A Sequential Cooperative Game Theoretic Approach to Scheduling Multiple Large-scale Applications in Grids

Rubing Duan[a], Radu Prodan[b], Xiaorong Li[a]

[a] *Institute of High Performance Computing, Agency for Science, Technology and Research, 1 Fusionopolis Way, #6 Connexis, Singapore 138632*
[b] *Institute of Computer Science, University of Innsbruck, Technikerstr. 21a, 6020 Innsbruck, Austria*

Abstract

Scheduling large-scale applications in heterogeneous distributed computing systems is a fundamental NP-complete problem that is critical to obtaining good performance and execution cost. In this paper, we address the scheduling problem of an important class of large-scale Grid applications inspired from real-world, characterized by a huge number of homogeneous, concurrent, and computationally-intensive tasks that are the main sources of performance, cost, and storage bottlenecks. We propose a new formulation of this problem based on a cooperative distributed game theory-based method applied using three algorithms with low time complexity for optimizing three important metrics in scientific computing: execution time, economic cost, and storage requirements. We present comprehensive experiments using simulation and real-world applications that demonstrate the effectiveness of our approach in terms of time and fairness compared to other related algorithms.

Keywords: Grid computing, game theory, scheduling, performance, economic cost, storage

1. Introduction

Over the last decade, distributed computing systems including *grids* and *clouds* have evolved towards a worldwide infrastructure providing scientific applications with dependable, consistent, pervasive, and inexpensive access to geographically-distributed high-end computational capabilities. To program such a large and scalable infrastructure, loosely coupled-based coordination models of legacy software components such as workflows [1] have emerged as one of the most successful programming paradigms in the scientific community. One of the most challenging NP-complete problems that researchers try to solve is how to schedule large-scale scientific applications to distributed and heterogeneous resources such that certain objective functions such as total execution time (called from hereon *makespan*) in academic Grids or economic cost (in short

DOI: https://doi.org/10.1016/j.future.2013.09.001

cost from hereon) in business or market-oriented Clouds are optimized, and certain execution constraints such as storage requirements are fulfilled. From the end-users' perspective, both minimizing cost or execution time are preferred functionalities, whereas from the system's perspective fairness can be considered as a good motivation. Currently, only a few schemes can deal with both perspectives, such as optimizing one user objective (e.g. makespan, cost) while providing a good fairness to all users. On the other hand, many applications can generate huge data sets in a relatively short time, such as the Large Hadron Collider [2] expected to produce $5-6$ petabytes of data per year, which must be accommodated and handled through appropriate scheduling storage constraints.

Traditionally, this scheduling problem has been addressed in the form of a centralized meta-scheduling service [3] that tries to map activities (or tasks) of single or multiple applications to individual processors [4, 5]. This approach has two drawbacks. First, there can be no single meta-scheduler in a distributed environment like the Grid where individual applications are controlled and managed by different actors with potentially different goals and interests. Second, access to a remote computational resource is usually mediated by a resource manager or job queuing system [6] that prohibits direct access to processors. Moreover, in a business Cloud the total number of processors available at a site may not even be public information. Since there are many applications which are competitors for the use of available resources, several issues arise such as the efficient resource allocation for different applications taking into account their individual performance, cost, storage, and other constraint requirements, the ability to implement allocation schemes in a distributed manner with no centralized decision point, and the fair use of resources from system perspective.

In this paper, we address these issues by proposing three scheduling schemes for an important class of large-scale applications characterized by large sets of independent and identical activities interconnected through simple control flow and data flow dependencies:

- *Makespan scheduling* (see Section 3) minimizes the expected execution time of applications (known to be an NP-complete problem [7]) based on a decentralized cooperative game theory algorithm. We compare the performance of our approach with six related heuristics and show that, for our special class of applications, the game theoretic algorithm is superior in complexity, quality of result, and fairness. Our proposed algorithm may not be suited for highly heterogeneous applications for which the scheduling problem cannot be properly formulated as a typical and solvable game, consisting of phases which can be specifically defined so that game players can bargain with no dependencies between each other.

- *Cost scheduling* (see Section 4) minimizes the cost of execution while guaranteeing a user deadline. We present a solution to this problem consisting of three steps: deadline assignment, partitioning, and cost optimization.

- *Storage-aware scheduling* (see Section 5) minimizes the makespan and cost of all applications while taking into account the space constraints they

require for storing the produced data.

The rest of the paper is structured as follows. Motivated by real-world applications and real Grid testbeds, we introduce in Section 2 the application and the System models, followed by the problem definition targeted by this paper. Section 3, Section 4, and Section 5 describe in detail the performance, cost, and storage-aware algorithms validated and compared against related methods through simulated experiments, as well as real-world applications in the Austrian Grid environment [8]. Section 6 reviews the most relevant related work and Section 7 concludes the paper.

2. Model

We describe in this section the abstract application and computational resource models used in this paper, motivated by real-world applications and real Austrian Grid testbeds.

2.1. Application model

We focus on large-scale applications characterized by a high number (thousands to millions) of *homogeneous parallel* (independent) *activities* that dominate their performance, interconnected through simple control and data flow dependency constructs. The next section will give a few real-world examples.

Definition 1. *Let* $\mathcal{AP} = (\mathcal{CS}, \mathcal{DD})$ *denote a* large-scale application *modeled as a directed acyclic graph, where* $\mathcal{CS} = \bigcup_{k=1}^{A} \mathcal{AC}_k$ *is the set of* A activities classes *and* $\mathcal{DD} = \left(\mathcal{AC}_s <^d \mathcal{AC}_d | \{\mathcal{AC}_s, \mathcal{AC}_d,\} \subset \mathcal{CS}\right)$ *is the set of data flow dependencies. We call* $\mathcal{AC}_s$ *the* predecessor *of* $\mathcal{AC}_d$ *and write:* $\mathcal{AC}_s = pred\,(\mathcal{AC}_d)$. *We define an* activity class $\mathcal{AC}_k$ *as a set of parallel independent activities* $\mathcal{AC}_k = \bigcup_{j=1}^{A_k} \mathtt{a}_j^{(k)}, k \in [1..A]$ *that have the same* activity type *and can be concurrently executed, where* A_k *is the* cardinality *of the activity class. The term activity type refers to an abstract functional description of activities. We call* atomic *or* sequential activity *an activity class of cardinality one.*

Examples of activity types are matrix multiplication, Fast Fourier Transform, or `poten`, `pgroups`, `lapw1`, and `lapw2` for our real-world pilot applications.

We further assume the availability of an *expected time to compute (ETC)* [9] matrix which delivers the *expected execution time* $p_i^{(k)}$ of activities in each class $k \in [1..A]$ on each resource $\mathtt{s}_i, i \in [1..M]$, including staging of the required input data. We obtain the ETC matrix using an own performance prediction service [10, 11] based on machine learning and similarity methods in that we proved in previous research to deliver accurate results for our targeted application classes.

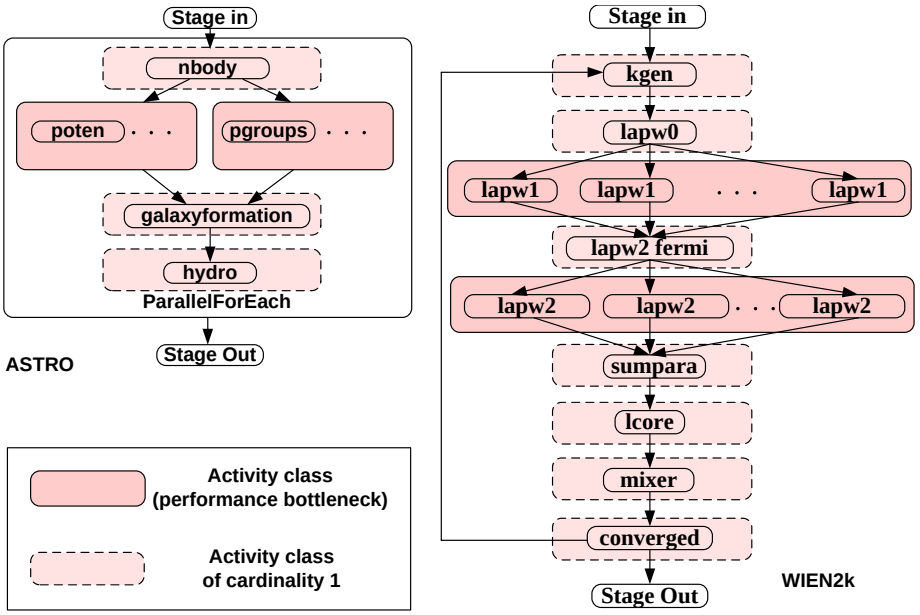


Figure 1: Real-world large-scale application examples.

2.2. Example of real-world applications

Our formal application model is motivated by several real-world applications from the astronomy, graphics rendering, hydrology, meteorology, theoretical chemistry domains that we encountered in our previous interdisciplinary research (see [12, 13, 14, 15]). In the following, we present two case studies that we use in this paper as as pilots to validate our generic methods: WIEN2k from theoretical chemistry and ASTRO from astronomy domains.

ASTRO [16] is an astronomical application that solves numerical simulations of the movements and interactions of galaxy clusters using an N-Body system. The computation starts with the state of the universe at some time in the past and is done until the current time. Galaxy potentials are computed for each time step. Finally, the hydrodynamic behavior and processes are calculated.

WIEN2k [17] is a program package for performing electronic structure calculations of solids using density functional theory based on the full-potential (linearized) augmented plane-wave ((L)APW) and local orbital method. We have ported this application onto distributed resources by splitting the monolithic code into several course-grain activities coordinated in a simple workflow illustrated in Figure 1. The `lapw1` and `lapw2` activity classes can be solved in parallel by a fixed number of homogeneous activities called k-points. A final activity named converged applied on several output files tests whether the problem convergence criterion is fulfilled.

The sources of performance bottlenecks in these applications are large sets of homogeneous activities such as `lapw1` and `lapw2` in WIEN2k, or `poten` and

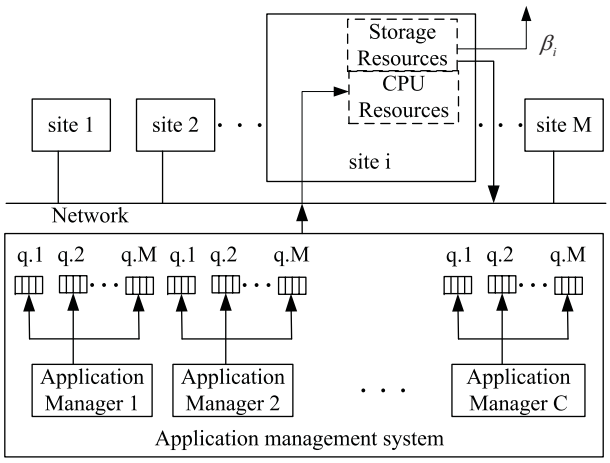


Figure 2: Computational resource model overview.

`pgroups` in ASTRO. The number of grid cells (i.e. number of `pgroups` and `poten` activities) of a real simulation in ASTRO is 128^3, while the number of `lapw1` and `lapw2` parallel activities in WIEN2k may be of tens of thousand for a good density of states. Currently, most related work only considers applications with tens or hundreds of activities, which are an order of magnitude lower than the size of our applications. Sequential activities are relatively trivial in large-scale applications and can be served and scheduled on-demand on the fastest or cheapest available processor.

2.3. System model

We define an abstract system model based on the characteristics of the Austrian Grid [8] which is the infrastructure in which we carry out our research. Computational resources consist of a set of *sites* connected to Internet, where each site is a homogeneous parallel computer. Access to each site is performed through a local job management (or queuing) system [6] administered using local policies. Activities or jobs arriving at each site may belong to multiple applications. The execution of each application is controlled by one *application manager* which competes with the other application managers for resources (see Figure 2). Most other scheduling approaches in the related work assume direct mapping of user jobs or activities to individual processors which we consider inappropriate for distributed computing where sites are exclusively managed by locally administered queuing systems. To support this more realistic model, the application manager maintains locally one queue for each site in order to schedule and limit the number of job submissions based on the site's activity processing rate. From the local queue, the jobs are submitted by the application managers to the gatekeepers of the remote computational resources such as the Grid Resource Allocation Manager (GRAM) of Globus toolkit [18].

