Scheduling for Large-Scale Systems

Anne Benoit, Loris Marchal, Yves Robert, Bora Uçar and Frédéric Vivien

0.1 Introduction

In this chapter, *scheduling* is defined as the activity that consists in mapping a task graph onto a target platform. The task graph represents the application, nodes denote computational tasks, and edges model precedence constraints between tasks. For each task, an assignment (choose the processor that will execute the task) and a schedule (decide when to start the execution) are determined. The goal is to obtain an efficient execution of the application, which translates into optimizing some objective function.

The traditional objective function in the scheduling literature is the minimization of the total execution time, or *makespan*. Section 0.2 provides a quick background on task graph scheduling, mostly to recall some definitions, notations, and well-known results.

In contrast to pure makespan minimization, this chapter aims at discussing scheduling problems that arise at large scale, and involve one or several different optimization criteria, that depart from, or complement, the classical makespan objective. Section 0.3 assesses the importance of key ingredients of modern scheduling techniques: (i) multi-criteria optimization; (ii) memory management; and (iii) reliability. These concepts are illustrated in the following sections, which cover four case studies. Section 0.4 is devoted to illustrate multi-criteria optimization techniques, that mix throughput, energy and reliability. Sections 0.5 and 0.6 illustrate different memory management techniques to schedule large task graphs. Finally, Section 0.7 discusses the impact of checkpointing techniques to schedule parallel jobs at very large-scale. We conclude this chapter by stating some research directions in Section 0.8, by defining terms in Section 0.9, and by pointing to additional sources of information in Section 0.10.

0.2 Background on Scheduling

0.2.1 Makespan Minimization

Traditional scheduling assumes that the target platform is a set of p identical processors, and that no communication cost is paid. In that context, a task graph is a directed acyclic vertex-weighted graph $G = (V, E, w)$, where the set V of vertices represents the tasks, the set E of edges represents precedence constraints between tasks ($e = (u, v) \in E$ if and only if $u \prec v$). and the weight function $w : V \longrightarrow \mathbb{N}^*$ gives the weight (or duration) of each task. Task weights are assumed to be positive integers. A schedule σ of a task graph is a function that assigns a start time to each task: $\sigma : V \longrightarrow \mathbb{N}^*$ such that $\sigma(u) + w(u) \leq \sigma(v)$ whenever $e = (u, v) \in E$. In other words, a schedule preserves the *dependence constraints* induced by the precedence relation $\prec$ and embodied by the edges of the dependence graph; if $u \prec v$, then the execution of u begins at time $\sigma(u)$ and requires $w(u)$ units of time, and the execution of v at time $\sigma(v)$ must start after the end

of the execution of u. Obviously, if there was a cycle in the task graph, no schedule could exist, hence the restriction to acyclic graphs (DAGs).

There are other constraints that must be met by schedules, namely, *resource constraints*. When there is an infinite number of processors (in fact, when there are as many processors as tasks), the problem is *with unlimited processors*, and denoted $P\infty|prec|Cmax$ in the literature [26]. We use the shorter notation Pb(∞) in this chapter. When there is only a fixed number p of available processors, the problem is *with limited processors*, and denoted Pb(p). In the latter case an allocation function $\mathsf{alloc} : V \longrightarrow \mathcal{P}$ is required, where $\mathcal{P} = \{1, \ldots, p\}$ denotes the set of available processors. This function assigns a target processor to each task. The resource constraints simply specify that no processor can be allocated more than one task at the same time:

$$\mathsf{alloc}(T) = \mathsf{alloc}(T') \Rightarrow \left\{ \begin{array}{ll} & \sigma(T) + w(T) \leq \sigma(T') \\ \text{or} & \sigma(T') + w(T') \leq \sigma(T). \end{array} \right.$$

This condition expresses the fact that if two tasks T and T' are allocated to the same processor, then their executions cannot overlap in time.

The *makespan* MS(σ, p) of a schedule σ that uses p processors is its total execution time: MS$(\sigma, p) = \max_{v \in V}\{\sigma(v) + w(v)\}$ (assuming that the first task(s) is (are) scheduled at time 0). The makespan is the total execution time, or finish time, of the schedule. Let MS$_{opt}(p)$ be the value of the makespan of an optimal schedule with p processors: MS$_{opt}(p) = \min_\sigma$ MS(σ, p). Because schedules respect dependences, we have MS$_{opt}(p) \geq w(\Phi)$ for all paths Φ in G (weights extend to paths in G as usual). We also have Seq $\leq p \times$ MS$_{opt}(p)$, where Seq $= \sum_{v \in V} w(v) =$ MS$_{opt}(1)$ is the sum of all task weights.

While Pb(∞) has polynomial complexity (simply traverse the graph and start each task as soon as possible using a fresh processor), problems with a fixed amount of resources are known to be difficult. Lettting $Dec(p)$ be the decision problem associated with Pb(p), and $Indep - tasks(p)$ the restriction of $Dec(p)$ to independent tasks (no dependence, i.e., when $E = \emptyset$) , well-known complexity results are summarized below:

- $Indep - tasks(2)$ is NP-complete but can be solved by a pseudo-polynomial algorithm. Moreover, $\forall\, \varepsilon > 0$, $Indep - tasks(2)$ admits a $(1 + \varepsilon)$-approximation whose complexity is polynomial in $\frac{1}{\varepsilon}$.
- $Indep - tasks(p)$ is NP-complete in the strong sense.
- $Dec(2)$ (and hence $Dec(p)$) is NP-complete in the strong sense.

Because Pb(p) is NP-complete, heuristics are used to schedule task graphs with limited processors. The most natural idea is to use greedy strategies: at each instant, try to schedule as many tasks as possible onto available processors. Such strategies deciding *not to deliberately keep a processor idle* are called *list scheduling* algorithms. Of course there are different possible strategies to decide which tasks are given priority in the (frequent) case where there are more free tasks than available processors. Here a *free* task is a task whose predecessors have all been executed, hence which is available and ready for execution itself. But a key result due to Graham [27] is that any list algorithm can be shown to achieve at most twice the optimal makespan. More precisely, the approximation factor with p processors is $\frac{2p-1}{p}$, and this bound is tight.

A widely used list scheduling technique is *critical path scheduling*. The selection criterion for ready tasks is based on the value of their bottom level, defined as the maximal weight of a path from the task to an exit task of the graph. Intuitively, the larger the bottom level, the more "urgent" the task. The *critical path* of a task is defined as its bottom level and is used to assign priority levels to tasks. Critical path scheduling is list scheduling where the priority level of a task is given by the value of its critical path. Ties are broken arbitrarily.

0.2.2 Dealing with Communication Costs

Thirty years ago, communication costs have been introduced in the scheduling literature. Because the performance of network communication is difficult to model in a way that is both precise and conducive to understanding the performance of algorithms, the vast majority of results hold for a very simple model, which is defined as follows.

The target platform consists of p identical processors that are part of of a fully connected clique. All interconnection links have same bandwidth. If a task T communicates data to a successor task T', the cost is modeled as

$$\text{cost}(T, T') = \begin{cases} 0 & \text{if } \mathsf{alloc}(T) = \mathsf{alloc}(T') \\ c(T, T') & \text{otherwise,} \end{cases}$$

where $\mathsf{alloc}(T)$ denotes the processor that executes task T, and $c(T, T')$ is defined by the application specification. The time for communication between two tasks running on the same processor is negligible. A schedule σ must still preserve dependences, a condition which now writes:

$$\forall e = (T, T') \in E, \begin{cases} \sigma(T) + w(T) \leq \sigma(T') & \text{if } \mathsf{alloc}(T) = \mathsf{alloc}(T') \\ \sigma(T) + w(T) + c(T, T') \leq \sigma(T') & \text{otherwise.} \end{cases}$$

This so-called *macro-dataflow* model makes two main assumptions: (i) communication can occur as soon as data are available; and (ii) there is no contention for network links. Assumption (i) is reasonable as communication can overlap with (independent) computations in most modern computers. Assumption (ii) is much more questionable (although it makes sense in a shared-memory environment). Indeed, there is no physical device capable of sending, say, $1,000$ messages to $1,000$ distinct processors, at the same speed as if there were a single message. In the worst case, it would take $1,000$ times longer (serializing all messages). In the best case, the output bandwidth of the network card of the sender would be a limiting factor. In other words, assumption (ii) amounts to assuming infinite network resources. Nevertheless, this assumption is omnipresent in the traditional scheduling literature.

Including communication costs in the model makes everything difficult, including solving $\text{Pb}(\infty)$, which becomes NP-complete in the strong sense. The intuitive reason is that a trade-off must be found between allocating tasks to either many processors (hence balancing the load but communicating intensively) or few processors (leading to less communication but less parallelism as well). Even the problem in which all task weights and communication costs have the same (unit) value, the so-called UET-UCT problem (unit execution time-unit communication time), is NP-hard [41].

With limited processors, list heuristics can be extended to take communication costs into account, but Graham's bound does not hold any longer. For instance, the *Modified Critical Path (MCP)* algorithm proceeds as follows. First, bottom levels are computed using a pessimistic evaluation of the longest path, accounting for each potential communication (this corresponds to the allocation where there is a different processor per task). These bottom levels are used to determine the priority of free tasks. Then each free task is assigned to the processor that allows its earliest execution, given previous task allocation decisions. It is important to explain further what "previous task allocation decisions" means. Free tasks from the queue are processed one after the other. At any moment, it is known which processors are available and which ones are busy. Moreover, for the busy processors, it is known when they will finish computing their currently allocated tasks. Hence, it is always possible to select the processor that can start the execution of the task under consideration the earliest. It may well be the case a currently busy processor that is selected.

The last extension of the "classical" line of work has been to handle heterogeneous platforms, i.e., platforms that consist of processors with different speeds and interconnection links with different bandwidths. The main principle remains the same, namely to assign the free task with highest priority to the "best"

processor, given already given decisions. Now the "best" processor is the one capable to finish the execution of the task at the earliest, rather than starting the execution of the task, because computing speeds are different. These ideas have led to the design of the list heuristic called HEFT, for *Heterogeneous Earliest Finish Time* [52]. We refer the reader to [14] for a complete description of HEFT, because we have omitted many technical (but important) implementation details. We stress that HEFT has received considerable acceptance, and is the most widely used heuristic to schedule a task graph on a modern computing platform.

0.2.3 Realistic Communication Models

Assuming an application that runs threads on, say, a node that uses multicore technology, the network link could be shared by several incoming and outgoing communications. Therefore, the sum of the bandwidths allotted by the operating system to all communications cannot exceed the bandwidth of the network card. The *bounded multi-port model* proposed by Hong and Prasanna [29] assesses that an unbounded number of communications can thus take place simultaneously, provided that they share the total available bandwidth.

A more radical option than limiting the total bandwidth is simply to forbid concurrent communications at each node. In the *one-port model*, a node can either send data or receive data, but not simultaneously. This model is thus very pessimistic as real-world platforms can achieve some concurrency of communication. On the other hand, it is straightforward to design algorithms that follow this model and thus to determine their performance a priori.

There are more complicated models such as those that deal with bandwidth sharing protocols [36, 37]. Such models are very interesting for performance evaluation purposes, but they almost always prove too complicated for algorithm design purposes. For this reason, when scheduling task graphs, one prefers to deal with the bounded multi-port or the one-port model. As stated above, these models represent a good trade-off between realism and tractability, and we use them for all the examples and case studies of this chapter.

0.3 Underlying Principles

As stated in the introduction, the case-studies provided in this chapter depart from pure makespan minimization. In this section, we briefly overview the main objectives that can be addressed instead of, or in complement to, the total execution time.

0.3.1 Multi-Criteria Optimization

The first case-study (Section 0.4) deals with energy consumption. Processors can execute tasks at different frequencies, and there is a trade-off to achieve between fast execution (which implies high frequencies) and low energy consumption (which requests low frequencies). To complicate matters, running at low frequencies increases the probability of a fault during the execution. Altogether, one faces a tri-criteria problem, involving makespan, energy and reliability.

How to deal with several objective functions? In some approaches, one would form a linear combination of the different objectives and treat the result as the new objective to optimize for. But it is not natural for the user to maximize a quantity like $0.6M+0.3E+0.1R$, where M is the makespan, E the energy consumption, and R the reliability. Instead, one is more likely to fix an execution deadline and a reliability threshold, and to search for the execution that minimizes the energy consumption under these makespan and reliability constraints. One single criterion is optimized, under the condition that the constraints on the other criteria are not violated.

