Scheduling Parallel Tasks under Multiple Resources: List Scheduling vs. Pack Scheduling

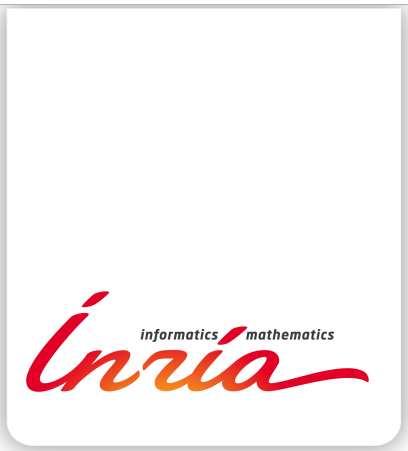
Hongyang Sun, Redouane Elghazi, Ana Gainaru, Guillaume Aupy, Padma Raghavan

Scheduling Parallel Tasks under Multiple Resources: List Scheduling vs. Pack Scheduling

Hongyang Sun, Redouane Elghazi, Ana Gainaru, Guillaume Aupy, Padma Raghavan

RESEARCH REPORT
N° 9140
January 2018
Project-Team Tadaam

ISSN 0249-6399 ISRN INRIA/RR--9140--FR+ENG



Scheduling Parallel Tasks under Multiple Resources: List Scheduling vs. Pack Scheduling

Hongyang Sun*, Redouane Elghazi†, Ana Gainaru*, Guillaume Aupy‡, Padma Raghavan*

Project-Team Tadaam

Research Report n° 9140 — January 2018 — 29 pages

* Vanderbilt University, Nashville TN, USA
† École Normale Supérieure de Lyon, France
‡ Inria & University of Bordeaux

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Abstract: Scheduling in High-Performance Computing (HPC) has been traditionally centered around computing resources (e.g., processors/cores). The ever-growing amount of data produced by modern scientific applications start to drive novel architectures and new computing frameworks to support more efficient data processing, transfer and storage for future HPC systems. This trend towards data-driven computing demands the scheduling solutions to also consider other resources (e.g., I/O, memory, cache) that can be shared amongst competing applications. In this paper, we study the problem of scheduling HPC applications while exploring the availability of multiple types of resources that could impact their performance. The goal is to minimize the overall execution time, or makespan, for a set of moldable tasks under multiple-resource constraints. Two scheduling paradigms, namely, list scheduling and pack scheduling, are compared through both theoretical analyses and experimental evaluations. Theoretically, we prove, for several algorithms falling in the two scheduling paradigms, tight approximation ratios that increase linearly with the number of resource types. As the complexity of direct solutions grows exponentially with the number of resource types, we also design a strategy to indirectly solve the problem via a transformation to a single-resource-type problem, which can significantly reduce the algorithms' running times without compromising their approximation ratios. Experiments conducted on Intel Knights Landing with two resource types (processor cores and high-bandwidth memory) and simulations designed on more resource types confirm the benefit of the transformation strategy and show that pack-based scheduling, despite having a worse theoretical bound, offers a practically promising and easy-to-implement solution, especially when more resource types need to be managed.

Key-words: Scheduling; KNL.

Ordonnancement de tâches parallèles sous multiples ressources : Ordonnancement de Listes vs ordonnancement de Packs

Résumé : L'ordonnancement en Calcul Haute-Performance est traditionnellement centré autour des ressources de calculs (processeurs, cœurs). Suite à l'explosion des quantités de données dans les applications scientifiques, de nouvelles architectures et nouveaux paradigmes de calcul apparaissent pour soutenir plus efficacement les calculs dirigés par l'accès aux données. Nous présentons ici des solutions algorithmiques qui prennent en compte cette multitude de ressources.

Mots-clés : Ordonnancement, KNL.

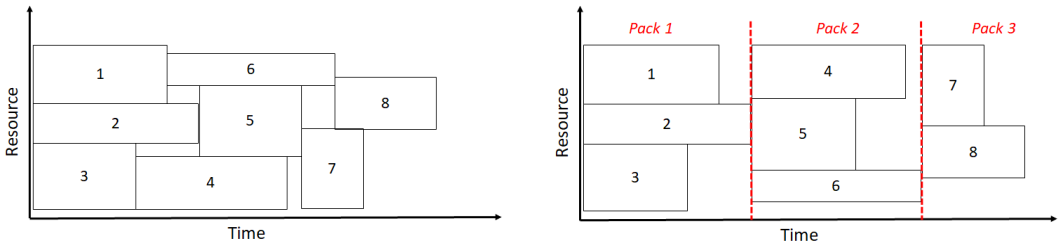
1 Introduction

Scientific discovery now relies increasingly on data and its management. Large-scale simulations produce an ever-growing amount of data that has to be processed and analyzed. As an example, it was estimated that the Square Kilometer Array (SKA) could generate an exabyte of raw data a day by the time it is completed [3]. In the past, scheduling in High-Performance Computing (HPC) has been mostly compute-based (with a primary focus on the processor/core resource) [32]. In general, data management is deferred as a second step separate from the data generation. The data created would be gathered and stored on disks for data scientists to study later. This will no longer be feasible, as studies have shown that these massive amounts of data are becoming increasingly difficult to process [11].

To cope with the big-data challenge, new frameworks such as *in-situ* and *in-transit* computing [9] have emerged. The idea is to use a subset of the computing resources to process data as they are created. The results of this processing can then be used to re-inject information in the subsequent simulation in order to move the data towards a specific direction. In addition, architectural improvements have been designed to help process the data at hand. Emerging platforms are now equipped with more levels of memory/storage (e.g., NVRAM, burst buffers, data nodes) and better data transfer support (e.g., high-bandwidth memory, cache-partitioning technology) that can be exploited by concurrent applications.

In view of such trend towards data-driven computing, this work presents techniques on how to efficiently schedule competing applications given this multi-tier memory hierarchy. We go even further by considering in our scheduling model any resource (e.g., I/O, memory, cache) that can be shared or partitioned amongst these applications and that can impact their performance. The goal is to design, analyze and evaluate scheduling solutions that explore the availability of these multiple resource types in order to reduce the overall execution time, or makespan, for a set of applications. The applications in this work are modeled as independent *moldable* tasks whose execution times vary depending on the different resources available to them (see Section 3 for the formal definition of moldable tasks). This general model is particularly representative of the in-situ/in-transit workflows, where different analysis functions have to be applied to different subsets of the data generated.

We focus on two scheduling paradigms, namely, list scheduling and pack scheduling. In *list scheduling*, all tasks are first organized in a priority list. Then, the tasks are assigned in sequence to the earliest available resources that can fit them. In *pack scheduling*, the tasks are first partitioned into a series of packs, which are then executed one after another. Tasks within each pack are scheduled concurrently and a pack cannot start until all tasks in the previous pack have completed. Both scheduling paradigms have been studied by the literature, mostly under a single-resource constraint (see Section 2 for a literature review). Figure 1 shows an example of applying both scheduling paradigms to a same set of tasks under a single-resource constraint.



(a) list scheduling (b) pack scheduling

Figure 1: An example of list scheduling and pack scheduling for a same set of tasks (packs separated by dotted red lines).

Compared to some more sophisticated scheduling algorithms that could guarantee better theoretical bounds (e.g., [24, 20]), list-based and pack-based algorithms often produce simple yet efficient schedules that can be easily implemented by practical runtime systems. Furthermore, these algorithms can be adopted to the online and/or heterogeneous scheduling environment with minimal change, thus offering more general applicability to a wide range of scheduling scenarios. Between these two paradigms, list scheduling can make better use of the available resources (at least in theory) by reducing the idle times between the tasks and thus maximizing the resource utilization. However, pack scheduling has been advocated by some recent studies [2, 29] for being practically valuable due to its ease of implementation (as batch processing) and for incurring less scheduling overhead. The relative merits of the two scheduling paradigms are yet to be evaluated under multiple-resource constraints.

In this paper, we provide a comprehensive study of the two scheduling paradigms through both theoretical analyses and experimental evaluations in the presence of multiple resource types. In this context, we make the following contributions:

- We present several scheduling algorithms under multiple-resource constraint in both list and pack scheduling paradigms, and prove tight approximation ratios for these algorithms ($2d$ for list and $2d + 1$ for pack, where d is the number of resource types).
- We design a transformation strategy that reduces the problem of scheduling multiple resource types to single-resource-type scheduling, thereby significantly reducing the algorithms' running times without compromising the approximation ratios.
- We conduct experimental evaluations of these scheduling algorithms on the Intel Xeon Phi Knights Landing (KNL) processor, which offers two resource management options (processor cores and high-bandwidth memory).

- We project the performance of both list- and pack-based scheduling solutions under more than two resource types using simulations on synthetic parallel workloads that extend some classical speedup profiles.

Overall, the experimental/simultion results confirm the benefit of the transformation strategy and show that pack scheduling, despite having a slightly worse theoretical bound, can indeed offer a practically promising yet easy-to-implement solution compared to its list counterpart, especially when more resource types need to be managed. The insights derived from these results will be especially relevant to the emerging architectures that will likely offer in the resource management systems more scheduling possibilities across the memory hierarchy to embrace the shift towards data-driven computing.

The rest of this paper is organized as follows. Section 2 reviews some related work. Section 3 formally defines the multiple-resource-type scheduling model. Section 4 presents several scheduling solutions and proves their approximation ratios. Section 5 is devoted to experimental evaluation of different algorithms and heuristics. Finally, Section 6 concludes the paper with hints on future directions.