Notation	*Semantics*
N	Total number of activities in the application
A	Number of activity classes
M	Number of computational reosurce sites
$\delta_i^{(k)}$	Number of activities of activity class $\mathcal{AC}_k$ scheduled on site $\mathtt{s}_i$
$\delta^{(k)}$	Number of activities in the activity class $\mathcal{AC}_k$ $\left(\delta^{(k)} = \sum_{i=1}^{M} \delta_i^{(k)}\right)$
$p_i^{(k)}$	Expected execution time of activity class $\mathcal{AC}_k$ on site $\mathtt{s}_i$
m_i	Total number of available processors on site $\mathtt{s}_i$
$\beta_i^{(k)}$	Job processing rate of activity class $\mathcal{AC}_k$ on site $\mathtt{s}_i$ $\left(\beta_i^{(k)} = \frac{\theta_i^{(k)}}{p_i^{(k)}}\right)$
$\theta_i^{(k)}$	Number of available processors for activity class $\mathcal{AC}_k$ on site $\mathtt{s}_i$
$\beta^{(k)}$	Job processing rate of activity class $\mathcal{AC}_k$ $\left(\beta^{(k)} = \sum_{i=1}^{M} \beta_i^{(k)}\right)$
$t_i^{(k)}$	Remaining execution time of activity class $\mathcal{AC}_k$ on site $\mathtt{s}_i$ $\left(t_i^{(k)} = \frac{\delta_i^{(k)}}{\beta_i^{(k)}}\right)$
$t^{(k)}$	Remaining execution time of activity class $\mathcal{AC}_k$ $\left(t^{(k)} = \max\left\{t_1^{(k)}, t_2^{(k)}, \ldots, t_M^{(k)}\right\}\right)$
c_k	Cost of executing activity class $\mathcal{AC}_k$
sl_i	Storage limit on site $\mathtt{s}_i$
$sr^{(k)}$	Storage requirement of activity class $\mathcal{AC}_k$
φ_i	Price per time unit of site $\mathtt{s}_i$

Table 1: Notation summary.

2.4. Problem statement

Our goal is to design new algorithms for scheduling a set of large-scale applications defined according to Definition 1 and consisting of a huge number of activities (for which existing algorithms do not scale) in a computational environment modeled in Section 2.3. Our algorithms aim to optimize two objective functions: aggregated makespan (see Section 3) and aggregated cost (see Section 4), and optionally fulfill storage constraints (see Section 5).

To facilitate the read, Table 1 summarizes the most important notations defined in the remainder of this paper.

3. Makespan scheduling

In this section we first formally formulate the makespan scheduling problem for the special class of large-scale applications introduced in Section 2.1 (see Definition 1) and then propose and experimentally validate a game theory-based algorithm to efficiently solve it.

Definition 2. *Suppose we have a set of n large-scale applications consisting of activities that can be categorized into A different activity classes, and a distributed computing environment consisting of M sites. The* makespan *C_i of an application $\mathcal{A}_i, i \in [1..n]$ is the maximum completion time of its activity classes. The objective of the* makespan scheduling *problem is to find a solution that assigns all activities to the computational resource sites such that the maximum makespan of all applications $\max_{i \in [1..n]} \{C_i\}$ is minimized.*

Makespan scheduling can be formulated as a cooperative game among the application managers which can theoretically generate the optimal solution,

although this is hard to achieve due to the problem's high complexity. We therefore observe that the problem can be further formulated and solved as a *sequential cooperative game* that we present in the following.

3.1. Formulation

The specification of a game in game theory requires the proper definition of three important parameters: the players, the strategies, and the payoff.

We consider a *A-player cooperative game* in which each of the A application managers (or *players*) attempts at a certain time instance to minimize the execution time $t^{(k)}$ of one activity class $\mathcal{AC}_k$ based on its total *number of activities* $\delta^{(k)}$ and its *processing rate* $\beta_i^{(k)}$ on each site $\mathtt{s}_i$. For clarity, we assume that each application manager handles the execution of one activity class. The objective of each manager is to minimize the execution time of its activity class which can be expressed as:

$$f_k(\Delta) = t^{(k)}(\Delta) = \frac{\delta^{(k)}}{\beta^{(k)}} = \frac{\delta^{(k)}}{\sum_{i=1}^{M} \frac{\theta_i^{(k)}}{p_i^{(k)}}}, k \in [1..A], \tag{1}$$

where Δ is an *activity distribution matrix* $\left(\delta_i^{(k)}\right)_{A \times M}$ representing *strategies* and $\theta_i^{(k)}$ is the number of processors that are allocated to activity class $\mathcal{AC}_k$ on site $\mathtt{s}_i$, which is the embodiment of *payoff* in the cooperative game. The term $\theta_i^{(k)}$ represents the *resource allocation* of activity class k on site $\mathtt{s}_i$ defined as the product between the number of processors m_i on $\mathtt{s}_i$ and the ratio between the weighted aggregated execution times of activity class $\mathcal{AC}_k$ on $\mathtt{s}_i$ and the aggregated execution time of all activity classes on $\mathtt{s}_i$:

$$\theta_i^{(k)} = m_i \cdot \frac{\delta_i^{(k)} \cdot p_i^{(k)} \cdot w_i^{(k)}}{\sum_{x=1}^{A} \delta_i^{(x)} \cdot p_i^{(x)} \cdot w_i^{(x)}}, \tag{2}$$

where $w_i^{(k)}$ is the weight of site $\mathtt{s}_i$ for activity class $\mathcal{AC}_k$:

$$w_i^{(k)} = \frac{\frac{\min\limits_{x \in [1..S]} \left\{p_x^{(k)}\right\}}{p_i^{(k)}}}{\sum_{y=1}^{M} \frac{\min\limits_{x \in [1..S]} \left\{p_x^{(k)}\right\}}{p_y^{(k)}}} = \frac{\frac{1}{p_i^{(k)}}}{\sum_{y=1}^{M} \frac{1}{p_y^{(k)}}}. \tag{3}$$

We use the weight $w_i^{(k)}$ to enhance the fairness of allocation because one site may have different suitability for different activities for various reasons such as the locality of data, the size of memory, the CPU frequency, or the I/O speed. Intuitively, if the execution time $t_i^{(k)}$ on site $\mathtt{s}_i$ for activity class $\mathcal{AC}_k$ is much shorter than on other sites, we set a higher priority for this activity class on this site and allocate more resources to it. The specific utilization of this weight will be explained when we introduce the notion of sequential game.

When the ideal load balance of activity class $\mathcal{AC}_k$ is achieved (the remaining execution time on each site is equal) the objective function can be defined as:

$$f_k(\Delta) = t^{(k)}(\Delta) = \begin{cases} \frac{p_i^{(k)} \cdot \delta_i^{(k)}}{\theta_i^{(k)}}, & \theta_i^{(k)} \geq 1; \\ 0, & \theta_i^{(k)} < 1. \end{cases} \tag{4}$$

Based on the allocation of resources and the ratio of processing rate on site $\mathtt{s}_i$ to the total processing rate of the activity class, we define the *activity distribution* as the product between the number of activities in $\mathcal{AC}_k$ and the ratio between the processing rate of $\mathcal{AC}_k$ on site $\mathtt{s}_i$ with respect to the total processing rate of $\mathcal{AC}_k$:

$$\delta_i^{(k)} = \delta^{(k)} \cdot \frac{\beta_i^{(k)}}{\beta^{(k)}} = \delta^{(k)} \cdot \frac{\frac{\theta_i^{(k)}}{p_i^{(k)}}}{\sum_{i=1}^{M} \frac{\theta_i^{(k)}}{p_i^{(k)}}}. \tag{5}$$

Accordingly, we have the following definition.

Definition 3. *A* makespan optimization cooperative game *consists of (see Table 1 for notations):*

- *Managers of A activity classes as players;*
- *A set of strategies Δ defined by the following set of constraints: (1) $\delta_i^{(k)} \geq 0$, (2) $\theta_i^{(k)} \leq \delta_i^{(k)}$, (3) $\delta_i^{(k)} = 0$, if $\theta_i^{(k)} < 1$, (4) $\sum_{k=1}^{A} \theta_i^{(k)} = m_i$, and (5) $\sum_{i=1}^{M} \delta_i^{(k)} = \delta^{(k)}$;*
- *For each player $k \in [1..A]$, the objective function $f_k(\Delta)$. The goal is to minimize simultaneously all $f_k(\Delta)$;*
- *For each player $k \in [1..A]$, the initial value of the objective function $f_k\left(\Delta^0\right)$, where $\Delta^0 = \left(\delta_i^{(k)}\right)^0_{A \times M}$ is a $A \times M$ matrix filled with initial distribution of activities. Δ^* denotes the optimal solution of the game.*

For our class of large-scale applications, the maximum makespan is almost equal to the aggregated makespan divided by the number of processors, therefore, the goal of our cooperative optimization game can be approximated to minimizing the aggregated makespan: $\min\left\{\sum_{k=1}^{A} f_k(\Delta)\right\}$*, subject to the constraints (1)–(5).*

Unfortunately, the exact solution to this problem which is also optimal is in general difficult to obtain. Because the problem has high complexity and $A \cdot M$ variables, the solution depends on the distribution of activities in the same class on different sites, and the distribution of activities in different classes on the same site. In other words, the change of one variable impacts the values of all other variables. To circumvent this difficulty, we approximate solution by further formulating this problem as *a sequential game* [19] in which players

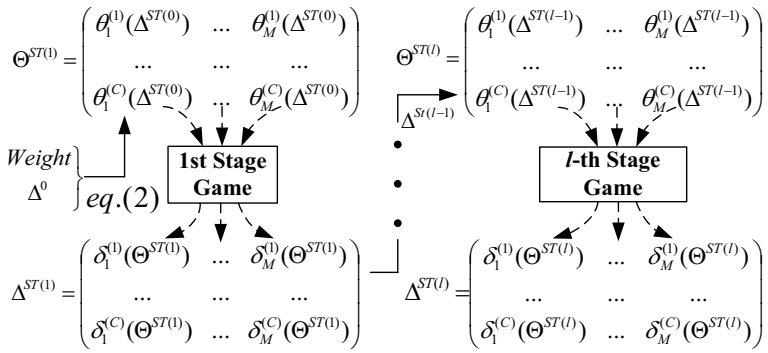


Figure 3: Sequential game theory-based allocation data flow.

select a strategy following a certain predefined order and observe the moves of the players who preceded them. Although the optimal solution is not directly achievable, we derive intermediate solutions in a set of *game stages*, based on the following inequality sequence:

$$\sum_{k=1}^{A} f_k^{ST(1)}\left(\Delta^{ST(0)}\right) \geq \sum_{k=1}^{A} f_k^{ST(2)}\left(\Delta^{ST(1)}\right) \geq \ldots \geq \sum_{k=1}^{A} f_k^{ST(l)}\left(\Delta^{ST(l-1)}\right) \geq \sum_{k=1}^{A} f_k\left(\Delta^{*}\right), \tag{6}$$

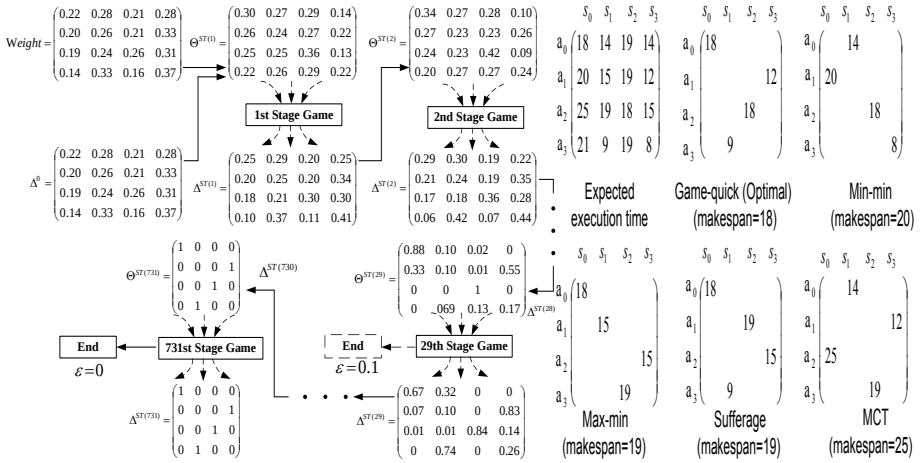
where ST denotes the stage of the sequential game, and $ST(l)$ the l^{th} game stage. At each stage, the players (managers of activity classes) provide a set of strategies (activity distributions) based on the allocation of resources in the last stage, and generate the new allocations by using Equation 2.

The first step in the sequential game is to initialize the distribution of activities $\Delta^{ST(0)}$. Every activity class is allocated a set of processors based on the processing rate on each site. At the initial stage $ST(0)$, every activity class assumes that all processors are available to it:

$$\delta_i^{(k)} = \delta^{(k)} \cdot \frac{\beta_i^{(k)}}{\beta^{(k)}} = \delta^{(k)} \cdot \frac{\frac{m_i}{p_i^{(k)}}}{\sum_{y=1}^{M} \frac{m_y}{p_y^{(k)}}}. \tag{7}$$

From Equation 7, we have the initial activity distribution $\Delta^{ST(0)}$.