0.3.2 Memory Management

The second and third case-studies (Sections 0.5 and 0.6) both deal with memory management issues. In Section 0.5, the goal is to traverse a task graph (an out-tree) to execute it with a prescribed amount of main memory. To execute a task, one needs to load the files produced by the parent of the task, the object code of the task itself, and all the files that it produces for its children. If there is not enough main memory available, secondary (out-of-core) memory must be used, at the price of a much slower execution. The ordering of the tasks chosen by the scheduler will have a dramatic impact on the peak amount of memory that is needed throughout the execution.

The problem investigated in Section 0.6 also deals with out-trees, and is expressed as follows: how to partition the leaves of the tree in fixed-size parts so that the total memory load is minimized? Here the load of a part if the sum of the weights of the sub-tree formed by the leaves in the part. Intuitively, the goal is to construct parts whose leaves share many ancestors, so that the cost of loading theses ancestors is paid for only a reduced umber of times. Both case-studies arise from scientific applications (such as multi frontal method for sparse linear algebra) that deploy large collections of tasks, and an efficient memory management is the key component to efficiency.

0.3.3 Reliability

The fourth case-study (Section 0.7) addresses a completely different problem. Instead of dealing with task graphs, it deals with a single job that must be executed by one or several processors. This (large) job is divisible, meaning that it can be split arbitrarily into chunks of different sizes. Processors are subject to failures, which leads to take a checkpoint after each successful execution of a chunk. Indeed, after a failure, the execution can resume at the state of the last checkpoint. Frequent checkpoints avoid to lose (in fact, re-execute) a big fraction of the work when a failure occurs, but they induce a big overhead. What is the optimal checkpointing policy? In some way, this problem can be viewed as a trade-off between makespan and reliability, but the framework is totally different from that of Section 0.4. The goal here is to minimize the expected makespan on a platform subject to failures. With the advent of future exascale machines that gather millions of processors, the problem has become central to the community.

0.4 Case Study: Tri-Criteria (Makespan, Energy, Reliability) Scheduling

In this section, we are interested in energy-aware scheduling: the aim is to minimize the energy consumed during the execution of the target application. Energy-aware scheduling has proven an important issue in the past decade, both for economical and environmental reasons. This holds true for traditional computer systems, not even to speak of battery-powered systems. More precisely, a processor running at frequency f dissipates f^3 watts per unit of time [8, 10, 16], hence it consumes $f^3 \times d$ joules when operated during d units of time. To help reduce energy dissipation, processors can run at different frequencies, i.e., different speeds. A widely used technique to reduce energy consumption is *dynamic voltage and frequency scaling (DVFS)*, also known as speed scaling [8, 10, 16]. Indeed, by lowering supply voltage, hence processor clock frequency, it is possible to achieve important reductions in power consumption; faster speeds allow for a faster execution, but they also lead to a much higher (supra-linear) power consumption. Processors can have arbitrary speeds, and can vary them continuously in the interval $[f_{\min}, f_{\max}]$. This model of continuous speeds is unrealistic (any possible value of the speed, say $\sqrt{e^{\pi}}$, cannot be obtained), but it is theoretically appealing [10].

Obviously, minimizing the energy consumption makes sense only when coupled with some performance bound to achieve, otherwise, the optimal solution always is to run each processor at the slowest possible speed. We consider a directed acyclic graph (DAG) of n tasks with precedence constraints, and the goal is to schedule such an application onto a fully homogeneous platform consisting of p identical processors. This problem has been widely studied with the objective of minimizing the total execution time, or *makespan*, and it is well known to be NP-complete [13]. Since the introduction of DVFS, many papers have dealt with the optimization of energy consumption while enforcing a deadline, i.e., a bound on the makespan [8, 10, 16, 6].

There are many situations in which the mapping of the task graph is given, say by an ordered list of tasks to execute on each processor, and we do not have the freedom to change the assignment of a given task. Such a problem occurs when optimizing for legacy applications, or accounting for affinities between tasks and resources, or even when tasks are pre-allocated [46], for example for security reasons. While it is not possible to change the allocation of a task, it is possible to change its speed. This technique, which consists in exploiting the slack due to workload variations, is called slack reclaiming [33, 44]. In our previous work [6], assuming that the mapping and a deadline are given, we have assessed the impact of several speed variation models on the complexity of the problem of minimizing the energy consumption. Rather than using a local approach such as backfilling [55, 44], which only reclaims gaps in the schedule, we have considered the problem as a whole.

While energy consumption can be reduced by using speed scaling techniques, it was shown in [58, 18] that reducing the speed of a processor increases the number of transient fault rates of the system; the probability of failures increases exponentially, and this probability cannot be neglected in large-scale computing [40]. In order to make up for the loss in *reliability* due to the energy efficiency, different models have been proposed for fault-tolerance. In this section, we consider *re-execution*: a task that does not meet the reliability constraint is executed twice. This technique was also studied in [58, 57, 43], or in the case-study of Section 0.7. The goal is to ensure that each task is reliable enough, i.e., either its execution speed is above a threshold, ensuring a given reliability of the task, or the task is executed twice to enhance its reliability. There is a clear trade-off between energy consumption and reliability, since decreasing the execution speed of a task, and hence the corresponding energy consumption, is deteriorating the reliability. This calls for tackling the problem of considering the three criteria (makespan, reliability, energy) simultaneously. This tri-criteria optimization brings dramatic complications: in addition to choosing the speed of each task, as in the makespan/energy bi-criteria problem, we also need to decide which subset of tasks should be re-executed (and then choose both execution speeds).

Given an application with dependence constraints and a mapping of this application on a homogeneous platform, we present in this section theoretical results for the problem of minimizing the energy consumption under the constraints of both a reliability threshold per task and a deadline bound.

0.4.1 Tri-Criteria Problem

Consider an application task graph $\mathcal{G} = (V, \mathcal{E})$, where $V = \{T_1, T_2, \ldots, T_n\}$ is the set of tasks, $n = |V|$, and where $\mathcal{E}$ is the set of precedence edges between tasks. For $1 \leq i \leq n$, task T_i has a weight w_i, that corresponds to the computation requirement of the task. We also consider particular class of task graphs, such as *linear chains* where $\mathcal{E} = \cup_{i=1}^{n-1}\{T_i \rightarrow T_{i+1}\}$, and *forks* with $n+1$ tasks $\{T_0, T_1, T_2, \ldots, T_n\}$ and $\mathcal{E} = \cup_{i=1}^{n}\{T_0 \rightarrow T_i\}$.

We assume that tasks are mapped onto a parallel platform made up of p identical processors. Each processor has a set of available speeds that is continuous (in the interval $[f_{\min}, f_{\max}]$). The goal is to minimize the energy consumed during the execution of the graph while enforcing a deadline bound and matching a reliability threshold. To match the reliability threshold, some tasks are executed once at a speed

high enough to satisfy the constraint, while some other tasks need to be re-executed. We detail below the conditions that are enforced on the corresponding execution speeds. The problem is therefore to decide which task to re-execute, and at which speed to run each execution of a task.

In this section, for the sake of clarity, we assume that a task is executed at the same (unique) speed throughout execution, or at two different speeds in the case of re-execution. In Section 0.4.2, we show that this strategy is indeed optimal. We now detail the three objective criteria (makespan, reliability, energy), and then define formally the problem.

The **makespan** of a schedule is its total execution time. We consider a *deadline bound* D, which is a constraint on the makespan. Let $\mathcal{E}xe(w_i, f)$ be the execution time of a task T_i of weight w_i at speed f. We assume that the cache size is adapted to the application, therefore ensuring that the execution time is linearly related to the frequency [38]: $\mathcal{E}xe(w_i, f) = \frac{w_i}{f}$. When a task is scheduled to be re-executed at two different speeds $f^{(1)}$ and $f^{(2)}$, we always account for both executions, even when the first execution is successful, and hence $\mathcal{E}xe(w_i, f^{(1)}, f^{(2)}) = \frac{w_i}{f^{(1)}} + \frac{w_i}{f^{(2)}}$. In other words, we consider a worst-case execution scenario, and the deadline D must be matched even in the case where all tasks that are re-executed fail during their first execution.

To define the **reliability**, we use the fault model of Zhu et al. [58, 57]. *Transient* failures are faults caused by software errors for example. They invalidate only the execution of the current task and the processor subject to that failure will be able to recover and execute the subsequent task assigned to it (if any). In addition, we use the reliability model introduced by Shatz and Wang [49], which states that the radiation-induced transient faults follow a Poisson distribution. The parameter λ of the Poisson distribution is then:

$$\lambda(f) = \tilde{\lambda_0}\, e^{\tilde{d}\frac{f_{\max}-f}{f_{\max}-f_{\min}}}, \tag{1}$$

where $f_{\min} \leq f \leq f_{\max}$ is the processing speed, the exponent $\tilde{d} \geq 0$ is a constant, indicating the sensitivity of fault rates to DVFS, and $\tilde{\lambda_0}$ is the average fault rate corresponding to $f_{\max}$. We see that reducing the speed for energy saving increases the fault rate exponentially. The reliability of a task T_i executed once at speed f is $R_i(f) = e^{-\lambda(f)\times\mathcal{E}xe(w_i,f)}$. Because the fault rate is usually very small, of the order of 10^{-6} per time unit in [9, 43], 10^{-5} in [5], we can use the first order approximation of $R_i(f)$ as

$$R_i(f) = 1 - \lambda(f) \times \mathcal{E}xe(w_i, f) = 1 - \lambda_0\, e^{-df} \times \frac{w_i}{f},$$

where $d = \frac{\tilde{d}}{f_{\max}-f_{\min}}$ and $\lambda_0 = \tilde{\lambda_0} e^{df_{\max}}$. This equation holds if $\varepsilon_i = \lambda(f) \times \frac{w_i}{f} \ll 1$. With, say, $\lambda(f) = 10^{-5}$, we need $\frac{w_i}{f} \leq 10^3$ to get an accurate approximation with $\varepsilon_i \leq 0.01$: the task should execute within 16 minutes. In other words, large (computationally demanding) tasks require reasonably high processing speeds with this model (which makes full sense in practice).

We want the reliability R_i of each task T_i to be greater than a given threshold, namely $R_i(f_{\texttt{rel}})$, hence enforcing a local constraint dependent on the task $R_i \geq R_i(f_{\texttt{rel}})$. If task T_i is executed only once at speed f, then the reliability of T_i is $R_i = R_i(f)$. Since the reliability increases with speed, we must have $f \geq f_{\texttt{rel}}$ to match the reliability constraint. If task T_i is re-executed (speeds $f^{(1)}$ and $f^{(2)}$), then the execution of T_i is successful if and only if one of the attempts do not fail, so that the reliability of T_i is $R_i = 1 - (1 - R_i(f^{(1)}))(1 - R_i(f^{(2)}))$, and this quantity should be at least equal to $R_i(f_{\texttt{rel}})$.

The **total energy consumption** corresponds to the sum of the energy consumption of each task. Let E_i be the energy consumed by task T_i. For one execution of task T_i at speed f, the corresponding energy consumption is $E_i(f) = \mathcal{E}xe(w_i, f) \times f^3 = w_i \times f^2$, which corresponds to the dynamic part of the classical energy models of the literature [8, 10, 16, 6]. Note that we do not take static energy into account, because all processors are up and alive during the whole execution.

If task T_i is executed only once at speed f, then $E_i = E_i(f)$. Otherwise, if task T_i is re-executed at speeds $f^{(1)}$ and $f^{(2)}$, it is natural to add up the energy consumed during both executions, just as we add up both execution times when enforcing the makespan deadline. Again, this corresponds to the worst-case execution scenario. We obtain $E_i = E_i(f_i^{(1)}) + E_i(f_i^{(2)})$. In this work, we aim at minimizing the total energy consumed by the schedule in the worst-case, assuming that all re-executions do take place. This worst-case energy is $E = \sum_{i=1}^{n} E_i$.

Some authors [57] consider only the energy spent for the first execution, which seems unfair: re-execution comes at a price both in the deadline and in the energy consumption. Another possible approach would be to consider the expected energy consumption, which would require to weight the energy spent in the second execution of a task by the probability of this re-execution to happen. This would lead to a less conservative estimation of the energy consumption by averaging over many execution instances. However, the makespan deadline should be matched in all execution scenarios, and the execution speeds of the tasks have been dimensioned to account for the worst-case scenario, so it seems more important to report for the maximal energy that can be consumed over all possible execution instances.

