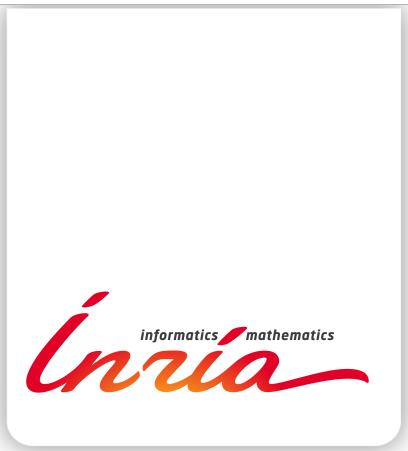
Co-Scheduling Algorithms for High-Throughput Workload Execution

Guillaume Aupy, Manu Shantharam, Anne Benoit, Yves Robert, Padma Raghavan
{guillaume.aupy, anne.benoit, yves.robert}@ens-lyon.fr; shantharam.manu@gmail.com; raghavan@cse.psu.edu

RESEARCH REPORT
N° 8293
April 29, 2013
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8293--FR+ENG



Co-Scheduling Algorithms for High-Throughput Workload Execution

Guillaume Aupy*, Manu Shantharam†, Anne Benoit*‡, Yves Robert*§‡, Padma Raghavan¶
{guillaume.aupy, anne.benoit, yves.robert}@ens-lyon.fr;
shantharam.manu@gmail.com; raghavan@cse.psu.edu

Project-Team ROMA

Research Report n° 8293 — April 29, 2013 — 21 pages

Abstract: This paper investigates co-scheduling algorithms for processing a set of parallel applications. Instead of executing each application one by one, using a maximum degree of parallelism for each of them, we aim at scheduling several applications concurrently. We partition the original application set into a series of packs, which are executed one by one. A pack comprises several applications, each of them with an assigned number of processors, with the constraint that the total number of processors assigned within a pack does not exceed the maximum number of available processors. The objective is to determine a partition into packs, and an assignment of processors to applications, that minimize the sum of the execution times of the packs. We thoroughly study the complexity of this optimization problem, and propose several heuristics that exhibit very good performance on a variety of workloads, whose application execution times model profiles of parallel scientific codes. We show that co-scheduling leads to to faster workload completion time and to faster response times on average (hence increasing system throughput and saving energy), for significant benefits over traditional scheduling from both the user and system perspectives.

Key-words: Scheduling, approximation algorithms, efficient heuristics

* LIP, Ecole Normale Supérieure de Lyon, France
† University of Utah, USA
‡ Institut Universitaire de France, France
§ University of Tennessee Knoxville, USA
¶ Pennsylvania State University, USA

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Algorithmes d'ordonnancement par lots de tâches parallélisables

Résumé : Nous étudions des algorithmes d'ordonnancement par lots pour un ensemble de tâches parallélisables. Plutôt que d'ordonnancer les tâches séquentiellement en utilisant le degré maximum de parallélisme pour chacune d'entre elles, nous cherchons à ordonnancer plusieurs tâches de manière concurrente. L'idée est de partitionner l'ensemble des tâches en une série de lots, qui seront exécutés les uns après les autres. Un lot est composé de plusieurs tâches, et un certain nombre de processeurs est alloué à chacune de ces tâches, sous la contrainte que le nombre total de processeurs alloués à un lot ne dépasse pas le nombre maximum de processeurs disponibles. L'objectif est de trouver une partition de l'ensemble des tâches en lots, et une allocation de processeurs optimale pour la minimisation de la somme des temps d'exécution de chaque lot.

Nous étudions la complexité de ce problème d'optimisation, et proposons plusieurs heuristiques qui obtiennent d'excellentes performances sur plusieurs ensembles de tâches représentatifs des calculs parallèles scientifiques. Notre technique d'ordonnancement par lots permet non seulement d'obtenir un gain sur le temps d'exécution total, mais obtient également d'excellents résultats pour le temps de réponse moyen.

Mots-clés : Ordonnancement, algorithmes d'approximation, heuristiques

1 Introduction

The execution time of many high-performance computing applications can be significantly reduced when using a large number of processors. Indeed, parallel multicore platforms enable the fast processing of very large size jobs, thereby rendering the solution of challenging scientific problems more tractable. However, monopolizing all computing resources to accelerate the processing of a single application is very likely to lead to inefficient resource usage. This is because the typical speed-up profile of most applications is sub-linear and even reaches a threshold: when the number of processors increases, the execution time first decreases, but not linearly, because it suffers from the overhead due to communications and load imbalance; at some point, adding more resources does not lead to any significant benefit.

In this paper, we consider a pool of several applications that have been submitted for execution. Rather than executing each of them in sequence, with the maximum number of available resources, we introduce co-scheduling algorithms that execute several applications concurrently. We do increase the individual execution time of each application, but (i) we improve the efficiency of the parallelization, because each application is scheduled on fewer resources; (ii) the total execution time will be much shorter; and (iii) the average response time will also be shorter. In other words, co-scheduling increases platform yield (thereby saving energy) without sacrificing response time.

In operating high performance computing systems, the costs of energy consumption can greatly impact the total costs of ownership. Consequently, there is a move away from a focus on peak performance (or speed) and towards improving energy efficiency [12, 20]. Recent results on improving the energy efficiency of workloads can be broadly classified into approaches that focus on dynamic voltage and frequency scaling, or alternatively, task aggregation or co-scheduling. In both types of approaches, the individual execution time of an application may increase but there can be considerable energy savings in processing a workload.

More formally, we deal with the following problem: given (i) a distributed-memory platform with p processors, and (ii) n applications, or tasks, T_i, with their execution profiles ($t_{i,j}$ is the execution time of T_i with j processors), what is the best way to *co-schedule* them, i.e., to partition them into packs, so as to minimize the sum of the execution times over all packs. Here a pack is a subset of tasks, together with a processor assignment for each task. The constraint is that the total number of resources assigned to the pack does not exceed p, and the execution time of the pack is the longest execution time of a task within that pack. The objective of this paper is to study this co-scheduling problem, both theoretically and experimentally, We aim at demonstrating the gain that can be achieved through co-scheduling, both on platform yield and response time, using a set of real-life application profiles.

On the theoretical side, to the best of our knowledge, the complexity of the co-scheduling problem has never been investigated, except for the simple case when one enforces that each pack comprises at most $k = 2$ tasks [21]. While the problem has polynomial complexity for the latter restriction (with at most $k = 2$ tasks per pack), we show that it is NP-complete when assuming at most $k \geq 3$ tasks per pack. Note that the instance with $k = p$ is the general, unconstrained, instance of the co-scheduling problem. We also propose an approximation algorithm for the general instance. In addition, we propose an optimal processor assignment procedure when the tasks that form a pack are given. We use these two results to derive efficient heuristics. Finally, we discuss how to optimally solve small-size instances, either through enumerating partitions, or through an integer linear program: this has a potentially exponential cost,

but allows us to assess the absolute quality of the heuristics that we have designed. Altogether, all these results lay solid theoretical foundations for the problem.

On the experimental side, we study the performance of the heuristics on a variety of workloads, whose application execution times model profiles of parallel scientific codes. We focus on three criteria: (i) cost of the co-schedule, i.e., total execution time; (ii) packing ratio, which evaluates the idle time of processors during execution; and (iii) response time compared to a fully parallel execution of each task starting from shortest task. The proposed heuristics show very good performance within a short running time, hence validating the approach.

The paper is organized as follows. We discuss related work in Section 2. The problem is then formally defined in Section 3. Theoretical results are presented in Section 4, exhibiting the problem complexity, discussing sub-problems and optimal solutions, and providing an approximation algorithm. Building upon these results, several polynomial-time heuristics are described in Section 5, and they are thoroughly evaluated in Section 6. Finally we conclude and discuss future work in Section 7.

2 Related work

In this paper, we deal with pack scheduling for parallel tasks, aiming at makespan minimization (recall that the makespan is the total execution time). The corresponding problem with sequential tasks (tasks that execute on a single processor) is easy to solve for the makespan minimization objective: simply make a pack out of the largest p tasks, and proceed likewise while there remain tasks. Note that the pack scheduling problem with sequential tasks has been widely studied for other objective functions, see Brucker et al. [4] for various job cost functions, and Potts and Kovalyov [18] for a survey. Back to the problem with sequential tasks and the makespan objective, Koole and Righter in [13] deal with the case where the execution time of each task is unknown but defined by a probabilistic distribution. They showed counter-intuitive properties, that enabled them to derive an algorithm that computes the optimal policy when there are two processors, improving the result of Deb and Serfozo [7], who considered the stochastic problem with identical jobs.