The resource allocation of the l^{th} stage $\Theta^{ST(l)}$, where $\Theta = \left(\theta_i^{(k)}\right)_{A\times M}$ is the *resource allocation matrix*, is calculated based on the activity distribution of the last stage $\Delta^{ST(l-1)}$. Accordingly, the activity distribution of the l^{th} stage $\Delta^{ST(l)}$ is calculated based on the resource allocation of l^{th} stage $\Theta^{ST(l)}$. These steps fully embody the idea of a sequential cooperative game. From Equation 2



(a) Game-quick stages.

(b) Related algorithms schedule comparison.

Figure 4: Game-quick makespan scheduling example.

and Equation 5, we can derive as also shown in Figure 3:

$$\Theta^{ST(l)} = \Theta\left(\Delta^{ST(l-1)}\right); \tag{8}$$

$$\Delta^{ST(l)} = \Delta\left(\Theta^{ST(l)}\right). \tag{9}$$

3.2. Game-quick algorithm

In this section we present an algorithm called *Game-quick* which implements the game theoretical makespan scheduling method formalized in the previous section. We accompany our presentation by a small example depicted in Figure 4a. The first ETC matrix presents the expected execution times of four activities $\{\mathtt{a}_0, \mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3\}$ on four sites $\{\mathtt{s}_0, \mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3\}$. For the sake of clarity, we restrict in this example the size of each site to one processor only.

The Game-quick algorithm depicted in pseudocode in Algorithm 1 receives as input a set of applications $\mathcal{AS}$ consisting of a workflow of activity classes conforming to our model introduced in Section 2.1. The algorithm has an outermost sequential loop (lines 2–19) that iteratively applies the cooperative game theoretic algorithm to the activity classes ready to be executed in three phases.

Phase 1. After adding the activity classes ready to be executed to the set of players *GP* (lines 3–5), we generate an initial distribution of activities Δ^0 and a weight matrix (see lines 3–11), as shown in Figure 4a. In this case, the two matrices are identical because we only have one processor on each cluster and one activity in each class. In this phase, users are also allowed to set performance constraints or to filter undesired sites by simply setting the weights of

Algorithm 1 Game-quick makespan scheduling algorithm.

Require: $\mathcal{AS}$: set of applications; A: number of activity classes; M = number of sites; m_i: number of processors on site $\mathbf{s}_i (i \in [1..m])$; $\left(p_i^{(k)}\right)_{A \times M}$: ETC matrix; $\delta^{(k)}$: number of activities of class $k (k \in [1..A])$; ϵ: optimization threshold;

Require: sl_i: storage limit of site $\mathbf{s}_i (i \in [1..M])$; $sr^{(k)}$: storage requirements of activity class $\mathcal{AC}_k (k \in [1..A])$; // *Optional parameters for the Game-storage algorithm (see Algorithm 3)*

Ensure: $\Delta^{ST(l)}$: activity distribution matrix; $\Theta^{ST(l)}$: resource allocation matrix

1: $GP \leftarrow \emptyset$ // *initialize the set of game players*
2: **repeat**
3: **for all** $\mathcal{AP} \in \mathcal{AS}$ **do** // ***Phase 1:*** *Initialize* Δ^0 *and the weight of activity classes; optionally apply constraints*
4: **for all** $\mathcal{AC}_k \in \mathcal{AP} \wedge \mathcal{AC}_k$ not yet scheduled $\wedge$ (pred $(\mathcal{AC}_k) = \emptyset \vee (\mathcal{AC}_j$ is completed, $\forall \mathcal{AC}_j \in$ pred $(\mathcal{AC}_k))$) **do** // *Take the next not scheduled activity class*
5: $GP \leftarrow GP \cup \mathcal{AC}_k$// *Add* $\mathcal{AC}_k$ *to the set of game players*
6: **for all** $i \in [1..M]$ **do** // *For every site* $\mathbf{s}_i$
7: Calculate $w_i^{(k)}$ by applying Equation 3
8: Calculate $\delta_i^{(k)}$ by applying Equation 5 to build Δ^0
9: **end for**
10: **end for**
11: **end for**
12: **repeat** // ***Phase 2:*** *search the final distribution of activities and allocation of resources*
13: Calculate $\Theta^{ST(l)} = \left(\theta_i^{(k)}\right)_{|GP| \times M}$ by applying Equation 8
14: $\Theta \leftarrow$ GAME-STORAGE$(\Theta, \Delta, m, sr, sl, p)$ // *Optionally apply storage constraints by calling Algorithm 3*
15: Calculate $\Delta^{ST(l)} = \left(\delta_i^{(k)}\right)_{|GP| \times M}$ by applying Equation 9
16: **until** $\sum_{k=1}^{|GP|} \left(t^{(k)}\left(\Delta^{ST(l-1)}\right) - t^{(k)}\left(\Delta^{ST(l)}\right)\right) \leq \epsilon$
17: **wait** for an activity class to complete
18: $GP \leftarrow GP - \mathcal{AC}, \forall \mathcal{AC} \in GP \wedge \mathcal{AC}$ completed // ***Phase 3:*** *remove completed activity classes and repeat Phases 1-2*
19: **until** $\forall \mathcal{AP} \in \mathcal{AS}$ completed

the applications for these sites to zero which prevents mapping of any activities to those sites. To assure that all constraints are satisfied, they can be verified again in the third phase.

Phase 2. Every iteration of the `repeat` loop (lines 12–16) is one game stage, where every stage consists of M *sub-games* (i.e. one per site). In each sub-game, all activity classes compete for resource allocation and those with relatively large weights win the sub-game on one site and obtain more resources in the next stage. These activity classes, however, cannot win everywhere due to the weight definition (i.e. the weight sum of one activity class is 1), therefore, winners of the sub-game on one site must be losers on other sites and vice-versa. This process repeats until no more performance can be gained. The further processing of the algorithm depends on the evaluation result at line 16: $\sum_{k=1}^{A} \left(t^{(k)}\left(\Delta^{ST(n-1)}\right) - t^{(k)}\left(\Delta^{ST(n)}\right)\right) > \epsilon$, where ϵ can be used to control the number of stages and the degree of optimization. The input and output data flow of each game stage has been shown in Figure 3. More specifically, we apply Equation 8 at line 13 to generate the first resource allocation matrix $\Theta^{ST(1)}$ (see also Figure 4a). Based on $\Theta^{ST(1)}$, we use Equation 9 at line 15 to generate the first activity distribution $\Delta^{ST(1)}$. Thereafter, we repeat the iteration until we reach the upper limit of optimization. In addition, we can use ϵ to control the number of stages.

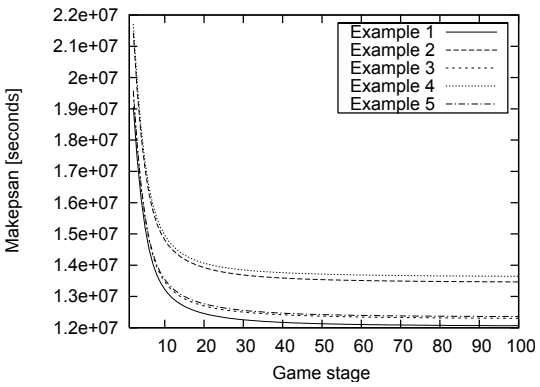


Figure 5: Game-quick convergence.

Phase 3. Finally, the earliest completed activity classes are eliminated. To utilize the released resources by the completed activity classes, we repeat the first two phases to recompute the distribution of the remaining classes until all applications are completed.

3.3. Convergence

The Game-quick convergence is very fast, as shown in Figure 5. For this experiment, we randomly generated five examples that assign $10^2 \times 10^4$ activities to $10^2 \times 10^2$ processors. Within $30 - 40$ stages, more than 90% of them complete the optimization process (of 600 stages in total). The reason for the fast convergence is that, to some extent, every activity class is a winner on certain sites and can achieve a performance improvement. At the beginning of a game, every activity class moves its workload to the sites that are more efficient for them and bargain for resources. If they cannot successfully bargain and get more resources, they move the workloads to less powerful sites. Finally, all activity classes reach a balance when no further improvement can be achieved. The difference in the makespan is explained by the different problem sizes generated.

3.4. Comparison with related algorithms

Figure 4a presents a small example in which Game-quick outperforms six well-known list scheduling heuristics designed for makespan minimization of N activities onto n processors (see Figure 4b), categorized in two classes:

$\mathcal{O}(n \cdot N)$ *complexity algorithms.* [20, 21] iterate once over the list of activities and schedules them as follows: *Minimum Execution Time (MET)* assigns each activity to the machine that delivers its minimum execution time; *Minimum Completion Time (MCT)* assigns each activity to the machine that delivers its minimum completion time; *Opportunistic Load Balancing (OLB)* minimizes the global load imbalance without considering the activity execution time.

CPUs	*Inconsistent*					*Consistent*				
	$\mathcal{AC}_1$	$\mathcal{AC}_2$	*Game-quick*	*Min-min*	*Max-min*	$\mathcal{AC}_1$	$\mathcal{AC}_2$	*Game-quick*	*Min-min*	*Max-min*
$S_1 : P_1$	15	8	8, 9	8, 15	15, 8	15	9	9, 9	9, 15	15, 9
$S_1 : P_2$	15	8	8, 9	8, 15	15, 8	15	9	9, 9	9, 15	15, 9
$S_1 : P_3$	15	8	8, 15	8, 15	15, 8	10	8	10, 10	8, 10	10, 8
$S_1 : P_1$	10	9	10, 10	9, 10	10, 9	10	8	10, 10	8, 10	10, 8
$S_1 : P_2$	10	9	10, 10	9, 10	10, 9	10	8	10, 9	8, 10	10, 8
Makespan			20	23	23			19	24	24

Table 2: Game-quick, Min-min, and Max-min comparison for small-sized ETC matrices.

$\mathcal{O}(n \cdot N^2)$ *complexity algorithms.* [22, 23] iterate over all activities before selecting one for scheduling according to the following criteria: *Min-min* selects the activity with the shortest minimum completion time; *Max-min* selects the activity with the largest minimum completion time; *Sufferage* selects the activity with the largest difference between the fastest and second fastest minimum completion times. The selected activity is assigned to the machine that delivers its earliest completion time.

For our example in Figure 4b, Game-quick gives a makespan of 18 which is also optimal, Min-min gives a makespan of 20, Max-min and Sufferage give a makespan of 19, and MCT performs the worst and gives a makespan of 25. MET assigns all tasks to $\mathtt{s}_3$ and gives the worse makespan of 49, hence, we do not show its mapping. Figure 4a presents the intermediate data generated by Game-quick for this scenario. The algorithm completes at stage 29 if $\epsilon = 0.1$ and at stage 731 if $\epsilon = 0$. For the experiments in this study, we set ϵ to zero.

To further compare the quality of the Game-quick solutions against the absolute optimum, we consider a small-sized problem consisting of two sites with three respectively two homogeneous processors each, and two activity classes containing five activities each. We use in our simulation both consistent matrices and inconsistent matrices, outlined in Table 2. The solution delivered by Game-quick is optimal in both cases, while Min-min and min-max deliver equal results for inconsistent matrices and worse results for the consistent cases. However, as we will demonstrate in the following sections, Min-min and Max-min require significantly longer scheduling times, especially in the case of large problems.

3.5. Complexity

The time complexity of Game-quick is $O(l \cdot A \cdot M)$ corresponding to the three algorithm nested loops (lines 2, 3, and 6 in the first phase of Algorithm 1), where l is the number of stages of the sequential game, A is the number of activity classes, and M is the number of sites, respectively (the work performed within each stage is constant). The second phase does not have an impact on the complexity, since the work performed in each stage is constant and depends linearly on the number of sites and on a small number of game players only. Most importantly, the complexity is independent of the total number of activities, which is a huge advantage against other related approaches.

For empirical evidence, Table 3a displays the number of stages and the execution time of Game-quick for different computational resource and application

Sites × *No. proc.*	*Classes* × *Activities/class*	*No. Stages*	*Game-quick* [*millisec.*]	*Min-Min* [*millisec.*]
10×10	10×10	310	2	15
10×10	10×100	334	2	22
10×10^2	$10^2 \times 10^3$	476	23	3.109
10×10^2	$10^2 \times 10^4$	484	25	29.512
$10^2 \times 10^2$	$10^2 \times 10^3$	597	362	485.597
$10^2 \times 10^2$	$10^2 \times 10^4$	593	362	$>$ 1 hour
$10^2 \times 10^2$	$10^3 \times 10^3$	632	11.065	$>$ 1 hour
$10^2 \times 10^2$	$10^3 \times 10^4$	627	11.856	$>$ 1 hour

(a) Game-quick stages and execution times.