We are now ready to define the optimization problem: given an application graph $\mathcal{G} = (V, \mathcal{E})$, mapped onto p homogeneous processors with continuous speeds, TRI-CRIT is the problem of deciding which tasks should be re-executed and at which speed each execution of a task should be processed, in order to minimize the total energy consumption E, subject to the deadline bound D and to the local reliability constraints $R_i \geq R_i(f_{\texttt{rel}})$ for each $T_i \in V$.

We also introduce variants of the problems for particular application graphs: TRI-CRIT-CHAIN is the same problem as TRI-CRIT when the task graph is a linear chain, mapped on a single processor; and TRI-CRIT-FORK is the same problem as TRI-CRIT when the task graph is a fork, and each task is mapped on a distinct processor.

0.4.2 Complexity Results

Note that the formal proofs for this section can be found in [7]. As stated in Section 0.4.1, we start by proving that there is always an optimal solution where each task is executed at a unique speed:

Lemma 1. *It is optimal to execute each task at a unique speed throughout its execution.*

The idea is to consider a task whose speed changes during the execution; we exhibit a speed such that the execution time of the task remains the same, but where both energy and reliability are maybe improved, by convexity of the functions.

Next we show that not only a task is executed at a single speed, but that its re-execution (whenever it occurs) is executed at the same speed as its first execution:

Lemma 2. *It is optimal to re-execute each task (whenever needed) at the same speed as its first execution, and this speed f is such that $f_i^{(inf)} \leq f < \frac{1}{\sqrt{2}} f_{rel}$, where*

$$\lambda_0 w_i \frac{e^{-2df_i^{(inf)}}}{(f_i^{(inf)})^2} = \frac{e^{-df_{rel}}}{f_{rel}}. \tag{2}$$

Similarly to the proof of Lemma 1, we exhibit a unique speed for both executions, in case they differ, so that the execution time remains identical but both energy and reliability are improved. If this unique speed is greater than $\frac{1}{\sqrt{2}} f_{\texttt{rel}}$, then it is better to execute the task only once at speed $f_{\texttt{rel}}$, and if f is lower than $f_i^{(\texttt{inf})}$, then the reliability constraint is not matched. Note that both lemmas can be applied to any

solution of the TRI-CRIT problem, not just optimal solutions. We are now ready to assess the problem complexity:

Theorem 1. *The* TRI-CRIT-CHAIN *problem is NP-hard, but not known to be in NP.*

Note that the problem is not known to be in NP because speeds could take any real values. The completeness comes from SUBSET-SUM [23]. The problem is NP-hard even for a linear chain application mapped on a single processor (and any general DAG mapped on a single processor becomes a linear chain).

Even if TRI-CRIT-CHAIN is NP-hard, we can characterize an optimal solution of the problem:

Proposition 1. *If $f_{rel} < f_{\max}$, then in any optimal solution of* TRI-CRIT-CHAIN*, either all tasks are executed only once, at constant speed $\max(\frac{\sum_{i=1}^{n} w_i}{D}, f_{rel})$; or at least one task is re-executed, and then all tasks that are not re-executed are executed at speed f_{rel}.*

In essence, Proposition 1 states that when dealing with a linear chain, we should first slow down the execution of each task as much as possible. Then, if the deadline is not too tight, i.e., if $f_{\mathtt{rel}} > \frac{\sum_{i=1}^{n} w_i}{D}$, there remains the possibility to re-execute some of the tasks (and of course it is NP-hard to decide which ones). Still, this general principle *"first slow-down and then re-execute"* can guide the design of efficient heuristics (see [7]).

While the general TRI-CRIT problem is NP-hard even with a single processor, the particular variant TRI-CRIT-FORK can be solved in polynomial time:

Theorem 2. *The* TRI-CRIT-FORK *problem can be solved in polynomial time.*

The difficulty to provide an optimal algorithm for the TRI-CRIT-FORK problem comes from the fact that the total execution time must be shared between the source of the fork, T_0, and the other tasks that all run in parallel. If we know D', the fraction of the deadline allotted for tasks $T_1, \ldots, T_n$ once the source has finished its execution, then we can decide which tasks are re-executed and all execution speeds. Indeed, if task T_i is executed only once, it is executed at speed $f_i^{(once)} = \min(\max(w_i/D', f_{\mathtt{rel}}), f_{\max})$. Otherwise, it is executed twice at speed $f_i^{(twice)} = \min(\max(2w_i/D', f_{\min}, f_i^{(\mathtt{inf})}), f_{\max})$, where $f_i^{(\mathtt{inf})}$ is the minimum speed at which task T_i can be executed twice (see Lemma 2). The energy consumption for task T_i is finally $E_i = \min\left(w_i \times f_i^{(once)}, 2w_i \times f_i^{(twice)}\right)$, and the case that reaches the minimum determines whether the task is re-executed or not. There remains to find the optimal value of D', which can be obtained by studying the function of the total energy consumption, and bounding the value of D' to a small number of possibilities. Note however that this algorithm does not provide any closed-form formula for the speeds of the tasks, and that there is an intricate case analysis due to the reliability constraints.

If we further assume that the fork is made of identical tasks (i.e., $w_i = w$ for $0 \leq i \leq n$), then we can provide a closed-form formula. However, Proposition 2 illustrates the inherent difficulty of this *simple* problem, with several cases to consider depending on the values of the deadline, and also the bounds on speeds ($f_{\min}$, $f_{\max}$, $f_{\mathtt{rel}}$, etc.). First, since the tasks all have the same weight $w_i = w$, we get rid of the $f_i^{(\mathtt{inf})}$ introduced above, since they are all identical (see Equation (2)): $f_i^{(\mathtt{inf})} = f^{(\mathtt{inf})}$ for $0 \leq i \leq n$. Therefore we let $f_{\min} = \max(f_{\min}, f^{(\mathtt{inf})})$ in the proposition below:

Proposition 2. *In the optimal solution of* TRI-CRIT-FORK *with at least three identical tasks (and hence $n \geq 2$), there are only three possible scenarios: (i) no task is re-executed; (ii) the n successors are all re-executed but not the source; (iii) all tasks are re-executed. In each scenario, the source is executed at speed f_{src} (once or twice), and the n successors are executed at the same speed f_{leaf} (once or twice).*

For a deadline $D < \frac{2w}{f_{\max}}$*, there is no solution. For a deadline* $D \in \left[\frac{2w}{f_{\max}}, \frac{w}{f_{rel}}\frac{(1+2n^{\frac{1}{3}})^{\frac{3}{2}}}{\sqrt{1+n}}\right]$*, no task is re-executed (scenario (i)) and the values of* f_{src} *and* f_{leaf} *are the following:*

- *if* $\frac{2w}{f_{\max}} \leq D \leq \min\left(\frac{w}{f_{\max}}(1+n^{\frac{1}{3}}), w(\frac{1}{f_{rel}}+\frac{1}{f_{\max}})\right)$*, then* $f_{src} = f_{\max}$ *and* $f_{leaf} = \frac{w}{Df_{\max}-w}f_{\max}$*;*
- *if* $\frac{w}{f_{\max}}(1+n^{\frac{1}{3}}) \leq w(\frac{1}{f_{rel}}+\frac{1}{f_{\max}})$*, then*
  - *if* $\frac{w}{f_{\max}}(1+n^{\frac{1}{3}}) < D \leq \frac{w}{f_{rel}}\frac{1+n^{\frac{1}{3}}}{n^{\frac{1}{3}}}$*, then* $f_{src} = \frac{w}{D}(1+n^{\frac{1}{3}})$ *and* $f_{leaf} = \frac{w}{D}\frac{1+n^{\frac{1}{3}}}{n^{\frac{1}{3}}}$*;*
  - *if* $\frac{w}{f_{rel}}\frac{1+n^{\frac{1}{3}}}{n^{\frac{1}{3}}} < D \leq \frac{2w}{f_{rel}}$*, then* $f_{src} = \frac{w}{Df_{rel}-w}f_{rel}$ *and* $f_{leaf} = f_{rel}$*;*
- *if* $\frac{w}{f_{\max}}(1+n^{\frac{1}{3}}) > w(\frac{1}{f_{rel}}+\frac{1}{f_{\max}})$*, then*
  - *if* $w(\frac{1}{f_{rel}}+\frac{1}{f_{\max}})) < D \leq \frac{2w}{f_{rel}}$*, then* $f_{src} = \frac{w}{Df_{rel}-w}f_{rel}$ *and* $f_{leaf} = f_{rel}$*;*
- *if* $\frac{2w}{f_{rel}} < D \leq \frac{w}{f_{rel}}\frac{(1+2n^{\frac{1}{3}})^{\frac{3}{2}}}{\sqrt{1+n}}$*, then* $f_{src} = f_{leaf} = f_{rel}$*.*

Note that for larger values of D, depending on $f_{\min}$, we can move to scenarios (ii) and (iii) with partial or total re-execution. The case analysis becomes even more painful, but remains feasible. Intuitively, the property that all tasks have the same weight is the key to obtaining analytical formulas, because all tasks have the same minimum speed $f^{(\inf)}$ dictated by Equation (2). Beyond the case analysis itself, the result of Proposition 2 is interesting: we observe that in all cases, the source task is executed faster than the other tasks. This shows that Proposition 1 does not hold for general DAGs, and suggests that some tasks may be more critical than others. A hierarchical approach, that categorizes tasks with different priorities, may guide the design of efficient heuristics (see [7]).

0.4.3 Conclusion

In this case-study, we have accounted for the energy cost associated to task re-execution in a more realistic and accurate way than the best-case model used in [57]. Coupling this energy model with the classical reliability model used in [49], we have been able to formulate a tri-criteria optimization problem: how to minimize the energy consumed given a deadline bound and a reliability constraint? The "antagonistic" relation between speed and reliability renders this tri-criteria problem much more challenging than the standard bi-criteria (makespan, energy) version. We have assessed the intractability of this tri-criteria problem, even in the case of a single processor. In addition, we have provided several complexity results for particular instances. Note that several polynomial time heuristics are discussed in [7].

Further work could tackle the design of efficient approximation algorithms. However, this is quite a challenging work for arbitrary DAGs, and one may try to focus on special graph structures, e.g., series-parallel graphs. However, looking back at the complicated case analysis needed for an elementary fork-graph with identical weights (Proposition 2), we cannot underestimate the difficulty of this problem. Also, it could be interesting to study the expected energy consumption, instead of the worst-case energy consumption. However, the complexity of the problems is likely to increase drastically. Finally, we point out that energy reduction and reliability will be even more important objectives with the advent of massively parallel platforms, made of a large number of clusters of multi-cores. More efficient solutions to the tri-criteria optimization problem (makespan, energy, reliability) could be achieved through combining replication with re-execution. A promising (and ambitious) research direction would be to search for the best trade-offs that can be achieved between these techniques that both increase reliability, but whose impact on execution time and energy consumption is very different. The comprehensive set of theoretical results presented in this case-study provides solid foundations for further studies, and constitute a partial yet important first step for solving the problem at very large scale.

0.5 Case Study: Memory-Aware Scheduling

In this section we review the latest results in memory-aware scheduling. When processing an application on a computational grid, it is likely that the involved data will have a very high volume. Such large data sizes may prevent the processing of some task on given resources, and more often, may prevent the processing of two data-intensive tasks on the same resource, because of the input and output data needed for these tasks. In this case, some of the data may need to be discarded from the main storage system (such as memory) and to be written to a secondary storage system (such as disk), to be able to proceed with the execution.

In Grid computing, workflows are usually modeled as Directed Acyclic Graph (DAG), where nodes represent computational tasks and edges represent the dependencies among tasks. The dependencies are in the form of input and output files: each node accepts a set of (large) files as input, and produces a set of (large) files as output, each of them accepted by different child tasks. The execution scheme of such a workflow corresponds to a traversal of the DAG, where visiting a node translates into reading the associated input files and producing output files. Even on a single machine, finding a way to traverse the DAG to minimize the needed storage is a difficult problem: when restricting the problem to uniform sizes, it is similar to the pebble game problem, which has been proven NP-hard by Sethi [48] (the general problem allowing recomputation, that is, re-pebbling a vertex which have been pebbled before, has been proven PSPACE complete by Gilbert, Lengauer and Tarjan [25]). Latest studies focus on the restricted problem of traversing tree-shaped DAGs to minimize the memory demand on a single machine.

For convenience we refer to the two-levels of storage as *main memory* and *secondary memory*, and also as *in-core* and *out-of-core*. Many combinations such as cache and RAM, or RAM and disk, or even disk and tape, lead to the same association of a faster but smaller storage device together with a larger but slower device. The difficulty remains the same for all combinations: find an execution scheme that makes the best use of *main memory*, and minimizes accesses to *secondary memory*.