2 Related Work

2.1 Parallel Task Models

Many parallel task models exist in the scheduling literature. Feitelson [10] classified parallel tasks into three categories, namely, rigid tasks, moldable tasks and malleable tasks. A *rigid* task requires a fixed amount of resources (e.g., number of processors) to execute. A *moldable* task can be executed with a varying amount of resources, but once the task has started the resource allocation cannot be changed. The most flexible model is the one of *malleable* tasks, which allows the amount of resources executing the task to vary at any time during the execution. For the latter two models, two important parameters are the speedup function and the total area (or work). The *speedup function* relates the execution time of a task to the amount of resources allocated to it, and the *total area* is defined as the product of execution time and resource allocation. Many prior works [2, 5, 24] have assumed that the speedup of a task is a non-decreasing function of the amount of allocated resources (hence the execution time is a non-increasing function) and that the total area is a non-decreasing function of the allocated resources. One example is the well-known *Amdahl's law* [1], which specifies the speedup of executing a parallel task with s sequential fraction using p processors as $1/(s + \frac{1-s}{p})$. Another example used by some scheduling literature [4, 27, 16] is the speedup function p^α, where p represents the amount of allocated resources to the task and $\alpha \leq 1$ is a constant. In particular, this function has been observed for processor resources when executing parallel matrix operations [26] as well as for characterizing the cache behaviors in terms of the miss rate of data access (which is directly related to the execution time) [17]. We refer to this function as the *power law*.

All works above apply to the case with a single resource type, and, to the best of our knowledge, no explicit speedup model is known to include multiple resource types. In Section 5, we extend these speedup functions to construct synthetic parallel tasks with multiple-resource demands. We also perform a profiling study of the Stream benchmark [23] on a recent Intel Xeon Phi architecture that includes two resource types (processor core and high-bandwidth memory).

2.2 Parallel Task Scheduling

Scheduling a set of independent parallel tasks to minimize the makespan is known to be strongly NP-complete [13], thus much attention has been directed at designing approximation or heuristic algorithms. Most prior works focused on allocating a single resource type (i.e., processors) to the tasks, and the constructed schedules are either pack-based (also called shelf-based or level-based) or list-based.

Approximation Algorithms

Scheduling rigid tasks with contiguous processor allocation can be considered as a rectangle packing or 2D-strip packing problem. For this problem, Coffman et al. [6] showed that the Next-Fit Decreasing Height (NFDH) algorithm is 3-approximation and the First-Fit Decreasing-Height (FFDH) algorithm is 2.7-approximation. Both algorithms pack rectangles onto shelves, which are equivalent to creating pack-based schedules. The first result (i.e., 3-approximation) has also been extended to the case of moldable task scheduling [2, 31]. Turek et al. [31] presented a strategy to extend any algorithm for scheduling rigid tasks into an algorithm for scheduling moldable tasks in polynomial time. The strategy preserves the approximation ratio provided that the makespan for the rigid-task problem satisfies certain conditions. The complexity of such an extension was improved in [22] with possibly a worse schedule than the one obtained in [31] but without compromising the approximation ratio. Thanks to these strategies, a 2-approximation algorithm was devised for moldable tasks based on list scheduling [31, 22], extending the same ratio previously known for rigid tasks [31, 12]. Using dual-approximation techniques, Mounie et al. [24] presented a 1.5-approximation algorithm for moldable tasks while assuming that the total area of a task is a non-decreasing function of the allocated processors. Also for moldable tasks, Jansen and Porkolab [20] presented, for any $\epsilon > 0$, a $(1+\epsilon)$-approximation scheme when the number of processors is a fixed constant.

While all results above are for scheduling under a single resource type, only a few papers have considered scheduling under multiple resource types. Garey and Graham [12] proved that a simple list-scheduling algorithm is $(d + 1)$-approximation for rigid tasks, where d is the number of resource types. He et al. [19] proved the same asymptotic result for scheduling a set of malleable jobs, each represented as a direct acyclic graph (DAG) of unit-size tasks. For moldable tasks, Shachnai and Turek [28] presented a technique to transform a

c-approximation algorithm on a single resource type to a $c \cdot d$-approximation algorithm on d types of resources. Partially inspired by these results, this paper presents techniques and algorithms with improved approximations for pack scheduling and new algorithms for list scheduling under multiple resource types.

Heuristics

Some papers have proposed heuristic solutions for multiple-resource-type scheduling under various different models and objectives. Leinberger et al. [21] considered rigid tasks and proposed two heuristics that attempt to balance the usage of different resource types through backfilling strategies. He et al. [18] studied online scheduling of DAG-based jobs and proposed a multi-queue balancing heuristic to balance the sizes of the queues that contain ready tasks under different resource types. Ghodsi et al. [14] proposed Dominant Resource Fairness (DRF), a multiple-resource-type scheduling algorithm for user tasks with fixed resource demands. It aims at ensuring the resource allocation fairness among all users by identifying the dominant resource share for each user and maximizing the minimum dominant share across all users. Grandl et al. [15] considered scheduling malleable tasks under four resource types (CPU, memory, disk and network), and designed a heuristic, called Tetris, that packs tasks to a cluster of machines. This is similar to pack-based scheduling but instead of creating packs in time it creates them in space. Tetris works by selecting a task with the highest correlation between the task's peak resource demands and the machine's resource availabilities, with the aim of minimizing resource fragmentation. NoroozOliaee et al. [25] considered a similar cluster scheduling problem but with two resources only (CPU and memory). They showed that the simple Best Fit plus Shortest Task First scheduling outperforms other packing heuristics (e.g., First Fit, FCFS) in terms of resource utilization and task queueing delays.

3 Model

This sections presents a formal model of the scheduling problem, which we call d-Resource-Scheduling. Suppose a platform has d different types of resources subject to allocation (e.g., processor, memory, cache). For each resource type i, there is a total amount $P^{(i)}$ of resources available. Consider a set of n independent tasks (or jobs), all of which are released at the same time on the platform, so the problem corresponds to scheduling a batch of applications in HPC environment. For each task j, its execution time $t_j(\vec{p}_j)$ is a function of the *resource allocation vector* $\vec{p}_j = (p_j^{(1)}, p_j^{(2)}, \cdots, p_j^{(d)})$, where $p_j^{(i)}$ denotes the amount of the i-th resource allocated to task j. In reality, the execution times are typically obtained by application profiling or interpolation and curve-fitting from historic data. Suppose $\vec{p}_j$ and $\vec{q}_j$ are two resource allocation vectors, and we define $\vec{p}_j \preceq \vec{q}_j$ if $p_j^{(i)} \leq q_j^{(i)}$ for all $1 \leq i \leq d$. Here, the execution time is assumed to be a non-increasing function of the resource allocation, i.e., $\vec{p}_j \preceq \vec{q}_j$

implies $t_j(\vec{p}_j) \geq t_j(\vec{q}_j)$. This means that increasing the allocation of any one type of resource without decreasing the others will not increase the execution time[1]. We also assume that the resource allocations $p_j^{(i)}$'s and the total amount of resources $P^{(i)}$'s are all integer values. This holds naturally true for discrete resources such as processors[2]. For other resources (e.g., memory, cache), it can be justified as most practical resource management systems allocate the resources in discrete chunks (e.g., memory blocks, cache lines). Note that some tasks may not require all resource types in order to execute, hence its resource allocation $p_j^{(i)}$ for a particular resource can be zero, in which case the execution time remains validly defined. In contrast, other tasks may require a minimum amount of certain resource in order to execute. In this case, the execution time can be define as infinity for any amount of resource below this threshold. The model is flexible to handle both scenarios.

In this paper, we focus on *moldable* task scheduling [10], where the amount of resources allocated to a task can be freely selected by the scheduler at launch time but they cannot be changed after the task has started the execution. Moldable task scheduling strikes a good balance between practicality and performance by incurring less overhead than malleable task scheduling yet achieving more flexible task executions than rigid task scheduling. For the d-Resource-Scheduling problem, the scheduler needs to decide, for each task j, a resource allocation vector $\vec{p}_j$ along with a starting time s_j. At any time t, a task is said to be *active* if it has started but not yet completed, i.e., $t \in [s_j, s_j + t_j(\vec{p}_j))$. Let J_t denote the set of active tasks at time t. For a solution to be valid, the resources used by the active tasks at any time should not exceed the total amount of available resources for each resource type, i.e., $\sum_{j \in J_t} p_j^{(i)} \leq P^{(i)}$ for all t and i. The objective is to minimize the maximum completion time, or the *makespan*, of all tasks, i.e., $T = \max_j(s_j + t_j(\vec{p}_j))$.

Since d-Resource-Scheduling is a generalization of classical makespan minimization problem with a single resource type, it is strongly NP-complete. Hence, we are interested in designing approximation and heuristic algorithms. In this paper, we will focus on and compare two major scheduling paradigms, namely, *list scheduling* and *pack scheduling*. A scheduling algorithm S is said to be *c-approximation* if its makespan satisfies $T_{\mathrm{S}} \leq c \cdot T_{\mathrm{OPT}}$ for any instance, where T_{OPT} denotes the makespan by an optimal moldable scheduler. Note that the optimal scheduler needs not be restricted to either list scheduling or pack scheduling.