To the best of our knowledge, the problem with parallel tasks has not been studied as such. However, it was introduced by Dutot et al. in [8] as a moldable-by-phase model to approximate the moldable problem. The moldable task model is similar to the pack-scheduling model, but one does not have the additional constraint (pack constraint) that the execution of new tasks cannot start before all tasks in the current pack are completed. Dutot et al. in [8] provide an optimal polynomial-time solution for the problem of pack scheduling identical independent tasks, using a dynamic programming algorithm. This is the only instance of pack-scheduling with parallel tasks that we found in the literature.

A closely related problem is the rectangle packing problem, or 2D-Strip-packing. Given a set of rectangles of different sizes, the problem consists in packing these rectangles into another rectangle of size $p \times m$. If one sees one dimension (p) as the number of processors, and the other dimension (m) as the maximum makespan allowed, this problem is identical to the variant of our problem where the number of processors is pre-assigned to each task: each rectangle r_i of size $p_i \times m_i$ that has to be packed can be seen as the task T_i to be computed on p_i processors, with $t_{i,p_i} = m_i$. In [22], Turek et al. approximated the rectangle packing problem using *shelf-based* solutions: the rectangles are assigned to *shelves*, whose placements correspond to constant time

values. All rectangles assigned to a shelf have equal starting times, and the next shelf is placed on top of the previous shelf. This is exactly what we ask in our pack-scheduling model. This problem is also called level packing in some papers, and we refer the reader to a recent survey on 2D-packing algorithms by Lodi et al. [16]. In particular, Coffman et al. in [6] show that level packing algorithm can reach a 2.7 approximation for the 2D-Strip-packing problem (1.7 when the length of each rectangle is bounded by 1). Unfortunately, all these algorithms consider the number of processors (or width of the rectangles) to be already fixed for each task, hence they cannot be used directly in our problem for which a key decision is to decide the number of processors assigned to each task.

In practice, pack scheduling is really useful as shown by recent results. Li et al. [15] propose a framework to predict the energy and performance impacts of power-aware MPI task aggregation. Frachtenberg et al. [9] show that system utilization can be improved through their schemes to co-schedule jobs based on their load-balancing requirements and inter-processor communication patterns. In our earlier work [21], we had shown that even when the pack-size is limited to 2, co-scheduling based on speed-up profiles can lead to faster workload completion and corresponding savings in system energy.

Several recent publications [2, 5, 11] consider co-scheduling at a single multicore node, when contention for resources by co-scheduled tasks leads to complex tradeoffs between energy and performance measures. Chandra et al. [5] predict and utilize inter-thread cache contention at a multicore in order to improve performance. Hankendi and Coskun [11] show that there can be measurable gains in energy per unit of work through the application of their multi-level co-scheduling technique at runtime which is based on classifying tasks according to specific performance measures. Bhaduria and McKee [2] consider local search heuristics to co-schedule tasks in a resource-aware manner at a multicore node to achieve significant gains in thread throughput per watt.

These publications demonstrate that complex tradeoffs cannot be captured through the use of the speed-up measure alone, without significant additional measurements to capture performance variations from cross-application interference at a multicore node. Additionally, as shown in our earlier work [21], we expect significant benefits even when we aggregate only across multicore nodes because speed-ups suffer due to of the longer latencies of data transfer across nodes. We can therefore project savings in energy as being commensurate with the savings in the time to complete a workload through co-scheduling. Hence, we only test configurations where no more than a single application can be scheduled on a multicore node.

3 Problem definition

The application consists of n independent tasks $T_1, \ldots, T_n$. The target execution platform consists of p identical processors, and each task T_i can be assigned an arbitrary number $\sigma(i)$ of processors, where $1 \leq \sigma(i) \leq p$. The objective is to minimize the total execution time by co-scheduling several tasks onto the p resources. Note that the approach is agnostic of the granularity of each processor, which can be either a single CPU or a multicore node.

Speedup profiles – Let $t_{i,j}$ be the execution time of task T_i with j processors, and $work(i,j) = j \times t_{i,j}$ be the corresponding work. We assume the following for

$1 \leq i \leq n$ and $1 \leq j < p$:

$$\text{Non-increasing execution time: } t_{i,j+1} \leq t_{i,j} \tag{1}$$

$$\text{Non-decreasing work: } work(i,j) \leq work(i,j+1) \tag{2}$$

Equation (1) implies that execution time is a non-increasing function of the number of processors. Equation (2) states that efficiency decreases with the number of enrolled processors: in other words, parallelization has a cost! As a side note, we observe that these requirements make good sense in practice: many scientific tasks T_i are such that $t_{i,j}$ first decreases (due to load-balancing) and then increases (due to communication overhead), reaching a minimum for $j = j_0$; we can always let $t_{i,j} = t_{i,j_0}$ for $j \geq j_0$ by never actually using more than j_0 processors for T_i.

Co-schedules – A co-schedule partitions the n tasks into groups (called *packs*), so that (i) all tasks from a given pack start their execution at the same time; and (ii) two tasks from different packs have disjoint execution intervals. See Figure 1 for an example. The execution time, or *cost*, of a pack is the maximal execution time of a task in that pack, and the cost of a co-schedule is the sum of the costs of each pack.

k-IN-p-CoSchedule optimization problem – Given a fixed constant $k \leq p$, find a co-schedule with at most k tasks per pack that minimizes the execution time. The most general problem is when $k = p$, but in some frameworks we may have an upper bound $k < p$ on the maximum number of tasks within each pack.

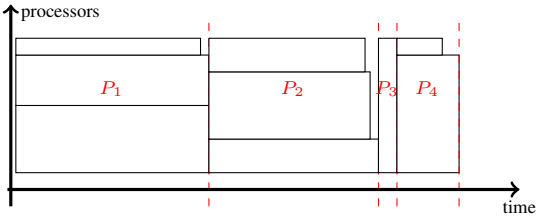


Figure 1: A co-schedule with four packs P_1 to P_4.

4 Theoretical results

First we discuss the complexity of the problem in Section 4.1, by exhibiting polynomial and NP-complete instances. Next we discuss how to optimally schedule a set of k tasks in a single pack (Section 4.2). Then we explain how to compute the optimal solution (in expected exponential cost) in Section 4.3. Finally, we provide an approximation algorithm in Section 4.4.

4.1 Complexity

Theorem 1. *The* 1-IN-p-CoSchedule *and* 2-IN-p-CoSchedule *problems can both be solved in polynomial time.*

Proof. This result is obvious for 1-IN-p-CoSchedule: each task is assigned exactly p processors (see Equation (1)) and the minimum execution time is $\sum_{i=1}^{n} t_{i,p}$.

This proof is more involved for 2-IN-p-CoSchedule, and we start with the 2-IN-2-CoSchedule problem to get an intuition. Consider the weighted undirected graph

$G = (V, E)$, where $|V| = n$, each vertex $v_i \in V$ corresponding to a task T_i. The edge set E is the following: (i) for all i, there is a loop on v_i of weight $t_{i,2}$; (ii) for all $i < i'$, there is an edge between v_i and $v_{i'}$ of weight $\max(t_{i,1}, t_{i',1})$. Finding a perfect matching of minimal weight in G leads to the optimal solution to 2-IN-2-COSCHEDULE, which can thus be solved in polynomial time.

For the 2-IN-p-COSCHEDULE problem, the proof is similar, the only difference lies in the construction of the edge set E: (i) for all i, there is a loop on v_i of weight $t_{i,p}$; (ii) for all $i < i'$, there is an edge between v_i and $v_{i'}$ of weight $\min_{j=1..p}(\max(t_{i,p-j}, t_{i',j}))$. Again, a perfect matching of minimal weight in G gives the optimal solution to 2-IN-p-COSCHEDULE. We conclude that the 2-IN-p-COSCHEDULE problem can be solved in polynomial time. □

Theorem 2. *When $k \geq 3$, the k-IN-p-COSCHEDULE problem is strongly NP-complete.*

Proof. We prove the NP-completeness of the decision problem associated to k-IN-p-COSCHEDULE: given n independent tasks, p processors, a set of execution times $t_{i,j}$ for $1 \leq i \leq n$ and $1 \leq j \leq p$ satisfying Equations (1) and (2), a fixed constant $k \leq p$ and a deadline D, can we find a co-schedule with at most k tasks per pack, and whose execution time does not exceed D? The problem is obviously in NP: if we have the composition of every pack, and for each task in a pack, the number of processors onto which it is assigned, we can verify in polynomial time: (i) that it is indeed a pack schedule; (ii) that the execution time is smaller than a given deadline.