Throughout the case-study we consider *out-trees* where a task can be executed only if its parent has already been executed. However, all results equivalently apply to *in-trees*, where tasks are processed from the leaves up to the root, as explained below. Each task (or node) i in the tree is characterized by the size f_i of its input file (data needed before the execution and received from its parent), and by the size n_i of its execution file. During execution, non-leaf nodes generate several output files, one for each child, which can have different sizes. A task can be processed only in-core; its execution is feasible only if all its files (input, output, and execution) fit in currently available memory. More formally, let M be the size of the main memory, and S the set of files stored in this memory when the scheduler decides to execute task i. Note that S must contain the input file of task i. The processing of task i is possible if we have:

$$MemReq(i) = f_i + n_i + \sum_{j \in Children(i)} f_j \leq M - \sum_{j \in S, j \neq i} f_j \tag{3}$$

where $MemReq(i)$ denotes the memory requirement of task i (and $Children(i)$ its child nodes in the tree). Once i has been executed, its input file and execution file can be discarded, and replaced by other files in main memory; the output files can either be kept in main memory, in order to execute some child of the task, or they can temporarily be stored into secondary memory (and retrieved later when the scheduler decides to execute the corresponding child of i). The volume of accesses (reads or writes) to secondary memory is referred to as the *I/O volume*.

Clearly, the traversal, i.e., the order chosen to execute the tasks, plays a key role in determining which amount of main memory and I/O volume are needed for a successful execution of the whole tree. More precisely, there are two main problems which the scheduler must address:

MINMEMORY Determine the minimum amount of main memory that is required to execute the tree without any access to secondary memory.

MINIO Given the size M of the main memory, determine the minimum I/O volume that is required to execute the tree.

Obviously, a necessary condition for the execution to be successful is that the size M of main memory exceeds the largest memory requirement over all tasks:

$$\max_i MemReq(i) \leq M$$

However, this condition is not sufficient, and a much larger main memory size may be needed for the MINMEMORY problem.

This problem has two main motivations. The first one (chronologically) arose from numerical linear algebra. Tree workflows (assembly or elimination trees) appear during the factorization of sparse matrices, and the huge size of the files involved makes it absolutely necessary to reduce the memory requirement of the factorization. The trees arising in this context are in-trees (as said before, there is no difference between in-trees and out-trees). Liu [34] discusses how to find the best traversal for the MINMEMORY problem when the traversal is required to correspond to a postorder traversal of the tree. In the follow-up study [35], an exact algorithm is proposed to solve the MINMEMORY problem, without the postorder constraint on the traversal. Another exact algorithm has been proposed recently [30]. Although its theoretical worst-case complexity is similar to Liu's algorithm, its performs better on elimination trees arising from real sparse matrice factorizations.

The second motivation for this problem comes from computations and simulations involving large objects, such as the accurate modeling of the electronic structure of atoms and molecules in quantum chemistry, or in some computational physics code. In this context, Liu's exact algorithm has been recently rediscovered by Lam et al. [32]. In the following, we first propose a formal model for the problem. Then we review state-of-the-art results and algorithms for the MINMEMORY and MINIO problems.

0.5.1 Problem Modeling

We consider a tree workflow $\mathcal{T}$ composed of p nodes, or tasks, numbered from 1 to p. Nodes in the tree have an input file, an execution file (or program), and several output files (one per child). More precisely:

- Each node i has an input file of size f_i. If i is not the root, its input file is produced by its parent $parent(i)$; if i is the root, its input file can be of size zero, or contain input from the outside world.
- Each node i in the tree has an execution file of size n_i.
- Each non-leaf node i in the tree, when executed, produces a file of size f_j for each $j \in Children(i)$. Here $Children(i)$ denotes the set of the children of i. If i is a leaf-node, then $Children(i) = \emptyset$ and i produces a file of null size: we then consider that the terminal data produced by leaves are directly written to the secondary memory or sent to the outside world, independently from the I/O mechanism.

The memory requirement $MemReq(i)$ of node i is the total amount of main memory that is needed to execute node i, as underlined in Equation (3). After i has been processed, its input file and program can be discarded, while its output files can either be kept in main memory (to process the children of i) or be stored in secondary memory temporarily.

0.5.2 In-Core Traversals and the MINMEMORY Problem

For the MINMEMORY problem, we are given a tree $\mathcal{T}$ with p nodes and an initial amount of memory M. A traversal is an ordering of the p nodes that specifies at which step whey are executed. A traversal must obey precedence constraints (a node is always scheduled after its parent) and must never exceed the available memory. A formal definition of a traversal is given below.

Definition 1 (INCORETRAVERSAL)**.** *Given a tree $\mathcal{T}$ and a amount M of available memory, the problem* INCORETRAVERSAL$(\mathcal{T}, M)$ *consists in finding a feasible in-core traversal σ described by a permutation of the nodes of a tree $\mathcal{T}$ such that:*

$$\forall i \neq root, \quad \sigma(parent(i)) < \sigma(i) \tag{4}$$

$$\forall i, \quad \sum_{\sigma(j)<\sigma(i)} \left(\sum_{k \in Children(j)} f_k - f_j \right) + n_i + \sum_{k \in Children(i)} f_k \leq M \tag{5}$$

In this definition, Equation (4) accounts for precedence constraints and Equation (5) deals with memory constraints. A postorder traversal is a traversal where nodes are visited according to some top-down postorder ordering of the tree nodes. Hence, in a postorder traversal, after processing a vertex i, the whole subtree rooted in i is completely processed.

Definition 2 (MINMEMORY)**.** *Given a tree $\mathcal{T}$, determine the minimum amount of memory M such that* INCORETRAVERSAL$(\mathcal{T}, M)$ *has a solution.* MINMEMORY-POSTORDER *is the same problem restricted to postorder traversals.*

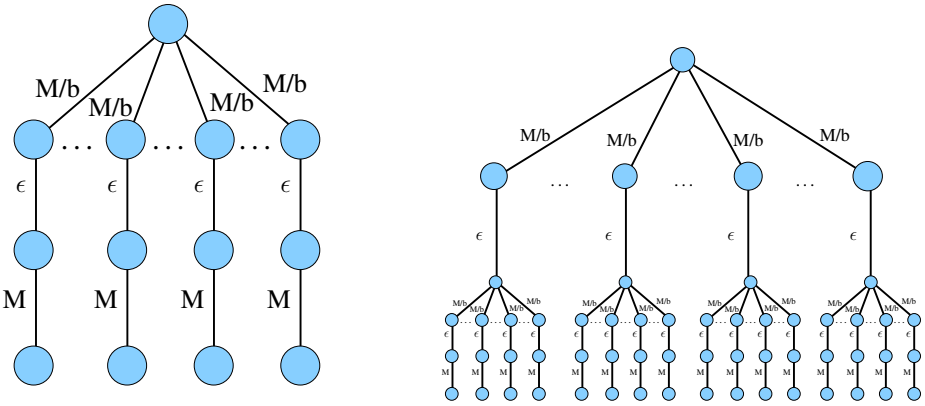
Here, we have chosen to define the problem for out-trees and top-down traversals. However, it is possible to define a similar problem for in-trees and bottom-up traversals. Both problems are equivalent. Consider an in-tree $\mathcal{T}$ and the out-tree $\mathcal{T}'$ obtained when reversing all its edges. Then, from any traversal of $\mathcal{T}$ we can obtain a valid traversal of $\mathcal{T}'$ with same memory footprint simply by reversing the order of visited nodes. This, all results mentioned in the following apply both for in-trees and out-trees.

Postorder traversals

Postorder traversals are very natural for the MINMEMORY problem, and they are widely used in sparse matrix software like `MUMPS` [2, 4]. Liu [34] has characterized the best postorder traversal, leading to a fast but sub-optimal solution for MINMEMORY. In a nutshell, the best postorder is obtained by guaranteeing that in the resulting order, the children of a node are listed in the increasing order of the memory requirement of their respective subtrees. This can be done with a recursive algorithm, which computes a memory-optimal schedule for each subtree: at each level subtrees are sorted in non increasing order of $\max_{i \in subtree}(MemReq(i)) - f_{subroot})$ (where $subroot$ is the root of the current subtree). In the following, we denote this algorithm by $PostOrder$. Although it has a low time complexity ($O(p \log(p))$) and it is widely used in practice, it may require arbitrarily more main memory than the optimal traversal.

Theorem 3. *Given any arbitrarily large integer K, there exist trees for which the best postorder traversal requires at least K times the amount of main memory needed by the optimal traversal for* MINMEMORY.

The proof of this results relies on the family of trees illustrated on Figure 1, where all branches are identical and all tasks have a zero length execution file. On the single-level tree of Figure 1(a), any postorder



(a) One level. (b) Two levels.

Figure 1: First levels of the graph for the proof of Theorem 3. Here b is the number of children of the nodes with more than one child.

traversal requires an amount of $M + \varepsilon + (b-1)M/b$ main memory, while the optimal traversal (which alternates between branches) only requires $M_{\min} = M + \varepsilon$. Now replace each leaf by a copy of the harpoon graph, as shown in Figure 1(b). The value of $M_{\min}$ is unchanged, while a postorder traversal requires $M + \varepsilon + 2(b-1)M/b$. Iterating the process K times leads to the desired result.

The tree used to prove that postorder may require an arbitrarily large memory compared from the optimal is very different to usual trees encountered in practice: it has many edges with zero weight, and on a path from the root, the edge weights alternate between large and small values. It is thus interesting to compare the actual performance of $PostOrder$ in terms of the memory requirement of the resulting traversal with respect to the optimal value on realistic trees. To this goal, one can use assembly trees that arise from the factorization of matrices obtained from the University of Florida Sparse Matrix Collection (see [30] for details on the experimental dataset). The optimal memory requirement of a traversal is computed using the exact algorithms presented below. The experiments show that in 95.8% of the cases, $PostOrder$ is optimal. In the non-optimal cases, $PostOrder$ requires up to 18% more memory than the optimal solution (the average increase is in memory is around 1%).

Exact algorithms

Liu [35] proposes an algorithm for MINMEMORY which is optimal among all possible traversals, not only postorder ones. It is a recursive bottom-up traversal of the tree which, at each node of the tree, combines the optimal traversals built for all subtrees (see Algorithms 1 and 2). The combination is based on the notion of *Hill-Valley Segments*: the schedule corresponding to a subtree is split into a sequence of segments. Each segment ends at a (local) minimum memory: to minimize the memory, it is better to totally process a segment before switching to the processing of another subtree. Then, the sequence of segments computed for all subtrees are ordered by non-increasing $hill - valley$ value and merged. The motivation is then to process first segments with high memory peak (hill) but small residual memory requirement (valley).

Liu provides a complex proof for the optimality of the obtained schedule. It also states that the worst-case complexity of this algorithm is $O(p^2)$ using some sophisticated multi-way merging algorithm.

Another optimal algorithm, the $MinMem$ has been proposed in [30] to minimize memory during the tree traversal. It is based on an advanced tree exploration routine: the $Explore$ algorithm described in

Algorithm 1 *LiuMinMem* (T)

Input: tree T rooted at r
if r *is a leaf* **then**
  return the sequence consisting of the node r
else
  Let $T_1, \ldots, T_k$ be the subtrees rooted at the children of r **for** $i = 1, \ldots, k$ **do**
    $S_i \leftarrow LiuMinMem(T_i)$
  return $S = Combine(T, S_1, \ldots, S_k)$

Algorithm 2 *Combine* $(T, S_1, \ldots, S_k)$

for $i = 1, \ldots, k$ **do**
  Compute the memory occupation at each step of S_i when processing subtree T_i
  Split S_i into a series of m segments such that:
    - the memory occupation at the end of the first segment is the minimum memory (valley) in S_i
    - the memory occupation at the end of the ith segment is the minimum memory in S_i restricted to segments i to m
  For each segment s, compute its hill (maximum memory) and valley (minimum memory) values
Sort all segments by hill-valley value
return this ordering of the nodes

Algorithm 3 *Explore* (T, M^{avail})

Input: tree T rooted at r and available memory M^{avail}
Output: $\langle L, m, S \rangle$ where L is a state (described by the input files it contains) with minimum memory m which can be reached with memory M, and S a schedule to reach it.
if r *is a leaf and there is enough memory for its processing* **then**
  return $\langle \emptyset, 0, (r) \rangle$
if *there is not enough memory to process* r **then**
  return $\langle \emptyset, \infty, \emptyset \rangle$
$S \leftarrow (r)$
$CurrentCut \leftarrow \{Children\ of\ r\}$
$stopped \leftarrow 0$
while $stopped \neq 1$ **do**
  $stopped \leftarrow 1$
  foreach $j \in CurrentCut$ **do**
    Compute the available memory for the processing of the subtree at j:
    $M_j^{\text{avail}} \leftarrow M^{\text{avail}} - \sum_{k \in CurrentCut \setminus \{j\}} f_k$
    $\langle L_j, m_j, S_j \rangle \leftarrow Explore(T, M_j^{\text{avail}})$
    if *exploring the subtree at* j *leads to state with less memory:* $m_j \leq f_j$ **then**
      $stopped \leftarrow 0$
      $CurrentCut \leftarrow CurrentCut \setminus \{j\} \cup L_j$
      append traversal S_j to the end of S: $S \leftarrow S \oplus S_j$
$m \leftarrow \sum_{j \in CurrentCut} f_j$
return $\langle CurrentCut, m, S \rangle$

Algorithm 3. The *Explore* algorithm requires a tree T to explore and an amount of available memory M_{avail}. With these parameters, the algorithm computes state with minimal residual memory that can be reached with memory M_{avail}. If the whole tree can be processed, then the minimal residual memory is zero. Otherwise, the algorithm stops before reaching the bottom of the tree, because some parts of the tree require more memory than what is available. In this case, the state with minimal residual memory corresponds to a *cut* in the tree: some subtrees are not yet processed, and the input files of their root nodes are still stored in memory. The *Explore* algorithm outputs the cut with minimal memory occupation, as well as a possible traversal to reach this state with the provided memory.

