Malleable task-graph scheduling with a practical speed-up model

Loris Marchal, Bertrand Simon, Oliver Sinnen, Frédéric Vivien

Malleable task-graph scheduling with a practical speed-up model

Loris MARCHAL, Bertrand SIMON, Oliver SINNEN, Frédéric VIVIEN

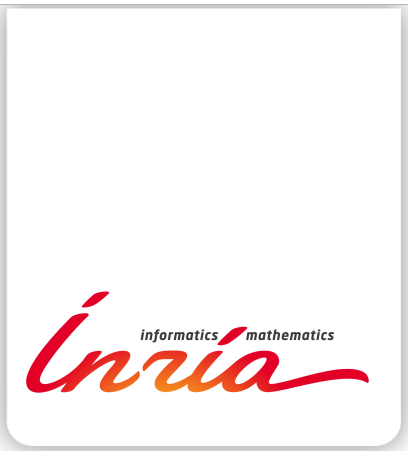
RESEARCH REPORT

N° 8856

February 2016

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8856--FR+ENG



Malleable task-graph scheduling with a practical speed-up model

Loris MARCHAL*, Bertrand SIMON†, Oliver SINNEN‡,
Frédéric VIVIEN§

Project-Team ROMA

Research Report n° 8856 — February 2016 — 34 pages

Abstract: Scientific workloads are often described by Directed Acyclic task Graphs. Indeed, DAGs represent both a model frequently studied in theoretical literature and the structure employed by dynamic runtime schedulers to handle HPC applications. A natural problem is then to compute a makespan-minimizing schedule of a given graph. In this paper, we are motivated by task graphs arising from multifrontal factorizations of sparse matrices and therefore work under the following practical model. We focus on malleable tasks (i.e., a single task can be allotted a time-varying number of processors) and specifically on a simple yet realistic speedup model: each task can be perfectly parallelized, but only up to a limited number of processors. We first prove that the associated decision problem of minimizing the makespan is NP-Complete. Then, we study a widely used algorithm, PropScheduling, under this practical model and propose a new strategy GreedyFilling. Even though both strategies are 2-approximations, experiments on real and synthetic data sets show that GreedyFilling achieves significantly lower makespans.

Key-words: Scheduling, Task graph, Malleable tasks, Approximation algorithms

* Loris MARCHAL is with CNRS, France. E-mail: loris.marchal@ens-lyon.fr
† Bertrand SIMON is with ENS de Lyon, France. E-mail: bertrand.simon@ens-lyon.fr
‡ Oliver SINNEN is with Dpt. of Electrical and Computer Engineering, University of Auckland, New Zealand. E-mail: o.sinnen@auckland.ac.nz
§ Frédéric VIVIEN is with Inria, France. E-mail: frederic.vivien@inria.fr

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Ordonnancement de Graphes de Tâches Malléables avec un modèle pragmatique

Résumé : Les applications de calcul scientifique sont souvent décrites comme des graphes de tâches dirigés et acycliques. En effet, ces graphes représentent à la fois un modèle étudié fréquemment dans la littérature théorique ainsi qu'une structure employée par les ordonnanceurs dynamiques de runtime pour traiter des applications HPC. Un problème naturel consiste donc à calculer un ordonnancement d'un graphe donné minimisant le temps d'exécution. Dans ce rapport, nous nous concentrons sur des graphes de tâches provenant de factorisations multifrontales de matrices creuses et travaillons donc dans le modèle pragmatique suivant. Nous étudions les tâches malléables (i.e., une tâche peut être allouée à un nombre variant de processeurs) et plus précisément un modèle d'accélération simple mais réaliste: chaque tâche peut être parallélisée parfaitement, mais seulement jusqu'à un nombre limite de processeurs. Nous commençons par prouver que le problème de décision associé à la minimisation du makespan est NP-complet. Ensuite, nous étudions sous notre modèle un algorithme largement répandu, PropScheduling, et proposons une nouvelle stratégie, GreedyFilling. Même si ces deux stratégies sont des 2-approximations, des expériences sur des bases de données réelles et artificielles montrent que GreedyFilling atteint des makespans significativement plus courts.

Mots-clés : Ordonnancement, Graphe de tâches, Tâche malléable, Algorithmes d'approximation

Contents

1 Introduction

Complex computations are often described as Directed Acyclic Graphs (DAGs), where nodes represent computational tasks and edges represent dependences between these tasks. This formalism is both very common in the theoretical scheduling literature [8] and sees an increasing interest in High Performance Computing: to cope with the complexity and heterogeneity in modern computer design, many HPC applications are now expressed as task graphs and rely on dynamic runtime schedulers such as StarPU [1], KAAPI [14], StarSS [30], and PaRSEC [4]. Even the OpenMP standard now includes DAG scheduling constructs [29].

Task graphs are helpful to express the structure of applications and to take advantage of the potential parallelism they express, sometimes called *inter-task parallelism*. However, tasks are often coarse grain and each task can be executed in parallel on several processing cores. To achieve good performance, we also want to take into account this *intra-task parallelism*. There have been a number of studies to mix both sources of parallelism when scheduling task graphs, such as [34, 10]. The main difficulty in this context is to come up with an expressive and yet tractable model for tasks. Task characteristics are summarized through a *speed-up* function that relates the task execution time to the number of processors it is allocated to. In the present paper, we focus on a simple model where the speedup function is perfect —that is, equal to the number of processor— until it reaches a given threshold, after which it stalls and stays constant. This speedup function models well the performance of linear algebra kernels which are our present concern, and it has been studied both in theoretical scheduling [6] and for practical schedulers [28, 35]. Contrarily to most existing studies, we also assume that tasks are *preemptible* (a task may be interrupted and resumed later), *malleable* (the number of processors allocated to a task can vary over time) and we allow *fractional allocation* of processors. We claim that this model is reasonable firstly because changing allocation of processors is easily achieved using the time sharing facilities of operating system schedulers or hypervisors: actual runtime schedulers are able to dynamically change the allocation of a task [19]. Secondly, given preemption and malleability, it is possible to transform any schedule with rational allocation to a schedule with integral allocation using McNaughton's wrap-around rule [26]. Hence, we can consider rational allocations that are simple to design and analyze, and then transform them into integral ones when needed.

The here presented study is motivated and driven by task graphs coming from sparse linear algebra, and especially from the factorization of sparse matrices using the multifrontal method. Liu [24] explains that the computational dependencies and requirements in Cholesky and LU factorization of sparse matrices using the multifrontal method can be modeled as a task tree, called the *assembly tree*. Our target are therefore such trees and the

experimental evaluation will focus on them. Having that said, the proposed algorithms in this paper are not limited to trees, but apply to series-parallel graphs (SP-graphs) or general DAGs. We will describe and analyse them correspondingly for the sake of generality.

The contributions of this paper are the following. We apply the perfect but limited speedup model to scheduling graphs with malleable tasks. We first show that this problem is NP-complete. We analyse an allocation scheme called Proportional Mapping which is commonly used by runtime schedulers and show that it is a 2-approximation algorithm with tight ratio. We propose and analyse a simple greedy scheduler, called Greedy-Filling, and prove that it also is a 2-approximation algorithm. We perform simulations both on synthetic series-parallel graphs and on real task trees and show that the Greedy-Filling heuristic usually performs better than Proportional Mapping.

2 Related Work

In this section, we thoroughly review the related work on malleable task graph scheduling for models of tasks that are close or similar to our model. We also present some basic resuts on series-parallel graphs.

Models of parallel tasks. The literature contain numerous models for "parallel tasks"; names and notations are varying and their usage is not always consistent. The most simple model for parallel tasks is the model of *rigid* tasks, sometimes simply called *parallel tasks* [18]. A rigid task must always be executed on the same number of processors (that must be simultaneously available). In the model of *moldable* tasks, the scheduler has the freedom to chose on which number of processors to run a task, but this number cannot change during the execution. This model is sometimes called *multiprocessor tasks* [7]. The most general model is that of *malleable* tasks: the number of processors executing a task can change in any way at any time throughout the task execution. However, numerous articles use the name malleable to denote moldable tasks like, for instance, [18, 23, 21]. Depending on the variants, moldable and malleable tasks can run on any number of processors, from 1 to p, or each task T_i may have a maximum parallelism which is often denoted by δ_i [7, 2].Furthermore, depending on the assumptions, tasks may be preempted to be restarted later on the same set of processors, or on a potentially different ones (preemption+migration). It should be noted that the model of malleable tasks is a generalization of the model of moldable tasks with preemption and migration.

An important feature of the models for moldable and malleable tasks is the task speed-up functions that relate a task execution time to the number of processors it uses. Some authors, like Hunold [20], do not make

any assumptions on the speed-up functions. More generally, people assume the task execution time is a non-increasing function of the number of processors [20, 23, 25, 12]. Another classical assumption is that the work is a non-decreasing function [20, 23, 12] —the work is the product of the execution time and of the number of processor used— which defines the model sometimes called *monotonous penalty assumptions*. Some other work consider that the speed-up function is a concave function [25]. Several of the models considered in the literature satisfy all above assumptions: non-decreasing concave speed-up function and non-decreasing work. This is for instance the case of the model studied by Prasanna and Musicus [32, 33] where $p_i(k) = \frac{p_1(1)}{k^\alpha}$ where α is a task-independent constant between 0 and 1 [32, 33, 17]. Another instance is the model we use in this work, that is, the linear model [2, 39, 6, 28, 40]: $p_i(k) = \frac{p_i}{k}$. Kell and Havill [22] added to that model an overhead affine in the number of processors used: $p_i(k) = \frac{p_i}{k} + (k-1)c$. This model is also closely related to the Amdahl's law where $p_i(k) = \frac{p_i^{(p)}}{k} + p_i^{(s)}$. This law is considered the experimental evaluation of [12].

Finally, the number of processors alloted to a task can, depending on the assumptions, either only take integer values, or can also take rational ones [32, 33, 17, 25].

Results for moldable tasks. Du and Leung [9] have shown that the problem of scheduling moldable tasks with preemption and arbitrary speed-up functions is NP-complete.

In the scope of the monotonous penalty model, Lepère, Trystram, and Woeginger [23] presented a $3 + \sqrt{5} \approx 5.23606$ approximation algorithm for general DAGs, and a $\frac{3+\sqrt{5}}{2} + \epsilon \approx 2.61803 + \epsilon$ approximation algorithm for series-parallel graphs and DAGs of bounded width.

Wang and Cheng presented [39] a $3 - \frac{2}{p}$-approximation algorithm to minimize the makespan while scheduling moldable task graphs with linear speed-up and maximum parallelism δ_j (problem $P|prec, any, spdp\text{-}lin, \delta_j|C_{\max}$).

Results for malleable tasks. The problem of scheduling independent malleable tasks with linear speedups, maximum parallelism per task, and with integer allotments, that is $P|var, spdp\text{-}lin, \delta_j|C_{\max}$, can be solved in polynomial time [38, 6] using a generalization of McNaughton's wrap-around rule [26]. Drozdowski and Kubiak showed in [6] that this problem becomes NP-hard when dependences are introduced: $P|prec, var, spdp\text{-}lin, \delta_j|C_{\max}$ is NP-hard. Balmin et al. [28] present a 2-approximation algorithm for this problem. Their algorithm builds integral allotments by first scheduling the DAG on an infinite number of processors and then using the optimal algorithm for independent tasks to build an integral-allotment schedule for each

interval of the previous schedule during which a constant number of processors greater than p was used. According to Theorem 5 of the present paper, this algorithm is also a $2 - \frac{\delta_{\min}}{p}$ approximation for makespan minimization with fractional allotments.

Makarychev and Panigrahi [25] consider the problem $P|prec, var|C_{\max}$ under the monotonous penalty assumption and when allotments are rational. They provide a $(2+\epsilon)$-approximation algorithm, of unspecified complexity (their algorithm relies on the resolution of a rational linear program; this linear program is not explicitly given). Furthermore, they prove that there is no "online algorithm with sub-polynomial competitive ratio" (an online algorithm is an algorithm that considers tasks one after the other, as in our greedy algorithms).

Series-parallel graphs. Series-parallel graphs can be recognized and decomposed into a tree of series and parallel combination in linear time [37]. It is well-known that that series-parallel graphs capture the structure of many real-world scientific workflows [3], A possible way to extend algorithms designed for series-parallel graphs to general graphs is to first transform a graph into a series-parallel graph, using a process sometimes called SPization [15, 27] before applying a specialized algorithm for SP-graphs. This was for example done in [5]. However, note that no SPization algorithm guarantees that the length of the critical path is increased by only a constant ratio.

3 Application model

We consider a workflow of tasks whose precedence constraints are represented by a task graph $G = (V, E, w, \delta)$ of n nodes, or tasks: a task can only be executed after the termination of all its predecessors. We assume that G is a *series-parallel* graph. Such graphs are built recursively as series or parallel composition of two or more smaller SP-graphs, and the base case is a single task. Trees can be seen as a special-case of series-parallel graphs. A tree can be turned into an SP-graph by simply adding one dummy task without computation cost, that has an edge with every leaf of the tree.

