Scheduling Rigid, Evolving Applications on Homogeneous Resources

Cristian Klein, Christian Pérez

▶ **To cite this version:**

Cristian Klein, Christian Pérez. Scheduling Rigid, Evolving Applications on Homogeneous Resources. [Research Report] RR-7205, INRIA. 2010. <inria-00456769v2>

HAL Id: inria-00456769

https://hal.inria.fr/inria-00456769v2
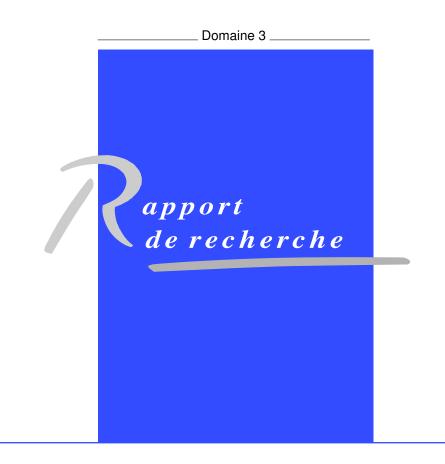
Submitted on 17 Feb 2010

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Scheduling Rigid, Evolving Applications on Homogeneous Resources

Cristian KLEIN — Christian PÉREZ

N° 7205 — version 2

initial version 15 February 2010 — revised version 16 February 2010

Domaine 3

Rapport de recherche

ISRN INRIA/RR--7205--FR+ENG

ISSN 0249-6399

Scheduling Rigid, Evolving Applications on Homogeneous Resources

Cristian KLEIN, Christian PÉREZ

Abstract: Classical applications executed on clusters or grids are either rigid/moldable or workflow-based. However, the increase of resource computing and storage capabilities has leveraged more complex applications. For example, some code coupling applications exhibit changing resource requirements without being a workflow. Executing them on current batch schedulers leads to an inefficient resource usage, as a block of resources has to be reserved for the whole duration of the application. This paper studies the problem of offline scheduling of rigid and evolving applications on homogeneous resources. It proposes several scheduling algorithms and evaluates them based on simulations. Results show that significant makespan and resource usage improvement can be achieved with short scheduling computing time.

Key-words: clusters, resource management, scheduling, evolving application, rigid application

Ordonnancement d'applications rigides et évolutives sur des ressources homogènes

Résumé :   Traditionnellement, les applications exécutées sur des grappes ou des grilles de calcul sont soit rigides / moldables, soit des flots de travail. Cependant, l'accroissement des capacités de calcul et de stockage a fait apparaître des applications de plus en plus complexe. Par exemple, les applications de couplage de code exhibent des changements de leurs besoins en ressources, sans que leur structure ne soit un flot de travail. Leur exécution via les ordonnanceurs actuels de ressources (batch) conduit à une utilisation inefficace des ressources car les ressources doivent être réservées pour toute la durée de l'exécution de l'application. Cet article étudie le problème de l'ordonnancement hors ligne des applications rigides et évolutives sur des ressources homogènes comme une grappe. Plusieurs algorithmes d'ordonnancement sont proposés et évalués via des simulations. Les résultats montrent que la durée total d'exécution et l'utilisation des ressources peuvent être grandement améliorées pour un faible cout d'ordonnancement.

Mots-clés : grappes, gestion des ressources, ordonnancement, application évolutive, application rigide

1 Introduction

The increasing computing capabilities offered by parallel and distributed architectures enable the design of more and more complex applications. For example, accuracy of scientific numerical simulations is increased by coupling several codes, each code typically simulating a phenomenon. Due to the increase in computing capability, the current trend is to execute such codes concurrently rather than one after the other. Moreover, some codes are only active during some phases of the applications, leading to non-constant resource requirements. Progress is being made in designing advanced programming models to express such applications. For example, STCM [1] is a component model that enables the use of both spatial and temporal composition.

However, current RMSs do not offer an adequate level of functionality to efficiently support applications written in such programming models. They only handle simple abstractions, such as rigid jobs or workflows. Code-coupled applications are usually not a workflow as coupled codes exchange messages.

This issue is studied within the French ANR COOP project[1] that aims at improving the relationships between programming models and Resource Management Systems (RMSs). Within this project, EDF R&D is providing applications in various simulation fields such as neutronics or thermo-hydraulics. This paper studies whether it is valuable for RMSs to take into account the evolvement of resource requirement of such applications. It deals with rigid, fully-predictable, and evolving applications on homogeneous clusters. The paper proposes offline scheduling algorithms and evaluates them with simulations.