When called on a tree T, the algorithm first checks if the current node can be executed. If not, the algorithm stops. Otherwise, it recursively proceeds in its subtree. The optimal cut is initialized with its children, and iteratively improved. All the nodes in the cut are explored: if the cut L_j found in the subtree of a child j has a smaller memory occupation than the child itself, the cut is updated by removing child j, and by adding the corresponding cut L_j. When no more nodes in the cut can be improved, then the algorithm outputs the current cut.

The *Explore* algorithm can be used to check whether a given tree can be processed using a given memory. A possible way to compute the minimum memory to process the tree is then to perform a binary search on the available memory. However, there are several ways to speed-up such an algorithm (for details, see [30]):

- We can transform the *Explore* algorithm so that it also returns the minimum increment in the available memory needed to explore more nodes. This helps to reduce the number of calls to *Explore*.
- When calling *Explore* several times on the same subtree with an increasing memory, the beginning of the subtree will be explored many times. To avoid this, it is possible to store the current state of exploration, as well a the current schedule. This ensures that nodes are not visited several times.

Using these optimizations, it is possible to prove that the $MinMem$ algorithm based on the *Explore* procedures has a similar time complexity with $LiuMinMem$ ($O(p^2)$). Although both exact algorithms have the same worst-case time complexity, it is interesting to note that $MinMem$ is faster than $LiuMinMem$ on realistic trees: Experiments on the same set of trees described above show that $MinMem$ is the fastest algorithm in 80% of the cases. This is explain by the fact that $LiuMinMem$ has to sort segments when merging schedules obtained for subtrees. On the contrary, $MinMem$ does not use sorting. This can make a significant difference if, for example, for each node, the size of the input file is larger than the size of the output files, $MinMem$ will run in $O(n)$ for a tree of n nodes.

0.5.3 Out-of-Core Traversals and the MINIO Problem

Out-of-core processing enables solving large problems, when the size of the data cannot fit into the main memory. In this case, some temporary data are copied into the secondary memory, and unloaded from the main memory, so as to leave room for other computations. Since secondary memory has a smaller access rate, the usual objective is to limit the volume of I/O operations.

Defining traversals that perform I/O operations is more complicated than defining in-core traversals: in addition to determining the ordering of the nodes (the permutation σ), at each step we have to identify which files are written into secondary memory (if necessary). When a task i is scheduled for execution but its input file was moved to secondary memory, that file must be read and loaded back into the main memory before processing task i. Thus, a given file is written at most once in the secondary memory. Moreover, it a file is to be written in the secondary memory, it is always beneficial to write it as soon as it is created. Thus, we can model the I/O operations via a binary function δ_{IO} such that $\delta_{IO}(i) = 1$ if and only if the input file of task i (of size f_i) should be moved to secondary memory. Formally, a valid out-of-core traversal can be

defined as follows.

Definition 3 (OUTOFCORETRAVERSAL)**.** *Given a tree $\mathcal{T}$ and a fixed amount of main memory M, the problem* OUTOFCORETRAVERSAL$(\mathcal{T}, M)$ *consists in finding an out-of-core traversal, described by a permutation σ of the nodes in $\mathcal{T}$ (corresponding to the schedule of computations), and a binary function δ_{IO} (corresponding to I/O operations), such that:*

$$\forall i \neq root, \quad \sigma(parent(i)) < \sigma(i) \tag{6}$$

$$\forall i, \quad \sum_{\sigma(j)<\sigma(i)} \left(\sum_{k \in Children(j)} f_k - f_j \right) - \sum_{\substack{\sigma(parent(j))<\sigma(i)<\sigma(j) \\ \delta_{IO}(j)+1}} f_j + n_i + \sum_{k \in Children(i)} f_k \leq M \tag{7}$$

Then the amount of data written in secondary memory is given by

$$IO = \sum_{\delta_{IO}(i)=1} f_i$$

In Equation (7), the term $\sum_{\substack{\sigma(parent(j))<\sigma(i)<\sigma(j) \\ \delta_{IO}(j)+1}} f_j$ corresponds to the files that have been written into secondary memory at step $\sigma(i)$. The MINIO problem, which asks for an out-of-core traversal with the minimum amount of I/O volume, can be defined as follows.

Definition 4 (MINIO)**.** *Given a tree $\mathcal{T}$, and a fixed amount of main memory M, determine the minimum I/O volume IO needed by a solution of* OUTOFCORETRAVERSAL$(\mathcal{T}, M)$.

The following three variants of the problem are NP-complete.

Theorem 4. *Given a tree $\mathcal{T}$ with p nodes, and a fixed amount of main memory M, consider the following problems:*
(i) given a postorder traversal σ of the tree, determine the I/O schedule so that the resulting I/O volume is minimized,
(ii) determine the minimum I/O volume needed by any postorder traversal of the tree,
(iii) determine the minimum I/O volume needed by any traversal of the tree.
The (decision version of) each problem is NP-complete.

Note that (iii) is the original MINIO problem. Also note that the NP-completeness of (i) does not a priori imply that of (ii), because the optimal postorder traversal could have a particular structure. The same comment applies for (ii) not implying (iii). The proof of these three results all relies on a reduction from 2-Partition problem [23], using the tree illustrated on Figure 2. Scheduling this tree with a memory $M = 2S$ an maximum amount IO of $IO = 2/S$ requires to process first task T_{big}, and thus to select a subset of a_is (the input file of tasks T_i) which sums exactly to $S/2$. For more details on the proofs for all variants, see [30].

0.5.4 Perspectives and Open Problems

We have gathered in this section existing results on the problem of scheduling trees under memory constraints. Many problems are still open. The first one concerns splittable files. Indeed, the NP-completeness

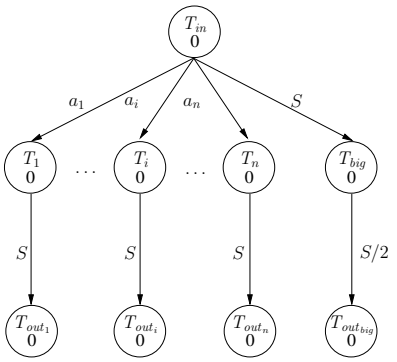


Figure 2: Reduction of an instance of the 2-partition problem for the NP-completeness proof of MINIO.

of the MINIO problem comes from the difficulty to find a subset of files whose sizes sum up to a given bound. However, most memory systems nowadays are paginated, and allows to write only portions of files to secondary memory. Therefore, studying the complexity of the MINIO problem with splittable files is very relevant. This problem is still open.

The results presented here only concern the processing of trees on a single machine. It is crucial to extend these results for other type of graphs (like series-parallel graphs, or any DAGs), and for parallel machines. It is likely that most problems will be NP-complete, but the results on trees may be useful to derive good heuristics, in the best case with approximation bounds. Even if approximation algorithms seem out of reach for general traversals, there is hope to derive an approximation algorithm for postorder traversals of trees, which are simpler to analyze than arbitrary traversals.

0.6 Case Study: Partitioning Tree Structured Computations

The problem addressed in this case-study arises in the solution of linear systems with a direct method, and with an out-of-core environment [1]. In order to make the section self-contained, we investigate the combinatorial problem (in Sections 0.6.1-to-0.6.3) without linear algebra notions, and refer the curious reader to the related reference items. We briefly discuss the mentioned application and one more related scenario in Section 0.6.4.

0.6.1 Definitions

Let $T = (V, E)$ be a rooted tree where all the edges are towards the root node r. Each node u, except the root, has a parent. Each node u, except the leaves, has a set of children, which are the nodes whose parent is u. For each node u, the vertices in the unique path from u to r are called the ancestors of u, and they are denoted by $P(u)$. The nodes u and r are included in $P(u)$. A subtree of T rooted at a node u contains all nodes v for which $u \in P(v)$; we use $T(u)$ to denote that subtree and use $L(u)$ to denote the leaf nodes in the subtree $T(u)$. A topological ordering of a tree $T = (V, E)$ is an ordering of the nodes V in such a way that any node is numbered before its parent; the root node is ordered last. A postorder traversal of a tree is a topological ordering where the nodes in each subtree are numbered consecutively.

There is a weight associated with each internal node of T. We use $w(u)$ to denote the weight of the node u. Leaf nodes are special and have zero weight. We are given a set of n tasks $t_1, t_2, \ldots, t_n$. Each task is associated with a unique leaf node. In order to execute the task t_i, one starts from a leaf node ℓ_i and visits all

the internal nodes in $P(\ell_i)$. Visiting an internal node u entails a cost, $w(u)$. A task requires a unit amount of memory storage throughout its execution. The task executions are nonpreemptive, i.e., when a task is started, it continues until the root node. One can execute two or more tasks concurrently at each internal node, where the cost payed at a node u is always $w(u)$, irrespective of the number of concurrent tasks. As each task requires a unit amount of memory storage, the number of concurrent tasks is limited. In such a case, one takes as many tasks as possible and executes them concurrently till the root node. The problem then becomes that of partitioning the tasks into groups so that each group can be stored in the memory while the total cost is minimized.

We now define the problem formally. The cost associated with a task t_i is

$$cost(t_i) = \sum_{u \in P(\ell_i)} w(u) \ . \tag{8}$$

If we execute a set S of tasks concurrently, the associates cost is

$$cost(S) = \sum_{u \in P(S)} w(u) \ \text{ where } \ P(S) = \bigcup_{t_i \in S} P(\ell_i) \ .$$

Let B be the maximum number of tasks one can execute concurrently. Then we have the following TREEPARTITIONING problem. Given a tree T, a set of tasks $S = \{t_1, t_2, \ldots, t_n\}$, and an integer $B \leq n$, partition S into a number of subsets $S_1, S_2, \ldots, S_K$ such that $|S_k| \leq B$, for $k = 1, \ldots, K$ and the total cost

$$\sum_{k=1}^{K} cost(S_k) \tag{9}$$

is minimum. The number of parts K is not specified (but for feasibility we have $K \geq \lceil n/B \rceil$).

0.6.2 Complexity Results

Before introducing the complexity results (and algorithms), we present a lower bound for the cost of an optimal partition.

Theorem 5. *Let T be a node weighted tree, $w(u)$ be the weight of node u, B be the maximum allowed size of a partition, and $L(u)$ be the number of leaf nodes in the subtree rooted at u. Then we have the following lower bound, denoted by η, on the optimal solution $c^\star$ of the* TREEPARTITIONING *problem:*

$$\eta = \sum_{u \in T} w(u) \times \left\lceil \frac{L(u)}{B} \right\rceil \leq c^\star \ .$$

Proof. Follows easily by noting that the leaf nodes of the subtree $T(u)$ will be partitioned among at least $\left\lceil \frac{L(u)}{B} \right\rceil$ parts. □

Each internal node is on a path from (at least) one leaf node, therefore $\lceil L(u)/B \rceil$ is at least 1, and we have $\sum_u w(u) \leq c^\star$.