Each task $T_i \in V$ is associated with a ***weight*** w_i that corresponds to the work that needs to be done to complete the task. By extension, the weight of a subgraph of G is the sum of the weights of the tasks it is composed of. The ***start time*** t_i of a task T_i is defined as the time when the processing of its work starts for the first time. We denote by p the total number of identical processors available to schedule G. Tasks are assumed to be preemptible, malleable, with linear speed-up, and maximum parallelism per task. That is, each task T_i may be allocated a fractional, time-varying amount $p_i(t)$ of processors at time t. Moreover, task T_i is associated with a ***threshold***

δ_i in the number of processors, which limits the speed-up of the task. For $p_i(t) \leq \delta_i$, the task is perfectly parallel, and for $p_i(t) > \delta_i$, the speed-up is equal to the threshold. The completion or ***finish time*** of task T_i is thus defined as the smallest value f_i such that

$$\int_0^{f_i} \min(p_i(t), \delta_i)dt = w_i.$$

The objective is to minimize the *makespan* of the application, that is the latest task finish time. Using Graham's notation, this problem corresponds to $P|prec, var, frac, spdp\text{-}lin, \delta_j|C_{\max}$, where *frac* denotes fractional allocations. As mentioned in the introduction, it is possible to remove the assumption of fractional allocation without degrading the makespan thanks to malleability: from any fractional schedule, we consider each interval when tasks get a constant fractional number of processors, and we transform it into an integer allocation using McNaughton's wrap-around rule [26]. This may only add a number of preemptions proportional to the number of tasks for each interval.

In the following, we will often use the length of the *critical path* of a task T_i, which is defined as the minimum time needed to complete all the tasks on any path from this task to any output task of the graph, provided that an unlimited number of processors is available. This corresponds to the classical notion of bottom-level [36], when the duration of each task is set to w_i/δ_i. By extension, the critical path of the entire graph G is the longest critical path of all its tasks.

Some of the algorithms presented in this paper also apply to some restricted variants of the problem. A notable one is the case of *moldable* tasks, which prohibit any variation in the set of processors used by a task: in this case, $p_i(t)$ must be constant on one time interval, and null elsewhere.

4 Complexity

Task malleability and perfect speed-up make this problem much easier than most scheduling problems. However, quite surprisingly, adding thresholds to limit the possible parallelism is sufficient to make it NP-complete[1].

Theorem 1. *The problem of minimizing the makespan is NP-complete.*

Proof. We start by proving that this problem belongs to NP. Without loss of generality, we restrict to schedules which allocate a constant share of processors to each task between any two task completions. Note that from a schedule that does not respect this condition, we can construct a schedule with the same completion times simply by allocating the average share of processors to each task in each such interval. Given a schedule that respects

[1] A similar result already appeared in [6], however its proof is more complex and not totally specified, which makes it difficult to check.

this restriction, it is easy to check that it is valid in time polynomial in the number of tasks.

To prove completeness, we perform a reduction from the 3SAT problem which is known to be NP-complete [13]. An instance $\mathcal{I}$ of this problem consists of a boolean formula, namely a conjunction of m disjonctive clauses, $C_1, \ldots, C_m$, of 3 literals each. A literal may either be one of the n variables $x_1 \ldots x_n$ or the negation of a variable. We are looking for an assignment of the variables which leads to a TRUE evaluation of the formula.

Instance definition. From $\mathcal{I}$, we construct an instance $\mathcal{J}$ of our problem. This instance is made of $2n+1$ chains of tasks and $p = 3$ processors. The first $2n$ chains corresponds to all possible literals of instance $\mathcal{I}$; they are denoted L_{x_i} or $L_{\overline{x_i}}$ and called *literal chains*. The last chain is intended to mimic a variable "processor profile", that is a varying number of available processors over time for the other chains, and is denoted by L_{pro}.

Our objective is that for every *pair of literal chains* (L_{x_i} and $L_{\overline{x_i}}$), one of them starts at some time $t_i = 2(i-1)$ and the other at time $t_i + 1$. The one starting at time $t_i + 1$ will have the meaning of TRUE. We will construct the chains such that (i) no two chains of the same pair can start both at time $t_i + 1$ and (ii) at least one chain L_{x_i} or $L_{\overline{x_i}}$ corresponding to one of the three literals of any given clause starts at time $t_i + 1$.

For any chain, we consider its *critical path* length, that is, the minimum time needed to process it provided that enough processors are available. The makespan bound M of instance $\mathcal{J}$ is equal to the critical path length of the last chain L_{pro}, and will be specified later. Thus, to reach M, all tasks of L_{pro} must be allocated their threshold, and no idle time may be inserted between them.

In constructing the chains, we only use tasks whose weight is equal to their threshold, so that their minimum computing time is one. Then, a chain is defined by a list of numbers $[a_1, a_2, \ldots]$: the i-th task of the chain has a threshold and a weight a_i. As a result, the critical path length of a chain is exactly the number of tasks it contains. We define $\varepsilon = 1/4n$ and present the general shape of a literal chain L_a, where a is either x_i or $\overline{x_i}$:

$$L_a = [1, \varepsilon, \underbrace{\varepsilon, \ldots, \varepsilon}_{2(n-i)}, \underbrace{\mathit{SelectClause}(a)}_{2m \text{ tasks}}, \underbrace{\varepsilon, \ldots, \varepsilon}_{2(n-i)}, 1]$$

The leftmost and rightmost parts of the chain are dedicated to ensuring that in each pair of literal chains, one of them starts at time $t_i = 2(i-1)$ and the other at time $t_i + 1$. The central part of the chain is devoted to clauses, and ensures that for each clause, at least one chain corresponding to a literal of

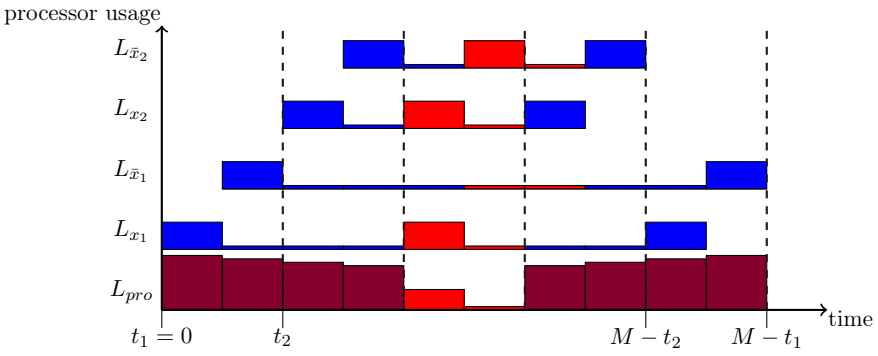


Figure 1: Example of a possible schedule for the instance $\mathcal{J}$ associated to the formula $x_1 \vee x_2 \vee \bar{x}_2$.

the clause starts at time $t_i + 1$:

$$SelectClause(a) = [InClause(C_1, a), \varepsilon, \ldots, InClause(C_m, a), \varepsilon]$$
$$\text{where } InClause(C_k, a) = \begin{cases} [1 - \frac{2}{3}n\varepsilon] & \text{if } a \text{ appears in } C_k \\ [\varepsilon] & \text{otherwise} \end{cases}$$

In total, the chain L_a includes $4n + 2m - 4i + 3$ tasks. Finally, the profile chain is defined as follows:

$$L_{\text{pro}} = [\underbrace{2, 2-\varepsilon, \ldots, 2-(2n-1)\varepsilon}_{2n \text{ tasks}}, \underbrace{L_{10}, L_{10}, \ldots, L_{10}}_{2m \text{ tasks}}, \underbrace{2-(2n-1)\varepsilon, \ldots, 2-\varepsilon, 2}_{2n \text{ tasks}}]$$

where $L_{10} = [1-(\frac{2}{3}n-2)\varepsilon, 3\varepsilon]$. The critical path length of L_{pro} defines $M = 2m + 4n$. Figure 1 presents a valid schedule for the instance corresponding to the formula $(x_1 \vee x_2 \vee \overline{x_2})$, which corresponds to the assignment $\{x_1 = 0,\ x_2 = 0\}$.

From a truth assignment to a valid schedule. We assume here that we are given a truth assignment of the variables of $\mathcal{I}$: let v_i denote the value of variable x_i in this assignment. We construct the following schedule for $\mathcal{J}$: for all chains, each task is allocated a number of processors equal to its threshold and no idle time is inserted between any two consecutive tasks. Chain L_{pro} starts at time 0 while chain L_{x_i} (respectively $L_{\overline{x_i}}$) starts at time $t_i + 1$ if v_i is TRUE (resp. FALSE), otherwise it starts at time t_i. It is straightforward to check that this schedule is valid. Here, we only concentrate on the most critical part, namely the central part which corresponds to the clauses. We count the number of processors used during on time interval $[2n + 2(k-1), 2n + 2k]$ which corresponds to clause C_k:

- In the first half of this interval, at most 2 literal chains can have a task of size $1 - \frac{2}{3}n\varepsilon$ since at least one in the tree literals of the clause is true.

Together with the other literal chains and the profile, the maximum processor occupancy is at most:

$$\underbrace{2\left(1-\frac{2}{3}n\varepsilon\right)}_{\text{FALSE literal chains}} + \underbrace{(2n-2)\varepsilon}_{\text{other literal chains}} + \underbrace{1-\left(\frac{2}{3}n-2\right)\varepsilon}_{L_{\text{pro}}} = 3.$$

(Remark that because $\varepsilon = 1/4n$, $1-\frac{2}{3}n\varepsilon > \varepsilon$.)

- In the second half of this interval, at most 3 literal chains can have a task of size $(1-\frac{2}{3}n\varepsilon)$, which may result in a maximum number of busy processors of:

$$\underbrace{3\left(1-\frac{2}{3}n\varepsilon\right)}_{\text{TRUE literal chains}} + \underbrace{(2n-3)\varepsilon}_{\text{other literal chains}} + \underbrace{3\varepsilon}_{L_{\text{pro}}} = 3.$$

The resulting schedule has a makespan of M and is thus a solution to $\mathcal{J}$.

From a valid schedule to a truth assignment. We now assume that instance $\mathcal{J}$ has a valid schedule S and we aim at reconstructing a solution for $\mathcal{I}$. We first prove some properties on the starting times of chains through the following lemma. The proof is done by induction on i, by carefully checking when the first and last tasks of chains $L_{x_i}, L_{\overline{x_i}}$ may be scheduled, given the resources which are not used by the previous chains and by L_{pro}.

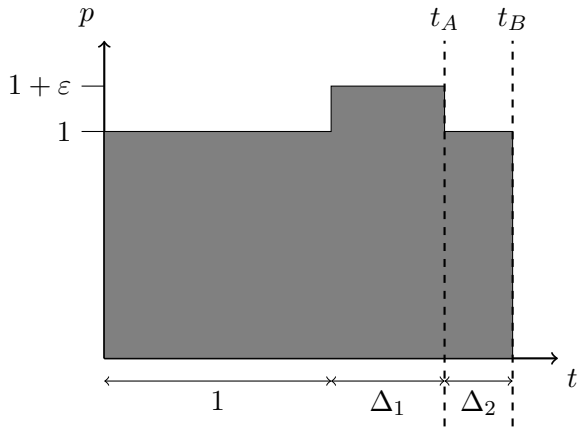
Lemma 2. *In any valid schedule S for $\mathcal{J}$,*

i. each pair of chains $L_{x_i}, L_{\overline{x_i}}$ is completely processed during time interval $[t_i, M - t_i]$,
ii. one of them is started at time t_i and the other one at time t_{i+1},
iii. all tasks of both chains are allocated their threshold,
iv. there is no idle time between any two consecutive tasks of each chain.

Proof. We prove the lemma by induction on i. It is obviously true for $i = 0$ (as no chain L_{x_0} exists). We assume the lemma true for $i < n$ and consider chains $L_{x_i}, L_{\overline{x_i}}$. We first notice that given the induction properties, no processor is available before t_i and after $M - t_i$, which proves (i). The time span available for the remaining chains is thus $M - 2t_i = 4n + 2m - 4i + 4$ while the critical path of chains L_{x_i} and $L_{\overline{x_i}}$ is $4n+2m-4i+3$: these chains cannot be started after $t_i + 1$ to be completed within the time span.

We consider the first task of chain L_{x_i} and the first task of chain $L_{\overline{x_i}}$. Both tasks have weight 1. Let A denote the first of these two tasks to complete (at a time t_A) and let B be the other one (which completes at time t_B). Given the $2(i-1)$ chains already scheduled, the number of processors available during interval $[t_i, t_i + 1]$ is 1 and during interval $[t_i + 1, t_i + 2]$ is

$1+\varepsilon$. A and B both complete at or after time t_i+1. We note $t_A = t_i+1+\Delta_1$ and $t_B = t_i + 1 + \Delta_1 + \Delta_2$ ($\Delta_1 \geq 0$ and $\Delta_2 \geq 0$). Note that because of the critical path length of the remaining tasks of both chains and the limited time span, $\Delta_1 \leq 1$ and $\Delta_1 + \Delta_2 \leq 1$. The following figure illustrates the previous notations and the amount of processors available for tasks A and B (note that after time t_A, B may use only $\delta_B = 1$ processor).