The rest of the article is structured as follows: Section 2 characterizes applications from the RMS's point of view and presents related work. Section 3 gives a few definitions and notations used throughout the paper and formally introduces the problem. Section 4 proposes algorithms to solve the stated problem, which are evaluated with simulations in Section 5. Finally, Section 6 concludes this paper and opens up some perspectives.

2 Context and Related Work

2.1 Properties of Applications with Respect to Resource Usage

An application can have different properties that may impact its resource utilization. Based on a classification inspired by [2], this paper considers two such properties: malleability and evolvement.

Malleability is the property of an application to reconfigure itself based on RMS-driven resource allocation changes. Several classes of applications can be distinguished: *rigid* (resources are fixed at compilation time), *moldable* (resource are set at the beginning of their execution – e.g. most MPI applications), and *malleable* (applications may reconfigure themselves at any time).

Evolvement characterises whether an application changes its resource requirements during execution. *Non-evolving* applications have the same requirements during their whole execution (classical MPI applications behave like this). *Evolving* applications have changing requirements during their execution. For example, a multi-stage application might require fewer resources during the initial pre-processing than during its main processing loop. In contrast to malleability, evolvement implies resource changes which are driven by applications. Depending on how much in advance the evolution can be predicted, an application can range from *fully-predictable* (evolution of resource requirements is known at submission) to *non-predictable*.

Contrary to [2], the presented classification takes into account applications whose resource allocation may change, both due to RMS-driven reallocation and internal requirements. For example, an application might be both malleable and evolving: the RMS may grow and shrink the application, however the application's minimum and maximum number of required nodes may change in time.

2.2 Related Work

Most batch schedulers consider jobs to be rigid and non-evolving. At submission time, the user has to give a wall-time and a number of nodes. Scheduling is done using either First Come First Serve (FCFS) [3] or Conservative Back-Filling (CBF) [4] strategies. FCFS chooses available resources at the end of the waiting queue and is used by default in PBS [5] and Sun Grid Engine [6]. CBF works similarly, but tries to fill in gaps in the waiting queue, with the condition of not delaying jobs which are already scheduled. CBF is used in resource managers such as Loadleveler [7] and OAR [8].

---

[1] http://coop.gforge.inria.fr

A lot of research has been dedicated to improving application malleability, since both resource utilization and job response time can be improved if jobs are moldable or malleable [9]. Since most applications are already moldable (e.g. MPI applications), several works have aimed at creating malleable programming models. For example, Dynaco [10] is a framework which allows applications to dynamically adapt when resources are preempted.

On the resource manager's side, to our knowledge, OAR is the only production-used batch scheduler which supports moldable jobs. While not directly supported by OAR, malleable jobs can be created using best-effort jobs [11]. As for grid resource managers, KOALA directly supports malleable jobs and is able to send grow/shrink messages to applications [12].

The most commonly studied evolving applications are workflows. There are two main approaches for running them: submitting tasks as individual jobs [13] or creating a big "pilot" job, inside which tasks are scheduled [14]. However, this abstraction is not usable for running applications whose components have spatial relationships [1].

3   Problem Statement

To accurately define the problem studied in this paper, let us first introduce some mathematical definitions and notations.

3.1   Definitions and Notations

Let an evolution profile (EP) be a sequence of *stages*, each stage being characterized by a *duration* and a *number of nodes*. Formally, $ep = \{(d_1, n_1), (d_2, n_2),$
$\ldots, (d_N, n_N)\}$, where N is the number of stages, d_i is the duration and n_i is number of nodes of stage i.

An EP can be used to represent three distinct concepts. First, it can represent application resource requirements. For example, $ep = \{(500, 5), (3600, 10)\}$ models a two-stage application with the first stage having a duration of $500\,\text{s}$ and requiring 5 nodes and the second stage having a duration of $3600\,\text{s}$ and requiring 10 nodes. Non-evolving, rigid applications can be represented by a single stage EP. We shall call these *requested EPs*.

Second, an EP can represent the number of nodes allocated to an application, including delaying and stretching. For example, an allocation of nodes to the previous two-stage application with a delay of $3600\,\text{s}$ and a stretch of $500\,\text{s}$ to the first stage is modelled with $ep_s = \{(3600, 0), (1000, 5), (3600, 10)\}$. When using an EP for this purpose, we shall call it *scheduled EP*.

