Computers & Operations Research 40 (2013) 1564–1578

Contents lists available at SciVerse ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

A Bee Colony based optimization approach for simultaneous job scheduling and data replication in grid environments

Javid Taheri [a,*], Young Choon Lee [a], Albert Y. Zomaya [a], Howard Jay Siegel [b,c]

[a] Center for Distributed and High Performance Computing, School of Information Technologies, J12, The University of Sydney, Sydney, NSW 2006, Australia
[b] Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523-1373, USA
[c] Department of Computer Science, Colorado State University, Fort Collins, CO 80523-1373, USA

ARTICLE INFO

Available online 25 November 2011

Keywords:
Bee colony optimization
Data replication
Grid computing
Job scheduling
Resource allocation

ABSTRACT

This paper presents a novel Bee Colony based optimization algorithm, named Job Data Scheduling using Bee Colony (JDS-BC). JDS-BC consists of two collaborating mechanisms to efficiently schedule jobs onto computational nodes and replicate datafiles on storage nodes in a system so that the two independent, and in many cases conflicting, objectives (i.e., makespan and total datafile transfer time) of such heterogeneous systems are concurrently minimized. Three benchmarks – varying from small- to large-sized instances – are used to test the performance of JDS-BC. Results are compared against other algorithms to show JDS-BC's superiority under different operating scenarios. These results also provide invaluable insights into data-centric job scheduling for grid environments.

1. Introduction

Grid computing has matured into an essential technology that enables the effective exploitation of diverse distributed computing resources to deal with large-scale and resource-intensive applications, such as those found in science and engineering. A grid usually consists of a large number of heterogeneous resources spanning across multiple administrative domains. The effective coordination of these heterogeneous resources plays a vital key role in achieving performance objectives. Grids can be broadly classified into two main categories, computational and data, based on their application focus. In recent years, the distinction between these two classes of grids is much blurred, mainly due to the ever increasing data processing demand in many scientific, engineering, and business applications, such as drug discovery, economic forecasting, seismic analysis, back-office data processing in support of e-commerce, Web services, etc. [1].

In a typical scientific environment such as in High-Energy Physics (HEP), hundreds of end-users may individually or collectively submit thousands of jobs to access peta-bytes of distributed HEP data. Given the large number of tasks resulting from splitting these bulk submitted jobs and the amount of data being used by them, their optimal scheduling along with allocating their demanding datafiles becomes a serious problem for grids—where jobs compete for scarce compute and storage resources among available nodes. The Compact Muon Solenoid (CMS) [2] and the Large Hadron Collider (LHC) [3] are two well known case studies for such applications and are used as a motivation to design many systems including the algorithm in this article. Both systems constantly submit thousands of parallel jobs to access many shared datafiles. In such systems, each job is an acyclic data flow of hundreds of tasks in which CMS/LHR executable modules must run them in parallel [4]. Table 1 shows a typical number of jobs from users and their computation and data related requirements for CMS jobs [5].

Grid schedulers are mainly divided into two types: (1) job-oriented and (2) data-oriented systems. In job-oriented systems, datafiles are fixed in location and jobs are scheduled, usually adhering to some objective such as power consumption [6,7]. In this case, the goal is to schedule jobs among computational nodes (CNs) to minimize the overall makespan of the whole system; here, it also is assumed that the overall transfer time of all datafiles are relatively negligible compare to executing jobs. The speed and number of available computer resources in different CNs and the network capacity between CNs and storage nodes (SNs) are typical considerations taken into account in such systems. For data-oriented systems, on the other hand, jobs are fixed in location and datafiles are moved or replicated in the system so that their accessibility by relevant jobs is increased. In contrast to the previous mode, here it is assumed that transfer time of all datafiles are much more time consuming than executing their dependent jobs. As a result, jobs will need less time to

* Corresponding author.
E-mail addresses: javid.taheri@sydney.edu.au (J. Taheri), albert.zomaya@sydney.edu.au (A.Y. Zomaya), hj@colostate.edu (H.J. Siegel).

0305-0548/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2011.11.012

Table 1
Typical job characteristics in CMS [5].

Number of simultaneously active users	100–1000
Number of jobs submitted per day	250–10,000
Number of jobs being processed in parallel	50–1000
Job turnaround time for jobs	0.2 s–5 months
Number of datasets that serve as input to a sub job	0–50
Average number of datasets accessed by a job	250–10,000K
Average size of the dataset accessed by a job	from 30 GB to 3 TB

download the associated datafiles to execute; and therefore, the total execution time (i.e., makespan plus transfer time) of the system is reduced. The available storage in SNs and the capacity of interconnected network links between CNs and SNs are typical considerations in such allocations. From a practical point of view, neither of these two types of systems is adequate to deal with cases in which both computational jobs and datafiles are equally influential factors for efficient system utilization. Therefore, inappropriate distribution of resources, large queues, reduced performance, and throughput degradation for the remainder of the jobs are some of the drawbacks of assuming systems fit into just one of these two types.