Since $w_A + w_B = 2$ work units have to be performed before time t_B, we have

$$1 + \Delta_1(1+\varepsilon) + \Delta_2 \geq 2$$

and thus $\Delta_2 \geq 1 - \Delta_1(1+\varepsilon)$ and $t_B \geq t_i + 2 - \Delta_1\varepsilon$.

We symmetrically apply the same reasoning to the last tasks C and D of these two chains, and their *starting* times t_C and t_D, assuming that C is started before D. By setting $t_D = M - t_i - 1 - \Delta'_1$, we get $t_C \leq M - t_i - 2 + \Delta'_1\varepsilon$. We distinguish between two cases, depending on the chains to which A, B, C, and D belong to:

- In the first case, we assume that A and D belong to the same chain. We consider the other chain, containing B and C. Because exactly $4(n-i)+2m+1$ tasks need to be processed between these two tasks, we have

  $$t_C \geq t_B + 4(n-i) + 2m + 1$$

  which gives

  $$\Delta'_1\varepsilon \geq 1 - \Delta_1\varepsilon$$

  We have $\Delta_1 \leq 1$ and similarly, $\Delta'_1 \leq 1$. Together with the previous inequality, this gives $\varepsilon \geq 1/2$ which is not possible since $\varepsilon = 1/4n$. Hence B and C cannot belong to the same chain.

- In the second case, we consider that A and C belongs to the same chain. Because exactly $4(n-i)+2m+1$ tasks need to be processed

between A and C (and between B and D), we have

$$t_C \geq t_A + 4(n-i) + 2m + 1 \quad \text{and} \quad t_D \geq t_B + 4(n-i) + 2m + 1.$$

This gives

$$2m + 4n - t_i - 2 + \Delta'_1\varepsilon \geq t_i + 1 + \Delta_1 + 4(n-i) + 2m + 1$$

and

$$2m + 4n - t_i - 1 - \Delta'_1 \geq t_i + 2 - \Delta_1\varepsilon + 4(n-i) + 2m + 1,$$

which are simplified (using $t_i = 2(i-1)$) into

$$\Delta'_1\varepsilon \geq \Delta_1 \text{ and } \Delta'_1 \leq \Delta_1\varepsilon.$$

This leads to $\Delta_1 \leq \Delta_1\varepsilon^2$. As $0 < \varepsilon < 1$, we have $\Delta_1 = 0$, so $t_A = t_i + 1$. Then, no processor can be allocated to B during $[t_i, t_{i+1}]$.

In other words, one task among the first task of L_{x_i} and the first task of $L_{\overline{x_i}}$ is fully processed during interval $[t_i, t_i + 1]$ and the other one is not processed before $t_i + 1$. Because of its critical path length, the chain starting second must be processed at full speed (each task being allocated a number of processors equal to its threshold) and without idle time in the interval $[t_i + 1, M - t_i]$, which in particular proves (ii). The last task of the chain starting at time t_i must then be completed at time $M - t_i - 1$, and thus this chain must also be processed at full speed and without idle time. This proves (iii) and (iv). □

For each literal chain which starts at time $t_i + 1$, we associate the value True in an assignment of the variables of $\mathcal{I}$, and we associate the value False to all other literals. Thanks to the previous lemma, we know that exactly one literal in the pair $(x_i, \overline{x_i})$ is assigned to True. Furthermore, not three tasks of size $1 - \frac{2}{3}n\varepsilon$ can be scheduled at time $2n + 2(k-1)$ because of the profile chain, as this would lead to a number of occupied processors of:

$$\underbrace{3\left(1 - \frac{2}{3}n\varepsilon\right)}_{\text{3 False literal chains}} + \underbrace{(2n-3)\,\varepsilon}_{\text{other literal chains}} + \underbrace{1 - \left(\frac{2}{3}n - 3\right)\varepsilon}_{L_{\text{pro}}} = 4 - \frac{2}{3}n\varepsilon = 4 - \frac{1}{6} > 3 = p.$$

Thus, at least one literal of each clause is set to True in our assignment. This proves that it is a solution to $\mathcal{I}$. □

Algorithm 1: PROPORTIONALMAPPING ($G = (V, E, w, \delta)$, p)

if *The top-level composition is the series composition of K sub-graphs* **then**
  Allocate $p_k = p$ processors to each subgraph
else The top-level composition is the parallel composition of K sub-graphs
  Allocate the ratio of processors $p_k = \frac{W_k}{\sum_K W_j} p$ to subgraph k, $1 \leq k \leq K$, where W_k is the weight of sub-graph k
Call PROPORTIONALMAPPING (sub-graph k, p_k) for each sub-graph k

5 Proportional Mapping

The first algorithm to be studied is the widely used "proportional mapping" [31]. In this approach, a sub-graph is allocated a number of processors that is proportional to the ratio of its weight to the sum of the weights of all sub-graphs under consideration. Based on the composition of the considered SP-graph G, Algorithm 1 allocates a share of processors to each sub-graph and eventually each task. A given graph G (with SP-graph characteristics) can be decomposed into its series and parallel components using an algorithm from [37]. Observe that the threshold δ_i is not considered in this proportional mapping.

The schedule corresponding to this proportional mapping is simply to start every task as early as possible (i.e. after predecessors have completed) with $\min\{\delta_i, p_i\}$ processors, as given in Algorithm 2.

Algorithm 2: PROPSCHEDULING ($G = (V, E, w, \delta)$, p)

Call PROPORTIONALMAPPING (G, p) to determine p_i for each task $T_i \in G$
Order tasks of G in topological order list L
foreach $T_i \in L$ **do**
  Start T_i with $\min\{\delta_i, p_i\}$ processors as early as possible, i.e. after all predecessors completed

Given the proportional mapping, there are always enough processors available to do that. It is worth noting that the created schedule is compatible with the moldable model; tasks use the same number of processors during their entire execution. As such, Algorithm 2 can also be used for the moldable model.

In the case of perfect parallelism (i.e., $\delta_i \geq p$, $\forall T_i \in G$), there is no idle time as all tasks of a parallel composition terminate at exactly the same time (due to the proportional mapping). Hence, this schedule achieves the optimal

makespan $M_\infty = \frac{\sum_{i \in G} w_i}{p}$. For the general case the following theorem holds.

Theorem 3. PROPSCHEDULING *is a 2-approximation algorithmfor makespan minimization.*

Proof. We first note that the optimal makespan without thresholds, M_∞, is a lower bound on the makespan M with thresholds. With M_{OPT} being the optimal schedule length, we have $M_\infty \leq M_{\text{OPT}} \leq M$ and we want to show $M \leq 2M_{\text{OPT}}$.

The critical path cp of G, as defined in Section 3, is a longest path in G, where length is defined as the sum of the work of each task on the path divided by its threshold, $len(cp) = \sum_{i \in cp} \frac{w_i}{\delta_i}$. Naturally, the critical path length is another lower bound on the optimal makespan, $len(cp) \leq M_{\text{OPT}}$.

Consider the schedule produced by PROPSCHEDULING. There is at least one path Φ in G from the entry task to the exit task, with no idle time between consecutive tasks. In other words, the next task starts execution when the previous one finishes.This follows from the fact that we start tasks as early as possible, so this is always true between the tasks of every serial composition of the SP-graph G, and it is true for at least one task in every parallel composition. The execution length of this path is the makespan M, because it includes no idle time and that it goes from entry to exit task. It is given by

$$M = \sum_{i \in \Phi} \frac{w_i}{\min\{\delta_i, p_i\}}$$

Let us divide the tasks of Φ into two sets: the set A of tasks that are executed with their threshold processors δ_i and the set B of tasks that are executed with the allocated number of processors $p_i < \delta_i$, with $A \cup B = \Phi$. We then have

$$M = \sum_{i \in A} \frac{w_i}{\delta_i} + \sum_{i \in B} \frac{w_i}{p_i}$$

The first term is per definition less than or equal to the length of the critical path $len(cp)$. The second term consists only of tasks that are executed with their proportionally allocated processors, so it is less than or equal to the optimal length without threshold M_∞. We then get the desired inequality:

$$M \;=\; \sum_{i \in A} \frac{w_i}{\delta_i} + \sum_{i \in B} \frac{w_i}{p_i} \;\leq\; len(cp) + M_\infty \;\leq\; 2M_{\text{OPT}}$$

□

So PROPSCHEDULING is a 2-approximation algorithm and the factor is asymptotically tight, as shown in the following corollary.

Corollary 4. *The approximation factor 2 for* PROPSCHEDULING *is asymptotically tight.*

Proof. Consider the tree of Figure 2, where k chains of two tasks each are connected to the root. The tasks are of two different types. The ones closer to the root (depicted as narrow and long) are called type S and have a threshold of $\delta_S = \epsilon$ and a size which is a multiple of ϵ, between 1 and k. The leaves are called type P (depicted as rectangular), have no threshold, i.e. $\delta_P \geq p$, and a computation size as given in the figure. The number of processors is $p = W$. Let ϵ be very small in comparison to W and k, and k be small in comparison to W.

With PROPSCHEDULING each chain is allocated p/k processors. Hence all chains are processed in parallel and the makespan is determined by the rightmost chain. In that chain, the S task takes k time units (due to $\delta_S = \epsilon$) plus the execution of the P task with all processors p/k:

$$M_{pro} = k + \frac{W}{p}k$$

In an optimal schedule, the processors are not all allocated statically, but first allocated to the rightmost chain to complete its P task, then to the one left to it (minus ϵ processors) and so on. The resulting makespan is:

$$M_{opt} = k + \frac{W}{p} + O(\epsilon)$$

The ratio of those makespans is then:

$$\frac{M_{pro}}{M_{opt}} = \frac{kp + kW}{kp + W} + O(1/\epsilon)$$

and with $p = W$ we get

$$\frac{M_{pro}}{M_{opt}} = \frac{2pk}{p(k+1)} + O(1/\epsilon)$$

When k and p tend to infinity, the ratio tends to 2.

□

PROPSCHEDULING has the following complexity. The recursive processing of sub-graphs in PROPORTIONALMAPPING is $O(|V|)$. A topological order for a directed acyclic graph is created in $O(|V| + |E|)$. Scheduling each task as early as possible is also $O(|V|+|E|)$, because each predecessor relationship corresponds to one edge which is only checked once. So the total complexity of PROPSCHEDULING is $O(|V| + |E|)$, which is $O(|V|)$ for the addressed SP-graphs (as $O(|E|) = O(|V|)$).

6 Greedy-filling

The proportional mapping studied in the previous section might be a common approach, but it does not make use of the malleability of tasks and

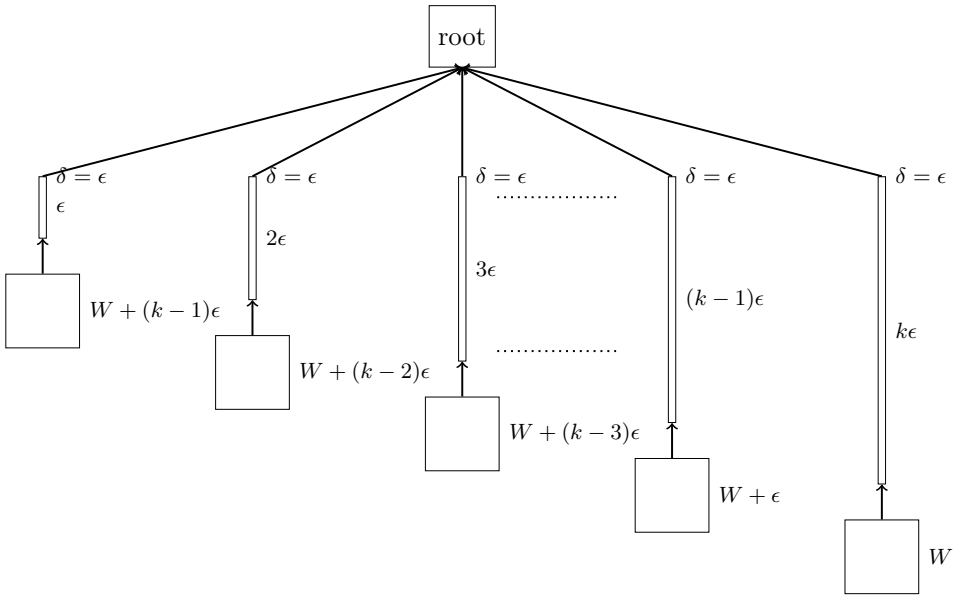
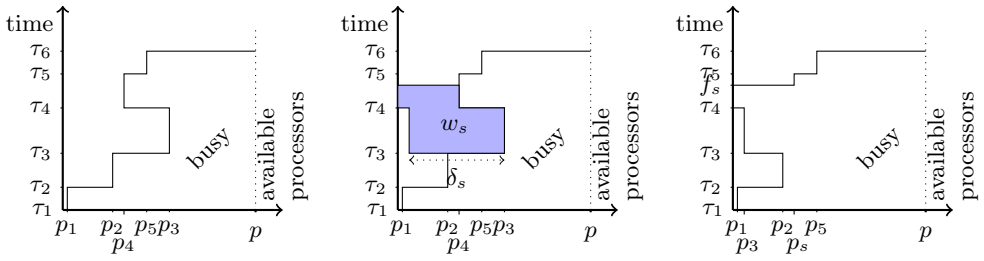


Figure 2: Tree for proof of tightness of approximation factor of PROPORTIONALMAPPING

is restricted to SP-graphs. In this section we study an algorithm, called GREEDYFILLING, that does not have these limitations. It considers one task at a time and greedily "fills" the possible empty spaces in the schedule.