Third, EP can represent resource occupation of a system. For example, if 10 nodes are busy for $1200\,\text{s}$, afterwards 20 nodes are busy for $3600\,\text{s}$, then $ep_r = \{(1200, 10), (3600, 20)\}$. We shall call it *resource EP*.

We define the *stretched* and *delayed* EPs of $ep = \{(d_1, n_1), \ldots, (d_N, n_N)\}$ as follows:

- $ep' = \{(d_1', n_1), \ldots, (d_N', n_N)\}$ is a *stretched* EP of ep, if $\forall i = 1 \ldots N, d_i' \geq d_i$;

- $ep'' = \{(d_0, 0), (d_1, n_1), \ldots, (d_N, n_N)\}$ is a *delayed* EP of ep, if $d_0 \geq 0$.

For manipulating EPs, we use the following helper functions:

- $ep(t)$ returns the number of nodes at time coordinate t,
  i.e. $ep(t) = n_1$ for $t \in [0, d_1[$, $ep(t) = n_2$ for $t \in [d_1, d_1 + d_2[$, etc.

- $\max(ep, t_0, t_1)$ returns the maximum number of nodes between t_0 and t_1
  i.e. $\max(ep, t_0, t_1) = \max_{t \in [t_0, t_1[} ep(t)$, and 0 if $t_0 = t_1$.

- $\text{loc}(ep, t_0, t_1)$ returns the end time coordinate of the last stage containing the maximum, restricted to $[t_0, t_1]$ i.e. $\text{loc}(ep, t_0, t_1) = t \Rightarrow$
  $\max(ep, t_0, t) = \max(ep, t_0, t_1) > \max(ep, t, t_1)$.

- $\text{delay}(ep, t_0)$ returns an evolution profile that is delayed with t_0.

- $ep_1 + ep_2$ is the sum of the two EPs, i.e. $\forall t, (ep_1 + ep_2)(t) = ep_1(t) + ep_2(t)$.

- $\text{chps}(ep)$ returns the set of time coordinates between stages (*change-points*)
  i.e. $\text{chps}(ep) = (d_0, d_0 + d_1, d_0 + d_1 + d_2, \ldots)$.

3.2 Formal Problem Statement

Based on the previous definitions and notations, the problem can be stated as follows. Let S be the set of nodes in a homogeneous cluster, having a cardinality of n_{max}. n_{app} applications having the requested EPs $ep^{(i)}, i = 1 \ldots n_{app}$ arrive at time 0. All applications are valid, which means $\forall t, ep^{(i)}(t) \leq n_{max}$. The problem is to compute $ep_s^{(i)}, i = 1 \ldots n_{app}$ scheduled EPs, $T = \{t_1, t_2, \ldots t_{M+1}\}$ a sequence of time coordinates, and $S_j^{(i)} \subseteq S$ the set of nodes allocated to application i during the time interval $[t_j, t_{j+1}[, \forall j = 1 \ldots M$, such that the following conditions are simultaneously met:

C1 $ep_s^{(i)}$ is a delayed and stretched version of $ep^{(i)}, \forall i = 1 \ldots n_{app}$;

C2 the mapping satisfies the scheduled EPs:
$$\max(ep_s^{(i)}, t_j, t_{j+1}) \leq |S_j^{(i)}| \leq n_{max}, \forall i = 1 \ldots n_{app}, \forall j = 1 \ldots M;$$

C3 one node is allocated to a single applications: $S_j^{(i)} \cap S_j^{(i')} = \emptyset, \forall i \neq i'$;

C4 applications may chose which resources to free.

C1 expresses that, when an evolving application starts or increases its requirements, it has to wait for the RMS to allocate resources to it. While this happens, resources which are already allocated to the application remain busy, but the computation is assumed not to make progress. The scheduled EP of the application will be a delayed and stretched version of its requested EP. The scheduled EP is the one that the RMS will have to fulfill (C2), by granting exclusive access to resources (C3).

Code-coupling applications usually have a "driver" process which is responsible for managing the application's components. It is crucial that the node on which the driver runs be not taken away from the application, that is why applications should be able to choose which nodes to free (C4). Preemption and migration of the applications are also excluded by this condition, since they imply involuntarily freeing resources.

The issue is to optimize makespan and resource waste.