There are three main phases of scheduling in such complex systems [8]: (1) resource discovery, (2) matchmaking, and (3) job execution. In the first phase, resource discovery, grid schedulers conduct a global search to generate a list of all available resources as well as their limitations and history profiles in a system. In the second phase, matchmaking, schedulers try to determine best choices for executing jobs and replicating datafiles. Capacities of CNs/SNs as well as quality of the network connecting them are among the basic characteristics that need to be considered by schedulers to perform this phase. In the last phase, job execution, schedulers produce commands for CNs and SNs to execute jobs and replicate datafiles, respectively. Here, schedulers do not interfere with details of such commands and leave CNs/SNs to perform their allocated commands, including – but not limited to – datafile staging or system cleanups.

In this work, the matchmaking process of schedulers was targeted and our contribution is a holistic scheduling approach to concurrently minimize two very important performance factors of a grid system, i.e., (1) makespan for executing all jobs, and (2) transfer time of all datafiles. Our approach adopts two collaborating mechanisms to schedule jobs and replicate datafiles with respect to their inter-dependencies as well as network bandwidth among CNs/SNs to host these jobs and datafiles.

The rest of this paper is organized as follows. Section 2 presents related works. Section 3 overviews our proposed framework. Section 4 presents the problem statement. Section 5 briefly introduces the Bee Colony optimization algorithm as well as our approach (JDS-BC). Section 6 demonstrates the performance of our approach in comparison with other techniques. Discussion and analysis is presented in Section 7, followed by conclusions in Section 8.

2. Related work

Several approaches already have been proposed to solve the bi-objective scheduling problem that is the focus of this work. Most of these methods make certain assumptions about the nature of jobs and datafiles to present a specific real system. Their solutions can be roughly categorized into two classes: online and batch [9]. In the online methods, it is assumed that jobs arrive one-by-one, usually following a predetermined distribution, and grid schedulers must immediately dispatch these jobs upon receiving them. In the batch methods (also known as batch-of-jobs or bulk) jobs are assumed to be submitted in bulk; and thus, grid schedulers need to allocate several jobs at the same time. Although the online mode can be a fair representation of small grids, CMS and LHR as well as many other massive systems always process the jobs in the batch mode. To date, most approaches usually use only one mode (online or batch) and only few exist that use both.

The European Data Grid (EDG) project was among the first that has created a resource broker for its workload management system based on an extended version of Condor [10]. The problem of bulk scheduling also has been addressed through shared sandboxes in the most recent versions of gLite from the EGEE project [11]. Nevertheless, these approaches only consider one of the priority and/or policy controls rather than addressing the complete suite of co-allocation and co-scheduling issues for bulk jobs. In another approach for data intensive applications, data transfer time was considered in the process of scheduling jobs [12]. This deadline based scheduling approach however could not be extended to cover bulk scheduling. In the Stork project [13], data placement activities in grids were considered as important as computational jobs; therefore, data-intensive jobs were automatically queued, scheduled, monitored, managed, and even check-pointed in this system/approach. Condor and Stork also were combined to handle both job and datafile scheduling to cover a number of scheduling scenarios/policies. This approach also lacks the ability to cover bulk scheduling. In another approach [14], jobs and datafiles are linked together by binding CNs and SNs into I/O communities. These communities then participate in the wide-area system where the Class Ad framework is used to express relationships among the stakeholders. This approach however does not consider policy issues in its optimization procedure. Therefore, although it covers co-allocation and co-scheduling, it cannot deal with bulk scheduling and its related managements issues such as reservation, priority and policy. The approach presented in [15] defines an execution framework to link CPUs and data resources in grids for executing applications that require access to specific datasets. Similar to Stork, bulk scheduling is also left uncovered in this approach.

In more complete works such as the Maui Cluster Scheduler in [16], all jobs are queued and scheduled based on their priorities. In this approach, which is only applicable for local environments (i.e., for clusters rather than grids), weights are assigned based on various objectives to manipulate priorities in scheduling decisions. The data aware approach of the MyGrid [17] project schedules jobs close to the datafiles they require. However, this traditional approach is not always very cost effective as the amount of available bandwidths is rapidly increasing nowadays. This approach also results in long job queues and adds undesired load on sites even when several jobs are moved to other less loaded sites. The GridWay scheduler [18] provides dynamic scheduling and opportunistic migration through a rather simplistic information collection and propagation mechanism. Furthermore, it has not been exposed to bulk scheduling of jobs yet.

The Gang scheduling [19] approach provides some sort of bulk scheduling by allocating similar jobs to a single location; it is specifically tailored toward parallel applications running in a cluster. XSufferage designed as an extension to the well-known Sufferage scheduling algorithm [20] to consider location of datafiles during the scheduling process [21]. This algorithm however only uses such information for better scheduling of jobs, not to (re)allocate/replicate the datafiles. The work in [22] proposes a framework based on GriPhyN [23] and used ChicSim [22] to simulate it. In their framework, they assumed that (1) each job only needs one processor and only one datafile to execute, (2) each job's execution time is linearly related to the size of its requested datafile, and (3) network links between all sites are identical. Based on their simplistic framework, they surprisingly discovered that it is not always necessary to consider both jobs and datafiles at the same time for a better scheduling.