An essential part in this heuristic is to keep track of how many processors are available at each time. For this, we propose to use a sorted set of pairs $\langle \tau_k, p_k \rangle$, called $availProcs$. Each pair represents the time τ_k from which on p_k processors are available until the time τ_{k+1} of the next pair $\langle \tau_{k+1}, p_{k+1} \rangle$. The sorted set property means that times are distinct and in ascending order. Also, for any two consecutive pairs, the numbers of processors are different (otherwise the later of the two pairs is not meaningful). All additions to, and modifications of, $availProcs$ keep these properties intact. In particular, if the numbers of processors of two consecutive pairs become identical, then the later pair is removed.Figure 3a illustrates this data structure with six pairs, $\langle \tau_k, p_k \rangle$, $k \in \{1, \ldots, 6\}$, depicting the available processors at the six times (note that $p_6 = p$).

Using the $availProcs$ data structure, the GREEDYFILLING algorithm is proposed in Algorithm 3. The principle of the algorithm is simple. First, each task is given a distinct priority. Then, the algorithm successively schedules the tasks, always choosing a free task T_s with highest priority. A task T is *free* when all of its parents have already been completely scheduled. When a task is scheduled, it is "poured" into empty spaces of the schedule. This means it starts as early as it can, which is at the earliest processor availability time from $availProcs$ that is at or after T_s's ready time $ready(s)$. A task T's



(a) Initial availability profile (b) Task T_s is added (c) After T_s was added

Figure 3: Processor availability behavior with GreedyFilling.

ready time is the highest finish time of its parents or zero if T has no parent (i.e., is a source task). The task T_s greedily uses as many processors as it can, which is the smaller of its threshold δ_s and the number of processors available at the time. Given that processor availability changes over time, we must determine the number of processors that can be used during each interval of availability times of $availProcs$ until T_s is completely scheduled. Figure 3b illustrates the adding of task T_s to the schedule. Its ready time, and hence it earliest start time, is τ_3 at which time it starts. During the interval $[\tau_3, \tau_4]$ the processors it can use is limited by its threshold δ_s. After τ_4 it uses all available processors p_k and completes at time f_s. This time becomes a new time-processors pair to be added to $availProcs$. The resulting availability profile after T_s has been added is given in Figure 3c.

It is interesting to note that if the total number of processors p and all thresholds are integers ($\delta_i \in \mathbb{N}, \forall T_i \in V$) then also all allocated processors p_i will be integers too.

Theorem 5. GreedyFilling *is a* $2 - \frac{\delta_{\min}}{p}$ *approximation for makespan minimization,with* $\delta_{\min} = \min_{T_i \in V} \delta_i$.

Proof. This proof is a transposition of the classical proof by Graham [16]. In any schedule produced by GreedyFilling, let T_1 be a task whose completion time is equal to the completion time of the processing of the whole task graph. We consider the last time t_1 prior to the start of the execution of T_1 at which not all processors were fully used. If the execution of T_1 did not start at time t_1 this is only because at least one ancestor T_2 of T_1 was executed at time t_1. Then, by induction we build a dependence path $\Phi = T_k \to \ldots \to T_2 \to T_1$ such that all processors are fully used during the execution of the entire schedule except, at least partially, during the execution of the tasks of Φ.

We consider the execution of any task T_i of Φ. At any time during the execution interval(s) of T_i (due to malleability it might be executed in disconnected intervals), either all processors are fully used, or some processors are (partially) idle and then, because of Step 10, δ_i processors are allocated

Algorithm 3: GREEDYFILLING ($G_{DAG} = (V, E, w, \delta)$, p)

```
1  Assign distinct priority priority(i) to each task T_i ∈ G_DAG
2  FreeTasks ← source tasks; availProcs ← ⟨0, p⟩
3  while FreeTasks ≠ ∅ do
4      T_s ← arg max_{T_j ∈ FreeTasks}{priority(j)}          // select task
5      w_Δ ← w_s                                  // Initialize work to be done
6      foreach ⟨τ_k, p_k⟩ ∈ availProcs in order with τ_k ≥ ready(s), skip if
       p_k = 0 do
7          if ⟨τ_k, p_k⟩ is last pair in availProcs then interval ← ∞
8          else interval ← τ_{k+1} − τ_k
9
10         usedProcs ← min{δ_s, p_k}
11         Replace ⟨τ_k, p_k⟩ by ⟨τ_k, p_k − usedProcs⟩ in availProcs
12         if w_Δ = w_s then t_s ← τ_k    // first part, set start time
13         if w_Δ/usedProcs > interval then
           w_Δ ← w_Δ − interval · usedProcs
14         else                          // rest of task fits in interval
15             f_s = τ_k + w_Δ/usedProcs                // set finish time
16             availProcs ← availProcs ∪ ⟨f_s, p_k⟩
17             break                              // break out foreach loop
18     FreeTasks ← FreeTasks \ {T_s} ∪ free children of T_s
```

to T_i. Therefore, during the execution of T_i, the total time during which not all processors are fully used is at most equal to $\frac{w_i}{\delta_i}$ and there are at most $p - \delta_i$ idle processors. Let *Idle* denote the sum of the idle areas, i.e., idle periods multiplied by idle processors, in the schedule. Then we have:

$$Idle \leq \sum_{i=1}^{k} \left(\frac{w_i}{\delta_i} \times (p - \delta_i) \right) \leq (p - \delta_{\min}) \times \sum_{i=1}^{k} \frac{w_i}{\delta_i} \leq (p - \delta_{\min}) M_{\text{OPT}}.$$

The last inequality comes from the fact that $\sum_{i=1}^{k} \frac{w_i}{\delta_i}$ is a lower bound on the execution time of the path Φ, and thus a lower bound on the optimal makespan.

Let *Used* denote the sum of the busy areas in the schedule. Then $Used = \sum_i w_i$ and thus $Used \leq p \times M_{\text{OPT}}$. Let M be the makespan of the considered schedule. Then we have:

$$p \times M = Idle + Used \leq (2p - \delta_{\min}) M_{\text{OPT}}.$$

□

Note that the above proof makes little reference to how the schedule of G_{DAG} has been constructed. The only important characteristic is that the

algorithm never leaves a processor deliberately idle if there are tasks that could be scheduled. Hence, the above approximation factor will also apply to other algorithms which adhere to that characteristic.

GreedyFilling has the following complexity. Assigning a priority to each task and always selecting the free task with the highest priority has a total complexity of $O(|V| \log |V|)$, for example by using a heap based priority queue for $FreeTasks$. Checking for newly freed tasks in the last line amortises to $O(|E|)$ for all tasks, as every edge is only checked once. For every task at most $O(|V|)$ processor availability intervals are checked, as the bounds of the intervals are finishing times of tasks of which there are at most $O(|V|)$. As this is done for each task, the complexity of this step is $O(|V|^2)$, which is also the total complexity of GreedyFilling as it dominates the other components.

7 Simulations

In this section, we compare the proposed GreedyFilling algorithm to two existing algorithms: PropScheduling, described earlier, and a 2-approximation, FlowFlex, proposed in [28] to solve a more general problem with multiple applications. In summary, FlowFlex first runs an algorithm similar to GreedyFilling on an infinite number of processors and then divide the obtained schedule into time intervals with constants allocations. On each such interval, if the total number of used processors exceed their actual number, the allocation is proportionally scaled down to fit the limit and the duration of this interval is increased. Note that PropScheduling does not take advantage of task malleability, while both GreedyFilling and FlowFlex do.

The following simulations are performed on two different datasets:

- First, we consider synthetic random SP-graphs of 5000 nodes, whose generation is detailed below. In order to compute a random SP-graph of $x > 1$ nodes, we obey the following recursive strategy: toss k uniformly in $[1, x-1]$; with a probability of $1/2$, build a series composition of two random SP-graphs of respectively k and $x-k$ nodes, and otherwise, build a parallel composition of these graphs. Then, in order to compute a random task (i.e., a random graph of $x = 1$ node), we choose its weight uniformly in $[1; 1000]$, and we choose its threshold to be proportional to its weight (with a factor of $f = 0.01$). This dataset has two variants, one where the threshold and the weight of a task are proportional ($\delta_i = \alpha \times w_i$, with $\alpha = 0.01$), denoted by Synth-Prop, and one where some randomness is introduced in this relation ($\alpha \in [0.001, 0.1]$ with log uniform distribution), denoted by Synth-Rand.
- Second, we consider the assembly trees of a set of actual sparse matri-

ces, which corresponds to the task graphs of the factorization of these matrices using multifrontal methods. In total, we consider more than 600 trees which may have a very large number of nodes, so we aggregate nodes such that at most 1000 nodes remain and their weight is larger than a given threshold (100 kFLOP). Here, the threshold of tasks is also proportional to their weight This dataset is denoted by TREES. The last dataset D_{rl} consists of assembly trees of a set of sparse matrices obtained from the University of Florida Sparse Matrix Collection (`http://www.cise.ufl.edu/research/sparse/matrices/`). The details concerning the computation of the data set can be found in [11]. The data set consists of more than 600 trees each containing between $2,000$ and $1,000,000$ nodes with a depth ranging from 12 to $75,000$. In order to simplify the dataset, we aggregated the trees by merging the smallest tasks to their parent, i.e., adding their weight, so that each tree now consists of $1,000$ nodes. Then, we set the threshold of each node being proportional to its weight.

In Figure 4, we present the makespan of PROPSCHEDULING, FLOWFLEX and GREEDYFILLING normalized by the classical lower bound on makespan: the maximum of the critical path and of the total work divided by the number of processors. Since all graphs do not have the same intrinsic parallelism, they exhibit different behaviors for the same number of processors. Thus, we first estimate the inter-task parallelism in a given graph using the following formula:

$$para = \frac{\text{makespan with all thresholds at 1 and infinite resources } (p = \infty)}{\text{makespan with all thresholds at 1 and } (p = 1)}$$

Then, the number of processors is normalized using this quantity: $p_{\text{norm}} = p/para$. We run the simulations on similar values of the normalized number of processors in order to better illustrate the behavior of the algorithms. In Figure 4, linked dots depict the average over all trees and the ribbon shows the 80% of the results that are between the first and ninth decile.

We first notice that PROPSCHEDULING and FLOWFLEX always give longer schedules. For very small or very large numbers of processors, the problem is simplified and all algorithms reach the same makespan: with very few processors, the thresholds do not influence the solution, and with an abundance of processors, there is no need to share the resources. Thus, the maximum gain of the proposed GREEDYFILLING is reached for an intermediate number of processors which depends on the algorithm and the intrinsic parallelism of the task graph. One may note that a threshold not proportional to the weight (as in SYNTH-RAND) leads to a larger maximum gap between the algorithms, which appears for fewer processors. Quantitatively, one can expect a maximum gain of 27% on a random graph, by switching from PROPSCHEDULING to GREEDYFILLING, and 24% when switching from FLOWFLEX to GREEDYFILLING.

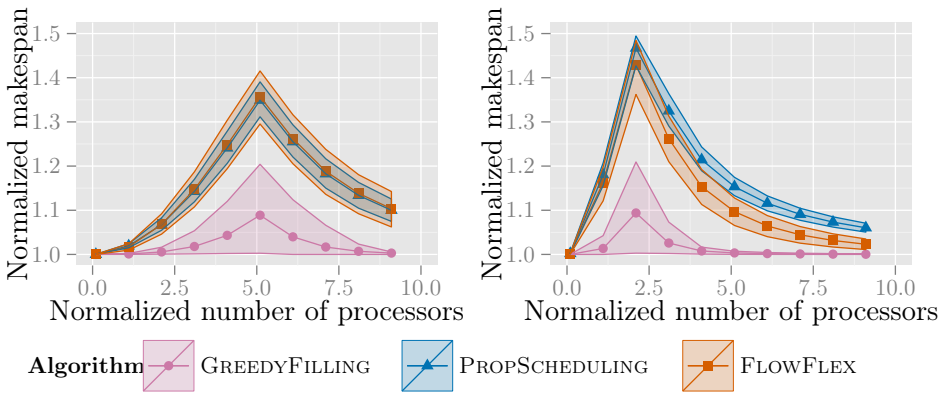


Figure 4: Normalized makespan of PropScheduling and GreedyFilling on Synth-Prop (left) and Synth-Rand (right).