4 Scheduling Rigid, Evolving Applications

This section aims at solving the above problem in three steps. First, the problem is simplified thanks to an algorithm that maps applications to node IDs given their scheduled EPs. Second, an algorithm which transforms requested EPs into scheduled EPs is presented. It requires a `fit` function which operates on two EPs. Third, several algorithms for computing a `fit` function are described.

4.1 A Mapping Algorithm from Node Numbers to Node IDs

Let S be the set of nodes in the system and n_{max} its cardinality. Let n_{app} be the number of applications to be scheduled, described by their scheduled EPs $ep_s^{(i)}, i = 1 \ldots n_{app}$, whose sum do not exceed resources: $\forall t, \sum_i ep_s^{(i)}(t) \leq n_{max}$.

Algorithm 1 computes a set of node IDs. In fact, it finds a sequence $SeqT$ of time coordinates and a set of node IDs allocated for each application i for all intervals of $SeqT$ such that conditions C2, C3 and C4 hold. The algorithm first computes the union of the change-points of all input EPs. For each change-point, it finds the applications which changed resource requirements and updates the set of resources as necessary. When an application decreases its node requirements, it lets the application selects which node IDs should be removed. When an application increases its node requirements, it determines new node IDs to add to the set of nodes of the application. Because of lack of space, the paper does not present the proof by induction that the algorithm respects the conditions C2, C3 and C4.

4.2 An Algorithm for Offline Scheduling of Rigid Evolving Applications

This section presents an algorithm which transforms multiple requested EPs into scheduled EPs. Thanks to Algorithm 1, these scheduled EPs can then be mapped to node IDs. More formally, given n_{max} nodes and n_{app} applications, submitted at the same time, whose resource requirements are described by $ep^{(i)}, i = 1 \ldots n_{app}$ with $ep^{(i)}(t) \leq n_{max}, \forall t$, the issue is to find scheduled EPs $ep_s^{(i)}$, s.t. $\sum_i ep_s^{(i)} \leq n_{max}$ and $ep_s^{(i)}$ are delayed and stretched EPs of $ep^{(i)}$ (C1).

Algorithm 1 Mapping Scheduled EPs to Node IDs

Input:
   S : set of available nodes in our system,
   $ep_s^{(i)}(t)$: scheduled EP of application i, s.t. $\forall t, \sum_i ep_s^{(i)}(t) \leq |S|$
Output:
   $SeqT = \{t_1, \ldots, t_{M+1}\}$: a sequence of time coordinates,
   $S_j^{(i)}$: the set of nodes allocated to application i between $[t_j, t_{j+1}[, j \in 1 \ldots M$

---

   assume $S_0^{(i)} = \emptyset, \forall i = 1 \ldots n_{app}$
   $SeqT \leftarrow \bigcup_i \text{chps}(ep_s^{(i)})$
   for $j = 1 \ldots |SeqT|$ **do**
      for all $i, |S_{j-1}^{(i)}| > ep_s^{(i)}(t_j)$ **do**
         $S' \leftarrow$ set of $|S_{j-1}^{(i)}| - ep_s^{(i)}(t_j)$ resources freed by application i
         $S = S \cup S'$
         $S_j^{(i)} \leftarrow S_{j-1}^{(i)}/S'$
      end for
      for all $i, |S_{j-1}^{(i)}| < ep_s^{(i)}(t_j)$ **do**
         $S' \leftarrow$ set of $ep_s^{(i)}(t_j) - |S_{j-1}^{(i)}|$ resources randomly chosen from S
         $S = S/S'$
         $S_j^{(i)} \leftarrow S_{j-1}^{(i)} \cup S'$
      end for
   end for

Algorithm 2 Off-line Scheduling of Rigid, Evolving Applications.

Input:
   $ep^{(i)}$: requested EP of the application i,
   n_{max} : maximum number of nodes in the system,
   $\text{fit}(ep_{src}, ep_{dst}, n_{max}) \rightarrow (t_s, ep_s)$: a fit function
Output:
   $ep_s^{(i)}$: scheduled EP of application i

   $ep_r \leftarrow$ empty EP
   for all $ep^{(i)}$ **do**
      $t_s^{(i)}, ep_s^{(i)} \leftarrow \text{fit}(ep^{(i)}, ep_r, n_{max})$
      $ep_s^{(i)} \leftarrow delay(ep_s^{(i)}, t_s^{(i)})$
      $ep_r \leftarrow ep_r + ep_s^{(i)}$
   end for

Algorithm 2 is a simple offline scheduling algorithm that solves this issue. It starts by initializing ep_r, the resource EP, representing how resource occupation evolves over time, to the empty EP. Then, it considers each requested EP, potentially stretching and delaying it using a helper fit function. The resulting EP is added to ep_r, effectively updating the resource occupation.