[1] This assumption is not restrictive, as we can discard an allocation $\vec{q}_j$ that satisfies $\vec{p}_j \preceq \vec{q}_j$ and $t_j(\vec{p}_j) < t_j(\vec{q}_j)$. This is because any valid schedule that allocates $\vec{q}_j$ to task j can be replaced by a valid schedule that allocates $\vec{p}_j$ to the task without increasing the execution time, thus rendering $\vec{q}_j$ useless.

[2] We do not allow fractional processor allocation (typically realized by timesharing a processor among several tasks) as assumed by some prior work.

4 Theoretical Analysis

In this section, we present polynomial-time algorithms for the d-RESOURCE-SCHEDULING problem under both list- and pack-scheduling paradigms, and we prove tight approximation ratios for these algorithms.

4.1 Preliminaries

We start with some preliminary definitions that will be used throughout the analysis. Table 1 provides a summary of the main notations used.

Definition 1. *Define* $\mathbf{p} = (\vec{p}_1, \vec{p}_2, \cdots, \vec{p}_n)^T$ *to be a* resource allocation matrix *for all tasks, where* $\vec{p}_j = (p_j^{(1)}, p_j^{(2)}, \cdots, p_j^{(d)})$ *is a resource allocation vector for task* j.

Definition 2. *Given a resource allocation matrix* $\mathbf{p}$ *for the* d-RESOURCE-SCHEDULING *problem, we define the following:*

- *For task* j, $a_j(\vec{p}_j) = \sum_{i=1}^{d} \frac{p_j^{(i)}}{P^{(i)}} \cdot t_j(\vec{p}_j)$ *is the task's* area[3]*;*
- $A(\mathbf{p}) = \sum_{j=1}^{n} a_j(\vec{p}_j)$ *is the* total area *of all tasks;*
- $t_{\max}(\mathbf{p}) = \max_j t_j(\vec{p}_j)$ *is the* maximum execution time *of all tasks;*
- $L(\mathbf{p}, s) = \max\left(\frac{A(\mathbf{p})}{s}, t_{\max}(\mathbf{p})\right)$ *for any* $s > 0$.

The last quantity $L(\mathbf{p}, s)$ is related to the lower bound of the makespan. In particular, we define

$$L_{\min}(s) = \min_{\mathbf{p}} L(\mathbf{p}, s) \ , \tag{1}$$

and we will show later that it is a lower bound on the makespan when s is set to be d.

In the following, we will present two general techniques for constructing the scheduling solutions. The first technique is a *two-phase* approach, similarly to the one considered in scheduling under a single resource type [22]:

- *Phase 1*: Determines a resource allocation matrix for all the tasks;
- *Phase 2*: Constructs a rigid schedule based on the fixed resource allocation of the first phase.

The second technique is a *transformation* strategy that reduces the d-RESOURCE-SCHEDULING problem to the 1-RESOURCE-SCHEDULING problem, which is then solved and whose solution is transformed back to the original problem.

Section 4.2 presents a resource allocation strategy (*Phase 1*). Section 4.3 presents a transformation strategy, followed by rigid task scheduling schemes (*Phase 2*) in Section 4.4 under both pack and list scheduling. Finally, Section 4.5 puts these different components together and presents several complete scheduling solutions.

[3]Rigorously, we define $a_j(\vec{p}_j) = \infty$ if $\exists i$, s.t. $p_j^{(i)} = 0$ and $t_j(\vec{p}_j) = \infty$.

Table 1: List of Notations.

For platform	
d	Number of resource types
$P^{(i)}$	Total amount of i-th type of resource ($i = 1, \cdots, d$)
For any task j	
$p_j^{(i)}$	Amount of i-th resource allocated to task j
$\vec{p}_j$	Resource allocation vector of task j
$t_j(\vec{p}_j)$	Execution time of task j with vector $\vec{p}_j$
$a_j(\vec{p}_j)$	Area of task j with vector $\vec{p}_j$
s_j	Starting time of task j
For set of all tasks	
n	Number of all tasks
$\mathbf{p}$	Resource allocation matrix for all tasks
$A(\mathbf{p})$	Total area of all tasks with matrix $\mathbf{p}$
$t_{\max}(\mathbf{p})$	Maximum execution time of all tasks with matrix $\mathbf{p}$
$T(\mathbf{p})$	Makespan of all tasks with matrix $\mathbf{p}$
$L(\mathbf{p}, s)$	Maximum of $A(\mathbf{p})/s$ and $t_{\max}(\mathbf{p})$ for any $s > 0$
$L_{\min}(s)$	Minimum $L(\mathbf{p}, s)$ over all matrix $\mathbf{p}$

4.2 A Resource Allocation Strategy

This section describes a resource allocation strategy for the first phase of the scheduling algorithm. It determines the amounts of resources allocated to each task, which are then used to schedule the tasks in the second phase as a rigid task scheduling problem.

The goal is to find efficiently a resource allocation matrix that minimizes $L(\mathbf{p}, s)$, i.e.,

$$\mathbf{p}_{\min}^{s} = \arg\min_{\mathbf{p}} L(\mathbf{p}, s) \ . \tag{2}$$

Let $P = \prod_{i=1}^{d}(P^{(i)} + 1)$ denote the number of all possible allocations for each task, including the ones with zero amount of resource under a particular resource type. A straightforward implementation takes $O(P^n)$ time, which grows exponentially with the number of tasks n. Algorithm 1 presents a resource allocation strategy $\mathrm{RA}_d(s)$ that achieves this goal in $O(nP(\log P + \log n + d))$ time. Note that when the amounts of resources under different types are in the same order, i.e., $P^{(i)} = O(P_{\max})$ for all $i = 1, \cdots, d$, where $P_{\max} = \max_i P^{(i)}$, the running time grows exponentially with the number of resource types d, i.e., the complexity contains $O(P_{\max}^d)$. Since the number of resource types is usually a small constant (less than 4 or 5), the algorithm runs in polynomial time under most realistic scenarios.

The algorithm works as follows. First, it linearizes and sorts all P resource allocation vectors for each task and eliminates any allocation that results in both a higher execution time and a larger total area (Lines 2-17), for such a vector can be replaced by another one that leads to the same or smaller

Algorithm 1: Resource Allocation Strategy $\mathrm{RA}_d(s)$

Input: Set of n tasks, execution time $t_j(\vec{p})\ \forall j, \vec{p}$ and resource limit $P^{(i)}\ \forall i$
Output: Resource allocation matrix $\mathbf{p}^s_{\min} = \arg\min_{\mathbf{p}} L(\mathbf{p}, s)$

begin
  $P \leftarrow \prod_{i=1}^{d}(P^{(i)}+1)$;
  $A \leftarrow 0$;
  for $j = 1$ *to* n **do**
    Linearize all P resource allocation vectors for task j in an array *res_alloc* and sort it in non-decreasing order of task execution time;
    $admissible\_allocations_j \leftarrow list()$;
    $min\_area \leftarrow \infty$;
    for $h = 1$ *to* P **do**
      $\vec{p} \leftarrow res\_alloc(h)$;
      if $a_j(\vec{p}) < min\_area$ **then**
        $min\_area \leftarrow a_j(\vec{p})$;
        $admissible\_allocations_j.append(\vec{p})$;
    $\vec{p}_j \leftarrow admissible\_allocations_j.last\_element()$;
    $A \leftarrow A + a_j(\vec{p}_j)$;
  Build a priority queue Q of n tasks with their longest execution time $t_j(\vec{p}_j)$'s as priorities;
  $L_{\min} = \infty$;
  while $Q.size() = n$ **do**
    $k \leftarrow Q.highest\_priority\_element()$;
    $\vec{p}_k \leftarrow admissible\_allocations_k.last\_element()$;
    $t_{\max} = t_k(\vec{p}_k)$;
    $L = \max\left(\frac{A}{s}, t_{\max}\right)$;
    if $L < L_{\min}$ **then**
      $L_{\min} \leftarrow L$;
      $\mathbf{p}^s_{\min} \leftarrow (\vec{p}_1, \vec{p}_2, \cdots, \vec{p}_n)^T$;
    if $\frac{A}{s} \geq t_{\max}$ **then**
      break;
    $admissible\_allocations_k.pop\_last()$;
    $Q.remove\_highest\_priority\_element()$;
    if $admissible\_allocations_k.nonempty()$ **then**
      $\vec{p}_k{}' \leftarrow admissible\_allocations_k.last\_element()$;
      $A \leftarrow A + a_k(\vec{p}_k{}') - a_k(\vec{p}_k)$;
      $Q.insert\_element(k)$ with priority $t_k(\vec{p}_k{}')$;

$L_{\min}(s)$ (Equation (1)). Each task then ends up with an array of *admissible allocations* in increasing order of execution time and decreasing order of total area. The complexity for this part is $O\big(n(P \log P + Pd)\big)$, which is dominated by sorting each task's allocations and computing its area. Then, the algorithm goes through all possible $t_{\max}$, i.e., the maximum execution time, from the remaining admissible allocations, and for each $t_{\max}$ considered, it computes the minimum total area A and hence the associated L value (Lines 18-40). This is achieved by maintaining the tasks in a priority queue with their longest execution times as priorities and updating the queue with a task's new priority when its next longest execution time is considered. The algorithm terminates either when at least one task has exhausted its admissible allocations, in which case the priority queue will have fewer than n tasks (Line 20), or when L becomes dominated by $\frac{A}{s}$ (Line 30), since the total area only increases and the maximum execution time decreases during this process. As the allocation is only changed for one task when a new $t_{\max}$ is considered, the total area can be updated by keeping track of the area change due to this task alone (Line 36). The algorithm considers at most nP possible $t_{\max}$ values and the complexity at each step is dominated by updating the priority queue, which takes $O(\log n)$ time, and by updating the total area, which takes $O(d)$ time, so the overall complexity for this part is $O\big(nP(\log n + d)\big)$.