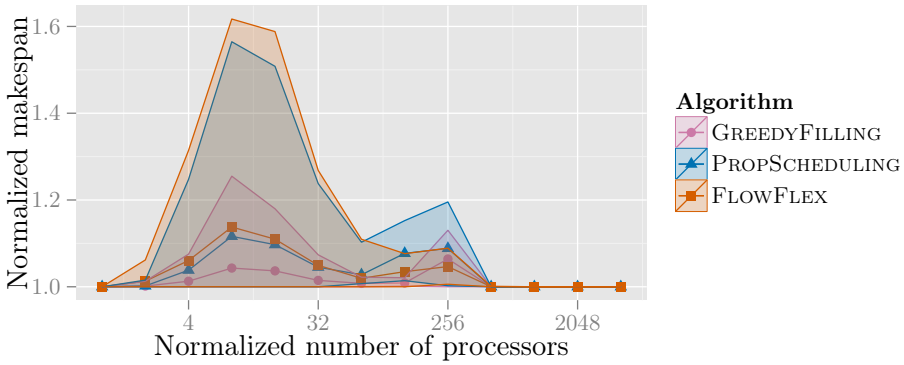


Figure 5: Normalized makespan of PropScheduling and GreedyFilling on a single family of Trees.

On the Trees dataset, GreedyFilling outperforms PropScheduling and FlowFlex in respectively 33% and 25% of the cases, and is outperformed by any of both in only 3% of the cases. For each tree, the algorithms exhibit the same behavior than the one outlined on Figure 4. However, the position and height of this characteristic shape widely vary among families (trees obtained from the same matrix using different ordering and amalgamation parameters). The maximum expected gain is smaller (around 15%). On Figure 5, we present the makespans achieved by the three algorithms normalized by the same lower bound as before on the 8 trees of a single family of Trees. The linked dots depict the average over this family and the ribbon shows the minimum and maximum makespans. The interested reader can find all results listed in Appendix A, each graph corresponding to a single family of Trees.

To explain the good behavior of GreedyFilling compared to its com-

petitors, we remark that it takes advantage of task malleability, contrarily to PropScheduling. Moreover, FlowFlex may produce gaps during its first phase, that is, time intervals with low processor utilization, which cannot be filled later; it only uses task malleability to cope with resource limitation.

8 Conclusion

In this paper, we have studied how to schedule graphs of malleable tasks with a perfect but bounded parallelism. We first proved the NP-completeness of the makespan minimization problem. Then, we show that the existing PropScheduling scheduling heuristic is a 2-approximation algorithm. Then we introduced GreedyFilling, a simple greedy policy that we also proved to be a 2-approximation algorithm. Using extensive simulations, we compare these two heuristics together and also against FlowFlex, another 2-approximation proposed in [28]. GreedyFilling gives the smallest makespan in most configurations and achieves noticeably lower makespans, both on synthetic and realistic graphs. For instance, on random graphs, for the maximum expected gain in makespan is around 25%.

This work could be extended by focusing on the more constrained moldable model in which the processing power alloted to a task cannot vary during its execution. Both theoretical and practical directions could be considered. First, it would be interesting to quantify the theoretical gain of allowing malleability and, then, to design certified and efficient heuristics for the moldable model.

References

[1] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23(2), 187–198 (2011)

[2] Beaumont, O., Bonichon, N., Eyraud-Dubois, L., Marchal, L.: Minimizing weighted mean completion time for malleable tasks scheduling. In: Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. pp. 273–284 (May 2012)

[3] Bharathi, S., Chervenak, A., Deelman, E., Mehta, G., Su, M.H., Vahi, K.: Characterization of scientific workflows. In: Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on. pp. 1 –10 (nov 2008)

[4] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., Dongarra, J.J.: PaRSEC: Exploiting heterogeneity for enhancing scalability. Computing in Science & Engineering 15(6), 36–45 (2013)

[5] Cordasco, G., Chiara, R.D., Rosenberg, A.L.: Assessing the computational benefits of area-oriented dag-scheduling. In: Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part I. pp. 180–192 (2011)

[6] Drozdowski, M., Kubiak, W.: Scheduling parallel tasks with sequential heads and tails. Annals of Operations Research 90(0), 221–246 (1999)

[7] Drozdowski, M.: Scheduling multiprocessor tasks — an overview. European Journal of Operational Research 94(2), 215 – 230 (1996)

[8] Drozdowski, M.: Scheduling parallel tasks – algorithms and complexity. In: Leung, J. (ed.) Handbook of Scheduling. Chapman and Hall/CRC (2004)

[9] Du, J., Leung, J.Y.T.: Complexity of scheduling parallel task systems. SIAM Journal on Discrete Mathematics 2(4), 473–487 (1989)

[10] Dutot, P., N'Takpé, T., Suter, F., Casanova, H.: Scheduling parallel task graphs on (almost) homogeneous multicluster platforms. IEEE TPDS 20(7), 940–952 (2009)

[11] Eyraud-Dubois, L., Marchal, L., Sinnen, O., Vivien, F.: Parallel scheduling of task trees with limited memory. Research Report RR-8606, INRIA (2014)

[12] Fan, L., Zhang, F., Wang, G., Liu, Z.: An effective approximation algorithm for the malleable parallel task scheduling problem. Journal of Parallel and Distributed Computing 72(5), 693–704 (2012), `http://www.sciencedirect.com/science/article/pii/S0743731512000238`

[13] Garey, M.R., Johnson, D.S.: Computers and Intractability, a Guide to the Theory of NP-Completeness. W. H. Freeman and Company (1979)

[14] Gautier, T., Besseron, X., Pigeon, L.: KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: International Workshop on Parallel Symbolic Computation. pp. 15–23 (2007)

[15] González-Escribano, A., van Gemund, A.J.C., Cardeñoso-Payo, V.: Mapping unstructured applications into nested parallelism. In: High Performance Computing for Computational Science - VECPAR 2002, 5th International Conference, Porto, Portugal, June 26-28, 2002, Selected Papers and Invited Talks. pp. 407–420 (2002)

[16] Graham, R.L.: Bounds for certain multiprocessing anomalies. Bell System Technical Journal 45(9), 1563–1581 (1966)

[17] Guermouche, A., Marchal, L., Simon, B., Vivien, F.: Scheduling trees of malleable tasks for sparse linear algebra. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015: Parallel Processing, Lecture Notes in Computer Science, vol. 9233, pp. 479–490. Springer Berlin Heidelberg (2015), http://dx.doi.org/10.1007/978-3-662-48096-0_37

[18] Günther, E., König, F., Megow, N.: Scheduling and packing malleable and parallel tasks with precedence constraints of bounded width. Journal of Combinatorial Optimization 27(1), 164–181 (2014)

[19] Hugo, A., Guermouche, A., Wacrenier, P., Namyst, R.: Composing multiple StarPU applications over heterogeneous machines: A supervised approach. IJHPCA 28(3), 285–300 (2014)

[20] Hunold, S.: One step toward bridging the gap between theory and practice in moldable task scheduling with precedence constraints. Concurrency and Computation: Practice and Experience 27(4), 1010–1026 (2015)

[21] Jansen, K., Zhang, H.: An approximation algorithm for scheduling malleable tasks under general precedence constraints. In: Deng, X., Du, D.Z. (eds.) Algorithms and Computation, Lecture Notes in Computer Science, vol. 3827, pp. 236–245. Springer Berlin Heidelberg (2005)

[22] Kell, N., Havill, J.: Improved upper bounds for online malleable job scheduling. Journal of Scheduling 18(4), 393–410 (2015), http://dx.doi.org/10.1007/s10951-014-0406-9

[23] Lepère, R., Trystram, D., Woeginger, G.J.: Approximation algorithms for scheduling malleable tasks under precedence constraints. International Journal of Foundations of Computer Science 13(04), 613–627 (2002), http://www.worldscientific.com/doi/abs/10.1142/S0129054102001308

[24] Liu, J.W.H.: The role of elimination trees in sparse factorization. SIAM Journal on Matrix Analysis and Applications 11(1), 134–172 (1990)

[25] Makarychev, K., Panigrahi, D.: Precedence-constrained scheduling of malleable jobs with preemption. In: ICALP 2014. pp. 823–834 (2014)

[26] McNaughton, R.: Scheduling with deadlines and loss functions. Management Science 6(1), 1–12 (1959)

[27] Mitchell, M.: Creating minimal vertex series parallel graphs from directed acyclic graphs. In: Australasian Symposium on Information Visualisation, InVis.au, Christchurch, New Zealand, 23-24 January 2004. pp. 133–139 (2004)

[28] Nagarajan, V., Wolf, J., Balmin, A., Hildrum, K.: Flowflex: Malleable scheduling for flows of mapreduce jobs. In: Middleware 2013, pp. 103–122. Springer (2013)

[29] OpenMP Architecture Review Board: OpenMP application program interface, version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf` (Jul 2013)

[30] Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with StarSs. IJHPCA 23(3), 284–299 (2009)

[31] Pothen, A., Sun, C.: A mapping algorithm for parallel sparse cholesky factorization. SIAM Journal on Scientific Computing 14(5), 1253–1257 (1993)

[32] Prasanna, G.N.S., Musicus, B.R.: Generalized multiprocessor scheduling and applications to matrix computations. IEEE TPDS 7(6), 650–664 (1996)

[33] Prasanna, G.N.S., Musicus, B.R.: The optimal control approach to generalized multiprocessor scheduling. Algorithmica 15(1), 17–49 (1996)

[34] Radulescu, A., Nicolescu, C., van Gemund, A.J.C., Jonker, P.: CPR: mixed task and data parallel scheduling for distributed systems. In: IPDPS'01. p. 39 (2001)

[35] Sbîrlea, A.S., Agrawal, K., Sarkar, V.: Elastic tasks: Unifying task parallelism and SPMD parallelism with an adaptive runtime. In: Euro-Par 2015. pp. 491–503 (2015)

[36] Sinnen, O.: Task Scheduling for Parallel Systems. Wiley series on parallel and distributed computing, Wiley (2007)

[37] Valdes, J., Tarjan, R.E., Lawler, E.L.: The recognition of series parallel digraphs. SIAM J. Comput. 11(2), 298–313 (1982)

[38] Vizing, V.: Minimization of the maximum delay in servicing systems with interruption. USSR Computational Mathematics and Mathematical Physics 22(3), 227 – 233 (1982)

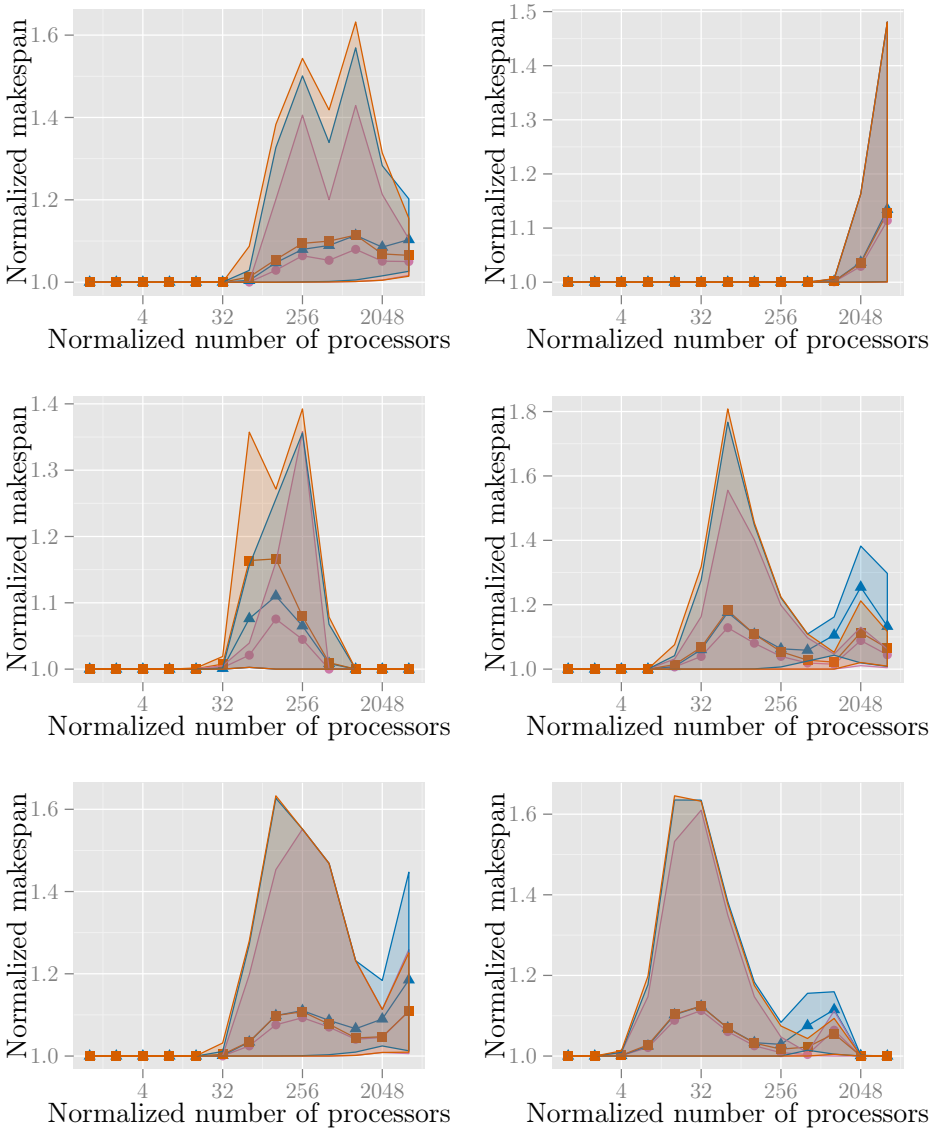
[39] Wang, Q., Cheng, K.H.: A heuristic of scheduling parallel tasks and its analysis. SIAM Journal on Computing 21(2), 281–294 (1992)