The fit function takes as input the maximum number of nodes in the system n_{max}, a source evolution profile ep_{src} and a destination evolution profile ep_{dst} and returns a time coordinate t_s and ep_s a stretched ep_{src}, such that $\forall t, ep_{dst}(t) + delay(ep_s, t_s)(t) \leq n_{max}$. A very simple fit implementation consists in delaying ep_{src} such that it starts after ep_{dst}.

Throughout the whole algorithm, the condition $ep_r(t) \leq n_{max}, \forall t$ is guaranteed by the post-conditions of the fit function. Since at the end of the algorithm $ep_r = \sum_i ep_s^{(i)}$, resources will not be exceeded.

Algorithm 2 guarantees that a scheduled application is not delayed nor stretched by any application submitted after it.

4.3 The `fit` Function

The core of the scheduling algorithm is the `fit` function, which stretches and delays a source EP over a destination EP. It returns a scheduled EP, so that the sum of the destination EP and scheduled EP does not exceed available resources.

4.3.1 Stretching

Because it can stretch an EP, the `fit` function is an element of the efficiency of a schedule. It is interesting for a `fit` algorithm to stretch an application so as to enable it to start earlier at the price of using more resources than without any stretch. Hence, there is a trade-off between the resource usage, the application's start time and its completion time.

In order to evaluate the impact of stretching, the proposed `fit` algorithm takes as parameter the *stretch limit*. This parameter expresses how many times the duration of a stage may be increased. For example, if the stretch limit is 2, a stage may not be stretched to more than twice its original duration. Having a stretch limit of 1 means applications will not be stretched, while an infinite stretch limit does not impose any limit on stretching.

4.3.2 Base `fit` Algorithm

Algorithm 3 is an algorithm that aims at efficiently computing the `fit` function, while allowing to choose different stretching limits. It operates recursively for each stage in ep_{src}, going through the following steps:

1. find the earliest time coordinate when the current stage can be placed, so that n_{max} is not exceeded (lines 8 – 11);

2. test if this placement forces a stretch on the previous stage, which exceeds the stretch limit (lines 14 – 17) or exceeds n_{max} (lines 18 – 21);

3. recursively try to place the next stage in ep_{src}, starting at the completion time of the current stage (line 24);

4. store the stretched version of the current stage in ep_s (line 31). The first stage is moved instead of being stretched (line 33).

The recursion ends when all stages have been successfully placed (lines 1 – 3).

Placement of a stage is first attempted at time coordinate t_0, which is 0 for the first stage, or the value given in Step 3 for the other stages. After every failed operation (placement or stretching) the time coordinate is advanced so that the same failure does not repeat:

- if placement failed, jump to the time coordinate after the encountered maximum (line 10);

- if stretching failed due to insufficient resources, jump to the time coordinate after the encountered maximum (computed at line 19, used at line 26)

- if stretching failed due to excessive stretch, jump to the first time coordinate which avoids excessive stretching (computed at line 15, used at line 26)

Since each stage, except the first, is individually placed at the earliest possible time coordinate and the first stage is placed so that the other stages are not delayed, the algorithm guarantees that the application has the earliest possible completion time. However, resource usage is not guaranteed to be optimal.

4.3.3 Post-processing Optimization

In order to reduce resource waste, while maintaining the guarantee that the application completes as early as possible, a *compacting* post-processing phase can be applied. After a first solution is found by the base `fit` algorithm, the stretched profile goes through a compacting phase: the last stage of the applications is placed so that it ends at the completion time found by the base algorithm. Then, the other stages are placed from right (last) to left (first), similarly to the base algorithm. In the worst case, no compacting occurs and the same EP is returned after the compacting phase.

The base `fit` algorithm with compacting first optimizes completion time then start time (it is optimal from stretching point-of-view), but because it acts in a greedy way, it might stretch stages which require more nodes, so it is not always optimal for resource waste.