Now, we show that the resource allocation matrix $\mathbf{p}^s_{\min}$ returned by Algorithm 1 while setting the parameter s to be d can be used by a rigid task scheduling strategy in the second phase to achieve good approximations.

Theorem 1. *If a rigid task scheduling algorithm* R_d *that uses the resource allocation matrix* $\mathbf{p}^d_{\min}$ *obtained by the strategy* $\mathrm{RA}_d(d)$ *produces a makespan*

$$T_{\mathrm{R}_d}(\mathbf{p}^d_{\min}) \leq c \cdot L_{\min}(d) \ , \tag{3}$$

then the two-phase algorithm $\mathrm{RA}_d(d)$+R_d *is c-approximation for the d-*Resource-Scheduling *problem.*

To prove the above theorem, let us define $\mathbf{p}_{\mathrm{OPT}}$ to be the resource allocation matrix of an optimal schedule, and let T_{OPT} denote the corresponding optimal makespan. We will show in the following lemma that $L_{\min}(d)$ serves as a lower bound on T_{OPT}. Then, Theorem 1 follows directly, since $T_{\mathrm{RA}_d(d)+\mathrm{R}_d} = T_{\mathrm{R}_d}(\mathbf{p}^d_{\min}) \leq c \cdot L_{\min}(d) \leq c \cdot T_{\mathrm{OPT}}$.

Lemma 1. $T_{\mathrm{OPT}} \geq L_{\min}(d)$.

Proof. We will prove that, given a resource allocation matrix $\mathbf{p}$, the makespan produced by any rigid task scheduler using $\mathbf{p}$ must satisfy $T(\mathbf{p}) \geq t_{\max}(\mathbf{p})$ and $T(\mathbf{p}) \geq \frac{A(\mathbf{p})}{d}$. Thus, for the optimal schedule, which uses $\mathbf{p}_{\mathrm{OPT}}$, we have $T_{\mathrm{OPT}} \geq \max\big(t_{\max}(\mathbf{p}_{\mathrm{OPT}}), \frac{A(\mathbf{p}_{\mathrm{OPT}})}{d}\big) = L(\mathbf{p}_{\mathrm{OPT}}, d) \geq L_{\min}(d)$. The last inequality is because $L_{\min}(d)$ is the minimum $L(\mathbf{p}, d)$ among all possible resource allocations including $\mathbf{p}_{\mathrm{OPT}}$ (Equation (1)).

The first bound $T(\mathbf{p}) \geq t_{\max}(\mathbf{p})$ is trivial since the makespan of any schedule should be at least the execution time of the longest task. For the second bound,

we have, in any valid schedule with makespan $T(\mathbf{p})$, that:

$$\begin{aligned} A(\mathbf{p}) &= \sum_{j=1}^{n} \sum_{i=1}^{d} \frac{p_j^{(i)}}{P^{(i)}} \cdot t_j(\vec{p}_j) \\ &= \sum_{i=1}^{d} \frac{1}{P^{(i)}} \sum_{j=1}^{n} p_j^{(i)} \cdot t_j(\vec{p}_j) \\ &\leq \sum_{i=1}^{d} \frac{1}{P^{(i)}} \cdot P^{(i)} \cdot T(\mathbf{p}) \\ &= d \cdot T(\mathbf{p}) \ . \end{aligned}$$

The inequality is because $P^{(i)} \cdot T(\mathbf{p})$ is the maximum volume for resource type i that can be allocated to all the tasks within a total time of $T(\mathbf{p})$. □

4.3 A Transformation Strategy

This section describes a strategy to indirectly solve the d-Resource-Scheduling problem via a transformation to the 1-Resource-Scheduling problem.

Algorithm 2 presents the transformation strategy TF, which contains three steps. First, any instance I of the d-Resource-Scheduling problem is transformed to an instance I' of the 1-Resource-Scheduling problem. Then, I' is solved by any moldable task scheduler under a single resource type. Lastly, the solution obtained for I' is transformed back to obtain a solution for the original instance I. The complexity of the transformation alone (without solving the instance I') is $O(nQd)$, dominated by the first step of the strategy, where Q is defined as the Least Common Multiple (LCM) of all the $P^{(i)}$'s. Note that if Q is in the same order as the maximum amount of resource $P_{\max} = \max_i P^{(i)}$ among all resource types (e.g., when $P^{(i)}$'s are in powers of two), the complexity becomes linear in $P_{\max}$. In contrast, the complexity of solving the problem directly (by relying on Algorithm 1) contains $P_{\max}^d$. Thus, the transformation strategy can greatly reduce an algorithm's running time, especially when more resource types need to be scheduled (see our simulation results in Section 5.4).

In the subsequent analysis, we will use the following notations:

- For the transformed instance I': Let $\mathbf{q} = (q_{1'}, q_{2'}, \cdots, q_{n'})^T$ denote the resource allocation for the transformed tasks. $L'_{\min}(s)$ is the minimum $L'(\mathbf{q}, s)$ as defined in Equation (1), and T'_{M_1} denotes the makespan produced by a moldable task scheduler M_1 for the 1-Resource-Scheduling problem.

- For the original instance I: Recall that $L_{\min}(s)$ denotes the minimum $L(\mathbf{p}, s)$ defined in Equation (1), and let $T_{\mathrm{TF}+\mathrm{M}_1}$ denote the makespan produced by the algorithm $\mathrm{TF} + \mathrm{M}_1$, which combines the transformation strategy TF and the scheduler M_1 for a single resource type.

Algorithm 2: Transformation Strategy TF

Input: Set of n tasks, execution time $t_j(\vec{p})\ \forall j, \vec{p}$, resource limit $P^{(i)}\ \forall i$
Output: Starting time s_j and resource allocation vector $\vec{p}_j\ \forall j$
(1) Transform d-Resource-Scheduling instance I to 1-Resource-Scheduling instance I';
- I' has the same number n of tasks as I;
- The resource limit for the only resource of I' is $Q = \operatorname{lcm}_{i=1\cdots d} P^{(i)}$;
- For any task j' in I', its execution time with any amount of resource $q \in \{0, 1, 2, \cdots, Q\}$ is defined as $t_{j'}(q) = t_j((\lfloor \frac{q \cdot P^{(i)}}{Q} \rfloor)_{i=1\cdots d})$;

(2) Solve the 1-Resource-Scheduling instance I';
(3) Transform 1-Resource-Scheduling solution S' back to d-Resource-Scheduling solution S;
- For any task j in I, its starting time is $s_j = s_{j'}$, where $s_{j'}$ is the starting time of task j' in I', and its resource allocation vector is $\vec{p}_j = (\lfloor \frac{q_{j'} \cdot P^{(i)}}{Q} \rfloor)_{i=1\cdots d}$, where $q_{j'}$ is the allocation of the only resource for task j' in I'.

We show that the transformation achieves good approximation for the d-Resource-Scheduling problem if the solution for the 1-Resource-Scheduling problem satisfies certain property.

Theorem 2. *For any transformed instance I' of the* 1-Resource-Scheduling *problem, if a moldable task scheduler* M_1 *under a single resource type produces a makespan*

$$T'_{\mathrm{M}_1} \leq c \cdot L'_{\min}(d)\ , \tag{4}$$

then the algorithm $\mathrm{TF}{+}\mathrm{M}_1$ *is c-approximation for the d-*Resource-Scheduling *problem.*

Before proving the approximation, we first show that the schedule obtained for I via the transformation strategy is valid and that the makespan for I' is preserved.

Lemma 2. *Any valid solution by scheduler* M_1 *for I' transforms back to a valid solution by algorithm* $\mathrm{TF} + \mathrm{M}_1$ *for I, and* $T_{\mathrm{TF}+\mathrm{M}_1} = T'_{\mathrm{M}_1}$.

Proof. According to Steps (1) and (3) of the transformation, the execution time for each task j in I is the same as that of j' in I', i.e., $t_j(\vec{p}_j) = t_{j'}(q_{j'})$. Hence, the makespan is equal for both I and I', since the corresponding tasks also have the same starting time, i.e., $s_j = s_{j'}$.

If the schedule for I' is valid, i.e., $\sum_{j' \in J'_t} q_{j'} \leq Q\ \forall t$, then we have $\sum_{j \in J_t} p_j^{(i)} = \sum_{j \in J_t} \lfloor \frac{q_{j'} \cdot P^{(i)}}{Q} \rfloor \leq \frac{P^{(i)}}{Q} \sum_{j' \in J'_t} q_{j'} \leq P^{(i)}\ \forall i, t$, rendering the schedule for I valid as well. □

The following lemma relates the makespan lower bound of the transformed instance to that of the original instance.

Lemma 3. $L'_{\min}(d) \leq L_{\min}(d)$.