Algorithm	*Time complexity*	*Exec. time* [*seconds*]
Game-quick	$O(l \cdot A \cdot M)$	< 0.4
MET	$O(m + N)$	< 1
OLB, MCT	$O(m \cdot N)$	$2 - 3$
Min-min (*et al.*)	$O\left(m \cdot N^2\right)$	$200 - 300$
GA-based	poor	$\gg 200 - 300$
A^*	exponential	$\gg 200 - 300$

(b) 10^5 activities and 10^3 processors.

Table 3: Algorithm complexity and execution time analysis.

sizes on a Dual Core Opteron 880 2.4 gigahertz processors and 16 gigabytes of memory. We can observe that Game-quick scales well with the computational resource size, since the number of game stages does not increase as fast as the number of processors and activities. Even for 10^6 activities and 10^4 processors, the algorithm only needs 593 stages and 0.36 seconds to complete the optimization, in contrast to the algorithms from the Min-min family which need more than one hour. Based on this empirical analysis, we use for Game-quick a maximum number of 1000 iterations that proves to be sufficient for convergence.

Table 3b shows more empirical results comparing Game-quick with MET, MCT, OLB, Min-min, Max-min, and Sufferage for scheduling 10^5 activities to 10^3 processors. The execution time of Game-quick is less than 0.3 seconds, while other algorithms might need several hours to generate comparable solutions. The exception is MET which has asymptotic complexity of $O(m + N)$, where m is the number of processors and N is the number of activities ($N \gg A$), and executes for less than one second. However, the results of MET have serious problems because it serializes most activities on the fastest site. OLB and MCT have asymptotic complexity of $O(m \cdot N)$, but their results are much worse than of Game-quick. Min-min, Max-min, and Sufferage have asymptotic complexity of $O\left(m \cdot N^2\right)$ and execute for an average of $200 - 300$ seconds.

Other algorithms such as the Heterogeneous Earliest Finish Time (HEFT) [24] algorithm degrade to Min-min for large-scale and simple-dependency applications. Genetic algorithm (GA)-based [9, 25] or A* [26] solutions scale poorly as the number of activities and processors increases, and their execution times are significantly higher than other algorithms (although they can decrease the makespans of Min-min by 5% to 10%, according to related work [9]). Other algorithms are similar to the ones discussed above or are not applicable to our problem, for example the Work Queue [27] suited for homogeneous clusters.

3.6. Experiments

In this section we evaluate through simulation the performance of Game-quick against related algorithms for different ETC matrices generated according to different degrees of machine and activity heterogeneity parameters selected from a uniform distribution in the specified ranges. Table 4 presents the details of the simulated environment. High machine heterogeneity in the range of

Config.	*No. Procs.*	*No. Clusters*	*No. Activ.*	*Activity Classes*	*Activity Heterog.*	*Machine Heterog.*
HiHi	900	10	157118	10	[1, 1000]	[1, 100]
HiLo	989	10	147871	10	[1, 1000]	[1, 10]
LoHi	900	10	149731	10	[1, 10]	[1, 100]
LoLo	1048	10	168208	10	[1, 10]	[1, 10]

(a) Consistent environment.

Config.	*No. Procs.*	*No. Clusters*	*No. Activ.*	*Activity Classes*	*Activity Heterog.*	*Machine Heterog.*
HiHi	982	10	131298	10	[1, 1000]	[1, 100]
HiLo	955	10	153395	10	[1, 1000]	[1, 10]
LoHi	955	10	173418	10	[1, 10]	[1, 100]
LoLo	1007	10	150156	10	[1, 10]	[1, 10]

(b) Inconsistent environment.

Table 4: Makespan scheduling simulation environment.

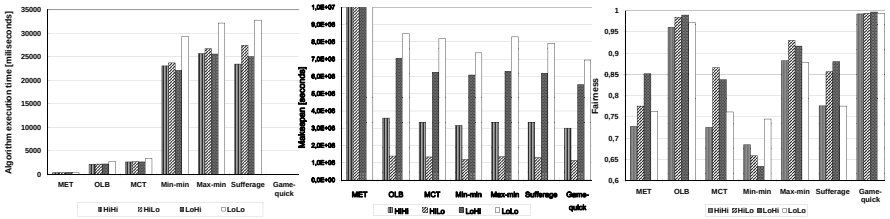
[1..100] causes a significant difference in activity execution times across sites, while high activity heterogeneity in the range of [1..1000] indicates that the expected execution times of different activities vary largely. We assume that the number of activities is randomly generated based on a uniform distribution in the range of [10000..20000], and that the number of processors on each site varies in the range of [64..128]. The activity classes are organized in workflows by having 10% dependence probability between each pair or activity classes and excluding cycles. We choose not to simulate larger application and computational resource sizes because of two reasons: (1) the complexity of the algorithms from the Min-min family that need several hours to complete making our entire simulation difficult; and (2) enlarging the simulation size will only increase the advantage of our game theoretic algorithm over the related heuristics.

We use two ETC simulation models: (1) *consistent* represents that, if a site A executes an activity faster than site B, then A executes all activities faster than B; and (2) *inconsistent* when the site A might be faster than B for some activities and slower for some others [9]. We evaluate our algorithm in four scenarios: high activity and high resource heterogeneity (HiHi), high activity and low resource heterogeneity (HiLo), low activity and high resource heterogeneity (LoHi), and low activity and low resource heterogeneity (LoLo).

Figure 6a shows the execution times of the algorithms which do not vary for consistent and inconsistent matrices, ranked from the fastest to the slowest as follows: Game-quick, MET, OLB, MCT, Min-min, Max-min, and Sufferage. In the next subsections we discuss the performance of all algorithms from the worst to the best in the consistent and inconsistent scenarios. To quantify the fairness of the algorithms, we use the *Jain's fairness index* [28]:

$$\frac{\left(\sum_{i=1}^{|\mathcal{AS}|} T_i\right)^2}{|\mathcal{AS}| \cdot \sum_{i=1}^{|\mathcal{AS}|} T_i^2}, \tag{10}$$

where $|\mathcal{AS}|$ is the number of applications and T_i is the execution time of appli-



(a) Algorithm execution time. (b) Makespan. (c) Fairness.

Figure 6: Game-quick scheduling results for consistent scenarios, 10^5 activities and 10^3 processors.

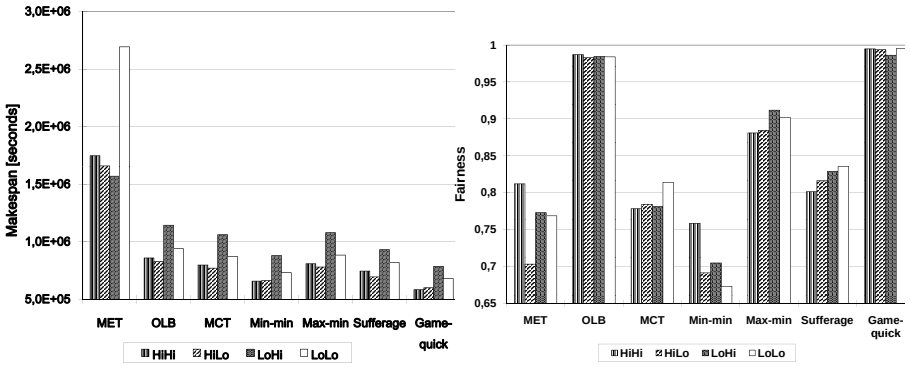
cation $\mathcal{AP}_i$. The fairness ranges from zero indicating the worst fairness, to one indicating the best fairness.

3.6.1. Consistent heterogeneity

Table 4a presents the input of four consistent heterogeneous scenarios and Figures 6b and 6c the corresponding simulation results considering the makespan and the fairness objectives. MET always gives the worst results because it maps all activities to the fastest machine. OLB usually performs the second worst because it selects resources without considering the activity execution time. Max-min gives poor results because it only fits the situation when a small number of activities are much larger than the others, which is never encountered in our simulated environment generated using uniform distributions. In addition, Max-min offers no fairness to smaller activities, hence, it performs worse than most algorithms. MCT performs quite well for high machine heterogeneity scenarios because it has a higher likelihood for selecting the fastest machine, especially for large activities, and poorly for low machine heterogeneity scenarios because it only considers the completion time and ignores the activity execution time. Sufferage performs quite similar to MCT for high machine heterogeneity and $5\% - 10\%$ better for low machine heterogeneity scenarios because it makes more intelligent decisions by considering the activity execution time. Min-min gives the second best results in each case due to the uniform distribution of activity execution times, but looses fairness because of handling the smallest activities first. Game-quick provides the best performance in all scenarios because it takes the best global decisions. It performs about 10% better than Min-min for the LoHi scenario, and 5% better for the other three. We can see that when fairness is ensured, the efficiency is also improved. We can further observe in Figure 6c that Game-quick always achieved almost perfect fairness of 0.99 in average.

3.6.2. Inconsistent heterogeneity

Table 4b presents the input of the four inconsistent scenarios investigated and Figure 7 depicts the results. In all four cases, MET gives the worst results because it maps most activities to the few fastest sites. MET could perform better than OLB when the fastest sites for different activity classes are evenly



(a) Makespan. (b) Fairness.

Figure 7: Game-quick scheduling results for inconsistent matrices.

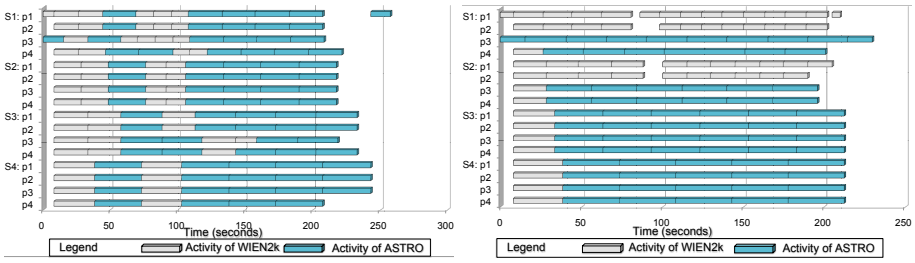
distributed, however, this special case rarely occurs. OLB, Max-min, MCT, and Sufferage perform worse for inconsistent than for consistent scenarios due to their design that cannot effectively handle the high heterogeneity of machines in inconsistent environments. In contrast, Min-min performs better for inconsistent than for consistent scenarios because the fastest machines are allocated evenly. Thus, Min-min is able to assign more activities to the fastest machines, though it also does not intentionally handle the change of the environment. Game-quick still provides the best mapping for the inconsistent cases for the same reason as for consistent scenarios.

3.7. Real-world experiments

In this section we report on the evaluation of the Game-quick algorithm for the WIEN2k and ASTRO scientific applications introduced in Section 2.2 and executed in the Austrian Grid. Our experimental testbed depicted in Table 5 consists of four parallel machines located at the University of Innsbruck, Salzburg, and Linz. We first evaluated the performance of Game-quick by comparing the makespan and the fairness of the computed scheduling plans against Min-min, which outperforms the other related heuristics for scheduling these two particular applications. The scheduling plans are matched by the real executions due to the proven accuracy of our prediction service [10, 11]. As shown in Figure 8a, Min-min gives a makespan of 258 and a fairness of 0.9466 which were improved by Game-quick by 12.17% and 5.42%, respectively (see Figure 8b). The fairness of Game-quick is almost perfect (0.9979). Finally, we can intuitively observe that the activities are highly interleaved in the Gantt chart produced by Min-min, which makes their completion time hard to predict. Contrarily, Game-quick yields an execution plan in which activities belonging to the same class are grouped in contiguous slots on the same sites which makes their execution more predictable.

Site	*Name*	*Size*	*Architecture*	*CPUs*	*Clock* [GHz]	*Resource Manager*	*Price* [units/hour]	*Location*
S_1	karwendel	Cluster	Operton 880	4(52)	2.4	SGE	4	Innsbruck
S_2	hc-ma	COW	Opteron	4(8)	2.2	SGE	2	Innsbruck
S_3	schafberg	Cluster	Itanium 2	4(16)	1.6	Fork	2	Salzburg
S_4	altix1	ccNUMA	Itanium 2	4(64)	1.6	PBS	1	Linz

Table 5: The Austrian Grid testbed.



(a) Min-min. (b) Game-quick.

Figure 8: Makespan scheduling for two real applications.

4. Cost scheduling

In this section, we apply the same cooperative game theoretic principles for economic cost scheduling, while guaranteeing a deadline.