Theorem 6. *The* TREEPARTITIONING *problem is NP-complete.*

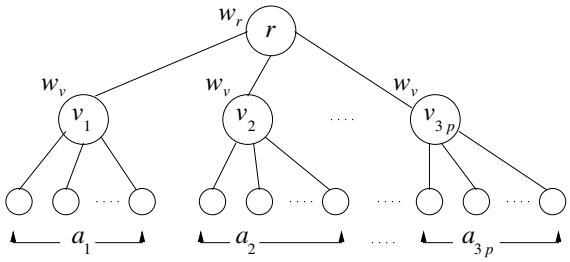


Figure 3: The instance of the TREEPARTITIONING problem corresponding to a given 3-PARTITION PROBLEM. The weight of each node is shown next to the node. The minimum cost of a solution for $B = Z$ to the TREEPARTITIONING problem is $p \times w_r + 3p \times w_v$ which is only possible when the children of each v_i are all in the same part, and when the children of three different internal nodes, say v_i, v_j, v_k, are put in the same part. This corresponds to putting the numbers a_i, a_j, a_k into a set for the 3-PARTITION problem which sums up to Z.

Proof. We consider the associated decision problem: given a tree T with n leaves, a value of B, and a cost bound c, does there exist a partitioning S of the n leaves into subsets whose size does not exceed B, and such that $cost(S) \leq c$? It is clear that this problem belongs to NP since if we are given the partition S, it is easy to check in polynomial time that it is valid and that its cost meets the bound c. We now have to prove that the problem is in the NP-complete subset.

To establish the completeness, we use a reduction from 3-PARTITION [23], which is NP-complete in the strong sense. Consider an instance $\mathcal{I}_1$ of 3-PARTITION: given a set $\{a_1, \ldots, a_{3p}\}$ of $3p$ integers, and an integer Z such that $\sum_{1 \leq j \leq 3p} a_j = pZ$, does there exist a partition of $\{1, \ldots, 3p\}$ into p disjoint subsets $K_1, \ldots, K_p$, each with three members, such that for all $1 \leq i \leq p$, $\sum_{j \in K_i} a_j = Z$?

We build the following instance $\mathcal{I}_2$ of our problem: the tree is a three-level tree composed of $N = 1 + 3p + pZ$ nodes: the root v_r, of cost w_r, has $3p$ children v_i, of the same cost w_v, for $1 \leq i \leq 3p$. In turn, each v_i has a_i children, each being a leaf node of zero cost. This instance $\mathcal{I}_2$ of the TREEPARTITIONING problem is shown in Fig. 3. We let $B = Z$ and ask whether there exists a partition of leaf nodes of cost $c = pw_r + 3pw_v$. Here w_r and w_v are arbitrary values (we can take $w_r = w_v = 1$). We note that the cost c corresponds to the lower bound shown in Theorem 5; in this lower bound, each internal node v_i is loaded only once, and the root is loaded p times, since it has $pZ = pB$ leaves below it. Note that the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. Indeed, because 3-PARTITION is NP-complete in the strong sense, we can encode $\mathcal{I}_1$ in unary, and the size of the instance is $O(pZ)$.

Now we show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ has a solution. Suppose first that $\mathcal{I}_1$ has a solution $K_1, \ldots, K_p$. The partition of leaf nodes corresponds exactly to the subsets K_i: we build p subsets S_i whose leaves are the children of vertices v_j with $j \in K_i$. Suppose now that $\mathcal{I}_2$ has a solution. To meet the cost bound, each internal node has to be loaded only once, and the root at most p times. This means that the partition involves at most p subsets to cover all leaves. Because there are pZ leaves, each subset is of size exactly Z. Because each internal node is loaded only once, all its leaves belong to the same subset. Altogether, we have found a solution to $\mathcal{I}_1$, which concludes the proof. □

We can further show that we cannot get a close approximation to the optimal solution in polynomial time.

Theorem 7. *Unless P=NP, there is no* $1 + o\left(\frac{1}{m}\right)$ *polynomial approximation for trees with* m *nodes in the* TREEPARTITIONING *problem.*

Proof. Assume that there exists a polynomial $1 + \frac{\varepsilon(m)}{m}$ approximation algorithm for trees with m nodes, where $\lim_{m\to\infty} \varepsilon(m) = 0$. Let $\varepsilon(m) < 1$ for $m \geq m_0$. Consider an arbitrary instance $\mathcal{I}_0$ of 3-PARTITION with a set $\{a_1, \ldots, a_{3p}\}$ of $3p$ integers, and an integer Z such that $\sum_{1\leq j\leq 3p} a_j = pZ$. Without loss of generality, assume that $a_i \geq 2$ for all i (hence $Z \geq 6$). We ask if we can partition the $3p$ integers of $\mathcal{I}_0$ into p triples of the same sum Z. Now we build an instance $\mathcal{I}_1$ of 3-PARTITION by adding X times the integer $Z-2$ and $2X$ times the integer 1 to $\mathcal{I}_0$, where $X = \max\left(\left\lceil \frac{m_0-1}{Z+3} \right\rceil - p, 1\right)$. Hence $\mathcal{I}_1$ has $3p+3X$ integers and we ask whether these can be partitioned into $p+X$ triples of the same sum Z. Clearly, $\mathcal{I}_0$ has a solution if and only if $\mathcal{I}_1$ does (the integer $Z-2$ can only be in a set with two 1s).

We build an instance $\mathcal{I}_2$ of TREEPARTITIONING from $\mathcal{I}_1$ exactly as we did in the proof of Theorem 6, with $w_r = w_v = 1$, and $B = Z$. The only difference is that the value p in the proof has been replaced by $p + X$ here, therefore the three-level tree now has $m = 1 + 3(p + X) + (p + X)Z$ nodes. Note that X has been chosen so that $m \geq m_0$. Just as in the proof of Theorem 6, $\mathcal{I}_1$ has a solution if and only if the optimal cost for the tree is $c^\star = 4(p + X)$, and otherwise the optimal cost is at least $4(p + X) + 1$.

If $\mathcal{I}_1$ has a solution, and because $m \geq m_0$, the approximation algorithm will return a cost at most

$$\left(1 + \frac{\varepsilon(m)}{m}\right) c^\star \leq \left(1 + \frac{1}{m}\right) 4(p + X) = 4(p + X) + \frac{4(p + X)}{m} .$$

But $\frac{4(p+X)}{m} = \frac{4(m-1)}{(Z+3)m} \leq \frac{4}{9} < 1$, so that the approximation algorithm can be used to determine whether $\mathcal{I}_1$, and hence $\mathcal{I}_0$, has a solution. This is a contradiction unless P=NP. □

0.6.3 Approximation Algorithm

Consider the heuristic POPART shown in Algorithm 4 for the TREEPARTITIONING problem. As seen in Algorithm 4, the POPART heuristic first orders the leaf nodes according to their rank in the postorder. It then puts the first B leaves in the first part, the next B leaves in the second part, and so on. This simple partitioning approach results in $\lceil n/B \rceil$ parts, for a tree with n leaf nodes, and puts B nodes in each part, except maybe in the last one. We have the following theorem which states that this simple heuristic obtains results that are at most twice the cost of an optimum solution.

Algorithm 4 POPART: A postorder based partitioning

Input $T = (V, E)$ with n leaves; each requested entry corresponds to a leaf node.
Input B: the maximum allowable size of a part.
Output $\Pi_{PO} = \{S_1, \ldots, S_K\}$ where $K = \lceil n/B \rceil$, a partition on the leaf nodes.
1: compute a postorder of the nodes of T
2: $\mathcal{L} \leftarrow$ sort the leaf nodes according to their rank in the postorder
3: $S_k = \{\mathcal{L}(i) : (k-1) \times B + 1 \leq i \leq \min\{k \times B, n\}$, for $k = 1, \ldots, \lceil n/B \rceil\}$

Theorem 8. *Let* Π_{PO} *be the partition obtained by the algorithm* POPART *and* $c^\star$ *be the cost of an optimum solution. Then*

$$cost(\Pi_{PO}) \leq 2 \times c^\star .$$

Proof. Consider node u. Since the leaves of the subtree rooted at u are sorted consecutively in $\mathcal{L}$, the contribution of node u to the cost will be at most $\left\lceil \frac{L(u)}{B} \right\rceil + 1$. Therefore the overall cost is at most

$$\begin{aligned} cost(\Pi_{PO}) &\leq \sum_u w(u) \times \left(\left\lceil \frac{L(u)}{B} \right\rceil + 1 \right) \\ &\leq \eta + \sum_u w(u) \\ &\leq 2 \times c^\star \end{aligned}$$

□

0.6.4 Application Scenarios

In the application of the TREEPARTITIONING problem described by Amestoy et al. [1], each task corresponds to a sparse triangular solve operation with a right-hand-side (RHS) vector containing a single nonzero. There is a tree which describes the data requirements for each solve operation. Each internal node of the tree corresponds to parts of the triangular matrices stored in a file at disk. Solve operation for a given RHS vector starts from a leaf node and climbs up to the root. At each internal node, the matrix parts associated with that node are read from a file, and their inverse are applied to the RHS vector. All the RHS vectors are of the same size, and there is a huge number of them. Due to the limited memory size, one can hold only a certain number of RHS vectors at once in memory. In this setting, one organizes the solve operations in epochs where at each epoch a set of RHS vectors are solved for. Any internal node is accessed at most once during an epoch, but it can be accessed in multiple epochs. The solution time of an epoch is dominated by the time spent in reading the files containing the parts of the triangular matrices. In order to minimize the completion time of the application, one has to minimize the total size of the files read in. This is an instance of the TREEPARTITIONING problem where the tasks are solve operations; B is the number of unit size RHS vectors that one can hold in memory; the cost associated with an internal node is the size of the corresponding file; and the objective function is the total size of the files read in.

Consider a hypothetical application whose tasks correspond to paths from a leaf node to the root in a given tree. Assume that each internal node is associated with a data. Consider the following parallelization approach. Each task is to be executed by a single processor. This necessitates replicating the nodes of the tree at different processors so that each task can be executed by the owner processor. This can be again described as an instance of the TREEPARTITIONING problem, where this time B is a function of the available processors. Consider again the same application, where we do not want to replicate the data. In a possible parallelization approach, the data should be partitioned, and the tasks should be migrated from one processor to the other whenever needed (e.g., when the parent of a node and the node itself are at different processors). This time, one partitions the internal nodes of a tree so that the leaf-to-root paths are split among as few parts as possible. This seemingly different problem is addressed elsewhere [24] with techniques that are very different than what we saw in this section. However, it is possible to express both problems using hypergraph models and use a common algorithm to obtain effective solutions [53].

0.7 Case Study: Checkpointing Strategies

Any hardware component may be subject to faults. The Mean Time Between Failures (MTBF) of a processor is typically of the order of one hundred years. The reader can thus very safely bet that his laptop processor

will not fail within the next month. Therefore, she does not need any fault-tolerance technique to safely run a computation on his laptop for, say, 12 hours. Things are completely different if the 12-hour computation is performed using a large number of processors. Indeed, the probability that one of the processors used by the application will fail during the 12 hours of the application execution, dramatically increases with the number of processors enrolled in the execution. For instance, the 45,208-processor Jaguar platform is reported to experience on the order of one failure per day [39, 11]. In other words, during any execution that would run for 12 hours using the entire Jaguar platform, there is one chance out of two that a fault will occur. Faults that cannot be automatically detected and corrected in hardware lead to application failures. Using fault-tolerance techniques is thus mandatory when using a large number of processors for a significant amount of time. Resilience—the ability to resist to failures—is thus a key challenge for modern and future high-performance computing (HPC) systems [19, 47].

The most used fault-tolerance technique is rollback recovery: after a failure, job execution is resumed from a previously saved fault-free execution state, or *checkpoint*. Rollback recovery implies frequent (usually periodic) *checkpointing* events at which the job state is saved to resilient storage. More frequent checkpoints lead to higher overhead during fault-free execution, but less frequent checkpoints lead to a larger loss when a failure occurs. The design of efficient *checkpointing strategies*, which specify when checkpoints should be taken, is thus key to high performance.

In this case study, we consider a single job to be executed on a predefined number of processors. The scheduling problem, here, is then to decide when (and how many times) to checkpoint the job in order to minimize its expected execution time.