Proof. We will show that given any resource allocation $\mathbf{p}$ for I, there exists an allocation $\mathbf{q}(\mathbf{p})$ for I' such that $L'(\mathbf{q}(\mathbf{p}), d) \leq L(\mathbf{p}, d)$. Thus, for $\mathbf{p}^d_{\min}$ that leads to $L_{\min}(d)$, there is a corresponding $\mathbf{q}(\mathbf{p}^d_{\min})$ that satisfies $L'(\mathbf{q}(\mathbf{p}^d_{\min}), d) \leq L(\mathbf{p}^d_{\min}, d) = L_{\min}(d)$, and therefore $L'_{\min}(d) = \min_{\mathbf{q}} L'(\mathbf{q}, d) \leq L'(\mathbf{q}(\mathbf{p}^d_{\min}), d) \leq L_{\min}(d)$.

Consider any resource allocation $\mathbf{p} = (\vec{p}_1, \vec{p}_2, \cdots, \vec{p}_n)^T$ for I. For each task j, let k_j denote its *dominating* resource type in terms of the proportion of resource used, i.e., $k_j = \arg\min_i \frac{p_j^{(i)}}{P^{(i)}}$. We construct $\mathbf{q}(\mathbf{p}) = (q_{1'}, q_{2'}, \cdots, q_{n'})^T$ for I' by setting, for each task j', an allocation $q_{j'} = \frac{p_j^{(k_j)} \cdot Q}{P^{(k_j)}}$. As $P^{(k_j)}$ divides Q, $q_{j'}$ is an integer and hence a valid allocation.

According to Step (1) of the transformation, the execution time of task j' in I' satisfies

$$t_{j'}(q_{j'}) = t_j((\lfloor \frac{p_j^{(k_j)} \cdot P^{(i)}}{P^{(k_j)}} \rfloor)_{i=1\cdots d}) \leq t_j(\vec{p}_j) \ .$$

The inequality is because $p_j^{(i)} \leq \lfloor \frac{p_j^{(k_j)} \cdot P^{(i)}}{P^{(k_j)}} \rfloor \ \forall i$, which we get by the choice of k_j and the integrality of $p_j^{(i)}$, and because we assumed that the execution time is a non-increasing function of each resource allocation (in Section 3). As a result, the area of each task j' in I' also satisfies

$$a_{j'}(q_{j'}) = \frac{q_{j'}}{Q} t_{j'}(q_{j'}) \leq \frac{p_j^{(k_j)}}{P^{(k_j)}} t_j(\vec{p}_j) \leq \sum_{i=1}^{d} \frac{p_j^{(i)}}{P^{(i)}} t_j(\vec{p}_j) = a_j(\vec{p}_j).$$

Hence, the maximum execution times and the total areas for the two instances I' and I satisfy $t'_{\max}(\mathbf{q}(\mathbf{p})) = \max_{j'} t_{j'}(q_{j'}) \leq \max_j t_j(\vec{p}_j) = t_{\max}(\mathbf{p})$ and $A'(\mathbf{q}(\mathbf{p})) = \sum_{j'} a_{j'}(q_{j'}) \leq \sum_j a_j(\vec{p}_j) = A(\mathbf{p})$. This leads to $L'(\mathbf{q}(\mathbf{p}), d) = \max\left(\frac{A'(\mathbf{q}(\mathbf{p}))}{d}, t'_{\max}(\mathbf{q}(\mathbf{p}))\right) \leq \max\left(\frac{A(\mathbf{p})}{d}, t_{\max}(\mathbf{p})\right) = L(\mathbf{p}, d)$. □

(Proof of Theorem 2). Based on the above results, we can derive:

$$\begin{aligned} T_{\text{TF+M}_1} &= T'_{\text{M}_1} && \text{(Lemma 2)} \\ &\leq c \cdot L'_{\min}(d) && \text{(Equation (4))} \\ &\leq c \cdot L_{\min}(d) && \text{(Lemma 3)} \\ &\leq c \cdot T_{\text{OPT}} \ . && \text{(Lemma 1)} \end{aligned}$$

4.4 Rigid Task Scheduling

This section presents strategies to schedule rigid tasks under fixed resource allocations. The strategies include both list-based and pack-based scheduling. The results extend the theoretical analyses of single-resource-type scheduling [31, 2] to account for the presence of multiple resource types.

4.4.1 List Scheduling

We first present a list-based scheduling strategy for a set of rigid tasks under multiple resource types. A list-based algorithm arranges the set of tasks in a list. At any moment starting from time 0 and whenever an existing task completes execution and hence releases resources, the algorithm scans the list of remaining tasks in sequence and schedules the first one that fits, i.e., there is sufficient amount of resource to satisfy the task under each resource type. Algorithm 3 presents the list scheduling strategy LS_d with d resource types, and it extends the algorithm presented in [31] for scheduling under a single resource type. The complexity of the algorithm is $O(n^2 d)$, since scheduling each task incurs a cost of $O(nd)$ by scanning the *taskList* and updating the resource availability for the times in *sortedTimeList*.

Algorithm 3: List Scheduling Strategy LS_d

Input: Resource allocation matrix $\mathbf{p}$ for the tasks and resource limit $P^{(i)}\ \forall i$
Output: Starting time $s_j\ \forall j$
begin
  Arrange the tasks in a list *taskList*;
  sortedTimeList $\leftarrow \{0\}$;
  while *taskList.nonempty*() **do**
    t = *sortedTimeList.pop_first*();
    for $j = 1$ *to taskList.size*() **do**
      if *task* j *fits at time* t **then**
        Schedule task j at time t, i.e., $s_j = t$;
        sortedTimeList.insert$(s_j + t_j(\vec{p}_j))$;
        Update available resources for all times before $s_j + t_j(\vec{p}_j)$;

The following lemma shows the performance of this list scheduling scheme.

Lemma 4. *For a set of rigid tasks with a resource allocation matrix* $\mathbf{p}$*, the list scheduling algorithm* LS_d *achieves, for any parameter* $s \geq 1$*, a makespan*

$$T_{\mathrm{LS}_d}(\mathbf{p}) \leq 2s \cdot L(\mathbf{p}, s)\ . \tag{5}$$

Proof. We will prove in the following that $T_{\mathrm{LS}_d}(\mathbf{p}) \leq 2 \cdot \max\left(t_{\max}(\mathbf{p}), A(\mathbf{p})\right)$. For any $s \geq 1$, it will then lead to $T_{\mathrm{LS}_d}(\mathbf{p}) \leq 2s \cdot \max\left(t_{\max}(\mathbf{p}), \frac{A(\mathbf{p})}{s}\right) = 2s \cdot L(\mathbf{p}, s)$.

First, suppose the makespan satisfies $T_{\mathrm{LS}_d}(\mathbf{p}) \leq 2t_{\max}(\mathbf{p})$, then the claim holds trivially. Otherwise, we will show $T_{\mathrm{LS}_d}(\mathbf{p}) \leq 2A(\mathbf{p})$, thus proving the claim. To this end, consider any time $t \in [0, \frac{T_{\mathrm{LS}_d}(\mathbf{p})}{2}]$, and define $t' = t + \frac{T_{\mathrm{LS}_d}(\mathbf{p})}{2}$. Since $t_{\max}(\mathbf{p}) < \frac{T_{\mathrm{LS}_d}(\mathbf{p})}{2}$ by our assumption, any task that is active at time t' has not been scheduled at time t. Define $U_t^{(i)} = \sum_{j \in J_t} \frac{p_j^{(i)}}{P^{(i)}}$ to be the total utilization for resource of type i at time t. Therefore, we should have $\exists i$, s.t. $U_t^{(i)} + U_{t'}^{(i)} \geq 1$, for otherwise any active task at time t' could have been scheduled by the

algorithm at time t or earlier. Thus, we can express the total area as:

$$\begin{aligned} A(\mathbf{p}) &= \int_{t=0}^{T_{\mathrm{LS}_d}(\mathbf{p})} \sum_{i=1}^{d} U_t^{(i)} dt \\ &= \int_{t=0}^{\frac{T_{\mathrm{LS}_d}(\mathbf{p})}{2}} \sum_{i=1}^{d} \left(U_t^{(i)} + U_{t'}^{(i)} \right) dt \\ &\geq \frac{T_{\mathrm{LS}_d}(\mathbf{p})}{2} \ . \end{aligned}$$

□

4.4.2 Pack Scheduling

We now present a pack-based scheduling strategy. Recall that a pack contains several concurrently executed tasks that start at the same time and the tasks in the next pack cannot start until all tasks in the previous pack have completed. Algorithm 4 presents the pack scheduling strategy PS_d with d resource types. It extends the algorithm presented in [2] for scheduling under a single resource type. Specifically, the tasks are first sorted in non-increasing order of execution times (which are fixed due to fixed resource allocations). Then, they are assigned one by one to the last pack if it fits, i.e., there is sufficient amount of resource under each resource type. Otherwise, a new pack is created and the task is assigned to the new pack. The complexity of the algorithm is $O\big(n(\log n + d)\big)$, which is dominated by the sorting of the tasks and by checking the fitness of each task in the pack.