Algorithm 3 Base $\mathtt{fit}(ep_{src}, ep_{dst}, n_{max}, l, i, t_0) \rightarrow (t_s, ep_s)$

Input:

$ep_{src} = \left\{ \left(d_{src}^{(1)}, n_{src}^{(1)} \right), \left(d_{src}^{(2)}, n_{src}^{(2)} \right), \ldots, \left(d_{src}^{(N_{src})}, n_{src}^{(N_{src})} \right) \right\}$: EP to stretch

$ep_{dst} = \left\{ \left(d_{dst}^{(1)}, n_{dst}^{(1)} \right), \left(d_{dst}^{(2)}, n_{dst}^{(2)} \right), \ldots, \left(d_{dst}^{(N_{dst})}, n_{dst}^{(N_{dst})} \right) \right\}$: destination EP

n_{max} : maximum number of nodes in the system,

l : maximum allowed stretching ($l \geq 1$),

i : index of stage from ep_{src} to start with (initially 1)

t_0 : first moment of time where ep_{src} is allowed to start (initially 0)

Output:

ep_s : stretched ep_{src} s.t. $ep_{dst}(t) + delay(ep_s, t_s) \leq n_{max}$ **or** $\varnothing$ if stretching failed

t_s : time when ep_s starts **or** time when stretching failed

---

    if $i > N_{src}$ {Termination condition, remaining ep_{src} is empty.} **then**
        return $(t_0, \text{empty EP})$
    end if
    $d \leftarrow d_{src}^{(i)}$ {duration of the current stage in ep_{src}}
5:   $n \leftarrow n_{src}^{(i)}$ {nodes of current stage in ep_{src}}
    $t_s \leftarrow t_0$
    loop
        {Attempt to put current stage from ep_{src} at moment t_s in ep_{dst}}
        if $n_{max} - \max(ep_{dst}, t_s, t_s + d) < n$ **then**
10:      **continue** $t_s \leftarrow loc(ep_{dst}, t_s, t_s + d)$
        end if

        {Check whether previous stage can be stretched}
        if $i > 1$ **then**
15:      $t_{eas} \leftarrow t_s - l \cdot d_{src}^{(i-1)}$ {Earliest Allowed Start of previous stage}
            if $t_{eas} > t_0 - d_{src}^{(i-1)}$ {Do we exceed stretch limit?} **then**
                return $t_s \leftarrow t_{eas}; ep_s \leftarrow \varnothing$
            else if $n_{max} - \max(ep_{dst}, t_0, t_s) < n_{src}^{(i-1)}$ **then**
                return $t_s \leftarrow loc(ep_{dst}, t_0, t_s); ep_s \leftarrow \varnothing$
20:      **end if**
        end if

        {Recurse for next stage in ep_{src}}
        $t_s^{tail}, ep_s \leftarrow \mathtt{fit}(ep_{src}, ep_{dst}, n_{max}, i + 1, t_s + d)$
25:     **if** $ep_s = \varnothing$ {i.e. stretching failed while processing next stage} **then**
            continue $t_s \leftarrow t_s^{tail}$
        end if

        {Actually stretch current stage and fill ep_s}
30:     **if** $i > 0$ **then**
            prepend $(t_s^{tail} - t_s, n)$ to ep_s
        else {Special case: do not stretch the first stage, move it}
            prepend (d, n) to ep_s; $t_s \leftarrow t_s^{tail} - d$
        end if
35:     **return** t_s, ep_s
    end loop

Name	Waste (%)			Reservation (relative)			Utilisation (%)			Makespan (relative)			Time (ms)		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
CBF	43	70	116	1	1	1	30	40	51	1	1	1	4.64	6.2	9.41
nos	0	0	0	.46	.58	.69	49	61	73	.49	.65	.82	11.4	24.7	55.8
mic	0	2	11	.47	.60	.71	50	63	75	.48	.64	.82	11.4	24.4	45.4
mic+c	0	ϵ	4	.46	.59	.70	53	63	75	.48	.63	.82	17.1	36.7	88.6
inf	0	7	22	.49	.63	.78	52	64	73	.49	.63	.78	11.4	23.4	49.2
inf+c	0	1	11	.46	.59	.71	55	64	74	.47	.62	.78	17.6	36	124

Table 1: Comparison of Scheduling Algorithms (System-centric Metrics)

4.4 Discussions

This section has presented a solution to the problem stated in Section 3. Since minimizing completion time and resource waste is difficult – it is a multiple criteria optimization, the presented strategies aim at minimizing completion time and reducing resource waste. However, these strategies are per-application optimizations. A more global strategy taking into account several applications at a time has not been studied. One motivation is to be able to move to online scheduling in future work.