We first prove the strong completeness of 3-IN-p-COSCHEDULE. We use a reduction from 3-PARTITION. Consider an arbitrary instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3n$ integers $a_1, \ldots, a_{3n}$, can we partition the $3n$ integers into n triplets, each of sum B? We can assume that $\sum_{i=1}^{3n} a_i = nB$, otherwise $\mathcal{I}_1$ has no solution. The 3-PARTITION problem is NP-hard in the strong sense [10], which implies that we can encode all integers $(a_1, \ldots, a_{3n}, B)$ in unary. We build the following instance $\mathcal{I}_2$ of 3-IN-p-COSCHEDULE: the number of processors is $p = B$, the deadline is $D = n$, there are $3n$ tasks T_i, with the following execution times: for all i, j, if $j < a_i$ then $t_{i,j} = 1 + \frac{1}{a_i}$, otherwise $t_{i,j} = 1$. It is easy to check that Equations (1) and (2) are both satisfied. For the latter, since there are only two possible execution times for each task, we only need to check Equation (2) for $j = a_i - 1$, and we do obtain that $(a_i - 1)(1 + \frac{1}{a_i}) \leq a_i$. Finally, $\mathcal{I}_2$ has a size polynomial in the size of $\mathcal{I}_1$, even if we write all instance parameters in unary: the execution time is n, and the $t_{i,j}$ have the same size as the a_i.

We now prove that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does. Assume first that $\mathcal{I}_1$ has a solution. For each triplet (a_i, a_j, a_k) of $\mathcal{I}_1$, we create a pack with the three tasks (T_i, T_j, T_k) where T_i is scheduled on a_i processors, T_j on a_j processors, and T_k on a_k processors. By definition, we have $a_i + a_j + a_k = B$, and the execution time of this pack is 1. We do this for the n triplets, which gives a valid co-schedule whose total execution time n. Hence the solution to $\mathcal{I}_2$.

Assume now that $\mathcal{I}_2$ has a solution. The minimum execution time for any pack is 1 (since it is the minimum execution time of any task and a pack cannot be empty). Hence the solution cannot have more than n packs. Because there are $3n$ tasks and the number of elements in a pack is limited to three, there are exactly n packs, each of exactly 3 elements, and furthermore all these packs have an execution time of 1 (otherwise the deadline n is not matched). If there were a pack (T_i, T_j, T_k) such that $a_i + a_j + a_k > B$, then one of the three tasks, say T_i, would have to use fewer than a_i processors, hence would have an execution time greater than 1. Therefore, for each

pack (T_i, T_j, T_k), we have $a_i + a_j + a_k \leq B$. The fact that this inequality is an equality for all packs follows from the fact that $\sum_{i=1}^{3n} a_i = nB$. Finally, we conclude by saying that the set of triplets (a_i, a_j, a_k) for every pack (T_i, T_j, T_k) is a solution to $\mathcal{I}_1$.

The final step is to prove the completeness of k-IN-p-COSCHEDULE for a given $k \geq 4$. We perform a similar reduction from the same instance $\mathcal{I}_1$ of 3-PARTITION. We construct the instance $\mathcal{I}_2$ of k-IN-p-COSCHEDULE where the number of processors is $p = B+(k-3)(B+1)$ and the deadline is $D = n$. There are $3n$ tasks T_i with the same execution times as before (for $1 \leq i \leq 3n$, if $j < a_i$ then $t_{i,j} = 1 + \frac{1}{a_i}$, otherwise $t_{i,j} = 1$), and also $n(k-3)$ new identical tasks such that, for $3n+1 \leq i \leq kn$, $t_{i,j} = \max\left(\frac{B+1}{j}, 1\right)$. It is easy to check that Equations (1) and (2) are also fulfilled for the new tasks. If $\mathcal{I}_1$ has a solution, we construct the solution to $\mathcal{I}_2$ similarly to the previous reduction, and we add to each pack $k-3$ tasks T_i with $3n+1 \leq i \leq kn$, each assigned to $B+1$ processors. This solution has an execution time exactly equal to n. Conversely, if $\mathcal{I}_2$ has a solution, we can verify that there are exactly n packs (there are kn tasks and each pack has an execution time at least equal to 1). Then we can verify that there are at most $(k-3)$ tasks T_i with $3n+1 \leq i \leq kn$ per pack, since there are exactly $(k-3)(B+1)+B$ processors. Otherwise, if there were $k-2$ (or more) such tasks in a pack, then one of them would be scheduled on less than $B+1$ processors, and the execution time of the pack would be greater than 1. Finally, we can see that in $\mathcal{I}_2$, each pack is composed of $(k-3)$ tasks T_i with $3n+1 \leq i \leq kn$, scheduled on $(k-3)(B+1)$ processors at least, and that there remains triplets of tasks T_i, with $1 \leq i \leq 3n$, scheduled on at most B processors. The end of the proof is identical to the reduction in the case $k = 3$. □

Note that the 3-IN-p-COSCHEDULE problem is NP-complete, and the 2-IN-p-COSCHEDULE problem can be solved in polynomial time, hence 3-IN-3-COSCHEDULE is the simplest problem whose complexity remains open.

4.2 Scheduling a pack of tasks

In this section, we discuss how to optimally schedule a set of k tasks in a single pack: the k tasks $T_1, \ldots, T_k$ are given, and we search for an assignment function $\sigma : \{1, \ldots, k\} \to \{1, \ldots, p\}$ such that $\sum_{i=1}^{k} \sigma(i) \leq p$, where $\sigma(i)$ is the number of processors assigned to task T_i. Such a schedule is called a 1-pack-schedule, and its *cost* is $\max_{1 \leq i \leq k} t_{i,\sigma(i)}$. In Algorithm 1 below, we use the notation $T_i \preccurlyeq_\sigma T_j$ if $t_{i,\sigma(i)} \leq t_{j,\sigma(j)}$:

Theorem 3. *Given k tasks to be scheduled on p processors in a single pack, Algorithm 1 finds a 1-pack-schedule of minimum cost in time $O(p \log(k))$.*

In this greedy algorithm, we first assign one processor to each task, and while there are processors that are not processing any task, we select the task with the longest execution time and assign an extra processor to this task. Algorithm 1 performs $p-k$ iterations to assign the extra processors. We denote by $\sigma^{(\ell)}$ the current value of the function σ at the end of iteration ℓ. For convenience, we let $t_{i,0} = +\infty$ for $1 \leq i \leq k$. We start with the following lemma:

Lemma: At the end of iteration ℓ of Algorithm 1, let $T_{i^\star}$ be the first task of the sorted list, i.e., the task with longest execution time. Then, for all i, $t_{i^\star,\sigma^{(\ell)}(i^\star)} \leq t_{i,\sigma^{(\ell)}(i)-1}$.

Proof. Let $T_{i^\star}$ be the task with longest execution time at the end of iteration ℓ. For tasks such that $\sigma^{(\ell)}(i) = 1$, the result is obvious since $t_{i,0} = +\infty$. Let us consider any

Algorithm 1: Finding the optimal 1-pack-schedule σ of k tasks in the same pack.

```
procedure Optimal-1-pack-schedule(T_1, ..., T_k)
begin
    for i = 1 to k do
        σ(i) ← 1
    end
    Let L be the list of tasks sorted in non-increasing values of ≼_σ ;
    p_available := p − k;
    while p_available ≠ 0 do
        T_i* := head(L);
        L := tail(L);
        σ(i*) ← σ(i*) + 1;
        p_available := p_available − 1;
        L := Insert T_i* in L according to its ≼_σ value;
    end
    return σ;
end
```

task T_i such that $\sigma^{(\ell)}(i) > 1$. Let $\ell' + 1$ be the last iteration when a new processor was assigned to task T_i: $\sigma^{(\ell')}(i) = \sigma^{(\ell)}(i) - 1$ and $\ell' < \ell$. By definition of iteration $\ell' + 1$, task T_i was chosen because $t_{i,\sigma^{(\ell')}(i)}$ was greater than any other task, in particular $t_{i,\sigma^{(\ell')}(i)} \geq t_{i^\star,\sigma^{(\ell')}(i^\star)}$. Also, since we never remove processors from tasks, we have $\sigma^{(\ell')}(i) \leq \sigma^{(\ell)}(i)$ and $\sigma^{(\ell')}(i^\star) \leq \sigma^{(\ell)}(i^\star)$. Finally, $t_{i^\star,\sigma^{(\ell)}(i^\star)} \leq t_{i^\star,\sigma^{(\ell')}(i^\star)} \leq t_{i,\sigma^{(\ell')}(i)} = t_{i,\sigma^{(\ell)}(i)-1}$. □