Algorithm 4: Packing Scheduling Strategy PS_d

Input: Resource allocation matrix $\mathbf{p}$ for the tasks and resource limit $P^{(i)}\ \forall i$
Output: Set of packs $\{B_1, B_2, \cdots\}$ containing the tasks and starting times S_m for each pack B_m

begin
- Sorted the tasks in non-increasing order of execution time $t_j(\vec{p}_j)$;
- $m \leftarrow 1$;
- $B_1 \leftarrow \emptyset$;
- $S_1 \leftarrow 0$;
- **for** $j = 1$ *to* n **do**
  - **if** *task* j *fits in pack* B_m **then**
    - $B_m \leftarrow B_m \cup \{j\}$;
  - **else**
    - $m \leftarrow m + 1$;
    - $B_m \leftarrow \{j\}$;
    - $S_m \leftarrow S_{m-1} + \max_{j \in B_{m-1}} t_j(\vec{p}_j)$;

The following lemma shows the performance of this pack scheduling scheme.

Lemma 5. *For a set of rigid tasks with a resource allocation matrix* $\mathbf{p}$*, the pack scheduling algorithm* PS_d *achieves, for any parameter* $s > 0$*, a makespan*

$$T_{\mathrm{PS}_d}(\mathbf{p}) \leq (2s+1) \cdot L(\mathbf{p}, s) \ . \tag{6}$$

Proof. We will prove in the following that $T_{\mathrm{PS}_d}(\mathbf{p}) \leq 2A(\mathbf{p}) + t_{\max}(\mathbf{p})$, which will then lead to $T_{\mathrm{PS}_d}(\mathbf{p}) \leq (2s+1) \cdot \max\left(\frac{A(\mathbf{p})}{s}, t_{\max}(\mathbf{p})\right) = (2s+1) \cdot L(\mathbf{p}, s)$.

Suppose the algorithm creates M packs in total. For each pack B_m, where $1 \leq m \leq M$, let A_m denote the total area of the tasks in the pack, i.e., $A_m = \sum_{j \in B_m} a_j(\vec{p}_j)$, and let T_m denote the total execution time of the pack, which is the same as the execution time of the longest task in the pack, i.e., $T_m = \max_{j \in B_m} t_j(\vec{p}_j)$.

Consider the time when the algorithm tries to assign a task j to pack B_m and fails due to insufficient amount of i-th resource, whose remaining amount is denoted by $p^{(i)}$. As the tasks are handled by decreasing execution time, we have $A_m \geq \left(1 - \frac{p^{(i)}}{P^{(i)}}\right) t_j(\vec{p}_j)$. Also, because task j cannot fit in pack B_m, it means that its allocation $p_j^{(i)}$ on resource i is at least $p^{(i)}$. This task is then assigned to pack B_{m+1}, so we have $A_{m+1} \geq a_j(\vec{p}_j) \geq \frac{p^{(i)}}{P^{(i)}} t_j(\vec{p}_j)$. Since task j is the first task put in the pack and all the following tasks have smaller execution times, we have $T_{m+1} = t_j(\vec{p}_j)$. Hence,

$$A_m + A_{m+1} \geq t_j(\vec{p}_j) = T_{m+1} \ .$$

Summing the above inequality over m, we get $\sum_{m=2}^{M} T_m \leq 2 \sum_{m=1}^{M} A_m = 2A(\mathbf{p})$. Finally, as the longest task among all tasks is assigned to the first pack, i.e., $T_1 = t_{\max}(\mathbf{p})$, we get the makespan $T_{\mathrm{PS}_d}(\mathbf{p}) = \sum_{m=1}^{M} T_m \leq 2A(\mathbf{p}) + t_{\max}(\mathbf{p})$. □

4.5 Putting Them Together

This section combines various strategies presented in the previous sections to construct several moldable task scheduling algorithms under multiple resource types. Two solutions are presented for the d-Resource-Scheduling problem under each scheduling paradigm (list vs. pack) depending on if the problem is solved directly or indirectly via a transformation to the 1-Resource-Scheduling problem. The following shows the combinations of the two solutions:

$$\textit{Direct-based solution:}\ \ \mathrm{RA}_d(d) + \mathrm{R}_d$$
$$\textit{Transform-based solution:}\ \ \mathrm{TF} + \mathrm{RA}_1(d) + \mathrm{R}_1$$

Here, RA denotes the resource allocation strategy (Algorithm 1), TF denotes the transformation strategy (Algorithm 2), and R denotes the rigid task scheduling strategy LS or PS (Algorithm 3 or 4). Four algorithms can be resulted from the above combinations, and we call them D(irect)-Pack, D(irect)-List, T(ransform)-Pack and T(ransform)-List, respectively.

We point out that when the LCM of all the $P^{(i)}$'s is dominated by $P_{\max}$ (e.g., when $P^{(i)}$'s are powers of two), the transform-based solution will reduce the overall complexity of the direct-based solution exponentially, thus significantly improving the algorithms' running times. The approximation ratios of

the transform-based solutions, however, will not be compromised. The following theorem proves the ratios of these algorithms.

Theorem 3. *The list-based algorithms (*D-List *and* T-List*) are* $2d$*-approximations and pack-based algorithms (*D-Pack *and* T-Pack*) are* $(2d+1)$*-approximations. Moreover, the bounds are asymptotically tight for the respective algorithms.*

Proof. We prove the approximation ratios for the two pack-based scheduling algorithms. The proof for the list-based algorithms is very similar and hence omitted.

For D-Pack ($\mathrm{RA}_d(d)$+PS_d): Using $\mathbf{p}^d_{\min}$ found by $\mathrm{RA}_d(d)$, we have $T_{\mathrm{PS}_d}(\mathbf{p}^d_{\min}) \leq (2d+1)\cdot L(\mathbf{p}^d_{\min}, d) = (2d+1)\cdot L_{\min}(d)$ from Lemma 5 (by setting $s = d$). Then, based on Theorem 1, we get $T_{\mathrm{RA}_d(d)+\mathrm{PS}_d} \leq (2d+1)\cdot T_{\mathrm{OPT}}$.

For T-Pack ($\mathrm{TF} + \mathrm{RA}_1(d) + \mathrm{PS}_1$): Applying PS_1 to the transformed instance using $\mathbf{q}^d_{\min}$ found by $\mathrm{RA}_1(d)$, we have $T'_{\mathrm{RA}_1(d)+\mathrm{PS}_1} = T'_{\mathrm{PS}_1}(\mathbf{q}^d_{\min}) \leq (2d+1)\cdot L'(\mathbf{q}^d_{\min}, d) = (2d+1)\cdot L'_{\min}(d)$ from Lemma 5 (again by setting $s = d$). Then, based on Theorem 2, we get $T_{\mathrm{TF}+\mathrm{RA}_1(d)+\mathrm{PS}_1} \leq (2d+1)\cdot T_{\mathrm{OPT}}$.

Now, we show that the approximation ratios are tight. To this end, we define in the following a partial profile for each task, i.e., the execution times for some resource allocation vectors, together with $t(0,\cdots,0) = \infty$. To adhere to the non-increasing execution time assumption (in Section 3), we define the execution time for each remaining vector $\vec{p}$ of the task as $t(\vec{p}) = \min_{\vec{q}\preceq\vec{p}} t(\vec{q})$, thus making these choices irrelevant to the scheduling algorithms.

For both pack-based algorithms, we construct an instance that contains $n = 2dP + d + 1$ tasks and total amount of resource $P^{(i)} = P$, $\forall i = 1,\cdots,d$. We refer to the first task as task 0, and define its execution time as $t_0(1,\cdots,1) = P$. The following describes the remaining $2dP + d$ tasks, which are indexed using $i \in [1,d]$ and $j \in [0,2P]$. Specifically, for each i, we have:

- $t_{i,0}(0,\cdots,0,P,P,0,\cdots,0) = 1$, where the P appears in positions i and $i-1$ (if $i-1 \geq 1$);
- $\forall j_1 \in [1,P],\ t_{i,j_1}(0,\cdots,0,P-1,0,\cdots,0) = 1$, where the $P-1$ appears in positions i;
- $\forall j_2 \in [P+1,2P],\ t_{i,j_2}(0,\cdots,0,2,0,\cdots,0) = 1$, where the 2 appears in positions i.

We can add different values in the range of $[0,\epsilon]$ to the above execution times in such a way that forces the algorithms to assign each task to a pack of its own, which can be achieved by placing them in the desired order. In particular, task 0 will be placed first. Then, for each $i \in [1,d]$, task $(i,0)$ will be placed, followed by tasks (i,j_1) and tasks (i,j_2) in alternating order. In this case, both pack-based algorithms will have a makespan $T = (2d+1)P + d + O(n\epsilon)$. On the other hand, the optimal algorithm is able to schedule task 0 together with tasks $(i,0)$ in $P + O(\epsilon)$ time, followed by tasks (i,j_2) in $2 + O(\epsilon)$ time, and finally tasks (i,j_2) in $2 + O(\epsilon)$ time, thus having a makespan $T_{\mathrm{OPT}} = P + 4 + O(\epsilon)$. By setting $\epsilon = \frac{1}{n}$, we have $\frac{T}{T_{\mathrm{OPT}}} = \frac{(2d+1)P+d+O(1)}{P+O(1)}$, so $\lim\limits_{P\to\infty} \frac{T}{T_{\mathrm{OPT}}} = 2d+1$.

For both list-based algorithms, we construct an instance that contains $n = 2d$ tasks and total amount of resource $P^{(i)} = 2P$, $\forall i = 1, \cdots, d$. For each task j, we define two relevant allocations:

- $t_j(0, \cdots, 0, P, 0, \cdots, 0) = 1$, where the P appears in position $\lceil \frac{j}{2} \rceil$;
- $t_j(P + 1, 0, \cdots, 0) = \frac{P-1}{P+1}$.