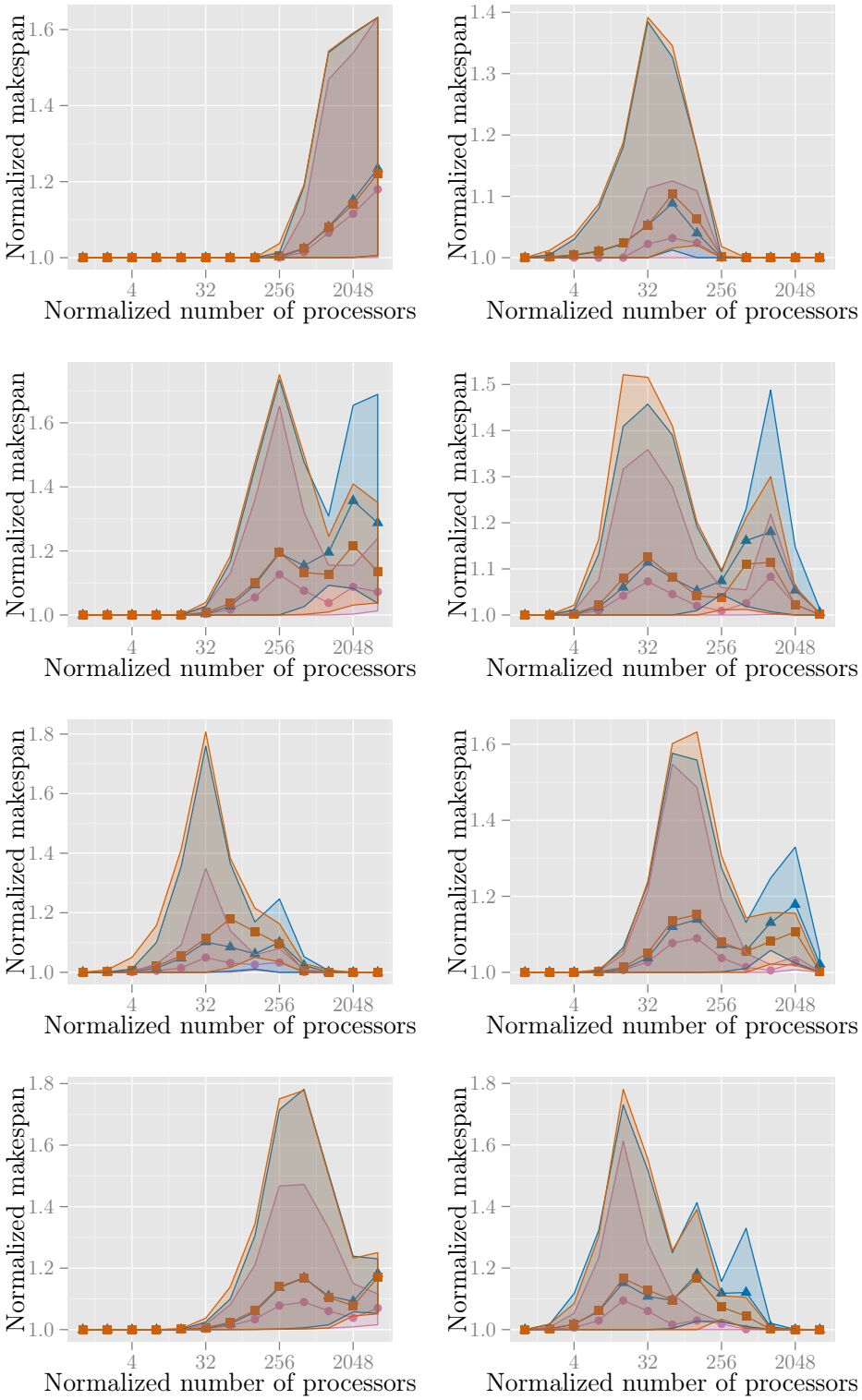
[40] Zinder, Y., Walker, S.: Scheduling flexible multiprocessor tasks on parallel machines. In: The 9th Workshop on Models and Algorithms for Planning and Scheduling Problems (2009), available online at `www.ctit.utwente.nl/library/proceedings/wp0911.pdf`.

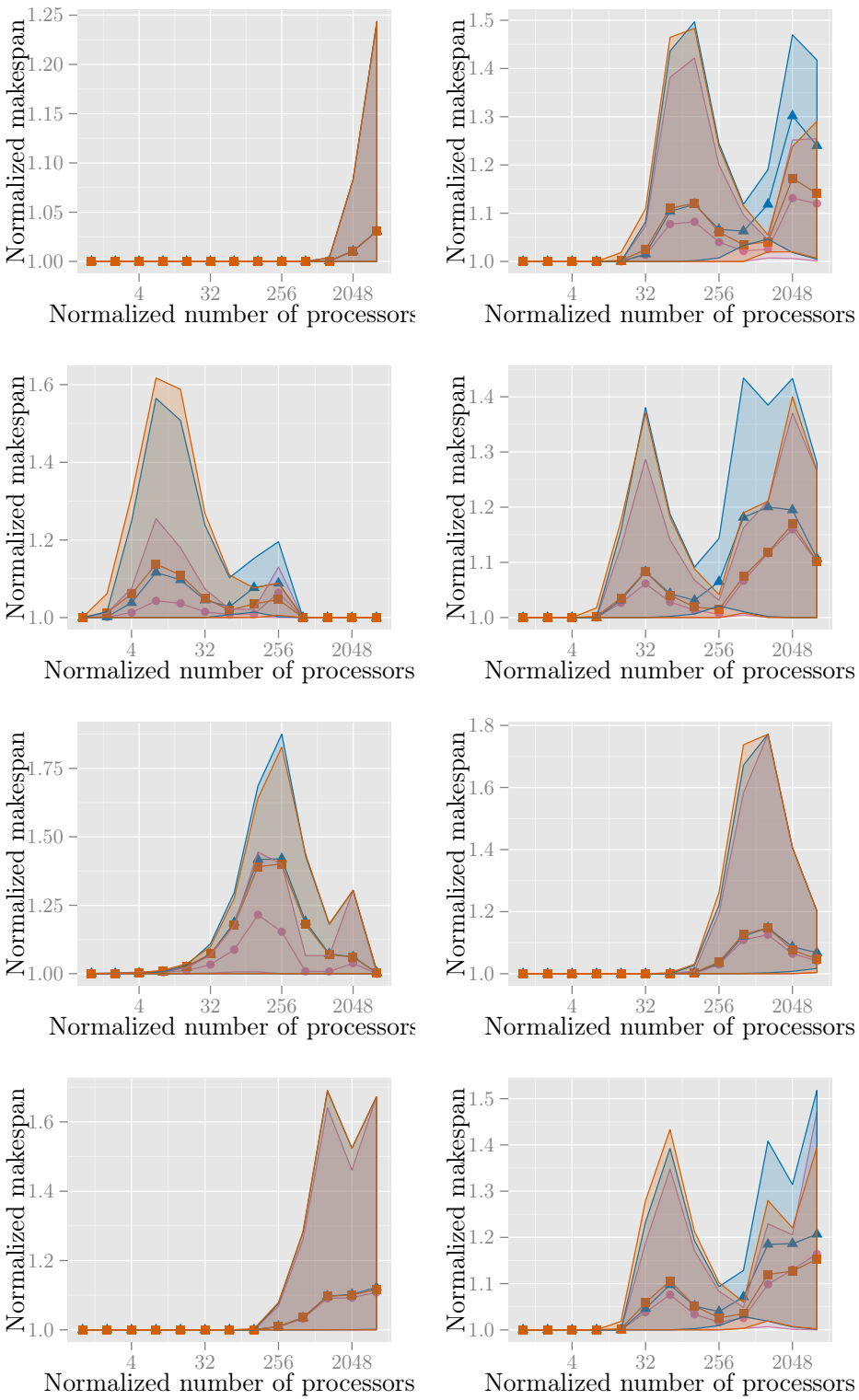
A Extensive simulation results

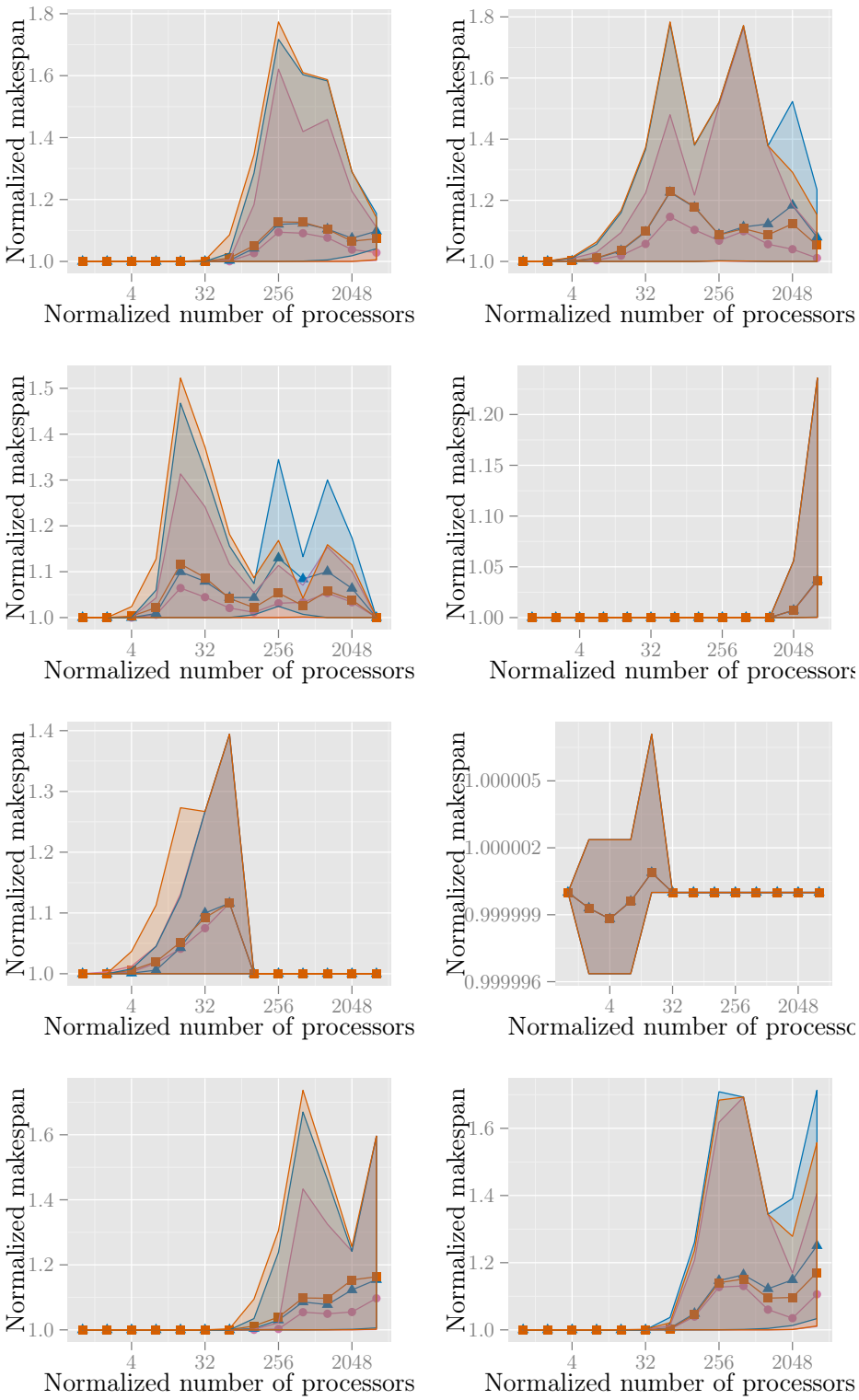
We list in this appendix the results presented in Section 7.