We are now ready to prove Theorem 3.

of Theorem 3. Let σ be the 1-pack-schedule returned by Algorithm 1 of cost $c(\sigma)$, and let $T_{i^\star}$ be a task such that $c(\sigma) = t_{i^\star,\sigma(i^\star)}$. Let σ' be a 1-pack-schedule of cost $c(\sigma')$. We prove below that $c(\sigma') \geq c(\sigma)$, hence σ is a 1-pack-schedule of minimum cost:

- If $\sigma'(i^\star) \leq \sigma(i^\star)$, then $T_{i^\star}$ has fewer processors in σ' than in σ, hence its execution time is larger, and $c(\sigma') \geq c(\sigma)$.
- If $\sigma'(i^\star) > \sigma(i^\star)$, then there exists i such that $\sigma'(i) < \sigma(i)$ (since the total number of processors is p in both σ and σ'). We can apply the previous Lemma at the end of the last iteration, where $T_{i^\star}$ is the task of maximum execution time: $t_{i^\star,\sigma(i^\star)} \leq t_{i,\sigma(i)-1} \leq t_{i,\sigma'(i)}$, and therefore $c(\sigma') \geq c(\sigma)$.

Finally, the time complexity is obtained as follows: first we sort k elements, in time $O(k \log k)$. Then there are $p - k$ iterations, and at each iteration, we insert an element in a sorted list of $k - 1$ elements, which takes $O(\log k)$ operations (use a heap for the data structure of L). □

Note that it is easy to compute an optimal 1-pack-schedule using a dynamic-programming algorithm: the optimal cost is $c(k, p)$, which we compute using the recurrence formula

$$c(i, q) = \min_{1 \leq q' \leq q} \{\max(c(i-1, q-q'), t_{i,q'})\}$$

for $2 \leq i \leq k$ and $1 \leq q \leq p$, initialized by $c(1, q) = t_{1,q}$, and $c(i, 0) = +\infty$. The complexity of this algorithm is $O(kp^2)$. However, we can significantly reduce the complexity of this algorithm by using Algorithm 1.

4.3 Computing the optimal solution

In this section we sketch two methods to find the optimal solution to the general k-IN-p-CoSCHEDULE problem. This can be useful to solve some small-size instances, albeit at the price of a cost exponential in the number of tasks n.

The first method is to generate all possible partitions of the tasks into packs. This amounts to computing all partitions of n elements into subsets of cardinal at most k. For a given partition of tasks into packs, we use Algorithm 1 to find the optimal processor assignment for each pack, and we can compute the optimal cost for the partition. There remains to take the minimum of these costs among all partitions.

The second method is to cast the problem in terms of an integer linear program:

Theorem 4. *The following integer linear program characterizes the k-IN-p-CoSCHEDULE problem, where the unknown variables are the $x_{i,j,b}$'s (Boolean variables) and the y_b's (rational variables), for $1 \leq i, b \leq n$ and $1 \leq j \leq p$:*

$$\begin{array}{ll} \textit{Minimize } \sum_{b=1}^{n} y_b & \textit{subject to} \\ \textit{(i)} \ \sum_{j,b} x_{i,j,b} = 1, & 1 \leq i \leq n \\ \textit{(ii)} \ \sum_{i,j} x_{i,j,b} \leq k, & 1 \leq b \leq n \\ \textit{(iii)} \ \sum_{i,j} j \times x_{i,j,b} \leq p, & 1 \leq b \leq n \\ \textit{(iv)} \ x_{i,j,b} \times t_{i,j} \leq y_b, & 1 \leq i, b \leq n, 1 \leq j \leq p \end{array} \tag{3}$$

Proof. The $x_{i,j,b}$'s are such that $x_{i,j,b} = 1$ if and only if task T_i is in the pack b and it is executed on j processors; y_b is the execution time of pack b. Since there are no more than n packs (one task per pack), $b \leq n$. The sum $\sum_{b=1}^{n} y_b$ is therefore the total execution time ($y_b = 0$ if there are no tasks in pack b). Constraint (i) states that each task is assigned to exactly one pack b, and with one number of processors j. Constraint (ii) ensures that there are not more than k tasks in a pack. Constraint (iii) adds up the number of processors in pack b, which should not exceed p. Finally, constraint (iv) computes the cost of each pack. □

4.4 Approximation algorithm

In this section we introduce PACK-APPROX, a 3-approximation algorithm for the p-IN-p-CoSCHEDULE problem. The design principle of PACK-APPROX is the following: we start from the assignment where each task is executed on one processor, and use Algorithm 2 to build a first solution. Algorithm 2 is a greedy heuristic that builds a co-schedule when each task is pre-assigned a number of processors for execution. Then we iteratively refine the solution, adding a processor to the task with longest execution time, and re-executing Algorithm 2. Here are details on both algorithms:

Algorithm 2. The k-IN-p-CoSCHEDULE problem with processor pre-assignments remains strongly NP-complete (use a similar reduction as in the proof of Theorem 2). We propose a greedy procedure in Algorithm 2 which is similar to the First Fit Decreasing Height algorithm for strip packing [6]. The output is a co-schedule with at most k tasks per pack, and the complexity is $O(n \log(n))$ (dominated by sorting).

Algorithm 3. We iterate the calls to Algorithm 2, adding a processor to the task with longest execution time, until: (i) either the task of longest execution time is already assigned p processors, or (ii) the sum of the work of all tasks is greater than p times the longest execution time. The algorithm returns the minimum cost found during execution. The complexity of this algorithm is $O(n^2 p)$ (in the calls to Algorithm 2 we do not need to re-sort the list but maintain it sorted instead) in the simplest version

presented here, but can be reduced to $O(n\log(n) + np)$ using standard algorithmic techniques.

Algorithm 2: Creating packs of size at most k, when the number $\sigma(i)$ of processors per task T_i is fixed.

```
procedure MAKE-PACK(n, p, k, σ)
begin
    Let L be the list of tasks sorted in non-increasing values of execution times t_{i,σ(i)};
    while L ≠ ∅ do
        Schedule the current task on the first pack with enough available processors and fewer than k tasks. Create a new pack if no existing pack fits;
        Remove the current task from L;
    end
    return the set of packs
end
```

Algorithm 3: PACK-APPROX

```
procedure PACK-APPROX(T_1, ..., T_n)
begin
    COST = +∞ ;
    for j = 1 to n do σ(j) ← 1;
    for i = 0 to n(p − 1) − 1 do
        Let A_tot(i) = Σ_{j=1}^{n} t_{j,σ(j)} σ(j);
        Let T_{j*} be one task that maximizes t_{j,σ(j)};
        Call MAKE-PACK (n, p, p, σ);
        Let COST_i be the cost of the co-schedule;
        if COST_i < COST then COST ← COST_i;
        if ( A_tot(i)/p > t_{j*,σ(j*)} ) or (σ(j*) = p) then return COST; /* Exit
            loop                                                          */
        else σ(j*) ← σ(j*) + 1; /* Add a processor to T_{j*}                 */
    end
    return COST;
end
```

Theorem 5. PACK-APPROX *is a 3-approximation algorithm for the* p-IN-p-COSCHEDULE *problem.*

Proof. We start with some notations:

- step i denotes the i^{th} iteration of the main loop of Algorithm PACK-APPROX;
- $\sigma^{(i)}$ is the allocation function at step i;
- $t_{\max}(i) = \max_j t_{j,\sigma^{(i)}(j)}$ is the maximum execution time of any task at step i;
- $j^\star(i)$ is the index of the task with longest execution time at step i (break ties arbitrarily);
- $A_{\text{tot}}(i) = \sum_j t_{j,\sigma^{(i)}(j)}\sigma^{(i)}(j)$ is the total work that has to be done at step i;

- COST_i is the result of the scheduling procedure at the end of step i;
- OPT denotes an optimal solution, with allocation function $\sigma^{(\text{OPT})}$, execution time COST_{OPT}, and total work
$$A_{\text{OPT}} = \sum_j t_{j,\sigma^{(\text{OPT})}(j)} \sigma^{(\text{OPT})}(j).$$

Note that there are three different ways to exit algorithm PACK-APPROX:

1. If we cannot add processors to the task with longest execution time, i.e., $\sigma^{(i)}(j^\star(i)) = p$;
2. If $\frac{A_{\text{tot}}(i)}{p} > t_{\max}(i)$ after having computed the execution time for this assignment;
3. When each task has been assigned p processors (the last step of the loop "for": we have assigned exactly np processors, and no task can be assigned more than p processors).