Based on the resource allocation strategy, the first allocation is discarded by both list-based algorithms (for having both longer execution time and larger area). Thus, using the second allocation, any list-based algorithm is forced to schedule the tasks sequentially one after another, resulting in a makespan $T = 2d\frac{P-1}{P+1}$. The optimal algorithm chooses instead the first allocation and schedules all tasks at the same time with a makespan $T_{\text{OPT}} = 1$. Therefore, we have $\lim_{P \to \infty} \frac{T}{T_{\text{OPT}}} = 2d$. □

5 Experiments

In this section, we conduct experiments and simulations whose goals are twofold: (i) to validate the theoretical underpinning of moldable task scheduling; and (ii) to compare the practical performance of various list- and pack-scheduling schemes under multiple resource types.

Experiments are performed on an Intel Xeon Phi Knights Landing (KNL) machine with 64 cores and 112GB available memory, out of which 16GB are high-bandwidth fast memory (MCDRAM) and the rest are slow memory (DDR). MCDRAM has ≈5x the bandwidth of DDR [30], thus offering a higher data-transfer rate. The applications we run are from a modified version of the Stream benchmark [23] configured to explore the two available resources of KNL (cores and fast memory) with different resource allocations. Simulations are further conducted to project the performance under more resource types using synthetic workloads that extend some classical speedup profiles.

5.1 Evaluated Algorithms

Each algorithm in Section 4.5 is evaluated with two variants, depending on if it is list-based or pack-based. A list-based algorithm can arrange the tasks according to two widely applied heuristics: *Shortest Processing Time First (SPT)* and *Longest Processing Time First (LPT)*, before scheduling them as shown in Algorithm 3. For pack-based scheduling, Algorithm 4 applies the *Next Fit (NF)* heuristic, which assigns a task to the last pack created. *First Fit (FF)* is another heuristic that assigns a task to the first pack in which the task fits. From the analysis, the approximation ratio can only be improved with this heuristic. Coupling these heuristics with the two solutions (direct vs. transformation) under the two scheduling paradigms (list vs. pack), we have a total of eight scheduling algorithms that we will evaluate in the experiments.

5.2 Application Profiling

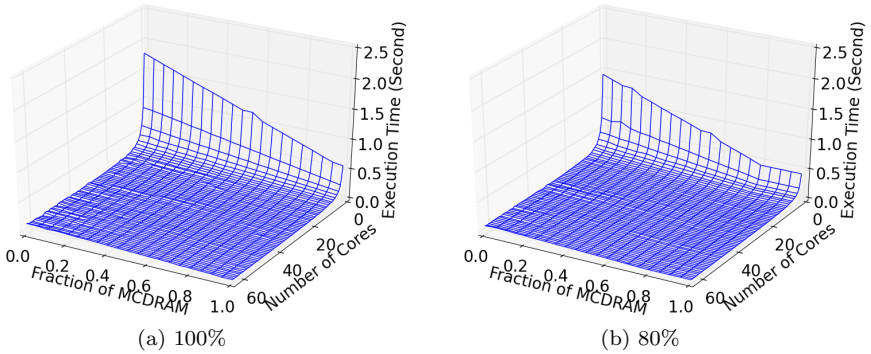
In order to manage the fast memory as a resource, we first configure the KNL machine in *flat* mode [30], where MCDRAM and DDR together form the entire addressable memory. We will also explore MCDRAM in *cache* mode later (see next subsection). The applications in the Stream benchmark are then modified by allocating a specified portion of their memory footprints on MCDRAM (and the remaining on DDR), thereby impacting the overall data-transfer efficiency and hence the execution time. The profiles are obtained by varying both the number of cores (1-64) and the amount of fast memory (16GB divided into 20 chunks) allocated to each application and recording their corresponding execution times. In addition, we vary the total memory footprints of the applications to create tasks of different sizes.

Figure 2 shows the profiles of the *triad* application when its memory footprint occupies 100%, 80% and 60% of the fast memory, respectively. Similar profiles are also observed for the other applications (e.g., *write*, *ddot*) in the benchmark. First, we observe that the execution time is inversely proportional to the number of allocated cores, suggesting that the application has nearly perfect speedup. Also, the execution time reduces linearly as more fast memory is allocated to the point where the application's memory footprint fits entirely in fast memory; after that the execution time remains fairly constant since allocating additional fast memory no longer benefits the execution. The profiled data for different applications under various sizes are subsequently used to compute the schedules by different algorithms, which are then implemented on the KNL node for performance comparison. We can see that the execution time the roofline curve once the allocated fast memory becomes smaller than the array size. Before this moment the execution time remains fairly constant. Note that in order for the total data size to be equal to a defined percentage of the fast memory each benchmark will work on array sizes of different sizes (since write requires only one array, while ddor works on two and triad on three).

In order to gather profiles for different application behaviours, we modified the stream benchmark to use the fast memory available in the KNL processor [7, 8]. We made experiments for the write, ddot and triad benchmarks variying the number of threads and the amount of fast memory used and recorded the execution time for each.

5.3 Experimental Results

In order to validate our theoretical analysis and to compare the performance of different algorithms, we implement a basic scheduler that executes the tasks according to the schedules given by these algorithms based on the application profiles. Each task is launched at the specified time using the specified amount of resources (cores and fast memory) on the KNL machine. If any required resource of a task is not available at start time because of the delays in the preceding tasks (possibly due to performance variations), the execution of the task will be correspondingly delayed until there are enough resources available. We



(a) 100%

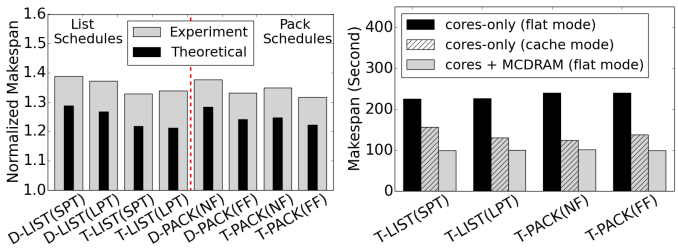
(b) 80%

(c) 60%

Figure 2: Profiles of the *triad* application in Stream benchmark when its memory footprint occupies 100%, 80% and 60% of the entire fast memory.

expect the delay (if any) to be small for each task but the effect can accumulate throughout the course of the schedule. This additional overhead is assessed in our experiments.

Figure 3(a) shows the makespans of the eight scheduling algorithms normalized by the theoretical lower bound (Equation (1) with $s = d$; see also Lemma 1) while running 150 tasks. We have also varied the number of tasks but observed no significant difference in their relative performance. First, we can see that the scheduling overhead, which manifests itself as the difference between the theoretical prediction and the experimental result, is less than 10% for all algorithms. In general, list scheduling (left four) and pack scheduling (right four) have comparable performance despite that the former admits a better theoretical bound. Moreover, transform-based solutions fare slightly better than direct-based solutions due to the balanced resource requirements of different applications in the benchmark. Lastly, in line with the intuition and conventional wisdom, *LPT* and *FF* are effective for reducing the makespan for list and pack scheduling,



(a) (b)

Figure 3: (a) Normalized makespans of eight scheduling algorithms; (b) Makespans of four transform-based scheduling algorithms in three different resource management configurations.

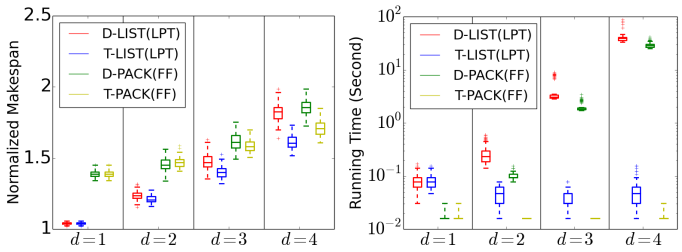
respectively, except for the T-List algorithm, which incurs a larger overhead with the *LPT* variant in the actual execution.

Figure 3(b) further compares the makespans of four transform-based algorithms when MCDRAM is not used in flat mode (in which case all data are allocated on the slow DDR memory), and when KNL is configured in cache mode (in which case MCDRAM serves as a last-level cache to DDR). In both cases, only one type of resource (i.e., cores) is scheduled, and as a result, the transformation produces the same solutions as the direct-based schedules. Not surprisingly, cores-only scheduling in flat mode has the worst makespan for all algorithms, since MCDRAM is completely unutilized in this case. Having MCDRAM as cache to DDR significantly improves the makespan, but due to interference from concurrently running applications, the makespan is still worse than scheduling both cores and MCDRAM in flat mode. The results confirm the benefit of explicitly managing multiple resource types for improving the application performance.

5.4 Simulation Results with More Resource Types

In this section, we conduct simulations to project the performance of list-based and pack-based scheduling algorithms under more resource types. We simulate up to four types of resources that could represent different type of components such as CPUs, GPUs, fast memory, I/O bandwidth. All elements that would be shared between different applications and needed for one execution. The total amount of discrete resource for each type is set to be 64, 32, 16 and 8, respectively. Due to the lack of realistic application profiles under multiple-resource constraints, we generate synthetic workloads by extending two classical speedup functions (described in Section 2.1) to model moldable applications:

- *Extended Amdahl's law* [1]: (i) $1/\left(s_0 + \sum_{i=1}^{d} \frac{s_i}{p^{(i)}}\right)$; (ii) $1/\left(s_0 + \frac{1-s_0}{\prod_{i=1}^{d} p^{(i)}}\right)$; (iii) $1/\left(s_0 + \max_{i=1..d} \frac{s_i}{p^{(i)}}\right)$.