Definition 4. *Suppose we have multiple large-scale applications consisting of a set of activities which can be categorized into A classes defined according to Definition 1. Suppose we have a set of M sites where each site has m_i homogeneous processors and a* price *per time unit $\varphi_i, i \in [1..M]$. The objective of the* cost scheduling *problem is to find a schedule that assigns the activities to sites such that the total cost $\sum_{i=1}^{M} \sum_{k=1}^{A} t_i^{(k)} \cdot \varphi_i$ is minimized, where $t_i^{(k)}$ is the* remaining execution time *of activity class $\mathcal{AC}_k$ on site $\mathtt{s}_i$ and φ_i is the price per time unit of site $\mathtt{s}_i$, and the* deadline *d_k of each activity class is guaranteed.*

To solve this problem, we introduce a cost scheduling algorithm consisting of two steps. The first step (see Section 4.1) is to assign *deadlines* to activity classes and to *partition* applications. The second step (see Section 4.3) is to apply a new cost scheduling algorithm called Game-cost to each partition, based on a similar idea as Game-quick. Similar as for the makespan scheduling, we assume the availability of an ETC matrix which delivers the expected execution time $p_i^{(k)}$ of activities in each class $k \in [1..A]$ on each site $\mathtt{s}_i, i \in [1..M]$.

4.1. Partitioning

Partitioning an application into smaller partitions and assigning them to different games are the two key steps in the design of our cost scheduling algorithm. Our partitioning method uses a static deadline assignment method called *effective deadline* [29, 30] in which the deadline of any activity is the

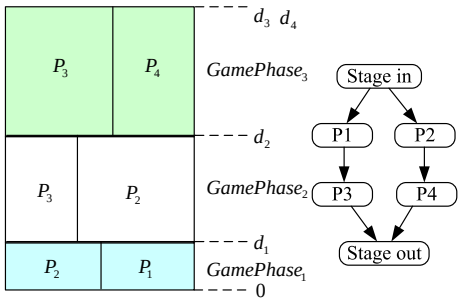


Figure 9: Sample partitioning of a cost scheduling game.

overall application deadline minus the total expected execution time of its subsequent activities. Figure 9 presents one deadline assignment and partitioning example of an application consisting of four activity classes. According to the specified deadline and work amount of each activity class, four deadlines d_1, d_2, d_3, and d_4 are assigned to the partitions P_1, P_2, P_3, and P_4 by using the effective deadline method. Thereafter, we sort the deadlines and identify *game phases* between two adjacent deadlines. In case a partition spawns across multiple phases such as P_3 in Figure 9, we split the work evenly between phases. In this example, the scheduling process is divided into three game phases, where phase one ranges between 0 and d_1, phase two between d_1 and d_2, and phase three between d_2 and d_3 (where $d_3 = d_4$). After partitioning the application, we apply our cost scheduling algorithm on each game phase independently.

4.2. Formulation

Similar to Game-quick, we model the cost scheduling problem as a *A-player cooperative game* in which A application managers as players attempt to minimize the costs of their own activity class, which depends on the number of activities in the class $\delta^{(k)}$ and their predicted execution time $p^{(k)}$, while guaranteeing a deadline. The objective function for each manager $k \in [1..A]$ is:

$$f_k(\Delta) = c_k(\Delta) = \sum_{i=1}^{M} p_i^{(k)} \cdot \delta_i^{(k)} \cdot \varphi_i, \tag{11}$$

where $\Delta = \left(\delta_i^{(k)}\right)_{A \times M}$ is the *activity distribution matrix* representing the number of activities from each class k scheduled to each site $\mathtt{s}_i$, $c_k(\Delta)$ is the *cost* of executing the activity class $\mathcal{AC}_k$ given the distribution Δ, and φ_i is the *price* per time unit of site $\mathtt{s}_i$. The distribution matrix represents the players' *strategies*.

When achieving the best price/performance ratio, the following *deadline* constraint for the activity distribution $\delta_i^{(k)}$ and resource allocation $\theta_i^{(k)}$ of activity

class $\mathcal{AC}_k$ on site $\mathtt{s}_i$ must hold:

$$d_{phase} \geq \frac{\delta_i^{(k)} \cdot p_i^{(k)}}{\theta_i^{(k)}}, \tag{12}$$

where d_{phase} is the deadline of the current phase.

The embodiment of *payoff* in our cooperative game is the *resource allocation matrix* $\Theta = \left(\theta_i^{(k)}\right)_{A \times M}$ representing the number of processors on site $\mathtt{s}_i$ allocated to each activity class $\mathcal{AC}_k$ according to following equation:

$$\theta_i^{(k)}(\Delta) = m_i \cdot \frac{\delta_i^{(k)} \cdot p_i^{(k)} \cdot cw_i^{(k)}}{\sum_{x=1}^{A} \delta_i^{(x)} \cdot p_i^{(x)} \cdot cw_i^{(x)}}, \tag{13}$$

where $cw_i^{(k)}$ is the importance *weight* of site $\mathtt{s}_i$ for activity class $\mathcal{AC}_k$:

$$cw_i^{(k)} = \frac{\frac{1}{\varphi_i \cdot p_i^{(k)}}}{\sum_{y=1}^{M} \frac{1}{\varphi_y \cdot p_y^{(k)}}}. \tag{14}$$

We use this importance weight to improve the fairness of resource allocation because one site may have preferences over a certain set of activity classes.

Definition 5. *The solution of the* cost scheduling *cooperative game is determined by solving the following minimization problem:* $\min\left\{\sum_{k=1}^{A}\left(c_k(\Delta)\right)\right\}$ *that satisfies all phase deadline constraints:*

$$\max_{\mathcal{AC}_k \in phase}\left\{t^{(k)}(\Delta)\right\} = \max_{\mathcal{AC}_k \in phase}\left\{\frac{\delta^{(k)}}{\sum_{i=1}^{M} \frac{\theta_i^{(k)}}{p_i^{(k)}}}\right\} \leq d_{phase}, \tag{15}$$

where $t^{(k)}$ *is* $\mathcal{AC}_k$*'s remaining execution time.*

Similar to Game-quick, the direct and exact solution to this optimization problem is difficult to obtain. We therefore model it as a *sequential game* based on following decreasing sequence:

$$\sum_{k=1}^{A} c_k^{ST(1)}\left(\Delta^{ST(0)}\right) \geq \sum_{k=1}^{A} c_k^{ST(2)}\left(\Delta^{ST(1)}\right) \geq \ldots \geq \sum_{k=1}^{A} c_k^{ST(l)}\left(\Delta^{ST(l-1)}\right) \geq \sum_{k=1}^{A} c_k\left(\Delta^*\right), \tag{16}$$

with the *termination condition* specified as follows:

$$\sum_{k=1}^{A} c_k^{ST(l+1)}\left(\Delta^{ST(l)}\right) \geq \sum_{k=1}^{A} c_k^{ST(l)}\left(\Delta^{ST(l-1)}\right), \tag{17}$$

meaning that Game-cost cannot reduce costs any more.

Algorithm 2 Game-cost scheduling algorithm.

Require: $\mathcal{P}$: set of partitions; A : number of activity classes; M : number of sites; m_i: number of processors on site $\mathtt{s}_i (i \in [1..m])$; $\left(p_i^{(k)}\right)_{A \times M}$: ETC matrix; $\delta^{(k)}$: number of activities of class $k (k \in [1..A])$; φ_i: price per time unit of site $\mathtt{s}_i (i \in [1..M])$; d_{phase}: deadline of current phase;
Require: sl_i: storage limit of site $\mathtt{s}_i (i \in [1..M])$; $sr^{(k)}$: storage requirements of activity class $\mathcal{AC}_k (k \in [1..A])$; // *Optional parameters for the Game-storage algorithm (see Algorithm 3)*
Ensure: $\Delta^{ST(l)}$: activity distribution matrix; $\Theta^{ST(l)}$: resource allocation matrix
Phase 1: *sort sites for each activity class by increasing performance/price ratio*
$GP \leftarrow \emptyset$
for all $\mathcal{AC}_k \in \mathcal{P}$ **do** // ***Phase 2:*** *initialize* $\Delta^{ST(0)}$ *and the weights of activity classes; optionally apply constraints*
  $GP \leftarrow GP \cup \mathcal{AC}_k$ // *Add* $\mathcal{AC}_k$ *to the set of game players*
  for all $\mathtt{s}_i \in Grid$ in sorted order **do**
    Calculate $cw_i^{(k)}$ by applying Equation 14
    $\delta_i^{(k)} \leftarrow \frac{m_i \cdot d_{phase}}{p_i^{(k)}}$

    if $\delta_i^{(k)} > \delta^{(k)} - \sum_{j=1}^{i-1} \delta_j^{(k)}$ **then** $\delta_i^{(k)} \leftarrow \delta^{(k)} - \sum_{j=1}^{i-1} \delta_j^{(k)}$
    end if
  end for
end for
repeat// ***Phase 3:*** *search the final distribution of activities and the allocation of resources*
  Calculate $\Theta^{ST(l)} = \left(\theta_i^{(k)}\right)_{|GP| \times M}$ by applying Equation 18
  $\Theta \leftarrow$ GAME-STORAGE$(\Theta, \Delta, m, sr, sl, p)$ // *Optionally apply storage constraints by applying Algorithm 3*
  Calculate $\Delta^{ST(l)} = \left(\delta_i^{(k)}\right)_{|GP| \times M}$ by applying Equation 19
  if $\max_{k \in phase} \{t_k(\delta)\} > d_{phase}$ **then continue** // *Deadline is not met*
  end if
until $\sum_{k=1}^{A} \left(c_k\left(\Delta^{ST(l-1)}\right) - c_k\left(\Delta^{ST(l)}\right)\right) \leq \epsilon$

According to Equations 12 and 13, the *resource allocation matrix* in the l^{th} stage denoted as $ST(l)$ is calculated based on the distribution of the last stage $ST(l-1)$ and can be further used to produce the new distribution matrix in the same stage:

$$\Theta^{ST(l)} = \Theta\left(\Delta^{ST(l-1)}\right); \tag{18}$$

$$\Delta^{ST(l)} = \Delta\left(\Theta^{ST(l)}\right) = \frac{d_{phase} \cdot \theta_i^{(k)}}{p_i^{(k)}}. \tag{19}$$

4.3. Game-cost algorithm

The Game-cost algorithm outlined in pseudo-code in Algorithm 2 receives as input a set of partitions, the expected execution time $p^{(k)}$ and the number of activities $\delta^{(k)}$ in each activity class AC_k, the number of processors m_i and the price per time unit φ_i of each site $\mathtt{s}_i$, as well as the deadline of the current phase d_{phase}. The algorithm consists of three phases.

Phase 1. The sites are first sorted according to the price/performance ratio for each activity class. In Figure 10, the ordered set of sites for activity classes $\mathcal{AC}_0$ and $\mathcal{AC}_1$ is $\{\mathtt{s}_1, \mathtt{s}_0\}$ in both cases.

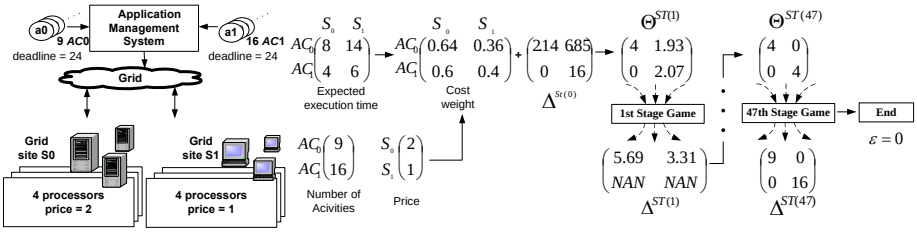


Figure 10: Game-cost example.

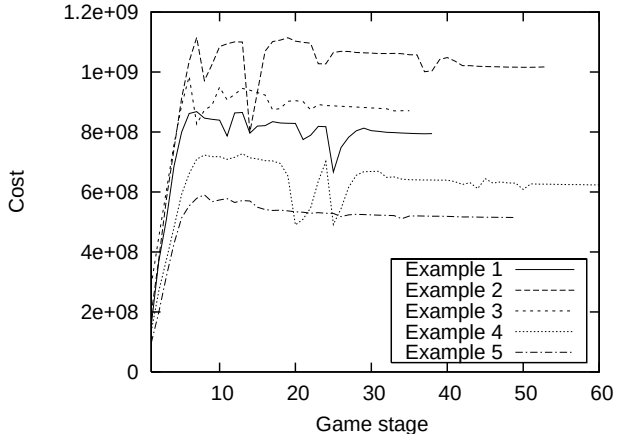


Figure 11: Game-cost convergence.