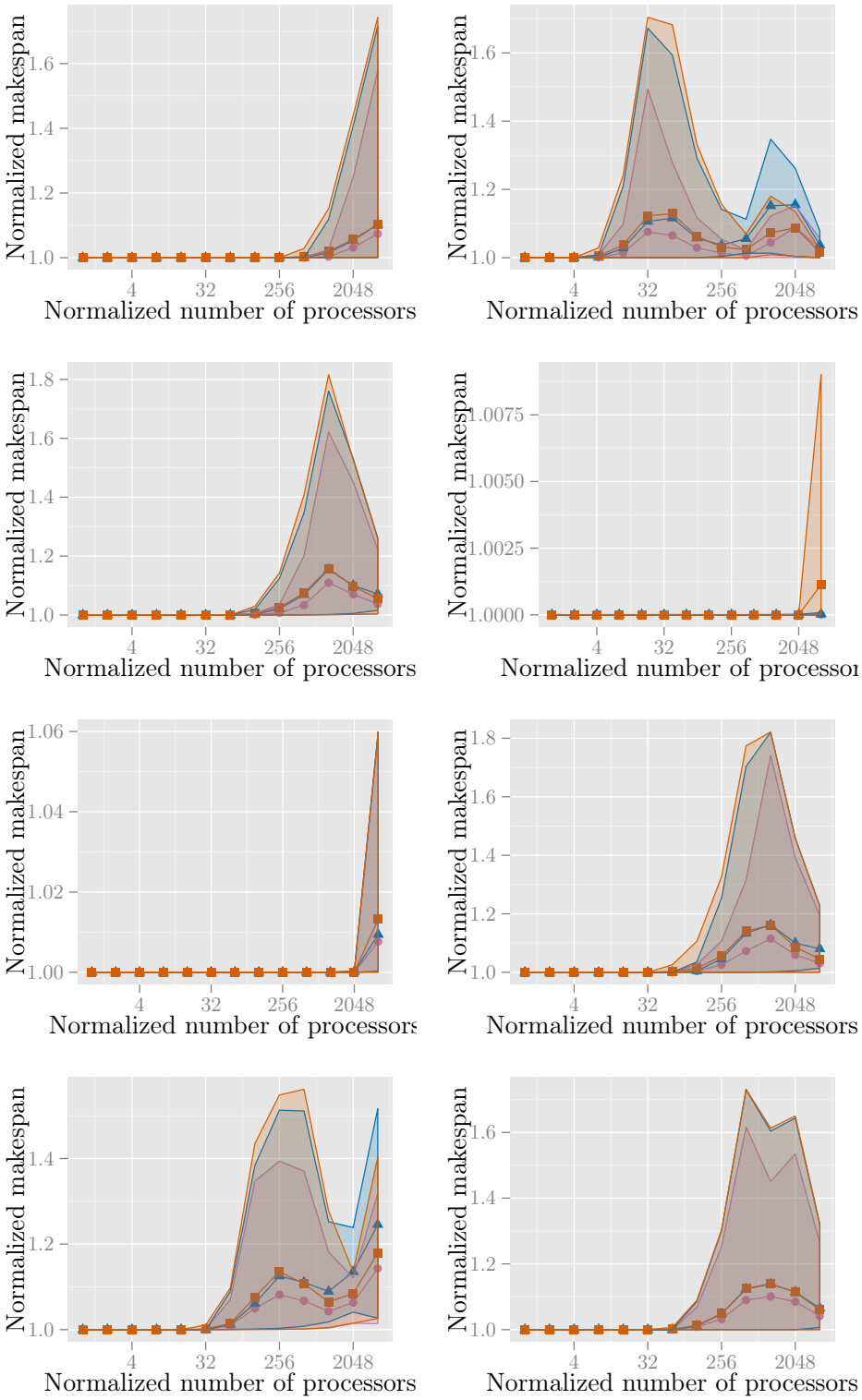
Figure 6: Legend common to all graphs of Appendix A.

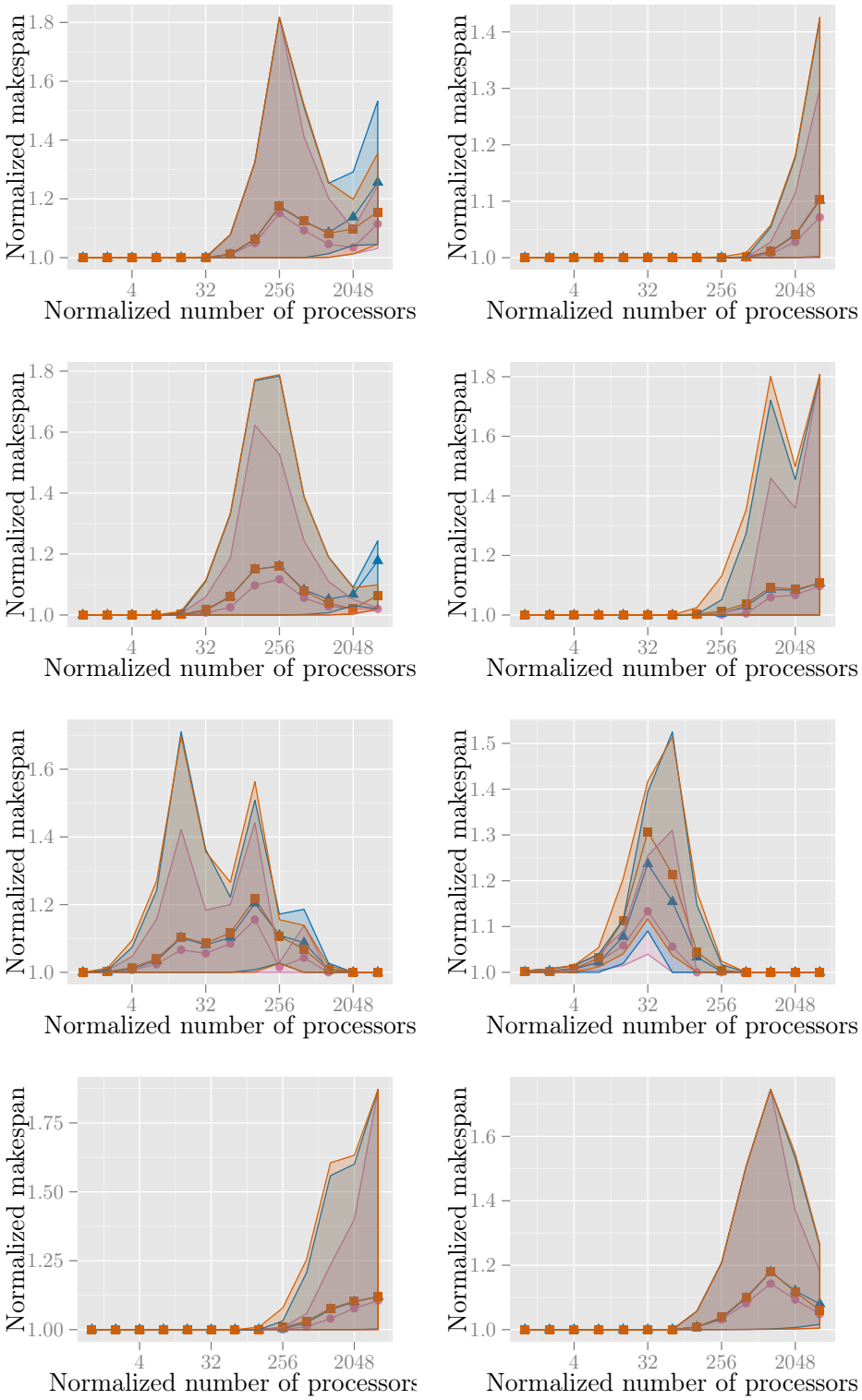


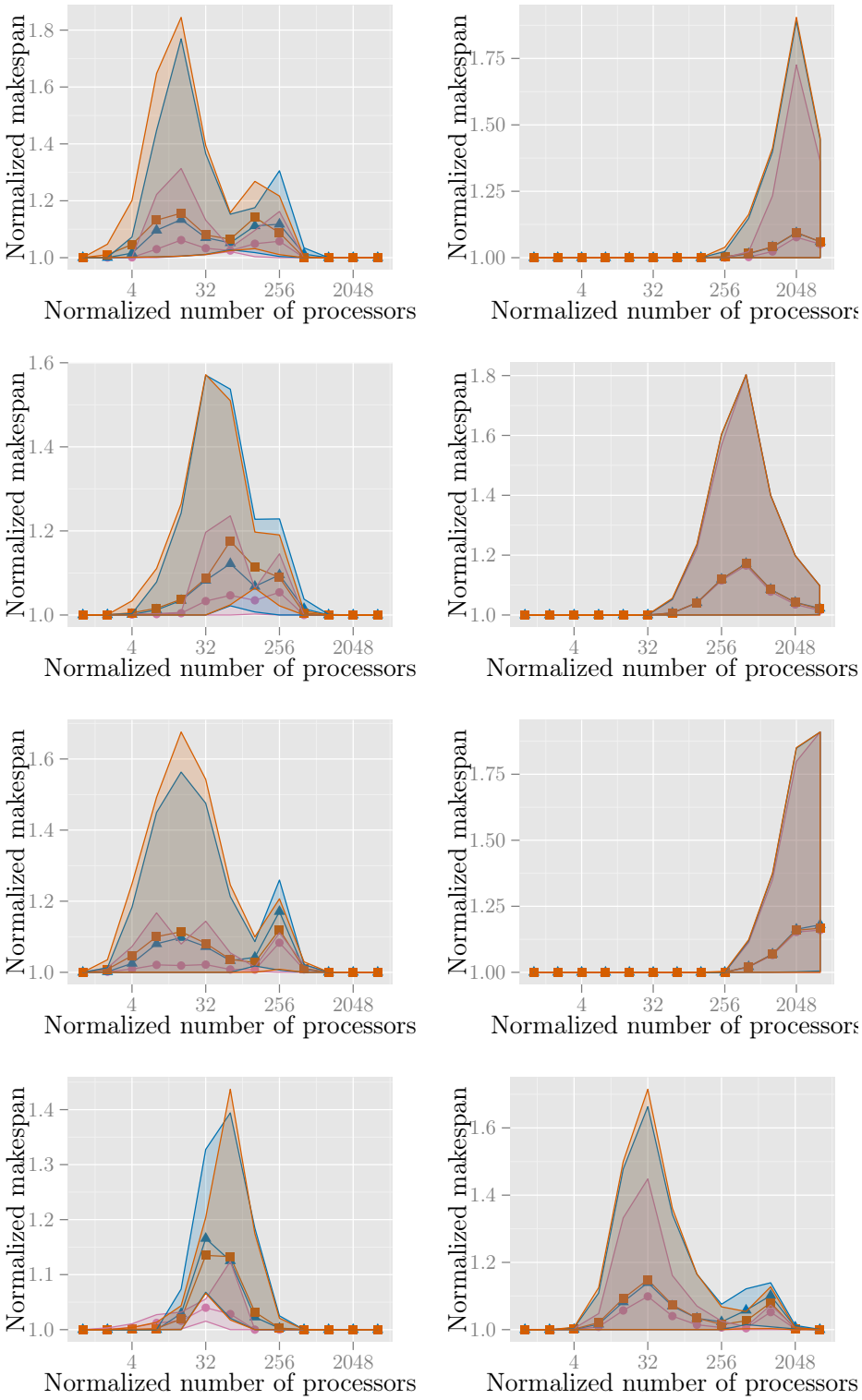

Normalized makespan
1.0
1.2
1.4
1.6
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.1
1.2
1.3
1.4
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.2
1.4
1.6
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.1
1.2
1.3
1.4
1.5
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.2
1.4
1.6
1.8
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.2
1.4
1.6
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.2
1.4
1.6
1.8
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.2
1.4
1.6
1.8
4
32
256
2048
Normalized number of processors