0.7.1 Problem Statement

We consider an application, or *job*, that executes on p *processors*. The job must complete $\mathcal{W}$ units of (divisible) work: there are no constraints on when the job state can be checkpointed. We assume coordinated checkpointing [54], i.e., that all processors checkpoint the job state simultaneously. We call *chunk* the fraction of the whole job that is executed between two consecutive checkpoints. Defining the sequence of chunk sizes is therefore equivalent to defining the checkpointing dates. We use C to denote the time needed to perform a checkpoint. The processors are subject to *failures*, each causing a *downtime* period, of duration D, followed by a *recovery* period, of duration R. The downtime accounts for software rejuvenation (i.e., rebooting [31, 15]) or for the replacement of the failed processor by a spare. Regardless, we assume that after a downtime the processor that failed is fault-free and begins a new lifetime at the beginning of the recovery period. This period corresponds to the time needed to restore the last checkpoint. Note that although C and R can fluctuate depending on cluster and network load, as in most works in the field we assume they are constant. Finally, we assume that failures can happen during recovery or checkpointing, but not during a downtime (otherwise, the downtime period could be considered part of the recovery period).

We consider the processors from time t_0 onward. We use $P_{suc}(x|\tau_1, ..., \tau_p)$ to denote the probability that none of the processors fail for the next x units of time, knowing that the last failure of processor P_i occurred τ_i units of time ago, for $1 \leq i \leq p$. The failure interarrival times on the different processors are defined by independent and identically distributed random variables. Let X_i be the random variable for the interarrival time on processor P_i between the last failure (that happened τ_i units of time ago) and the next one. Then we have:

$$P_{suc}(x|\tau_1, ..., \tau_p) = \mathbb{P}(X_1 \geq \tau_1 + x, ..., X_p \geq \tau_p + x \mid X_1 \geq \tau_1, ..., X_p \geq \tau_p) \ .$$

Note that we do *not* assume that the failure stochastic process is memoryless.

0.7.2 Minimizing the Expected Makespan

Our aim is to minimize the job execution time. However, as we are trying to execute the job on a failure-prone platform, this execution time cannot be known beforehand: it depends on whether and when failures will impact the job execution. Therefore, our objective will not be the minimization of the makespan, like for instance in Section 0.4, but the *expectation* of the makespan, that is, the average makespan, the average being taken over all possible failure scenarios.

Formal Problem Definition

A solution to our scheduling problem is fully defined by a function $f(\omega|\tau_1, ..., \tau_p)$ that returns the size of the next chunk to execute given the amount of work ω that has not yet been executed successfully ($f(\omega|\tau_1, ..., \tau_p) \leq \omega \leq \mathcal{W}$) and the amount of time τ_i elapsed since the last failure of processor P_i, $1 \leq i \leq p$. f is invoked at each decision point, i.e., after each checkpoint or recovery. Our goal is to determine a function f that defines an optimal solution. Assuming unit-speed processors without loss of generality, the time needed to execute a chunk of size w is $w + C$ if no failure occurs.

For a given amount of work ω and a time elapsed since the last failure τ, we define $T(\omega|\tau_1, ..., \tau_p)$ as the random variable that quantifies the time needed for executing ω units of work. Given a solution function f, let $\omega_1 = f(\omega|\tau_1, ..., \tau_p)$ denote the size of the first chunk. We can write the following recursion:

$$
\begin{aligned}
T(0|\tau_1, ..., \tau_p) &= 0 \\
T(\omega|\tau_1, ..., \tau_p) &= \begin{cases} \omega_1 + C + T(\omega - \omega_1|\tau_1 + \omega_1 + C, ..., \tau_p + \omega_1 + C) \\ \quad \text{if no processor fails during the next } \omega_1 + C \text{ units of time,} \\ T_{wasted}(\omega_1 + C|\tau_1, ..., \tau_p) + T(\omega|\tau'_1, ..., \tau'_p) \quad \text{otherwise.} \end{cases}
\end{aligned} \tag{10}
$$

The two cases above are explained as follows:

- If none of the processors fail during the execution and checkpointing of the first chunk (i.e., for $\omega_1 + C$ time units), there remains to execute a work of size $\omega - \omega_1$ and the time since the last failure is incremented on each processor by $\omega_1 + C$.
- If at least one processor fails before the successful completion of the first chunk and of its checkpoint, then some additional delays are incurred, as captured by the variable $T_{wasted}(\omega_1 + C|\tau_1, ..., \tau_p)$. The time wasted corresponds to the execution up to the failure, a downtime, and a recovery during which some additional failures (and downtimes) may happen. Regardless, once a successful recovery has been completed, there still remain ω units of work to execute, and the time since the last failure of a processor depends on whether it fails (and when) during the attempt to process the first chunk, or the subsequent downtime and recovery.

We define MAKESPAN formally as: find f that minimizes $\mathbb{E}(T(\mathcal{W}|\tau_1^0, ..., \tau_p^0))$, where $\mathbb{E}(X)$ denotes the expectation of the random variable X, and τ_i^0 the time elapsed since the last failure of processor P_i that happened before t_0.

Solving MAKESPAN for the Exponential Distribution

In this section we assume that the failure inter-arrival times of any processor follow an Exponential distribution with parameter λ, i.e., each $X_i = X$ has probability density $f_X(t) = \lambda e^{-\lambda t} dt$ and cumulative distribution $F_X(t) = 1 - e^{-\lambda t}$ for all $t \geq 0$. The advantage of the Exponential distribution, exploited time and again in the literature, is its "memoryless" property: the time at which the next failure occurs does not

depend on the time elapsed since the last failure occurred. Therefore, in this section, we simply write $T(\omega)$, $T_{wasted}(\omega)$, and $P_{suc}(\omega)$ instead of $T(\omega|\tau_1, ..., \tau_p)$, $T_{wasted}(\omega|\tau_1, ..., \tau_p)$, and $P_{suc}(\omega|\tau_1, ..., \tau_p)$.

A challenge for solving MAKESPAN is the computation of $T_{wasted}(\omega_1 + C)$. We rely on the following decomposition:

$$T_{wasted}(\omega_1 + C) = T_{lost}(\omega_1 + C) + T_{rec}\ , \qquad \text{where}$$

- $T_{lost}(x)$ is the amount of time spent computing before a failure, knowing that the next failure occurs within the next x units of time;
- T_{rec} is the amount of time needed by the system to recover from the failure (accounting for the fact that other failures may occur during recovery).

The expectation of the makespan can then be expressed as a "simple" recursion:

Lemma 3. *Let $\mathcal{W}$ be the amount of work to execute on p processors whose failure inter-arrival times follow iid Exponential distributions with parameter λ. The* MAKESPAN *problem is equivalent to finding a function f minimizing the following quantity:*

$$\mathbb{E}(T(\mathcal{W})) = P_{suc}(\omega_1 + C)\Big(\omega_1 + C + \mathbb{E}(T(\mathcal{W} - \omega_1 + \omega_1 + C))\Big) + (1 - P_{suc}(\omega_1 + C))\Big(\mathbb{E}(T_{lost}(\omega_1 + C)) + \mathbb{E}(T_{rec}) + \mathbb{E}(T(\mathcal{W}))\Big)$$

where $\omega_1 = f(\mathcal{W})$ and where $\mathbb{E}(T_{lost}(\omega))$ is given by

$$\mathbb{E}(T_{lost}(\omega)) = \frac{1}{p\lambda} - \frac{\omega}{e^{p\lambda\omega} - 1}.$$

The memoryless property makes it possible to solve the MAKESPAN problem analytically:

Theorem 9. *Let $\mathcal{W}$ be the amount of work to execute on p processors whose failure inter-arrival times follow iid Exponential distributions with parameter λ. Let $K_0 = \frac{p\lambda\mathcal{W}}{1+\mathbb{L}(-e^{-p\lambda C-1})}$, where $\mathbb{L}$, the Lambert function, is defined as $\mathbb{L}(z)e^{\mathbb{L}(z)} = z$. Then the optimal strategy to minimize the expected makespan is to split $\mathcal{W}$ into $K^* = \max(1, \lfloor K_0 \rfloor)$ or $K^* = \lceil K_0 \rceil$ same-size chunks, whichever minimizes $\psi(K^*) = K^*(e^{p\lambda(\frac{\mathcal{W}}{K^*}+C)}-1)$.*

The checkpointing strategy in Theorem 9 can be shown to be optimal among all deterministic and non-deterministic strategies, as a consequence of Proposition 4.4.3 in [45].

Interestingly, although we know the optimal solution with p processors, we are not able to compute the optimal expected makespan analytically. Indeed, $\mathbb{E}(T_{rec})$, for which there is a closed form in the case of sequential jobs [12], becomes quite intricate in the case of parallel jobs. This is because during the downtime of a given processor another processor may fail. During the downtime of that processor, yet another processor may fail, and so on. We would need to compute the expected duration of these "cascading" failures until all processors are simultaneously available. However, as the value of $\mathbb{E}(T_{rec})$ is independent of the choice of the chunk sizes, not knowing its value does not forbid to find the optimal chunk size function.

MAKESPAN and Arbitrary Distributions

Solving the MAKESPAN problem for arbitrary distributions is difficult because, unlike in the memoryless case, there is no reason for the optimal solution to use a single chunk size [51]. In fact, the optimal solution is very likely to use chunk sizes that depend on additional information that becomes available during the execution (i.e., failure occurrences to date). For sequential jobs, approximated dynamic programming solutions

have been proposed [12]. They cannot be extended to the parallel case as this would require to memorize the evolution of the time elapsed since the last failure for all possible failure scenarios for each processor, leading to a number of states exponential in p. As we cannot attempt to solve directly MAKESPAN, we introduce a new related optimization problem. This is especially important as failures in real-world computer systems are recognized not to follow Exponential distributions but rather distributions like the Weibull distribution.

0.7.3 Maximizing the Expectation of the Amount of Work Completed

Rather than directly minimizing the job expected makespan, as in the MAKESPAN problem, in the NEXTFAILURE optimization problem we try to maximize the expected amount of work completed before the next failure occurs. NEXTFAILURE amounts to optimizing the makespan on a "failure-by-failure" basis, selecting the next chunk size as if the next failure were to imply termination of the execution. Intuitively, solving NEXTFAILURE should lead to a good approximation of the solution to MAKESPAN, at least for large job sizes $\mathcal{W}$. Therefore, we use the solution of NEXTFAILURE in cases for which we are unable to solve MAKESPAN directly.

Formal Problem Definition

For a given amount of work ω and a set of times elapsed since the last failure τ_1, ..., τ_p, we define $W(\omega|\tau_1, ..., \tau_p)$ as the random variable that quantifies the amount of work successfully executed before the next failure. Given a solution function f, let $\omega_1 = f(\omega|\tau_1, ..., \tau_p)$ denote the size of the first chunk. We can write the following recursion:

$$\begin{aligned} W(0|\tau_1, ..., \tau_p) &= 0 \\ W(\omega|\tau_1, ..., \tau_p) &= \begin{cases} \omega_1 + W(\omega - \omega_1|\tau_1 + \omega_1 + C, ..., \tau_p + \omega_1 + C) \\ \quad \text{if the processor does not fail during the next } \omega_1 + C \text{ units of time,} \\ 0 \quad \text{otherwise.} \end{cases} \end{aligned} \tag{11}$$

This recursion is simpler than the one for MAKESPAN because a failure during the computation of the first chunk means that no work (i.e., no fraction of ω) will have been successfully executed before the next failure. We define NEXTFAILURE formally as: find f that maximizes $\mathbb{E}(W(\mathcal{W}|\tau_1^0, ..., \tau_p^0))$.

Solving NEXTFAILURE

Weighting the two cases in Equation (11) by their probabilities of occurrence, we obtain the expected amount of work successfully computed before the next failure:

$$\mathbb{E}(W(\omega|\tau_1, ..., \tau_p)) = P_{suc}(\omega_1 + C|\tau_1, ..., \tau_p)(\omega_1 + \mathbb{E}(W(\omega - \omega_1|\tau_1 + \omega_1 + C, ..., \tau_p + \omega_1 + C))).$$

Here, unlike for MAKESPAN, the objective function to be maximized can easily be written as a closed form, even for arbitrary distributions. Developing the expression above leads to the following result:

Proposition 3. *The* NEXTFAILURE *problem is equivalent to maximizing the following quantity:*

$$\mathbb{E}(W(\mathcal{W}|\tau_1^0, ..., \tau_p^0)) = \sum_{i=1}^{K} \omega_i \times \prod_{j=1}^{i} P_{suc}(\omega_j + C|t_1^j, ..., t_p^j) \ ,$$

where $t_i^j = \tau_i^0 + \sum_{\ell=1}^{j-1}(\omega_\ell + C)$ is the total time elapsed (without failure) before the start of the execution of chunk ω_j and since the last failure of processor P_i prior to the beginning of the application processing, and K is the (unknown) target number of chunks.