Phase 2. We add each activity class from the current partition to the set of players and sequentially compute the initial distribution of activities $\Delta^{ST(0)}$ from the fastest to the slowest site in terms of price/performance ratio.

Phase 3. The algorithm searches for the optimal distribution of activities and allocation of resources. At the beginning, activity classes are competitors on the site with the highest price/performance ratio. After the competition of one stage, winners get more processors from one resource and in the next stage losers compete for resources with the second highest price/performance ratio. This process repeats until no more costs can be reduced (i.e. 47 stages in Figure 10). However, in case of some tight deadlines and non-backtracking nature of the algorithm, it might not be possible to meet the deadlines for all activity classes. This happened for less than 1% of the involved activities in our study.

4.4. Convergence

The Game-cost convergence process is very fast as shown in Figure 11 with few numbers of stages due to deadline constraint limits. In this experiment, we randomly generated five examples by assigning $10^2 \times 10^4$ activities to $10^2 \times 10^2$ processors. The optimization almost completed after about $20 - 30$ stages,

No. Procs.	*No. Clusters*	*No. Activities*	*Activity Classes*	*Activity Heterog.*	*Machine Heterog.*	*Price Heterog.*
1023	10	13272	10	[1, 10]	[1, 10]	[1, 10]

Table 6: Cost scheduling simulation environment.

while the entire processes needed about 50 stages for this problem size. We can also observe that the Game-cost convergence curves can vary depending on the competition process and execution environment, exhibit some peaks and troughs, and the total cost declines. The cost grows fast in the beginning to meet the deadline, then slows down and flattens until it reaches a peak. The first peak means the completion of the competition on the site with the highest price/performance ratio. After the peak, the execution cost starts to decline very fast because many activities are moved to the site with the second highest price/performance ratio, and reach a trough. This process repeats until no more optimization can be achieved and all activity classes can meet their deadline. The number of peaks and peak sizes in graphs varies for different cases based on many factors such as prices, deadlines, number of tasks, and initial state of scheduling. Each peak means that the scheduling algorithm re-initialises the competition on one computing site due to violation of constraints. For example, too many tasks are distributed to the computing site due to the competition in the previous sites.

4.5. Complexity

The Game-cost time complexity is $O(l \cdot A \cdot M)$ and the space complexity is $O(A \cdot M)$, where l is the number of game stages, A the number of activity classes, and M the number of sites. Based on the empirical convergence analysis from Section 4.4 we use a maximum of 100 stages for the Game-cost algorithm which proves to be sufficient for convergence.

4.6. Experiments

In this section we compare the results delivered by the Game-cost algorithm with the MCT, OLB, Min-min, Max-min, and Sufferage heuristics modified and extended to incorporate cost and deadline control. Each time an machine is evaluated for an activity, the feasibility of the deadline is also computed. For Min-Min algorithm, in case multiple activities have the same shortest minimum completion time at one iteration, the activity with the highest cost weight is selected. The same selection mechanism is performed for the Max-Min and Sufferage algorithms, too. We performed the experiments in a simulated environment summarized in Table 6, where the expected execution times of activities are generated based on activity and machine heterogeneity, which are selected from a uniform distribution in the specified ranges.

The simulation results displayed in Figure 12 illustrate the following relative cost order of algorithms from best to worst: Game-cost, MCT, Sufferage, Min-min, Max-min, and OLB. Game-cost finds mappings whose costs are better than MCT by 27%, better than Sufferage by 45%, and better than other

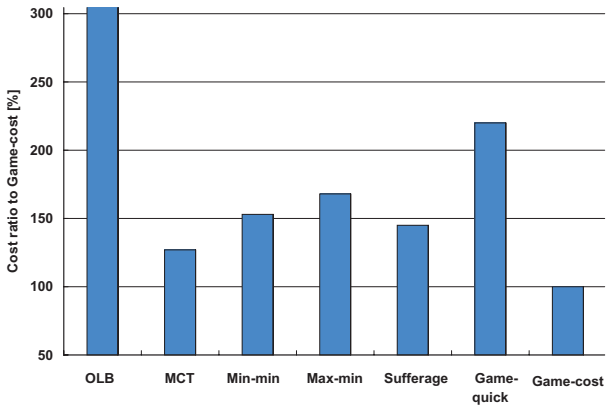
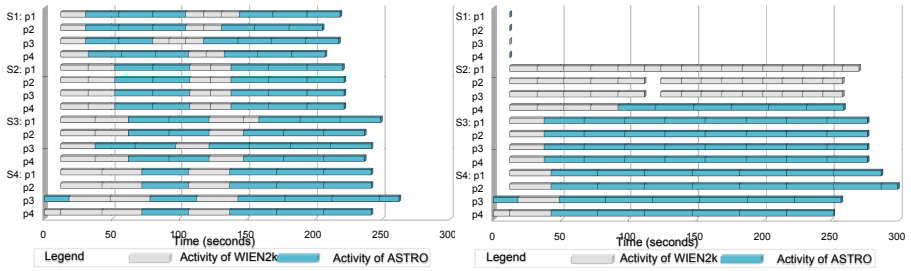


Figure 12: Cost scheduling results.

algorithms by at least 50%. OLB gives the worst results, because there is no cooperation between different activity classes, and the resources are selected based on their availability without considering activity execution time and prices of sites. Max-min gives poor results because it only fits the situation when some activities are much larger than the others, which is a very special situation seldom encountered in practice. In contrast to Max-min, Min-min only handles the smallest activities and ignores lager ones. However, the smallest activities are not the ones with the best performance/price ratio and, therefore, Min-min cannot perform well. Sufferage performs quite similar to Min-min due to similar reasons. MCT performs well, giving the second best results because it unconsciously selects activities with average sizes, and there is a larger likelihood that those activities statistically have the best performance/price ratio. Moreover, Game-cost algorithms can be combined with Game-quick to avoid problem when deadlines cannot be met. First, Game-quick is invoked to schedule the applications to meet the deadlines, and then the results are used as the input to Game-cost. This problem frequently occurs when for relatively short deadlines.

4.7. Real-world experiments

Similar as for the Game-quick algorithm, we evaluated Game-cost for the WIEN2k and AstroGrid scientific applications introduced in Section 2.2 and executed in the Austrian Grid. Our experimental testbed depicted in Table 5 consists of two clusters located at the University of Innsbruck and the University of Linz. Figure 13a and 13b present a scenario in which Game-cost outperforms Min-min (the fastest algorithm among the others for these two applications) in terms of cost, at the expense of a slightly larger makespan within the requested deadline. The reason for the higher makespan is the contradicting nature of the two objectives (makespan and cost), meaning that improving one of them automatically worsens the other. Intrusting is the fact that Game-cost schedules no activities on karwendel, which is the fastest and most expensive site in our environment. In this case, Min-min gives a cost of 7689 and a fairness of 0.9427,



(a) Min-min. (b) Game-cost.

Figure 13: Cost scheduling for two real applications.

which were improved by Game-cost by 45.35% and 5.82% respectively, while still keeping the makespan below the 300 second deadline. Once again, the fairness of Game-cost is almost perfect (0.9976). Similarly to Game-quick, we can intuitively observe that in case of Min-min the application activities are highly interleaved in the Gantt chart which makes their completion time hard to predict. Contrarily, Game-cost yields an execution plan in which activities belonging to the same class are grouped in contiguous slots on the same sites which makes their execution more predictable. The few process idling gaps in the Gantt chart are caused by the join synchronization of activity classes executions in the application control-flow and data-flow structure.

5. Storage-aware scheduling

In this section, we describe two storage-aware extensions to the performance and cost scheduling algorithms described in the previous two sections.

5.1. Storage-aware makespan scheduling

We first introduce a storage-aware makespan scheduling algorithm called *Game-storage* as an extension to Game-quick. The main idea of this method is to accumulate the optimization effects within many game stages until a certain load and storage supply and demand balance among activity classes is achieved. If there is enough storage for all activity classes, the storage supply will never be broken and Game-storage naturally degrades to Game-quick.

Definition 6. *The objective of the* storage-aware makespan scheduling *problem is to minimize the aggregated makespan of a set of large-scale applications as expressed by Definition 2, while additionally fulfilling the storage constraints expressed by the following condition:*

$$\sum_{k=1}^{A} sr^{(k)} \cdot \theta_i^{(k)} < sl_i, \tag{20}$$

Algorithm 3 Game-storage function.

Require: Θ: resource allocation matrix; Δ: activity distribution matrix, m_i: number of processors of site $\mathbf{s}_i$; sl_i: storage limit of site $\mathbf{s}_i (i \in [1..M])$; $sr^{(k)}$: storage requirements of activity class $\mathcal{AC}_k (k \in [1..A])$; $\left(p_i^{(k)}\right)_{A \times M}$: ETC matrix
Ensure: Θ: new resource allocation matrix
1: **function** GAME-STORAGE($\Theta, \Delta, m, sr, sl, p$)
2: **for all** $\mathcal{AC}_k \in \mathcal{AP}$ **do**
3: Calculate all $sw^{(k)}$ by applying Equation 21
4: **end for**
5: **for all** $\mathcal{AC}_k \in \mathcal{AP}$ **do**
6: **for all** $\mathbf{s}_i \in$ *Resources* **do**
7: **while** $\sum_{k=1}^{A} sr^{(k)} \cdot \theta_i^{(k)} > sl_i$ **do** // *Storage requirements are not fulfilled*
8: Recalculate $\theta_i^{(k)}$ by applying Equation 22
9: **end while**
10: **end for**
11: **end for**
12: **end function**

where C is the number of activity classes, $sr^{(k)}$ the storage requirements of activity class $\mathcal{AC}_k$, $\theta_i^{(k)}$ the available processors for activity class $\mathcal{AC}_k$ on site $\mathbf{s}_i$, and sl_i the storage limit on site $\mathbf{s}_i$.

In the beginning of the algorithm, Game-storage uses the same weight definition as Game-quick (see Equation 3) to generate positive impacts on the results of every game stage and enhance the fairness of allocation, because one site has different suitability for different activities. When a storage problem is detected, the algorithm uses the following adapted *storage weight* as the normalized value of expected storage requirements to adjust the resource allocation:

$$sw^{(k)} = \frac{\frac{1}{sr^{(k)}}}{\sum_{x=1}^{A} \frac{1}{sr^{(x)}}}, \tag{21}$$

where $sr^{(k)}$ denotes the *storage requirement* of activity class $\mathcal{AC}_k$. In both phases, the intermediate variable is the *resource allocation matrix* Θ representing the number of processors on each site allocated to each activity class, which accepts the effects from the weights and transfers the effects to the *activity distribution matrix* Δ, representing the activity distribution on each site for each activity class. At each stage, we recalculate the resource allocation based on previous results by applying the following equation:

$$\theta_i^{(k)} = m_i \cdot \frac{\delta_i^{(k)} \cdot p_i^{(k)} \cdot sw^{(k)}}{\sum_{x=1}^{A} \delta_i^{(x)} \cdot p_i^{(x)} \cdot sw^{(x)}} \tag{22}$$

(see Table 1). Through this phase, Game-storage can achieve the storage supply and demand balance, and can expand the throughput of the whole computing system by invoking Algorithm 3 at line 14 of Algorithm 1.

Figure 14 presents a scenario in which Game-storage outperforms other heuristics such as MCT, Sufferage, Min-min, and Max-min [22, 23]. There are two important input matrices in this figure: the expected execution time

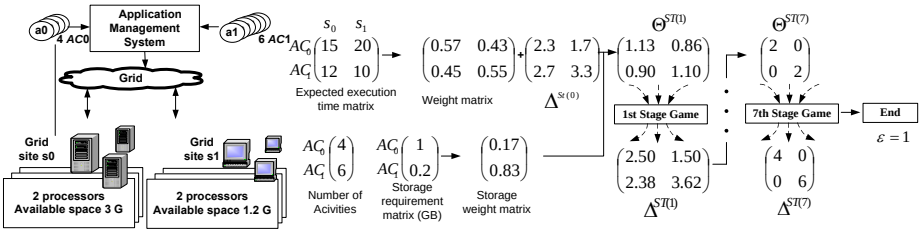
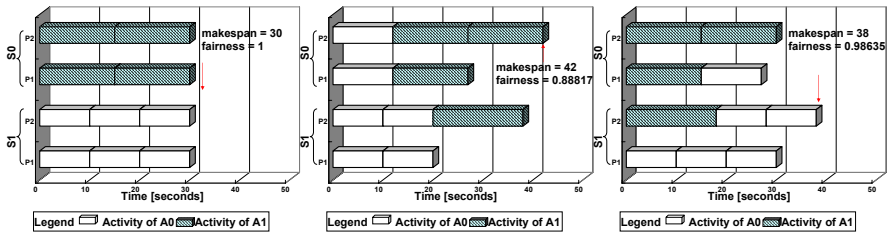


Figure 14: Game-storage example.