Lemma 1. *At the end of step i,* $COST_i \leq 3\max\left(t_{\max}(i), \frac{A_{tot}(i)}{p}\right)$.

Proof. Consider the packs returned by Algorithm 2, sorted by non-increasing execution times, $B_1, B_2, \ldots, B_n$ (some of the packs may be empty, with an execution time 0). Let us denote, for $1 \leq q \leq n$,

- j_q the task with the longest execution time of pack B_q (i.e., the first task scheduled on B_q);
- t_q the execution time of pack B_q (in particular, $t_q = t_{j_q,\sigma^{(i)}(j_q)}$);
- A_q the sum of the task works in pack B_q;
- p_q the number of processors available in pack B_q when j_{q+1} was scheduled in pack B_{q+1}.

With these notations, $\text{COST}_i = \sum_{q=1}^{n} t_q$ and $A_{\text{tot}}(i) = \sum_{q=1}^{n} A_q$. For each pack, note that $pt_q \geq A_q$, since pt_q is the maximum work that can be done on p processors with an execution time of t_q. Hence, $\text{COST}_i \geq \frac{A_{\text{tot}}(i)}{p}$.

In order to bound COST_i, let us first remark that $\sigma^{(i)}(j_{q+1}) > p_q$: otherwise j_{q+1} would have been scheduled on pack B_q. Then, we can exhibit a lower bound for A_q, namely $A_q \geq t_{q+1}(p - p_q)$. Indeed, the tasks scheduled before j_{q+1} all have a length greater than t_{q+1} by definition. Furthermore, obviously $A_{q+1} \geq t_{q+1} p_q$ (the work of the first task scheduled in pack B_{q+1}). So finally we have, $A_q + A_{q+1} \geq t_{q+1} p$.

Summing over all q's, we have: $2\sum_{q=1}^{n} \frac{A_q}{p} \geq \sum_{q=2}^{n} t_q$, hence $2\frac{A_{\text{tot}}(i)}{p} + t_1 \geq \text{COST}_i$. Finally, note that $t_1 = t_{\max}(i)$, and therefore $\text{COST}_i \leq 3 \max\left(t_{\max}(i), \frac{A_{\text{tot}}(i)}{p}\right)$. Note that this proof is similar to the one for the Strip-Packing problem in [6]. □

Lemma 2. *At each step i,* $A_{tot}(i+1) \geq A_{tot}(i)$ *and* $t_{\max}(i+1) \leq t_{\max}(i)$*, i.e., the total work is increasing and the maximum execution time is decreasing.*

Proof. $A_{\text{tot}}(i+1) = A_{\text{tot}}(i) - a + b$, where

- $a = work(j^\star(i), \sigma^{(i)}(j^\star(i)))$, and
- $b = work(j^\star(i), \sigma^{(i+1)}(j^\star(i)))$.

But $b = work(j^\star(i), \sigma^{(i)}(j^\star(i)) + 1)$and $a \leq b$ by Equation (2). Therefore, $A_{\text{tot}}(i+1) \geq A_{\text{tot}}(i)$. Finally, $t_{\max}(i+1) \leq t_{\max}(i)$ since only one of the tasks with the longest execution time is modified, and its execution time can only decrease thanks to Equation (1). □

Lemma 3. *Given an optimal solution* OPT, $\forall j, t_{j,\sigma^{(\text{OPT})}(j)} \leq COST_{\text{OPT}}$ *and* $A_{\text{OPT}} \leq pCOST_{\text{OPT}}$.

Proof. The first inequality is obvious. As for the second one, $p\text{COST}_{\text{OPT}}$ is the maximum work that can be done on p processors within an execution time of COST_{OPT}, hence it must not be smaller than A_{OPT}, which is the sum of the work of the tasks with the optimal allocation. □

Lemma 4. *For any step i such that $t_{\max}(i) > \text{COST}_{\text{OPT}}$, then $\forall j, \sigma^{(i)}(j) \leq \sigma^{(\text{OPT})}(j)$, and $A_{tot}(i) \leq A_{\text{OPT}}$.*

Proof. Consider a task T_j. If $\sigma^{(i)}(j) = 1$, then clearly $\sigma^{(i)}(j) \leq \sigma^{(\text{OPT})}(i)$. Otherwise, $\sigma^{(i)}(j) > 1$, and then by definition of the algorithm, there was a step $i' < i$, such that $\sigma^{(i')}(j) = \sigma^{(i)}(j) - 1$ and $\sigma^{(i'+1)}(j) = \sigma^{(i)}(j)$. Therefore $t_{\max}(i') = t_{j,\sigma^{(i')}(j)}$. Following Lemma 2, we have $t_{\max}(i') \geq t_{\max}(i) > \text{COST}_{\text{OPT}}$. Then necessarily, $\sigma^{(\text{OPT})}(j) > \sigma^{(i')}(j)$, hence the result. Finally, $A_{\text{tot}}(i) \leq A_{\text{OPT}}$ is a simple corollary of the previous result and of Equation (2). □

Lemma 5. *For any step i such that $t_{\max}(i) > \text{COST}_{\text{OPT}}$, then $\frac{A_{tot}(i)}{p} < t_{\max}(i)$.*

Proof. Thanks to Lemma 4, we have $\frac{A_{\text{tot}}(i)}{p} \leq \frac{A_{\text{OPT}}}{p}$. Lemma 3 gives us $\frac{A_{\text{OPT}}}{p} \leq \text{COST}_{\text{OPT}}$, hence the result. □

Lemma 6. *There exists $i_0 \geq 0$ such that $t_{\max}(i_0 - 1) > \text{COST}_{\text{OPT}} \geq t_{\max}(i_0)$ (we let $t_{\max}(-1) = +\infty$).*

Proof. We show this result by contradiction. Suppose such i_0 does not exist. Then $t_{\max}(0) > \text{COST}_{\text{OPT}}$ (otherwise $i_0 = 0$ would suffice). Let us call i_1 the last step of the run of the algorithm. Then by induction we have the following property, $t_{\max}(0) \geq t_{\max}(1) \geq \cdots \geq t_{\max}(i_1) > \text{COST}_{\text{OPT}}$ (otherwise i_0 would exist, hence contradicting our hypothesis). Recall that there are three ways to exit the algorithm, hence three possible definitions for i_1:

- $\sigma^{(i_1)}(j^\star(i_1)) = p$, however then we would have $t_{\max}(i_1) = t_{j^\star(i_1),p} > \text{COST}_{\text{OPT}} \geq t_{j^\star(i_1),\sigma^{(\text{OPT})}}$ (according to Lemma 3). This contradicts Equation (1), which states that $t_{j^\star(i_1),p} \leq t_{j^\star(i_1),k}$ for all k.
- $i_1 = n(p-1) - 1$, but then we have the same result, i.e., $\sigma^{(i_1)}(j^\star(i_1)) = p$ because this is true for all tasks.
- $t_{\max}(i_1) < \frac{A_{\text{tot}}(i_1)}{p}$, but this is false according to Lemma 5.

We have seen that PACK-APPROX could not have terminated at step i_1, however since PACK-APPROX terminates (in at most $n(p-1) - 1$ steps), we have a contradiction. Hence we have shown the existence of i_0. □

Lemma 7. $A_{tot}(i_0) \leq A_{\text{OPT}}$.

Proof. Consider step i_0. If $i_0 = 0$, then at this step, all tasks are scheduled on exactly one processor, and $\forall j, \sigma^{(i_0)}(j) \leq \sigma^{(\text{OPT})}(j)$. Therefore, $A_{\text{tot}}(i_0) \leq A_{\text{OPT}}$. If $i_0 \neq 0$, consider step $i_0 - 1$: $t_{\max}(i_0 - 1) > \text{COST}_{\text{OPT}}$. From Lemma 4, we have $\forall j, \sigma^{(i_0-1)}(j) \leq \sigma^{(\text{OPT})}(j)$. Furthermore, it is easy to see that $\forall j \neq j^\star(i_0 - 1), \sigma^{(i_0)}(j) = \sigma^{(i_0-1)}(j)$ since no task other than $j^\star(i_0 - 1)$ is modified. We also have the following properties:

- $t_{j^\star(i_0-1),\sigma^{(i_0-1)}(j^\star(i_0-1))} = t_{\max}(i_0 - 1)$;

- $t_{\max}(i_0 - 1) > t_{\text{OPT}}$ (by definition of step i_0);
- $t_{\text{OPT}} \geq t_{j^\star(i_0-1),\sigma^{(\text{OPT})}(j^\star(i_0-1))}$ (Lemma 3);
- $\sigma^{(i_0)}(j^\star(i_0 - 1)) = \sigma^{(i_0-1)}(j^\star(i_0 - 1)) + 1$.