(a) (b)

Figure 4: (a) Normalized makespans and (b) running times of four scheduling algorithms with up to four different resource types.

- *Extended power law* [26, 17]: (i) $1/\left(\sum_{i=1}^{d} \frac{s_i}{(p^{(i)})^{\alpha_i}}\right)$; (ii) $\prod_{i=1}^{d}(p^{(i)})^{\alpha_i}$; (iii) $1/\left(\max_{i=1..d} \frac{s_i}{(p^{(i)})^{\alpha_i}}\right)$.

The extension of both speedup laws comes in three different flavors (i.e., *sum, product, max*), characterizing how allocations of different resource types work together to contribute to the overall application speedup (i.e., *sequential, collaborative, concurrent*). In the simulations, we generate 150 tasks, each randomly selected from the six extended profiles. The sequential fraction s_0 is uniformly generated in $(0, 0.2]$, and the parallel fraction s_i for each resource type i is uniformly generated in $(0, 1]$ and normalized to make sure that $\sum_{i=1}^{d} s_i = 1 - s_0$. The parameter α_i is uniformly generated in $[0.3, 1)$. The total work of a task is chosen uniformly from a range normalized between 0 and 1 (the relative performance of the algorithms is not affected by the normalization).

Figure 4 plots the performance distributions of the scheduling algorithms with up to four resource types. Each boxplot is obtained by running 100 instances. Only *LPT* (for list) and *FF* (for pack) are included, since they have better overall performance. First, we observe that the normalized makespans indeed increase with the number of resource types, corroborating the theoretical analyses, although they are far below those predicted by the worst-case approximation ratios. Also, list-based schedules now produce consistently smaller makespans compared to the pack-based ones, which we believe is due to the larger variability in the simulated task profiles than that of the Stream benchmark. This allows the list-based schedules to explore more effectively the gaps between successive task executions (as reflected by the theoretical bounds). However, the makespan difference between the two scheduling paradigms becomes smaller with more resource types, suggesting that pack-based scheduling is indeed a promising solution when more resource types are to be managed. Finally, transform-based solutions are superior compared to the direct-based ones in terms of both makespan and algorithm running time (for which there are 2-3 orders of magnitude difference with more than two resource types). In general,

for workloads with balanced resource requirements, transform-based scheduling is expected to offer both fast and efficient scheduling solutions.

Note that it may seem in Figure 4 that the execution time increases when the number of resources increases. In practice it is only the ratio to the optimal solution with d resources that increases. However the value of that optimal solution decreases when we add resources since we have more wiggle room for optimization. Hence in practice the solution computed with more resources potentially performs better. This is what we observed experimentally when we computed a solution with one or with two types of resources in Figure 3.

6 Conclusion and Future Work

List scheduling and pack scheduling are two scheduling paradigms that have been studied extensively in the past. While most prior works focused on scheduling under a single resource type, in this paper we joined a handful of researchers on scheduling under multiple-resource constraints in these two scheduling paradigms. Our analyses and evaluation results show that scheduling becomes more difficult with increasing number of resource types, but considering all available resources is essential for improving the application performance. Despite the better theoretical bound of list scheduling, pack scheduling has been shown to have comparable or, in some scenarios, even better performance, especially when more types of resources are included in the scheduling decision. The results offer useful insights to the design of emerging architectures and systems that need to incorporate multiple dimensions in the resource management.

Future work will be devoted to the design of improved list or pack scheduling algorithms (or any other practical algorithms beyond the two scheduling paradigms) with multiple-resource constraints. An immediate open question is whether the $2d$-approximation for list scheduling can be improved, e.g., to $(d+1)$-approximation as proven by Garey and Graham for rigid tasks [12], which is equivalent to Phase 2 of our list-based algorithm but with fixed resource allocation. Given our list algorithm's lower bound of $2d$ for moldable tasks (Theorem 3), any improvement will likely be achieved through the design of more sophisticated resource allocation strategies (Phase 1) or through a more coupled design/analysis of the two phases. Prior work [24] on scheduling under a single-resource type has shed light on some possible directions (e.g., by using dual approximation).

Acknowledgements

We would like to thank Nicolas Denoyelle for sharing the code to run the modified Stream benchmark on the KNL processor. This research is supported in part by the National Science Foundation under the award CCF 1719674. Finally we would like to thank the reviewers for useful comments.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS'67*, pages 483–485, 1967.

[2] G. Aupy, M. Shantharam, A. Benoit, Y. Robert, and P. Raghavan. Co-scheduling algorithms for high-throughput workload execution. *Journal of Scheduling*, 19(6):627–640, 2016.

[3] H. Barwick. SKA telescope to generate more data than entire Internet in 2020. `https://www.computerworld.com.au/article/392735/ska_telescope_generate_more_data_than_entire_internet_2020/`, 2011.

[4] O. Beaumont and A. Guermouche. Task scheduling for parallel multifrontal methods. In *Euro-Par*, pages 758–766, 2007.

[5] K. P. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In *ICPP*, pages 72–75, 1990.

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

[7] N. Denoyelle. Instanciate the Cache Aware Roofline Model on single socket and multisocket systems. `https://github.com/NicolasDenoyelle/Locality-Aware-Roofline-Model`, 2017.

[8] N. Denoyelle, B. Goglin, A. Ilic, E. Jeannot, and L. Sousa. Modeling large compute nodes with heterogeneous memories with cache-aware roofline model. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation - 8th International Workshop, PMBS 2017, Denver, CO, USA, November 13, 2017, Proceedings*, pages 91–113, 2017.

[9] M. Dreher and B. Raffin. A flexible framework for asynchronous in situ and in transit analytics for scientific simulations. In *CCGrid*, pages 277–286, 2014.

[10] D. G. Feitelson. Job scheduling in multiprogrammed parallel systems (extended version). *IBM Research Report RC19790(87657)*, 1997.

[11] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. Scheduling the I/O of HPC applications under congestion. In *IPDPS*, pages 1013–1022, 2015.

[12] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. Comput.*, 4(2):187–200, 1975.

[13] 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, 1979.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 323–336, 2011.

[15] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.*, 44(4):455–466, Aug. 2014.

[16] A. Guermouche, L. Marchal, B. Simon, and F. Vivien. Scheduling trees of malleable tasks for sparse linear algebra. In *Proceedings of the Euro-Par Conference*, pages 479–490, 2015.

[17] A. Hartstein, V. Srinivasan, T. Puzak, and P. Emma. On the nature of cache miss behavior: Is it $\sqrt{2}$? *J. Instruction-Level Parallelism*, 2008.

[18] Y. He, J. Liu, and H. Sun. Scheduling functionally heterogeneous systems with utilization balancing. In *IPDPS*, pages 1187–1198, 2011.

[19] Y. He, H. Sun, and W.-J. Hsu. Adaptive scheduling of parallel jobs on functionally heterogeneous resources. In *ICPP*, page 43, 2007.

[20] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *SODA*, pages 490–498, 1999.

[21] W. Leinberger, G. Karypis, and V. Kumar. Job scheduling in the presence of multiple resource requirements. In *Supercomputing*, 1999.

[22] W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *SODA*, pages 167–176, 1994.

[23] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, 1991-2007. http://www.cs.virginia.edu/stream/.

[24] G. Mounié, C. Rapine, and D. Trystram. A 3/2-approximation algorithm for scheduling independent monotonic malleable tasks. *SIAM J. Comput.*, 37(2):401–412, 2007.

[25] M. NoroozOliaee, B. Hamdaoui, M. Guizani, and M. B. Ghorbel. Online multi-resource scheduling for minimum task completion time in cloud servers. In *INFOCOM Workshops*, 2014.

[26] G. N. S. Prasanna and B. R. Musicus. Generalized multiprocessor scheduling and applications to matrix computations. *IEEE Trans. Parallel Distrib. Syst.*, 7(6):650—664, 1996.

[27] P. Sanders and J. Speck. Efficient parallel scheduling of malleable tasks. In *IPDPS*, pages 1156–1166, 2011.

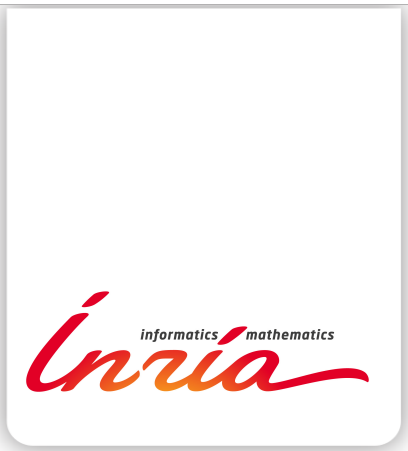
[28] H. Shachnai and J. J. Turek. Multiresource malleable task scheduling. Technical report, IBM T.J. Watson Research Center, 1994.

[29] M. Shantharam, Y. Youn, and P. Raghavan. Speedup-aware co-schedules for efficient workload management. *Parallel Proc. Letters*, 23(2), 2013.

[30] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016.

[31] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *SPAA*, 1992.

[32] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *JSSPP*, pages 44–60, 2003.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

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

ISSN 0249-6399