(a) Game-storage. (b) Min-min. (c) Max-min, MCT, and Sufferage.

Figure 15: Gantt-chart for the example in Figure 14 using different scheduling algorithms.

matrix for two activity classes $\mathcal{AC}_0 = \{\mathtt{a}_0, \ldots\}$ and $\mathcal{AC}_1 = \{\mathtt{a}_1, \ldots\}$ on two sites $\{\mathtt{s}_0, \mathtt{s}_1\}$, and the *storage requirement vector* expressed in gigabytes. From both inputs the we can generate the weight matrices.

We modified the Min-Min, Max-min, and Sufferage heuristics to operate on activity classes with incorporated storage control that does not schedule an activity on a site with not enough storage resources. For instance, Min-min cannot schedule two activities of activity class $\mathcal{AC}_0$ on site $\mathtt{s}_1$ due to its storage limit and therefore gives a makespan of 42 rather than 40. In this example, Game-storage gives a makespan of 30 which is optimal, Min-min a makespan of 42, while Max-min, MCT and Sufferage a makespan of 38 (see Figure 15).

The *convergence* process of Game-storage is fast, as shown in Figure 16. The number of stages is much lower than for makespan scheduling because the storage constraints limit the convergence processes. The convergence curves can vary depending on the competition process and execution environment, exhibit some fluctuations, and the total makespan declines. The application makespan decreases fast in the beginning, then slows down and has some fluctuations. The fluctuations mean that the storage supply and demand balance on a certain site is broken, and Game-storage needs to find a new balance on this site. After the fluctuations, it starts to decline because many activities are moved to the site with available storage resources. This process repeats until no more optimization can be achieved, and the storage constraints are fulfilled.

The *complexity* of Game-storage is identical to that of Game-quick.

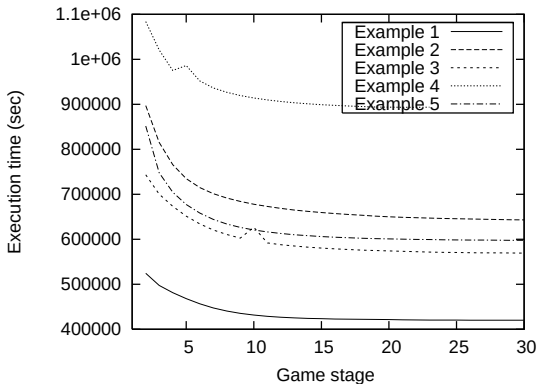


Figure 16: Game-storage convergence.

No. Procs.	*No. of Clusters*	*No. Activities*	*Activity Classes*	*Activity Heterog.*	*Machine Heterog.*	*Storage Heterog.*
1035	10	148690	10	$[1, 1000]$	$[1, 100]$	$[1, 1000]$

Table 7: Storage-aware scheduling simulation environment.

5.2. Storage-aware cost scheduling

Finally, we extend the cost Game-cost algorithm with storage constrains based on a similar idea as for the Game-storage algorithm.

Definition 7. *The objective of the* storage-aware cost scheduling *problem is to minimize the cost of scheduling a set of large-scale applications as defined in Definition 4, while fulfilling the storage constraints according to Equation 20.*

The first algorithm phase is to assign *deadlines* to activity classes and to *partition* the applications according to the assigned deadlines. Then, we apply our *cost scheduling* algorithm on each partition as presented in Section 4. After acquiring the information about activities and resources, we sort the resources for every activity class, and generate an initial distribution of activities, cost weight matrix, and storage weight vector. The difference between Game-cost and the new *Game-storage-cost* algorithm is that the former does not consider storage requirements for activities and storage limitations of sites. Game-storage-cost degrades to Game-cost when there are enough storage resources. Specifically, when storage problems are detected, we recalculate the resource allocation using Algorithm 3 at line 15 of Algorithm 2. The *convergence* curves of Game-storage-cost is similar to those of Game-cost. There are some peaks and troughs in the graph, but is difficult to distinguish between the effects of cost constraints and those of storage constraints. The time *complexity* of this Game-storage-cost is $O\left(l \cdot A^2 \cdot M^2\right)$ and the space complexity is $O\left(A \cdot M\right)$, where l is the number of game stages, A the number of activity classes, and M the number of sites.

5.3. Experiments

Table 7 presents the simulated computing environment, where the real values are randomly generated from a uniform distribution in the specified ranges. Since consistent matrix and inconsistent matrix generate similar results, we only present the results for inconsistent matrices that we consider more authentic for modeling a realistic computing environment. As expected, the execution time of Game-storage and Game-storage-cost is again significantly less than of the related algorithms needing only 198 milliseconds to complete (see Figure 17a). Not only are the time complexities lower, but Game-storage also gives the best makespan as shown in Figure 17b. The relative order of the algorithms from the best to worst is: Game-storage, Min-min, Sufferage, MCT, Max-min, OLB, and MET, for similar reasons as presented in Section 3.6. In terms of fairness, Game-storage always achieved almost perfect fairness of 0.99 in average, as shown in Figure 17c. When there are no storage constraints on the sites, Game-quick performs about $5 - 10\%$ better than Min-min and Game-cost achieves less costs than other algorithms by at least 28% (see Figure 17b). When there are storage constraints, Game-storage improves the performance of multiple workflows by at least 33%, and Game-storage-cost decrease costs by at least 74% (see Fig. 17d). Game-storage provides the best performance because makes the best global decisions in terms of simultaneous performance and storage optimization, while other heuristics can only find a compromise between the two objectives when storage requirements cannot be fulfilled, resulting in a potential waste of computing power. In terms of cost, Figure 17d illustrates that all algorithms need approximately twice the cost of Game-storage-cost.

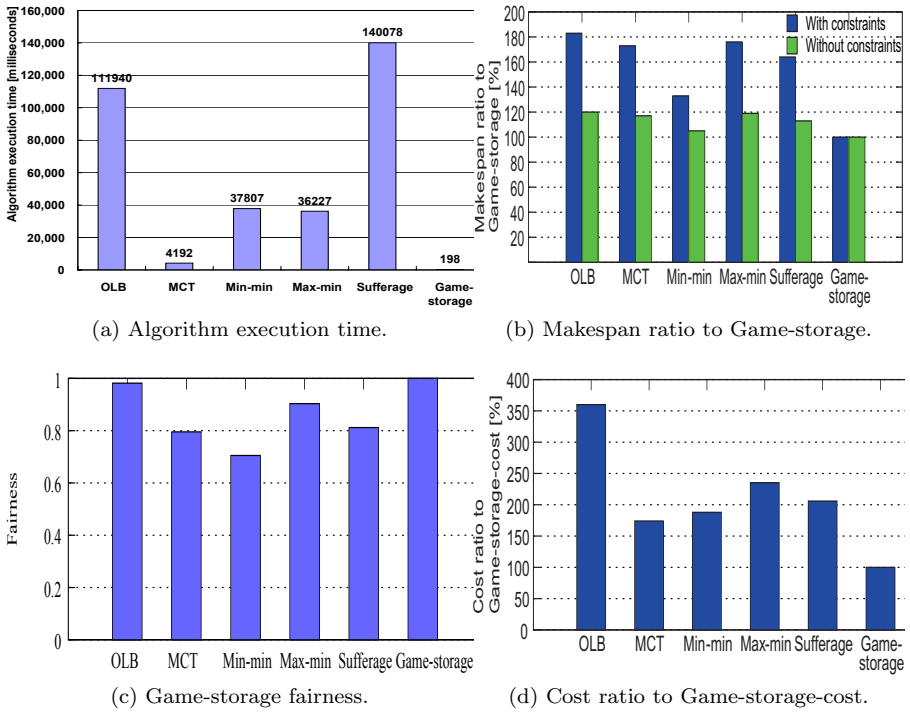
6. Related Work

Scheduling large-scale applications onto distributed computational resources is one of the most important and difficult research topics in high performance computing that led to the development of many different approaches and algorithms. In this section we can therefore cover only a part of two important or relevant areas of related work: large-scale application scheduling (Section 6.1), fairness (Section 6.2) and game theoretic algorithms (Section 6.3). The work described in this paper bridges these two categories.

6.1. Scheduling

The Directed Acyclic Graph Manager (DAGMan) [31] developed by the Condor project allows scheduling of large-scale workflow applications using opportunistic techniques such as matchmaking based on resource offers, resource requests, and cycle stealing with no support for advanced optimization heuristics.

The Grid Application Development Software (GrADS) project [4] continued the tradition of the AppLeS effort on developing techniques for scheduling MPI, iterative, master-slave, and workflow applications. Workflow scheduling is approached by adapting Max-Min, Min-Min, and Suffrage heuristics originally developed for throughput scheduling of independent tasks. We proposed in this



(a) Algorithm execution time.

(b) Makespan ratio to Game-storage.

(c) Game-storage fairness.

(d) Cost ratio to Game-storage-cost.

Figure 17: Storage-aware scheduling results.

paper an approach that proves to be more effective for the class of workflows consisting of large numbers of homogeneous and independent activities.

The scheduler in Gridbus [32] provides just-in-time mappings using Grid economy mechanisms. It makes scheduling decisions on where to place jobs on the Grid depending on the computational resources characteristics and users' Quality of Service (QoS) requirements. Yu and Buyya introduce in [33] a new type of genetic algorithm for large-scale heterogeneous environments for which the existing genetic operation algorithms cannot be directly applied. Due to their high time complexity, genetic algorithms are not practical for large-scale applications. The algorithm in [34] introduces economic cost as a part of the objective function for data and computation scheduling, but does not consider storage constraints and cannot globally solve the performance and cost optimization problem.

A comparison of eleven heuristics is also presented by Braun et al. [9]. All these methods, however, provide a centralized meta-scheduling approach in contrast to our distributed multi-application cost optimization method.

Casanova et al. [35] developed a task-graph scheduling heuristic for multiple homogeneous clusters that provides performance guarantees. Our work is different by targeting a heterogeneous Grid environment and an algorithm that

scales to a huge amount of homogeneous tasks. Furthermore, we do not limit our validation to simulations, but target real-world applications too.

Divisible loads are another class of applications that received significant attention in previous works [36], where the main focus was on how to flexibly partition an application into work chunks for minimizing the makespan (for example through single round or multi round algorithms [37]), while taking into consideration input/output communication overhead and connection latencies. In contrast, the activities that are part of the application class considered in this paper are atomic indivisible legacy codes.

Ramakrishnan et al. [38] solved the scheduling problem in a two-fold approach referred as storage-aware OLB in our experiments: (i) minimize the amount of space an application requires during execution by removing data files at runtime, and (ii) schedule the applications in a way which assures that the amount of data required and generated by the applications fit into individual resources. The algorithm does not consider fairness and cost.

6.2. Fairness

Fairness has been studied extensively and received various definitions for different problems and purposes in particular in the networking field through metrics such as TCP fairness, Max-min fairness, fairly shared spectrum efficiency, or Jain's fairness index. *TCP fairness* relates to congestion control mechanisms and requires that a new protocol receive no larger share of the network than a comparable TCP. *Max-min fairness* [39] is achieved by an allocation if and only if the allocation is feasible and an attempt to increase the allocation of any flow necessarily results in the decrease in the allocation of some other flow with an equal or smaller allocation. In packet radio wireless networks, *fairly shared spectrum efficiency* (FSSE) can be used as a combined measure of fairness and system spectrum efficiency, which is the aggregate throughput in the network divided by the utilized radio bandwidth. The FSSE is the portion of the system spectral efficiency that is shared equally among all active users.

In parallel and distributed computing, fairness measures if activities or tasks are receiving a fair share of resources. Different scheduling problems have been defined and addressed such as online scheduling, workflow scheduling, and proportional fair scheduling for operating system, networking, and real-time systems. Many online schedulers use queue length or delayed estimations to tune their solutions which provides only coarse control for single activities. For example, Condor includes some policies for matching resources to jobs [40], but does not consider the fairness for a whole set of activities. The fairness defined by other related works [41, 42, 43] implies that jobs experience similar slowdown or have a fair waiting time. Jain's fairness index used in our work to quantify fairness relates to the concept of proportional fairness and implies that applications with more computation should be allocated more resources. Nevertheless, no fairness definition fits for all cases of scheduling. We cannot say that the Jane's fairness index is the best metrics for all scheduling algorithms, however, to the purpose of our algorithms, Jane's fairness index tells us if an activity

class obtains enough resources to complete, as other classes submitted at the same time complete their execution.

6.3. Game theoretic scheduling

In terms of game theory-based algorithms, other researchers in performance-oriented distributed computing focused on system-level load balancing [44, 45] or resource allocation [46, 47], aiming to introduce economic and game theoretic aspects into computational questions.