The three first properties and Equation (1) allow us to say that $\sigma^{(i_0-1)}(j^\star(i_0 - 1)) < \sigma^{(\text{OPT})}(j^\star(i_0 - 1))$. Thanks to the fourth property, $\sigma^{(i_0)}(j^\star(i_0 - 1)) \leq \sigma^{(\text{OPT})}(j)$. Finally, we have, for all $j, \sigma^{(i_0)}(j) \leq \sigma^{(\text{OPT})}(j)$, and therefore $A_{\text{tot}}(i_0) < A_{\text{OPT}}$ by Equation (2). □

We are now ready to prove the theorem. For i_0 introduced in Lemma 6, we have:

$$\begin{aligned}
\text{COST}_{i_0} &\leq 3 \max\left(t_{\max}(i_0), \frac{A_{\text{tot}}(i_0)}{p}\right) \\
&\leq 3 \max\left(\text{COST}_{\text{OPT}}, \frac{A_{\text{OPT}}}{p}\right) \\
&\leq 3\text{COST}_{\text{OPT}}
\end{aligned}$$

The first inequality comes from Lemma 1. The second inequality is due to Lemma 6 and 7. The last inequality comes from Lemma 3, hence the final result. □

5 Heuristics

In this section, we describe the heuristics that we use to solve the k-IN-p-COSCHEDULE problem.

RANDOM-PACK– In this heuristic, we generate the packs randomly: as long as there remain tasks, randomly choose an integer j between 1 and k, and then randomly select j tasks to form a pack. Once the packs are generated, apply Algorithm 1 to optimally schedule each of them.

RANDOM-PROC– In this heuristic, we assign the number of processors to each task randomly between 1 and p, then use Algorithm 2 to generate the packs, followed by Algorithm 1 on each pack.

A word of caution– We point out that RANDOM-PACK and RANDOM-PROC are not pure random heuristics, in that they already benefit from the theoretical results of Section 4. A more naive heuristic would pick both a task and a number of processor randomly, and greedily build packs, creating a new one as soon as more than p resources are assigned within the current pack. Here, both RANDOM-PACK and RANDOM-PROC use the optimal resource allocation strategy (Algorithm 1) within a pack; in addition, RANDOM-PROC uses an efficient partitioning algorithm (Algorithm 2) to create packs when resources are pre-assigned to tasks.

PACK-APPROX– This heuristic is an extension of Algorithm 3 in Section 4.4 to deal with packs of size k rather than p: simply call MAKE-PACK (n, p, k, σ) instead of MAKE-PACK (n, p, p, σ). However, although we keep the same name as in Section 4.4 for simplicity, we point out that it is unknown whether this heuristic is a 3-approximation algorithm for arbitrary k.

PACK-BY-PACK (ε)– The rationale for this heuristic is to create packs that are well-balanced: the difference between the smallest and longest execution times in each pack should be as small as possible. Initially, we assign one processor per task (for $1 \leq i \leq n$, $\sigma(i) = 1$), and tasks are sorted into a list L ordered by non-increasing execution

times ($\preccurlyeq_\sigma$ values). While there remain some tasks in L, let $T_{i^\star}$ be the first task of the list, and let $t_{\max} = t_{i^\star,\sigma(i^\star)}$. Let V_{req} be the ordered set of tasks T_i such that $t_{i,\sigma(i)} \geq (1-\varepsilon)t_{\max}$: this is the sublist of tasks (including $T_{i^\star}$ as its first element) whose execution times are close to the longest execution time $t_{\max}$, and $\varepsilon \in [0,1]$ is some parameter. Let p_{req} be the total number of processors requested by tasks in V_{req}. If $p_{req} \geq p$, a new pack is created greedily with the first tasks of V_{red}, adding them into the pack while there are no more than p processors used and no more than k tasks in the pack. The corresponding tasks are removed from the list L. Note that $T_{i^\star}$ is always inserted in the created pack. Also, if we have $\sigma(i^\star) = p$, then a new pack with only $T_{i^\star}$ is created. Otherwise ($p_{req} < p$), an additional processor is assigned to the (currently) critical task $T_{i^\star}$, hence $\sigma(i^\star) := \sigma(i^\star) + 1$, and the process iterates after the list L is updated with the insertion of the new value for $T_{i^\star}$ Finally, once all packs are created, we apply Algorithm 1 in each pack, so as to derive the optimal schedule within each pack.

We have $0 < \varepsilon < 1$. A small value of ε will lead to balanced packs, but may end up with a single task with p processors per pack. Conversely, a large value of ε will create new packs more easily, i.e., with fewer processors per task. The idea is therefore to call the heuristic with different values of ε, and to select the solution that leads to the best execution time.

Summary of heuristics– We consider two variants of the random heuristics, either with one single run, or with 9 different runs, hence hoping to obtain a better solution, at the price of a slightly longer execution time. These heuristics are denoted respectively RANDOM-PACK-1, RANDOM-PACK-9, RANDOM-PROC-1, RANDOM-PROC-9. Similarly, for PACK-BY-PACK, we either use one single run with $\varepsilon = 0.5$ (PACK-BY-PACK-1), or 9 runs with $\varepsilon \in \{.1, .2, \ldots, .9\}$ (PACK-BY-PACK-9). Of course, there is only one variant of PACK-APPROX, hence leading to seven heuristics.

Variants– We have investigated variants of PACK-BY-PACK, trying to make a better choice than the greedy choice to create the packs, for instance using a dynamic programming algorithm to minimize processor idle times in the pack. However, there was very little improvement at the price of a much higher running time of the heuristics. Additionally, we tried to improve heuristics with up to 99 runs, both for the random ones and for PACK-BY-PACK, but here again, the gain in performance was negligible compared to the increase in running time. Therefore we present only results for these seven heuristics in the following.

6 Experimental Results

In this section, we study the performance of the seven heuristics on workloads of parallel tasks. First we describe the workloads, whose application execution times model profiles of parallel scientific codes. Then we present the measures used to evaluate the quality of the schedules, and finally we discuss the results.

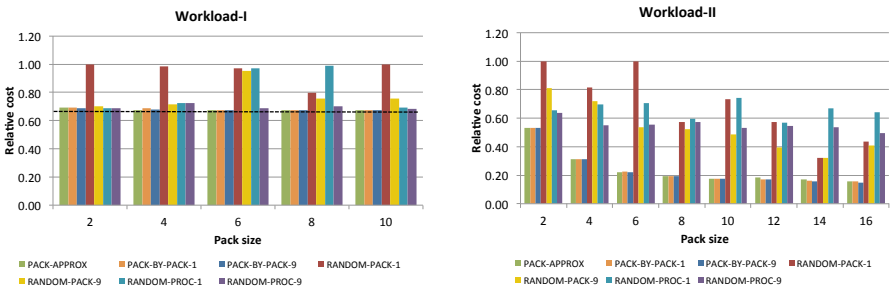
Workloads– Workload-I corresponds to 10 parallel scientific applications that involve VASP [14], ABAQUS [3], LAMMPS [17] and Petsc [1]. The execution times of these applications were observed on a cluster with Intel Nehalem 8-core nodes connected by a QDR Infiniband network with a total of 128 cores. In other words, we have $p = 16$ processors, and each processor is a multicore node.