5 Experiments and Analysis

This section evaluates the benefits and drawbacks of taking into account rigid, evolving applications. It is based on a series of experiments done with a home made simulator developed in Python. The experiments are first described, then the results are analyzed.

5.1 Experiments Description

The experiments compare two kinds of scheduling algorithms: the CBF algorithm and variations of Algorithm 2. Applications are seen by the CBF algorithm as rigid, non-evolving: the requested node number is the maximum number of nodes of all stages and the walltime is the sum of the durations of all stages.

Five versions of Algorithm 2 are considered to evaluate the impact of its options: (1) base fit with no stretching (`nos`), (2) base fit with stretch limit of 2 (`mic`), (3) base fit with limit stretch of 2 and compacting (`mic+c`), (4) base fit with infinite stretching (`inf`) and (5) base fit with infinite stretching with compacting (`inf+c`).

Two kinds of metrics are measured: system-centric and user-centric. The five system-centric metrics considered are: (1) resource `waste` – the area (*nodes × duration*) of resources (expressed as percent of total resources), which has been allocated to applications, but has not been used to make computations; (2) resource `reservation` – the resource area that has been reserved by applications; (3) resource `utilisation` – the resource area (expressed as percent of total resources) that has been effectively used for computations; (4) `Makespan` – the last completion time of the applications; (5) `Time` – the computation time taken by a scheduling algorithm to schedule one test on a laptop with a Intel®Core™2 Duo processor running at $2.53\,\mathrm{GHz}$.

The five user-centric metrics considered are: (1) per-test average job completion time (`Avg. JCT`); (2) per-test average job waiting time (`Avg. JWT`); (3) the number of stretched applications (`stretches`) as a percentage of the total number of applications in a test; (4) by how much was an application stretched (`Job Stretch`) as a percentage of its initial total duration; (5) per-job `waste` as a percentage of resources allocated to the application.

As we are not aware of any public archive of evolving application workloads, we created synthetic test-cases. It enables us to vary the parameters to cover more cases than the code-coupling applications studied in the COOP project.

A test case is made of a uniform random choice of the number of applications, their number of stages, as well as the duration and requested number of nodes of each stage. We tried various combinations that gave similar results. Table 1 and 2 respectively present the results for the system- and user-centric metrics of an experiment made of 1000 tests. The number of applications per test is within $[15, 20]$, the number of stages within $[1, 10]$, a stage duration within $[500, 3600]$ and the number of nodes per stage within $[1, 75]$.

Name	Avg. JCT (relative)			Avg. JWT (relative)			Stretches (%)			Job Stretch (%)			Job Waste (%)		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
CBF	1	1	1	1	1	1	0	0	0	0	0	0	0	67	681
nos	.42	.61	.84	.36	.55	.81	0	0	0	0	0	0	0	0	0
mic	.45	.61	.84	.36	.54	.80	0	22	56	0	4	76	0	2	75
mic+c	.44	.60	.84	.37	.54	.81	0	7	40	0	ϵ	60	0	ϵ	41
inf	.43	.62	.81	.27	.53	.76	0	26	62	0	19	884	0	6	360
inf+c	.44	.60	.81	.35	.53	.76	0	13	47	0	5	1354	0	1	119

Table 2: Comparison of Scheduling Algorithms (User-centric Metrics)

5.2 Analysis

Administrator's Perspective CBF is outperformed by all other strategies which improve effective resource utilisation, reduce makespan and drastically reduce resource waste within reasonable scheduling time. Compared to CBF, all algorithms reduce resource reservation, therefore saving energy [15]. `inf` behaves slightly poorer, while the other algorithms behave similarly.

There is a trade-off between resource waste and makespan (especially when looking at maximum values). However makespan differs less between algorithms than waste. If maintaining resources is very expensive, a system administrator may choose the `nos` algorithm, whereas if she wants to favour throughput, she would choose the `mic+c` algorithm.

User's Perspective When compared to CBF, the proposed algorithms always improve both per-job resource waste and average job completion time. When looking at maximum values, the trade-off between stretch / waste vs. completion time is again highlighted. Algorithms which favor stretching (`inf`, `inf+c`) reduce average waiting time, but not necessarily average completion time.

The results show that waste is not equally split among jobs, instead, few jobs are stretched a lot. Since most cluster / grid systems employ some kind of credit system, where users are penalised for using too many resources, using the `inf` and `inf+c` algorithm (which do not guarantee an upper bound on the waste) should be avoided. In the context of offline scheduling, the benefits of using `mic+c` instead of `nos` are small, at the expense of significant per-job resource waste. Therefore, users might prefer not to stretch their applications at all.