Penmatsa et al. [45] formulate the scheduling problem as a cooperative game where Grid sites try to minimize the expected response time of tasks, while Kwok et al. [47] investigate the impact of selfish behaviors of individual machine by taking into account the non-cooperativeness of machines.

Ghosh et al. [48] proposed a strategy that formulates an incomplete information, alternating-offers bargaining game on two variables: price per unit resource and percentage of bandwidth allocated. Compared to Ghoshs work, we use a more practical pricing model similar to the one used by Cloud resource providers such as Amazon Elastic Compute Cloud.

The ICENI project [49] solves the scheduling problem using a game theory algorithm that eliminates strictly dominated strategies where the least optimal solutions are continuously discarded. The feasibility of this algorithm is questionable due to its high time complexity. Apart from the game theory algorithm, ICENI provides scheduling solutions using random, best of n-random, and simulated annealing.

6.4. Contribution

The approach presented this paper successfully applies game theoretic concepts for scheduling multiple large-scale applications. Our work differs from the related work in that we present a more realistic system model and consider a important class of large-scale applications characterized by a large number of homogeneous independent activities interconnected through simple control flow and data flow dependencies. We introduced four algorithms for makespan, cost, and storage-aware optimization which, compared to other systems, consider intra- and inter-application cooperation. We formulated the scheduling problem as a new sequential cooperative game among several application managers controlling the execution of individual applications and proposed an algorithm that considers not only deadline, cost, and storage, but also provides fairness to all applications.

7. Conclusion

With increasing focus on large-scale applications on the distributed computational resources, it is important for a middleware to efficiently and effectively schedule and dynamically steer execution of large-scale applications. In this paper, we analyzed the main bottleneck of a special class of large-scale applications characterized by a large number of homogeneous activities, and presented

a scheduling solution based on a sequential cooperative game algorithm for three important metrics: makespan, cost, and storage. Experimental results based on simulation, as well as real applications in the Austrian Grid environment demonstrate that our approach delivers better solutions in terms of cost and fairness with less algorithm execution times than other existing approaches such as Min-min, Max-min, or Sufferage. Furthermore, we observed that the larger-scale the experiments are, the better results we achieve. For example, considering larger computing infrastructures up to thousands of processors to schedule the applications will only increase the gap between our game theoretic algorithms and the other classical heuristics, Min-min needing in this case hours to complete.

Our game theory-based scheduling algorithms possess great potential for improvement for large-scale applications in heterogeneous computing infrastructures. We plan to investigate how our algorithms can adapt to other metrics such as memory, security, resource availability, network bandwidth, or multiple virtual organizations. Furthermore, merely a handful of current research efforts consider the simultaneous optimization of multiple constraints. The pricing model used by our Game-cost algorithm series algorithms is based on the commodity market model, which specifies the service price according to the amount of usage. However, many other pricing models exist, including a bargaining, contract-net, or auctioning models and new algorithms incorporating multiple pricing models need also to be studied.

[1] I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields, Scientific Workflows for Grids, Workflows for e-Science, Springer Verlag, 2007.

[2] C. L. Smith, The Large Hadron Collider, Scientific American 283 (1) (2000) 70.

[3] S. S. Vadhiyar, J. J. Dongarra, A metascheduler for the grid, in: 11th International Symposium on High Performance Distributed Computing, IEEE Computer Society, 2002.

[4] F. Berman, et al., New Grid scheduling and rescheduling methods in the GrADS project, International Journal of Parallel Programming 33 (2-3) (2005) 209–229.

[5] T. D. Braun, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 61 (6) (2001) 810837.

[6] K. Czajkowski, I. Foster, N. Karonis, S. Martin, W. Smith, S. Tuecke, A resource management architecture for metacomputing systems, in: Job Scheduling Strategies for Parallel Processing Workshop, Vol. 1459 of Lecture Notes Computer Science, Springer Verlag, 1998, pp. 62–82.

[7] O. H. Ibarra, C. E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, J. ACM 24 (2) (1977) 280–289.

[8] J. Volkert, Austrian Grid: Overview on the project with focus on parallel applications, in: International Symposium on Parallel and Distributed Computing, 2006.

[9] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, R. F. Freund, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 61 (6) (2001) 810–837.

[10] R. Duan, F. Nadeem, J. Wang, R. Prodan, T. Fahringer, A hybrid intelligent approach for performance modeling and prediction of workflow activities in Grids, in: 9^{th} IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE Computer Society, 2009, pp. 339–347.

[11] F. Nadeem, T. Fahringer, Optimizing execution time predictions of scientific workflow applications in the grid through evolutionary programming, Future Generation Computer Systems 29 (4) (2013) 926–935.

[12] S. Ostermann, M. Brejla, R. Prodan, T. Fahringer, S. Schindler, Developing astrophysics workflow applications with the ASKALON environment in the Austrian Grid, in: J. Volkert, T. Fahringer, D. Kranzlmüller, R. Kobler, W. Schreiner (Eds.), 3^{rd} Austrian Grid Symposium, Vol. 269, Austrian Computer Society, 2009, pp. 114–128.

[13] K. Plankensteiner, J. Vergeiner, R. Prodan, G. Mayr, T. Fahringer, Porting LinMod to predict precipitation in the Alps using ASKALON on the Austrian Grid, in: J. Volkert, T. Fahringer, D. Kranzlmüller, R. Kobler, W. Schreiner (Eds.), 3^{rd} Austrian Grid Symposium, Vol. 269, Austrian Computer Society, 2009, pp. 103–114.

[14] R. Prodan, S. Ostermann, K. Plankensteiner, Performance analysis of Grid applications in the ASKALON environment, in: 10^{th} ACM/IEEE International Conference on Grid Computing, IEEE Computer Society, 2009, pp. 97–104.

[15] G. Morar, F. Schüller, S. Ostermann, R. Prodan, G. Mayr, Meteorological simulations in the Cloud with the ASKALON environment, in: I. Caragiannis, M. Alexander, R. M. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. L. Scott, J. Weidendorfer (Eds.), Euro-Par 2012: Parallel Processing Workshops, Vol. 7640 of Lecture Notes in Computer Science, Springer, 2013, pp. 68–78.

[16] W. Kapferer, W. Domainko, S. Schindler, E. V. Kampen, S. Kimeswenger, M. Mair, T. Kronberger, D. Breitschwerdt, Metal enrichment and energetics of galactic winds in galaxy clusters, Advances in Space Research 36 (2005) 682.

[17] K. Schwarz, P. Blaha, G. K. H. Madsen, Electronic structure calculations of solids using the wien2k package for material sciences, Computer Physics Communications 147 (71).

[18] I. Foster, Globus toolkit version 4: Software for service-oriented systems, Journal of Computer Science and Technology 21 (4) (2006) 513–520.

[19] R. B. Myerson, Game Theory: Analysis of Conflict, Harvard University Press, 1997.

[20] R. Armstrong, D. Hensgen, T. Kidd, The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions, in: 7th Heterogeneous Computing Workshop, IEEE Computer Sociery Press, 1998, pp. 79–87.

[21] M. Maheswaran, S. Ali, H. J. Siegel, D. A. Hensgen, R. F. Freund, Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems, in: Heterogeneous Computing Workshop, IEEE Computer Sociery Press, 1999, pp. 30–45.

[22] H. Casanova, D. Zagorodnov, F. Berman, A. Legrand, Heuristics for scheduling parameter sweep applications in grid environments, in: 9th Heterogeneous Computing Workshop, IEEE Computer Sociery Press, Cancun, Mexico, 2000, pp. 349–363.

[23] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, R. F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, Journal of Parallel and Distributed Computing 59 (2) (1999) 107–131.

[24] H. Topcuouglu, S. Hariri, M. you Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.

[25] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence, MIT Press, Cambridge, MA, USA, 1992.

[26] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 2nd Edition, Prentice-Hall et al, 2003.

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

[28] R. K. Jain, D.-M. W. Chiu, W. R. Hawe, A quantitative measure of fairness and discrimination for resource allocation in shared computer systems, Tech. Rep. DEC-TR-301, Eastern Research Lab (September 1984).
URL `http://www1.cse.wustl.edu/ jain/papers/ftp/fairness.pdf`

[29] R. Duan, Grid Workflow Enactment, Optimization, and Fault Tolerance, Ph.d. thesis, University of Innsbruck, Innsbruck, Austria (2008).

[30] B. Kao, H. Garcia-Molina, Deadline assignment in a distributed soft real-time system, IEEE Transactions on Parallel and Distributed Systems 8 (12) (1997) 1268–1274.

[31] Condor Project, DAGMan: Directed acyclic graph manager, http://www.cs.wisc.edu/condor/dagman/, university of Wisconsin-Madison.

[32] R. Buyya, S. Venugopal, The Gridbus toolkit for service oriented Grid and utility computing: An overview and status report, in: 1st International Workshop on Grid Economics and Business Models, IEEE Computer Society Press, 2004, pp. 19–36.

[33] J. Yu, R. Buyya, A budget constrained scheduling of workflow applications on utility grids using genetic algorithms, in: 1st Workshop on Workflows in Support of Large-Scale Science, IEEE Computer Society, 2006.

[34] S. Venugopal, R. Buyya., A deadline and budget constrained scheduling algorithm for e-scienceapplications on data Grids, in: 6th International Conference on Algorithms and Architectures forParallel Processing, Vol. 3719 of LNCS, Springer, 2005.

[35] P.-F. Dutot, T. N'Takp, F. Suter, H. Casanova, Scheduling parallel task graphs on (almost) homogeneous multicluster platforms, IEEE Transactions on Parallel and Distributed Systems 20 (7) (2009) 940–952.

[36] D. Ghose, T. G. Robertazzi (Eds.), Cluster Computing, Vol. 6 of Special issue on divisible load scheduling, Springer, 2003.

[37] Y. Yang, K. van der Raadt, Multiround algorithms for scheduling divisible loads, IEEE Trans. Parallel Distrib. Syst. 16 (11) (2005) 1092–1102, member-Casanova, Henri.

[38] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, M. Samidi, Scheduling data-intensiveworkflows onto storage-constrained distributed resources, in: 7th International Symposium on Cluster Computing and the Grid, IEEE Computer Society, 2007, pp. 401–409.

[39] J.-Y. L. Boudec, Rate adaptation, congestion control and fairness: A tutorial, Tech. rep., Ecole Polytechnique Federale de Lausanne (December 2008).

[40] R. Raman, M. Livny, M. H. Solomon, Matchmaking: An extensible framework for distributed resource management, Cluster Computing 2 (2) (1999) 129–138.

[41] G. Sabin, P. Sadayappan, Unfairness metrics for space-sharing parallel job schedulers, in: D. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), 11th International Workshop on Job Scheduling Strategies for Parallel Processing, Springer-Verlag Heidelberg, 2005.

[42] U. Schwiegelshohn, R. Yahyapour, Fairness in parallel job scheduling, Journal of Scheduling 3 (5) (2000) 297–320.

[43] H. Zhao, R. Sakellariou, Scheduling multiple DAGs onto heterogeneous systems, in: International Parallel and Distributed Processing Symposium, IEEE Computer Society, 2006.

[44] C. Kim, H. Kameda, An algorithm for optimal load balancing in distributed computer systems, IEEE Transactions on Computers 41 (3) (1992) 381–384.

[45] S. Penmatsa, A. T. Chronopoulos, Cooperative load balancing for a network of heterogeneous computers, in: IEEE (Ed.), Proc. of the 21st IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS 2006), Rhodes Island, Greece, 2006.

[46] J. Bredin, R. T. Maheswaran, C. Imer, T. Ba sar, D. Kotz, D. Rus, A game-theoretic formulation of multi-agent resource allocation, in: Proceedings of the Fourth International Conference on Autonomous Agents, ACM Press, Barcelona, Catalonia, Spain, 2000, pp. 349–356.

[47] Y. Kwok, S. Song, K. Hwang, Selfish grid computing: Game-theoretic modeling and nas performance results, in: 5th IEEE International Symposium on Cluster Computing and the Grid, Vol. 2, IEEE Computer Sociery Press, 2005, pp. 1143–1150.

[48] P. Ghosh, K. Basu, S. K. Das, A game theory-based pricing strategy to support single/multiclass job allocation schemes for bandwidth-constrained distributed computing systems, IEEE Trans. Parallel Distrib. Syst. 18 (3) (2007) 289–306.

[49] L. Young, S. McGough, S. Newhouse, J. Darlington, Scheduling architecture and algorithms within the iceni grid middleware, Tech. rep., UK e-Science All Hands Meeting, EPSRC (2003).