Unfortunately, there does not seem to be an exact solution to this problem. However, the recursive definition of $\mathbb{E}(W(\mathcal{W}|\tau_1, ..., \tau_p))$ lends itself naturally to a dynamic programming algorithm. Formally:

Proposition 4. *Using a time quantum u, and for any failure inter-arrival time distribution,* DPNEXTFAILURE *(Algorithm 1), called with parameters $(\mathcal{W}, \tau_1^0, ..., \tau_p^0)$, computes an optimal solution to* NEXTFAILURE *in time $O(\frac{\mathcal{W}}{u}^3)$.*

Algorithm 1: DPNEXTFAILURE (W,τ_1, ..., τ_p)

```
if W = 0 then return 0
best ← 0
chunksize ← 0
for ω = min{quantum, W} to W step quantum do
    (expected_work, 1st_chunk) ← DPNEXTFAILURE(W − ω, τ_1 + ω + C, ..., τ_p + ω + C)
    cur_exp_work ← P_suc(τ_1 + ω + C, ..., τ_p + ω + C | τ_1, ..., τ_p) × (ω + expected_work)
    if cur_exp_work > best then best ← cur_exp_work
    chunksize ← ω
return (best, chunksize)
```

0.7.4 Conclusion

In this case-study, we have considered the problem of deciding when to save the execution state of an application, i.e., when to take a checkpoint, in order to minimize the expectation of the execution time of an application in a failure-prone environment. We have showed that the optimal solution was to take periodic checkpoints when failures follow an exponential distribution. In the general case, it is not known how to solve the problem analytically. Instead, we have built an approximation of the optimal solution through a dynamic programming algorithm.

We have considered the problem in a simple setting where all checkpointing and recovery times were constant. Furthermore, we have analyzed the simplest checkpointing protocol, namely *coordinated checkpointing*. More complex protocols have been designed in order to (try to) alleviate the cost of the coordination [21, 20].

Rollback-recovery, as studied here, is only one of several fault-tolerance mechanisms. Another often used mechanism is *replication* (see [22] for a coupling of replication with rollback recovery). The question with replication is to decide which task should be replicated and how many times. The optimization objective could be the minimization of the expectation of the execution time (as in this study), or the maximization of the probability that the computation succeeds (reliability), or a multi-criteria optimization mixing reliability and performance.

0.8 Conclusion

In this chapter we have presented several case-studies which we believe to be representative of modern scheduling problems that arise at large-scale. Energy consumption, fault-tolerance, and memory manage-

ment are important objectives that must be taken into account in addition to pure makespan minimization.

The case-studies are also representative of the increased level of difficulty of modern scheduling problems, which typically involve multi-criteria optimization. A variety of mathematical tools and algorithmic techniques are used to solve such problems, and we hope to have convincingly illustrated many of them in this chapter. In addition, many bibliographical items have been provided within the text, and in Section 0.10 below, for further reference.

0.9 Defining Terms

Checkpoint and rollback recovery is the technique of saving the current state of the execution at different points in time; when a failure occurs, the application can rollback to, and recover from, the most recently saved state: execution resumes from that state instead of from the very beginning.

Dynamic voltage and frequency scaling (DVFS): by lowering supply voltage, hence processor clock frequency, it is possible to achieve important reductions in power consumption; faster speeds allow for a faster execution, but also lead to a much higher (supra-linear) power consumption.

Heterogeneity: resources are said to be heterogeneous when they do not have all the exact same characteristics. For instance, the set of processors in a Grid is heterogeneous if they do not all have the same processing capabilities or the same amount of memory.

Makespan: the makespan of an application is the time at which the application execution is completed.

Re-execution: a task whose execution is not reliable enough (for instance, because its execution speed is below a threshold) is executed twice (or more) to enhance its reliability.

Resource selection: is the problem of deciding which, among all available resources, should be used to solve a given problem.

Scheduling: is the problem of deciding at what time (when), and on which resource (where), to execute each of the (atomic) tasks that must be executed on the given platform.

0.10 Further Information

There is a very abundant literature on scheduling and load balancing strategies for parallel and/or distributed systems. The survey [50] is a good starting point to this work. The most famous heuristic to schedule a task graph on an heterogeneous platform is HEFT [52].

A good introduction to multi-criteria optimization is the book by Pinedo [42]. Section 0.4 is mainly based on [7]. Section 0.5 is mainly based on [30]. Liu proposes two optimal algorithms for optimal memory traversal of trees, the first one restricted to postorder traversals [34], the second one for any traversal [35]. For more information on the pebble game model such as complexity results for general DAGs, see the seminal work of Sethi [48] and Gilbert, Lengauer and Tarjan [25]).

The problem discussed in Section 0.6 is mainly based on a recent work by Amestoy et al. [1] The paper include some improved algorithms for the TREEPARTITIONING problem, as well as a hypergraph partitioning based formulation. The postorder-based algorithm discussed in Section 0.6.3 is implemented and incorporated in Mumps [3], a sparse direct solver.

Young is the author of a seminal study of periodic checkpointing policies [56]. His work was later extended by Daly [17]. Section 0.7 is mainly based on [12] which contains references to many studies of checkpointing policies. For non-coordinated checkpointing protocols, [28] by Guermouche et al. is a good starting point.

Bibliography

[1] P. Amestoy, I. S. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. Rouet, and B. Uçar. On computing inverse entries of a sparse matrix in an out-of-core environment. *SIAM Journal on Scientific Computing (SISC)*, 2012. To appear.

[2] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[3] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[4] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[5] I. Assayad, A. Girault, and H. Kalla. Tradeoff exploration between reliability power consumption and execution time. In *Proc. of Conf. on Computer Safety, Reliability and Security (SAFECOMP)*, Washington, DC, USA, 2011. IEEE CS Press.

[6] G. Aupy, A. Benoit, F. Dufossé, and Y. Robert. Reclaiming the energy of a schedule, models and algorithms. *Concurrency and Computation: Practice and Experience*, 2012. Also available as INRIA research report 7598 at `graal.ens-lyon.fr/~abenoit`.

[7] G. Aupy, A. Benoit, and Y. Robert. Energy-aware scheduling under reliability and makespan constraint. Research Report 7757, INRIA, France, Feb. 2012. Available at `graal.ens-lyon.fr/~abenoit`.

[8] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proc. of Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 113–121. IEEE CS Press, 2003.

[9] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proc. of Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, pages 170–177. ACM Press, 2003.

[10] N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM*, 54(1):1 – 39, 2007.

[11] L. Bautista Gomez, A. Nukada, N. Maruyama, F. Cappello, and S. Matsuoka. Transparent low-overhead checkpoint for GPU-accelerated clusters. Presentation at The fourth

workshop of the INRIA-Illinois Joint Laboratory on Petascale Computing. Available at https://wiki.ncsa.illinois.edu/display/jointlab/Workshop+Program.

[12] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *SC'2011, the IEEE/ACM Conference on High Performance Computing Networking, Storage and Analysis*. ACM Press, 2011.

[13] P. Brucker. *Scheduling Algorithms*. Springer, 2007.

[14] H. Casanova, A. Legrand, and Y. Robert. *Parallel Algorithms*. Chapman and Hall/CRC Press, 2008.

[15] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert. Proactive management of software aging. *IBM J. Res. Dev.*, 45(2):311–332, 2001.

[16] J.-J. Chen and T.-W. Kuo. Multiprocessor energy-efficient scheduling for real-time tasks. In *Proc. of Int. Conf. on Parallel Processing (ICPP)*, pages 13–20. IEEE CS Press, 2005.

[17] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *FGCS*, 22(3):303–312, 2004.

[18] V. Degalahal, L. Li, V. Narayanan, M. Kandemir, and M. J. Irwin. Soft errors issues in low-power caches. *IEEE Trans. Very Large Scale Integr. Syst.*, 13:1157–1166, October 2005.

[19] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, and M. Valero. The international exascale software project: A call to cooperative action by the global high-performance community. *Int. J. High Perform. Comput. Appl.*, 23(4):309–322, 2009.

[20] E. Elnozahy and J. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *Dependable and Secure Computing, IEEE Transactions on*, 1(2):97 – 108, april-june 2004.

[21] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey*, 34:375–408, 2002.

[22] K. Ferreira, J. Stearley, J. H. I. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the 2011 ACM/IEEE Conf. on Supercomputing*, 2011.

[23] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[24] J. Gil and A. Itai. How to pack trees. *J. Algorithms*, 32(2):108–132, 1999.

[25] J. R. Gilbert, T. Lengauer, and R. E. Tarjan. The pebbling problem is complete in polynomial space. *SIAM J. Comput.*, 9(3):513–524, 1980.

[26] R. Graham, E. Lawler, J. Lenstra, and A. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[27] R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[28] A. Guermouche, T. Ropars, M. Snir, and F. Cappello. HydEE: Failure containment without event logging for large scale send-deterministic MPI applications. In *Proceedings of IPDPS 2012*. IEEE CS Press, 2012.

[29] B. Hong and V. K. Prasanna. Adaptive allocation of independent tasks to maximize throughput. *IEEE Trans. Parallel Distributed Systems*, 18(10):1420–1435, 2007.

[30] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar. On optimal tree traversals for sparse matrix factorization. In *IPDPS'2011, the 25th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2011.

[31] N. Kolettis and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS '95*, page 381, Washington, DC, USA, 1995. IEEE CS.

[32] C.-C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. *Computer Languages, Systems & Structures*, 37(2):63–75, 2011.

[33] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proc. of Annual Design Automation Conf. (DAC)*, pages 806–809, 2000.

[34] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Software*, 12(3):249–264, 1986.

[35] J. W. H. Liu. An application of generalized tree pebbling to sparse matrix factorization. *SIAM J. Algebraic Discrete Methods*, 8(3), 1987.

[36] S. H. Low. A Duality model of TCP and queue management algorithms. *IEEE/ACM Trans. Networking*, 4(11):525–536, 2003.

[37] L. Massoulié and J. Roberts. Bandwidth sharing: Objectives and algorithms. *Transactions on Networking*, 10(3):320–328, june 2002.

[38] R. Melhem, D. Mosse, and E. Elnozahy. The interplay of power management and fault recovery in real-time systems. *IEEE Trans. on Computers*, 53:2004, 2003.

[39] E. Meneses. Clustering parallel applications to enhance message logging protocols. Presentation at The fourth workshop of the INRIA-Illinois Joint Laboratory on Petascale Computing. Available at http://charm.cs.illinois.edu/talks/ClusteringAppNcsaInriaWorkshop10.pdf.

[40] A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for BlueGene/L systems. In *Proc. of Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 64–73, 2004.

[41] C. Picouleau. Task scheduling with interprocessor communication delays. *Discrete Applied Mathematics*, 60(1-3):331–342, 1995.

[42] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 2008.

[43] P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In *Proc. of IEEE/ACM Int. Conf. on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 233–238, 2007.

[44] R. B. Prathipati. Energy efficient scheduling techniques for real-time embedded systems. Master's thesis, Texas A&M University, May 2004.

[45] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 2005.

[46] V. J. Rayward-Smith, F. W. Burton, and G. J. Janacek. Scheduling parallel programs assuming preallocation. In P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[47] V. Sarkar et al. Exascale software study: Software challenges in extreme scale systems, 2009. White paper available at: `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf`.

[48] R. Sethi. Complete register allocation problems. In *STOC'73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 182–195. ACM Press, 1973.

[49] S. M. Shatz and J.-P. Wang. Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems. *IEEE Transactions on Reliability*, 38:16–27, 1989.

[50] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.

[51] A. Tantawi and M. Ruschitzka. Performance analysis of checkpointing strategies. *ACM TOCS*, 2(2):123–144, 1984.

[52] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Systems*, 13(3):260–274, 2002.

[53] B. Uçar. Partitioning problems on trees and simple meshes. Presentation at 15th SIAM Conference on Parallel Processing for Scientific Computing (PP12), February 2012.

[54] L. Wang, P. Karthik, Z. Kalbarczyk, R. Iyer, L. Votta, C. Vick, and A. Wood. Modeling coordinated checkpointing for large-scale supercomputers. In *Proceedings of ICDSN'05*, pages 812–821, 2005.

[55] L. Wang, G. von Laszewski, J. Dayal, and F. Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with DVFS. In *Proc. of CCGrid'2010, the 10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, pages 368 –377, May 2010.

[56] J. W. Young. A first order approximation to the optimum checkpoint interval. *Comm. of the ACM*, 17(9):530–531, 1974.

[57] D. Zhu and H. Aydin. Energy management for real-time embedded systems with reliability requirements. In *Proc. of IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, pages 528–534, 2006.

[58] D. Zhu, R. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *Proc. of IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, pages 35–40, Washington, DC, USA, 2004. IEEE CS Press.