial Optimization 1 (2) (1997) 129–149.
[21] B. Hong, V. Prasanna, Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput, in: International Parallel and Distributed Processing Symposium IPDPS'2004, IEEE Computer Society Press, 2004.
[22] T.S. Hsu, J.C. Lee, D.R. Lopez, W.A. Royce, Task allocation on a network of processors, Transactions on Computers 49 (12) (2000) 1339–1353.
[23] S. Khuller, Y. Kim, On broadcasting in heterogenous networks, in: Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2004, pp. 1011–1020.
[24] P. Liu, Broadcast scheduling optimization for heterogeneous cluster systems, Journal of Algorithms 42 (1) (2002) 135–152.
[25] G. Manimaran, C.S.R. Murthy, A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis, IEEE Transactions on Parallel and Distributed Systems 9 (11) (1998) 1137–1152.
[26] M. Naedele, Fault-tolerant real-time scheduling under execution time constraints, in: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications, 1999, p. 392.
[27] M.G. Norman, P. Thanisch, Models of machines and computation for mapping in multicomputers, ACM Computing Surveys 25 (3) (1993) 103–117.
[28] Y. Oh, S.H. Son, Scheduling real-time tasks for dependability, Journal of Operational Research Society 48 (6) (1997) 629–639.
[29] X. Qin, H. Jiang, A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems, Parallel Computing 32 (5) (2006) 331–346.
[30] A. Rosenberg, Optimal schedules for cycle-stealing in a network of workstations with a bag-of-tasks workload, IEEE Transactions on Parallel and Distributed Systems 13 (2) (2002) 179–191.
[31] T. Saif, M. Parashar, Understanding the behavior and performance of non-blocking communications in MPI, in: Proceedings of Euro-Par 2004: Parallel Processing, LNCS, vol. 3149, Springer, 2004, pp. 173–182.
[32] B.A. Shirazi, A.R. Hurson, K.M. Kavi, Scheduling and Load Balancing in Parallel and Distributed Systems, IEEE Computer Science Press, 1995.
[33] O. Sinnen, L. Sousa, Experimental evaluation of task scheduling accuracy: implications for the scheduling model, IEICE Transactions on Information and Systems E86-D (9) (2003) 1620–1627.
[34] O. Sinnen, L. Sousa, Communication contention in task scheduling, IEEE Transactions on Parallel and Distributed Systems 16 (6) (2005) 503–515.
[35] Y. Sorel, Massively parallel computing systems with real-time constraints: the algorithm architecture adequation, in: Proceedings of Massively Parallel Comput. Syst., MPCS, 1994.
[36] H. Topcuoglu, S. Hariri, M.Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems 13 (3) (2002) 260–274.
[37] L. Wang, H.J. Siegel, V.R. Roychowdhury, A.A. Maciejewski, Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach, Journal of Parallel and Distributed Computing 47 (1) (1997) 8–22.