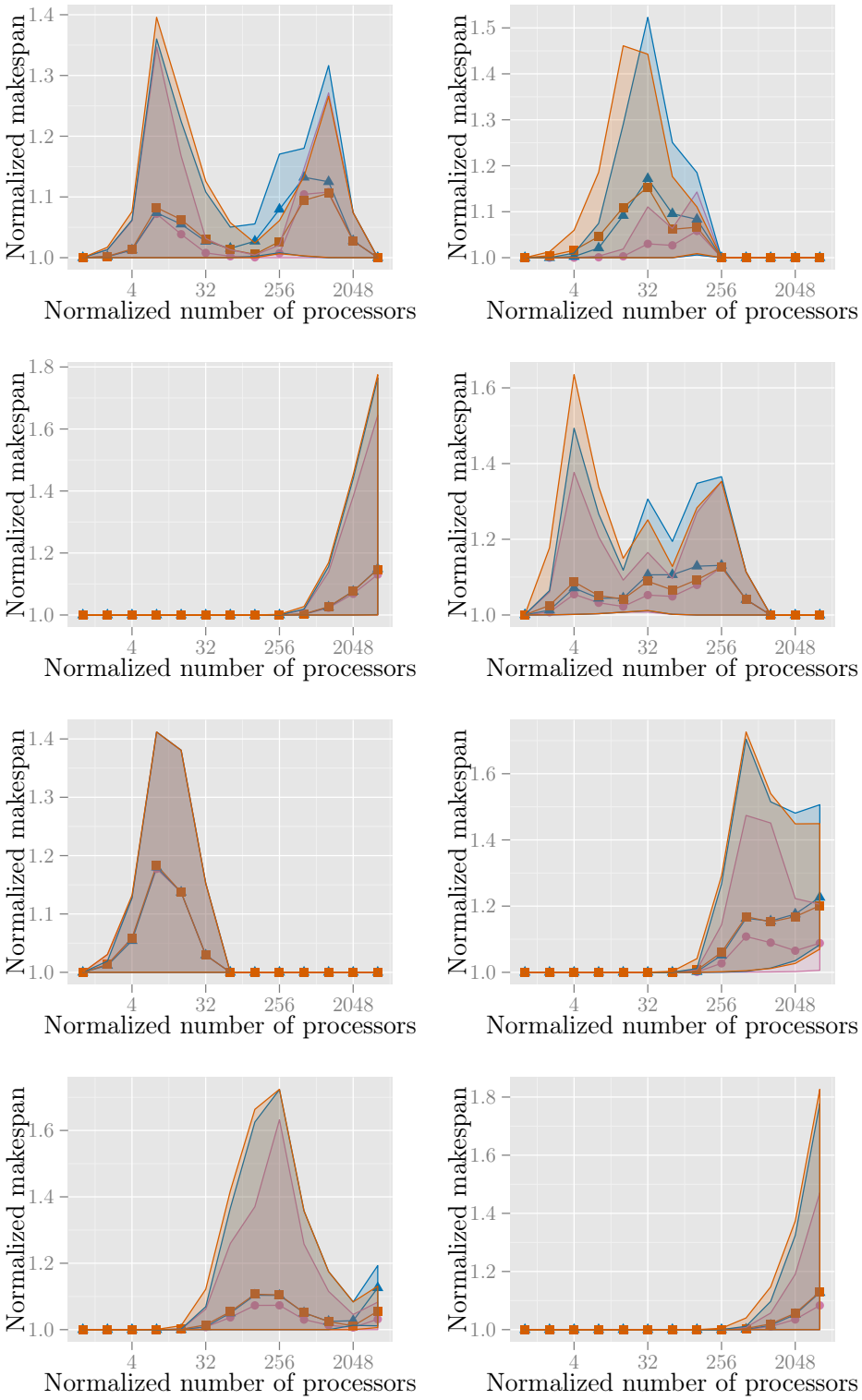

Normalized makespan
Normalized number of processors
4
32
256
2048
1.25
1.20
1.15
1.10
1.05
1.00
1.5
1.4
1.3
1.2
1.1
1.0
1.6
1.75
1.50
1.8

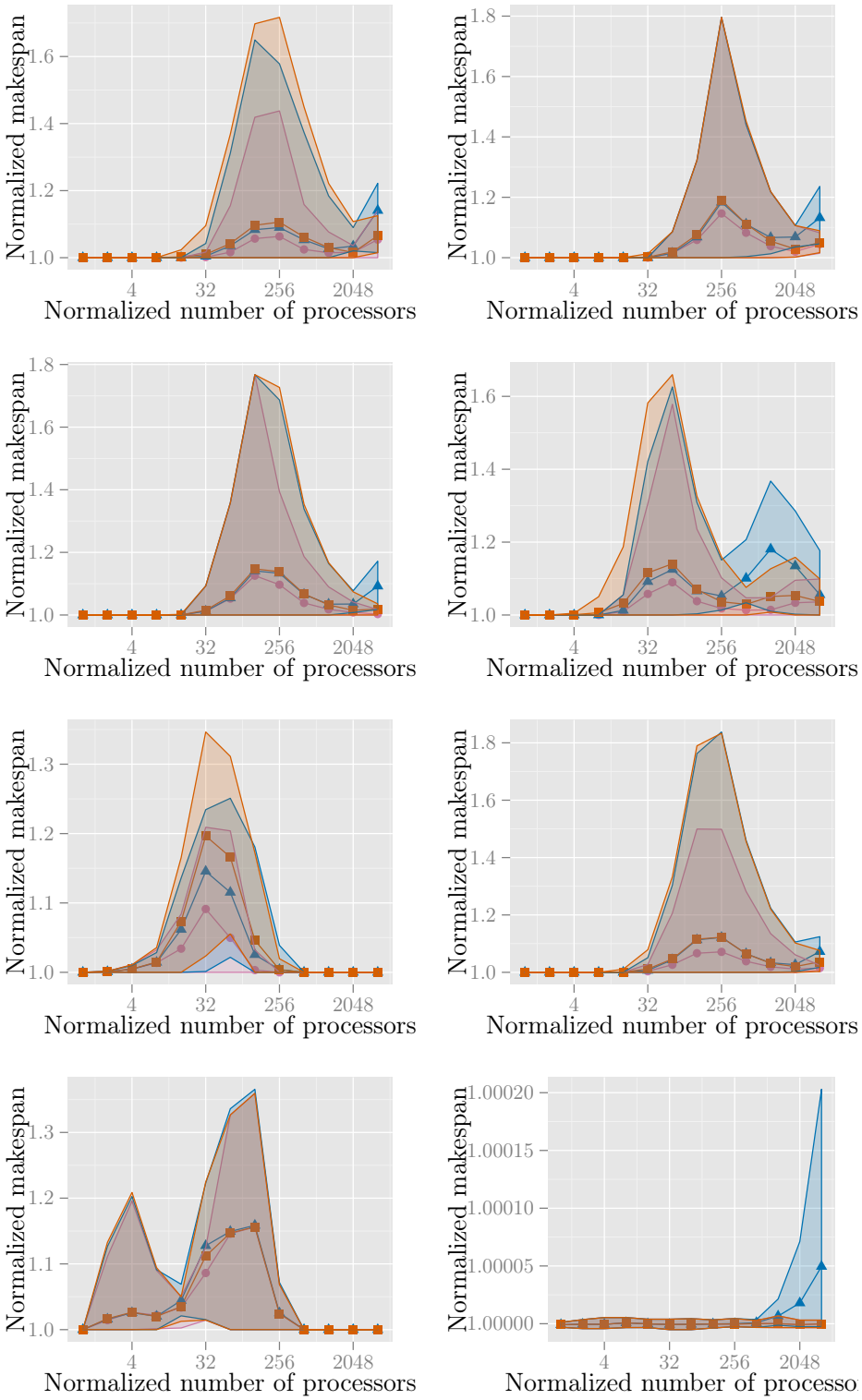

Normalized makespan
1.8
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.8
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.5
1.4
1.3
1.2
1.1
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.20
1.15
1.10
1.05
1.00
4
32
256
2048
Normalized number of processors
Normalized makespan
1.4
1.3
1.2
1.1
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.000005
1.000002
0.999999
0.999996
4
32
256
2048
Normalized number of processc
Normalized makespan
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors

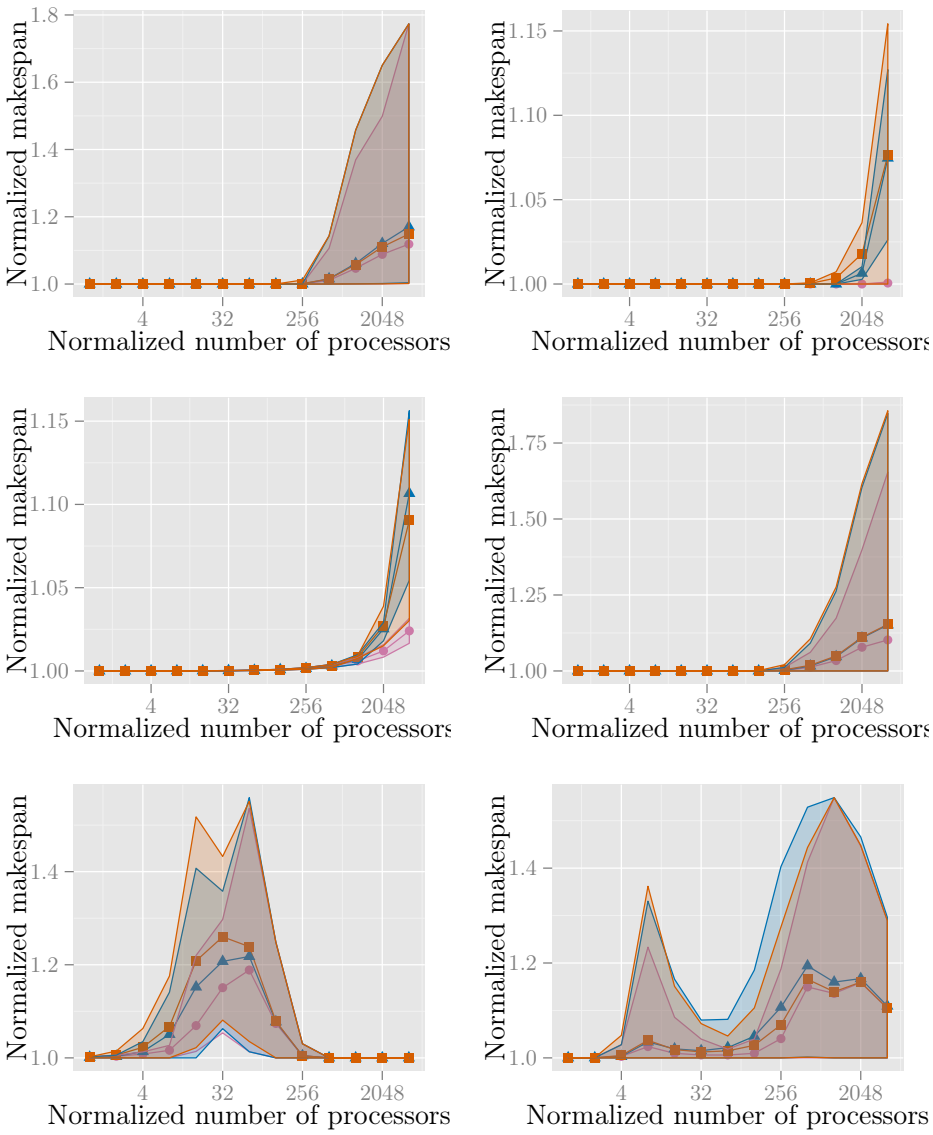

Normalized makespan
1.0
1.2
1.4
1.6
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.2
1.4
1.6
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.2
1.4
1.6
1.8
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0000
1.0025
1.0050
1.0075
4
32
256
2048
Normalized number of processor
Normalized makespan
1.00
1.02
1.04
1.06
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.2
1.4
1.6
1.8
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.2
1.4
4
32
256
2048
Normalized number of processors
Normalized makespan
1.0
1.2
1.4
1.6
4
32
256
2048
Normalized number of processors

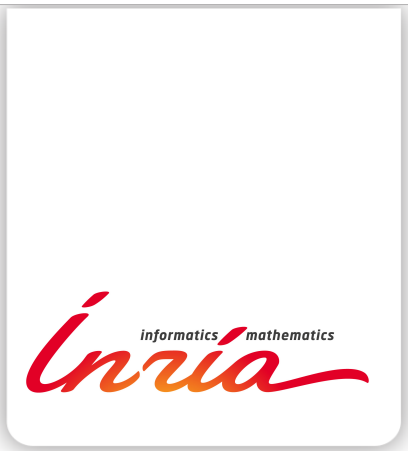

Normalized makespan
1.8
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.4
1.3
1.2
1.1
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.8
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.8
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.5
1.4
1.3
1.2
1.1
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.75
1.50
1.25
1.00
4
32
256
2048
Normalized number of processors
Normalized makespan
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors


Normalized makespan
1.8
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.75
1.50
1.25
1.00
4
32
256
2048
Normalized number of processors
Normalized makespan
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.8
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.75
1.50
1.25
1.00
4
32
256
2048
Normalized number of processors
Normalized makespan
1.4
1.3
1.2
1.1
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors


Normalized makespan
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.8
4
32
256
2048
Normalized number of processors


Normalized makespan
Normalized number of processors
4
32
256
2048
1.0
1.1
1.2
1.3
1.4
1.6
1.8
1.00000
1.00005
1.00010
1.00015
1.00020


Normalized makespan
1.8
1.6
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.15
1.10
1.05
1.00
4
32
256
2048
Normalized number of processors
Normalized makespan
1.15
1.10
1.05
1.00
4
32
256
2048
Normalized number of processors
Normalized makespan
1.75
1.50
1.25
1.00
4
32
256
2048
Normalized number of processors
Normalized makespan
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors
Normalized makespan
1.4
1.2
1.0
4
32
256
2048
Normalized number of processors



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