Workload-II is a *synthetic* test suite that was designed to represent a larger set of scientific applications. It models tasks whose parallel execution time for a fixed problem size m on q cores is of the form $t(m,q) = f \times t(m,1) + (1-f)\frac{t(m,1)}{q} + \kappa(m,q)$,

where f can be interpreted as the inherently serial fraction, and κ represents overheads related to synchronization and the communication of data. We consider tasks with sequential times $t(m,1)$ of the form cm, $cm\log_2 n$, cm^2 and cm^3, where c is a suitable constant. We consider values of f in $\{0, 0.04, 0.08, .16, .32\}$, with overheads $\kappa(m,q)$ of the form $\log_2 q$, $(\log_2 q)^2$, $q\log_2 q$, $\frac{m}{q}\log_2 q$, $\sqrt{m/q}$, and $m\log_2 q$ to create a workload with 65 tasks executing on up to 128 cores.
The same process was also used to develop Workload-III, our largest synthetic test suite with 260 tasks for 256 cores (and $p = 32$ multicore nodes), to study the scalability of our heuristics. For all workloads, we modified speedup profiles to satisfy Equations (1) and (2).

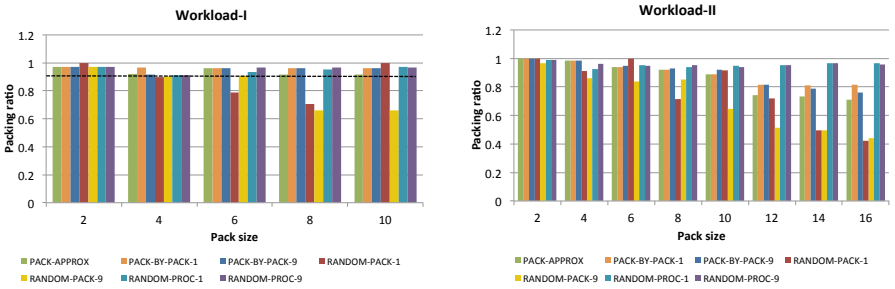
As discussed in related work (see Section 2) and [21], and confirmed by power measurement using Watts Up Pro meters, we observed only minor power consumption variations of less than 5% when we limited co-scheduling to occur across multicore nodes. Therefore, we only test configurations where no more than a single application can be scheduled on a given multicore node comprising 8 cores. Adding a processor to an application T_i which is already assigned σ_i processors actually means adding 8 new cores (a full multicore node) to the $8\sigma_i$ existing cores. Hence a pack size of k corresponds to the use of at most $8k$ cores for applications in each pack. For Workloads-I and II, there are 16 nodes and 128 cores, while Workload-III has up to 32 nodes and 256 cores.
Methodology for assessing the heuristics– To evaluate the quality of the schedules generated by our heuristics, we consider three measures: *Relative cost*, *Packing ratio*, and *Relative response time*. Recall that the cost of a pack is the maximum execution time of a task in that pack and the cost of a co-schedule is the sum of the costs over all its packs.

We define the relative cost as the cost of a given co-schedule divided by the cost of a 1-pack schedule, i.e., one with each task running at maximum speed on all p processors.
For a given k-IN-p-CoSCHEDULE, consider $\sum_{i=1}^{n} t_{i,\sigma(i)} \times \sigma(i)$, i.e., the total work performed in the co-schedule when the i-th task is assigned $\sigma(i)$ processors. We define the packing ratio as this sum divided by p times the cost of the co-schedule; observe that the packing quality is high when this ratio is close to 1, meaning that there is almost no idle time in the schedule.
An individual user could be concerned about an increase in response time and a corresponding degradation of individual productivity. To assess the impact on response time, we consider the performance with respect to a relative response time measure defined as follows. We consider a 1-pack schedule with the n tasks sorted in non-decreasing order of execution time, i.e., in a "shortest task first" order, to yield a minimal value of the response time. If this ordering is given by the permutation $\pi(i), i = 1, 2, \ldots, n$, the response time of task i is $r_i = \sum_{j=1}^{i} t_{\pi(j),p}$ and the mean response time is $R = \frac{1}{n}\sum_{i=1}^{n} r_i$. For a given k-IN-p-CoSCHEDULE with u packs scheduled in increasing order of the costs of a pack, the response time of task i in pack v, $1 \le v \le u$, assigned to $\sigma(i)$ processors, is: $\hat{r}_i = \sum_{\ell=1}^{v-1} cost(\ell) + t_{i,\sigma(i)}$, where $cost(\ell)$ is the cost of the ℓ-th pack for $1 \le \ell \le u$. The mean response time of the k-IN-p-CoSCHEDULE $\hat{R}$ is calculated using these values and we use $\frac{\hat{R}}{R}$ as the relative response time.
Results for small and medium workloads– For Workload-I, we consider packs of size $k = 2, 4, 6, 8, 10$ with 16 processors (hence a total of 128 cores). Note that we do not try $k = p = 16$ since there are only 10 applications in this workload. For Workload-II,



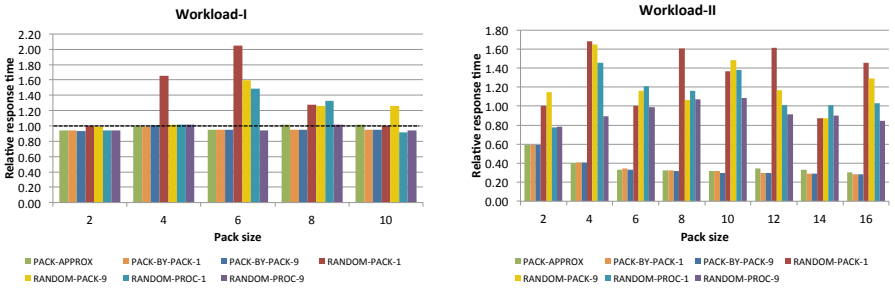
(a) (b)

Figure 2: Quality of co-schedules: Relative costs are shown in (a) for Workload-I and in (b) for Workload-II. The horizontal line in (a) indicates the relative cost of an optimal co-schedule for Workload-I.



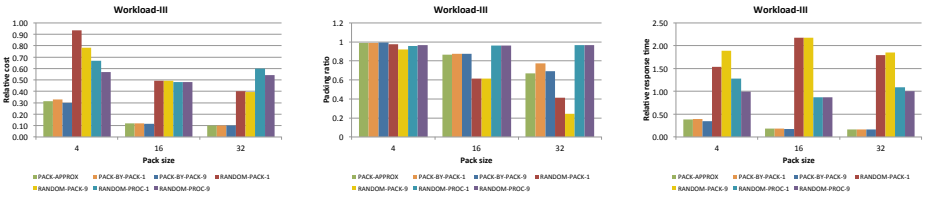
(a) (b)

Figure 3: Quality of packs: Packing ratios are shown in (a) for Workload-I and in (b) for Workload-II. The horizontal line in (a) indicates the packing ratio of an optimal co-schedule for Workload-I.



(a) (b)

Figure 4: Relative response times are shown in (a) for Workload-I and in (b) for Workload-II; values less than 1 indicate improvements in response times. The horizontal line in (a) indicates the relative response time of an optimal co-schedule for Workload-I.



(a) (b) (c)

Figure 5: Relative costs, packing ratios and relative response times of co-schedules for Workload-III on 256 cores.

we consider packs of size $k = 2, 4, 6, 8, 10, 12, 14, 16$.

Figure 2 shows the relative cost of co-schedules computed by the heuristics. For Workload-I (Figure 2(a)), the optimal co-schedule was constructed using exhaustive search. We observe that the optimal co-schedule has costs that are more than 35% smaller than the cost of a 1-pack schedule for Workload-I. Additionally, we observe that PACK-APPROX and PACK-BY-PACK compute co-schedules that are very close to the optimal one for all values of the pack size. Both RANDOM-PACK and RANDOM-PROC perform poorly when compared to PACK-BY-PACK and PACK-APPROX, especially when a single run is performed. As expected, RANDOM-PROC does better than RANDOM-PACK because it benefits from the use of Algorithm 2, and for this small workload, RANDOM-PROC-9 almost always succeed to find a near-optimal co-schedule. The results are similar for the larger Workload-II as shown in Figure 2(b), with an increased gap between random heuristics and the packing ones. Computing the optimal co-schedule was not feasible because of the exponential growth in running times for exhaustive search. With respect to the cost of a 1-pack schedule, we observe very significant benefits, with a reduction in costs of most than 80% for larger values of the pack size, and in particular in the unconstrained case where $k = p = 16$. This corresponds to significant savings in energy consumed by the hardware for servicing a specific workload.

Figure 3 shows the quality of packing achieved by the heuristics. The packing ratios are very close to one for PACK-BY-PACK and PACK-APPROX, indicating that our methods are producing high quality packings. In most cases, RANDOM-PROC and RANDOM-PACK also lead to high packing ratios.