Global Perspective From both perspectives, stretching jobs has a very limited benefit in the context of offline scheduling. Therefore, the `nos` algorithm seems to appear as the best choice. Taking into account evolvement of resource requirements of applications enables improvement of all metrics compared to CBF.

6 Conclusions

Some applications, such as code-coupling applications, can exhibit evolving resource requirements. As it may be difficult to obtain accurate evolvement information, this paper answers to the question whether this effort would be worthwhile in term of system and user perspectives. The paper has presented a model of rigid, fully-predictable evolving applications. Then, it has proposed an offline scheduling algorithm, with optional stretching capabilities. Experiments show that taking into account resource requirement evolvement leads to important improvements in all measured metrics – such as resource utilization and completion time. However, considered stretching strategies do not appear very valuable.

Future work can be divided in three directions. First, the problem can be extended to include evolving moldable and malleable applications. Second, as real applications are not fully-predictable, this assumption has to be changed and applications predictability has to be set to a limited horizon. Third, the problem has to be extended to online scheduling.

References

[1] Bouziane, H., Pérez, C., Priol, T.: A software component model with spatial and temporal compositions for grid infrastructures. In: Proc. 14th Intl. Euro-Par Conference (Euro-Par 08),. Volume 5168., Las Palmas de Gran Canaria, Spain, Springer Berlin / Heidelberg (26-29 August 2008) 698–708

[2] Feitelson, D.G., Rudolph, L.: Towards convergence in job schedulers for parallel supercomputers. In: IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, London, UK, Springer-Verlag (1996) 1–26

[3] Schwiegelshohn, U., Yahyapour, R.: Analysis of first-come-first-serve parallel job scheduling. In: SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (1998) 629–638

[4] Lifka, D.: The ANL/IBM SP scheduling system. In: Job Scheduling Strategies for Parallel Processing, Springer-Verlag (1995) 295–303

[5] Bayucan, A., Henderson, R., Lesiak, C., Mann, B., Proett, T., Tweten, D.: Portable batch system: External reference specification. Technical report, MRJ Technology Solutions (November 1999)

[6] Gentzsch, W.: Sun grid engine: towards creating a compute power grid. In: Proceedings of the first IEEE/ACM International Symposium on Cluster Computing and the Grid. (2001) 35–36

[7] Kannan, S., Roberts, M., Mayes, P., Brelsford, D., Skovira, J.: Workload Management with LoadLeveler. IBM Press (2001)

[8] Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., Richard, O.: A batch scheduler with high level components. CoRR **abs/cs/0506006** (2005)

[9] Hungershofer, J.: On the combined scheduling of malleable and rigid jobs. In: SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing, Washington, DC, USA, IEEE Computer Society (2004) 206–213

[10] Buisson, J., André, F., Pazat, J.L.: A framework for dynamic adaptation of parallel components. In: PARCO. (2005) 65–72

[11] C. Cera, M., Georgiou, Y., Richard, O., Maillard, N., O. A. Navaux, P.: Supporting MPI malleable applications upon the OAR resource manager. In: COLIBRI. (2009)

[12] Buisson, J., Sonmez, O., Mohamed, H., Lammers, W., Epema, D.: Scheduling malleable applications in multicluster systems. Technical Report TR-0092, Institute on Resource Management and Scheduling, CoreGRID (May 2007)

[13] Amar, A., Bolze, R., Bouteiller, A., Chouhan, P.K., Chis, A., Caniou, Y., Caron, E., Dail, H., Depardon, B., Desprez, F., Gay, J.S., Le Mahec, G., Su, A.: Diet: New developments and recent results. In et al. (Eds.), L., ed.: CoreGRID Workshop on Grid Middleware (in conjunction with EuroPar2006). Number 4375 in LNCS, Dresden, Germany, Springer (August 2006) 150–170

[14] Graciani Diaz, G., Tsaregorodtsev, A., Casajus Ramo, A.: Pilot Framework and the DIRAC WMS. In: CHEP 09, Prague Tchèque, République (May 2009)

[15] Orgerie, A.C., Lefèvre, L., Gelas, J.P.: Save watts in your grid: Green strategies for energy-aware framework in large scale distributed systems. In: ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems, Washington, DC, USA, IEEE Computer Society (2008) 171–178