Finally, Figure 4 shows that PACK-BY-PACK and PACK-APPROX produce lower cost schedules with commensurate reductions in response times. For Workload-II and larger values of the pack size, response time gains are over 80%, making k-IN-p-CO-SCHEDULE attractive from the user perspective.

Scalability– Figure 5 shows scalability trends for Workload-III with 260 tasks on 32 processors (hence a total of 256 cores.) Although the heuristics, including RANDOM-PACK and RANDOM-PROC, result in reducing costs relative to those for a 1-pack schedule, PACK-APPROX and PACK-BY-PACK are clearly superior, even when the random schemes are run 9 times. We observe that for pack sizes of 16 and 32, PACK-APPROX and PACK-BY-PACK produce high quality co-schedules with costs and response times that are respectively 90% and 80% lower than those for a 1-pack schedule. PACK-BY-PACK-1 obtains results that are very close to those of PACK-BY-PACK-9, hence even a single run returns a high quality co-schedule.

Running times– We report in Table 1 the running times of the seven heuristics. All

heuristics run within a few milliseconds, even for the largest workload. Note that PACK-APPROX was faster on Workload-II than Workload-I because its execution performed fewer iterations in this case. Random heuristics are slower than the other heuristics, because of the cost of random number generation. PACK-BY-PACK has comparable running times with PACK-APPROX, even when 9 values of ε are used.

	Workload-I	Workload-II	Workload-III
PACK-APPROX	0.50	0.30	5.12
PACK-BY-PACK-1	0.03	0.12	0.53
PACK-BY-PACK-9	0.30	1.17	5.07
RANDOM-PACK-1	0.07	0.34	9.30
RANDOM-PACK-9	0.67	2.71	87.25
RANDOM-PROC-1	0.05	0.26	4.49
RANDOM-PROC-9	0.47	2.26	39.54

Table 1: Average running times in milliseconds.

Summary of experimental results– Results indicate that heuristics PACK-APPROX and PACK-BY-PACK both produce co-schedules of comparable quality. PACK-BY-PACK-9 is slightly better than PACK-BY-PACK-1, at a price of an increase in the running time from using more values of ε. However, the running time remains very small, and similar to that of PACK-APPROX. Using more values of ε to improve PACK-BY-PACK leads to small gains in performance (e.g, 1% gain for PACK-BY-PACK-9 compared to PACK-BY-PACK-1 for $k = 16$ in Workload-II). However, these small gains in performance correspond to significant gains in system throughput and energy, and far outweigh the costs of computing multiple co-schedules. This makes PACK-BY-PACK-9 the heuristic of choice. Our experiments with 99 values of ε did not improve performance, indicating that large increases in the number of ε values may not be necessary.

7 Conclusion

We have developed and analyzed co-scheduling algorithms for processing a workload of parallel tasks. Tasks are assigned to processors and are partitioned into packs of size k with the constraint that the total number of processors assigned over all tasks in a pack does not exceed p, the maximum number of available processors. Tasks in each pack execute concurrently on a number of processors, and workload completes in time equal to sum of the execution times of the packs. We have provided complexity results for minimizing the sum of the execution times of the packs. The bad news is that this optimization problem is NP-complete. This does not come as a surprise because we have to choose for each task both a number of processors and a pack, and this double freedom induces a huge combinatorial solution space. The good news is that we have provided an optimal resource allocation strategy once the packs are formed, together with an efficient load-balancing algorithm to partition tasks with pre-assigned resources into packs. This load-balancing algorithm is proven to be a 3-approximation algorithm for the most general instance of the problem. Building upon these positive results, we have developed several heuristics that exhibit very good performance in our test sets. These heuristics can significantly reduce the time for completion of a workload for corresponding savings in system energy costs. Additionally, these savings

come along with measurable benefits in the average response time for task completion, thus making it attractive from the user's viewpoint.
These co-schedules can be computed very rapidly when speed-up profile data are available. Additionally, they operate at the scale of workloads with a few to several hundred applications to deliver significant gains in energy and time per workload. These properties present opportunities for developing hybrid approaches that can additionally leverage dynamic voltage and frequency scaling (DVFS) within an application. For example, Rountree et al. [19] have shown that depending on the properties of the application, DVFS can be applied at runtime through their Adagio system, to yield system energy savings of 5% to 20%. A potential hybrid scheme could start with the computation of a k-IN-p-COSCHEDULE for a workload, following which DVFS could be applied at runtime per application.
Our work indicates the potential benefits of co-schedules for high performance computing installations where even medium-scale facilities consume Megawatts of power. We plan to further test and extend this approach towards deployment in university scale computing facilities where workload attributes often do not vary much over weeks to months and energy costs can be a limiting factor.

Acknowledgments. This work was supported in part by the ANR *RESCUE* project. The research of Padma Raghavan was supported in part by the The Pennsylvania State University and grants from the U.S. National Science Foundation. The research of Manu Shantharam was supported in part by the U.S. National Science Foundation award CCF-1018881.

References

[1] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2012. `http://www.mcs.anl.gov/petsc`.

[2] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, New York, NY, USA, 2010. ACM.

[3] L. Borgesson. Abaqus. In O. Stephansson, L. Jing, and C.-F. Tsang, editors, *Coupled Thermo-Hydro-Mechanical Processes of Fractured Media - Mathematical and Experimental Studies*, volume 79 of *Developments in Geotechnical Engineering*, pages 565 – 570. Elsevier, 1996.

[4] P. Brucker, A. Gladky, H. Hoogeveen, M. Y. Kovalyov, C. Potts, T. Tautenhahn, and S. Van De Velde. Scheduling a batching machine. *Journal of Scheduling*, 1:31–54, 1998.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA 11*, pages 340–351. IEEE, 2005.

[6] E. G. Coffman Jr, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.

[7] R. K. Deb and R. F. Serfozo. Optimal control of batch service queues. *Advances in Applied Probability*, pages 340–361, 1973.

[8] P.-F. Dutot, G. Mounié, D. Trystram, et al. Scheduling parallel tasks: Approximation algorithms. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, 2003.

[9] E. Frachtenberg, D. Feitelson, F. Petrini, and J. Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 16(11):1066–1077, 2005.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[11] C. Hankendi and A. Coskun. Reducing the energy cost of computing through efficient co-scheduling of parallel workloads. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 994–999, 2012.

[12] S. Kamil, J. Shalf, and E. Strohmaier. Power efficiency in high performance computing. In *IPDPS*, pages 1–8. IEEE, 2008.

[13] G. Koole and R. Righter. A stochastic batching and scheduling problem. *Probability in the Engineering and Informational Sciences*, 15(04):465–479, 2001.

[14] G. Kresse and J. Hafner. Ab initio molecular dynamics for liquid metals. *Phys. Rev. B*, 47(1):558–561, Jan 1993.

[15] D. Li, D. S. Nikolopoulos, K. Cameron, B. R. de Supinski, and M. Schulz. Power-aware MPI task aggregation prediction for high-end computing systems. In *IPDPS 10*, pages 1–12, 2010.

[16] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, 2002.

[17] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.*, 117:1–19, March 1995.

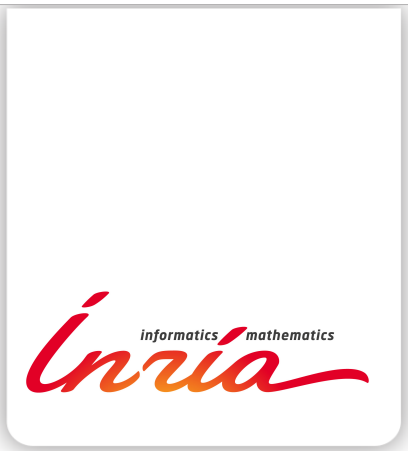
[18] C. N. Potts and M. Y. Kovalyov. Scheduling with batching: a review. *European Journal of Operational Research*, 120(2):228–249, 2000.

[19] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: making DVS practical for complex HPC applications. In *ICS 09*, pages 460–469, 2009.

[20] T. Scogland, B. Subramaniam, and W.-c. Feng. Emerging Trends on the Evolving Green500: Year Three. In *7th Workshop on High-Performance, Power-Aware Computing*, Anchorage, Alaska, USA, May 2011.

[21] M. Shantharam, Y. Youn, and P. Raghavan. Speedup-aware co-schedules for efficient workload management. *Parallel Processing Letters*, 2013, to appear.

[22] J. Turek, U. Schwiegelshohn, J. L. Wolf, and P. S. Yu. Scheduling parallel tasks to minimize average response time. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 112–121. Society for Industrial and Applied Mathematics, 1994.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399