Another approach was presented in [24] for a real data handling system called SAM [25] to help Condor-G in deciding where data-intensive jobs, which need multiple datafiles and probably on different SNs, should be executed. They were particularly interested to schedule jobs so that the minimum amount of data is moved. A distributed and scalable replication and scheduling approach, called DistReSS, was presented in [26]. Similar to [22], job execution times also were assumed proportional to the datafiles they need. Here, they used a *K*-means clustering algorithm to cluster sites and generate Virtual Clusters (VCs). VC-core/heads were responsible to handle and schedule jobs as well as replicate datafiles among their under control sites upon receiving any job. They also assumed all network links are identical and only a predefined number of datafiles exists in each VC. Data Intensive and Network Aware (DIANA) scheduling [6,8] is one of the complete approaches for simultaneous job and datafile scheduling/replication based on the real GILDA [27] and CMS [2] grid systems. In this approach, jobs are first assessed to determine their execution class. For data-intensive applications, jobs are migrated to the best available CN with minimum access (download) time to their required datafiles. For computationally intensive jobs, on the other hand, datafiles are migrated/replicated to the best available SN with minimum access (upload) time to their dependent jobs. In both cases, the decision is made based on: (1) capacity of SNs, (2) speed and number of computers/processors in CNs, and (3) network links connecting SNs and CNs.

Researchers in [9] assumed jobs to be moldable [28]; i.e., they can run on a variable number of processors. Here, they classified SNs and CNs as domains so that a replica is copied in each domain. They also assumed that each CN has a data cache that is large enough to contain all datafiles for the current run. They showed that having local cache has a positive impact in reducing the transfer time as almost half of the times CNs use/access their own cache. Their replication policy is based on detecting the hot files and only replicates those that are demanded more than the average. Close-to-Files [29] is another approach that considered both computation and transfer cost; but for one file only. Integrated Replication and Scheduling Strategy (IRS) [30] is another approach that decouples the replication and scheduling policies. Once IRS schedules jobs and completes them, it calculates the popularity of files and replicates them for the next batch of jobs, which may have different characteristics. The work in [31] proposed a bundle scheduler (HCS+HRS) that is primarily focused on sending jobs to CNs that already have the datafiles and thus takes less time to obtain them. They used a real Grid, Taiwan UniGrid Environment [32], to evaluate their work in which jobs were assumed to need a fixed number of 15 datafiles from a predetermined set of 50 available job types with no datafile overlap.

The researchers in [33] employed some intelligence into their decision making process to detect data types (physical, biology, chemical, etc.) and group them for better replications. Here they used an advanced metric, which includes the number of time a datafile is requested as well as its size, to detect hot datafiles. They also invented the notation of positive or negative distance between SNs to distribute datafiles. Using this distances, they replicated datafiles to achieve (1) the maximum distance among datafiles of different types, and (2) minimum distance of datafiles of the similar type. Unlike [31], the work in [34] assumed heterogeneous datafiles in their framework with only five distinct job types. Here, they scheduled jobs to minimize the amount of transferred data.

After close examination of all the aforementioned techniques, we have realized that most of these systems are usually tailor-made to either minimize makespan or transfer time of all datafiles in a system—with very few exceptions that consider both. As a result, most of these algorithms lack the potential to be extended to other systems, and in many cases even impossible to generalize. For example, algorithms designed for the online mode could rarely be extended to the batch mode and vice versa. Therefore, in this work, we tried to model our framework as generic as possible so that (1) it can be easily extended to any real system—i.e., both data- and job-oriented systems, (2) it can address the bulk scheduling mode appropriate for considerably large systems, and (3) it can support the general case of job to datafile dependency.

3. Framework

The different approaches in the literature make different assumptions to capture the complexity of solving this complicated scheduling problem. In this work, we tried to encompass as many features as possible from these approaches [1,6,8,9,22,24,26,31,33–36] and design our framework (shown in Fig. 1) to consist of heterogeneous: (1) CNs, (2) SNs, (3) interconnecting network, (4) schedulers, (5) users, (6) jobs, and (7) datafiles.

3.1. Computational nodes

In this framework, computer centers with heterogeneous computing elements are modeled as a collection of CNs; each CN (1) consists of several homogenous processors with identical characteristics, and (2) is equipped with a local storage capability. Fig. 2 shows a sample computer center consisting of four CNs with such storage capability. CNs are characterized by (1) their processing speed, and (2) their number of processors. The processing speed for each CN is a relative number to reflect the processing speed of a CN as compared with other CNs in the system. The number of processors for each CN determines its capability to execute moldable [28] jobs with certain degrees of parallelism in a non-preemptive fashion; i.e., jobs cannot interrupt execution of each other during their run-times.

3.2. Storage nodes

SNs are storage elements in the system that host datafiles required by jobs. Two types of SNs exist in this framework: isolated and attached. Isolated SNs are individual entities in the system that are only responsible to host datafiles and deliver

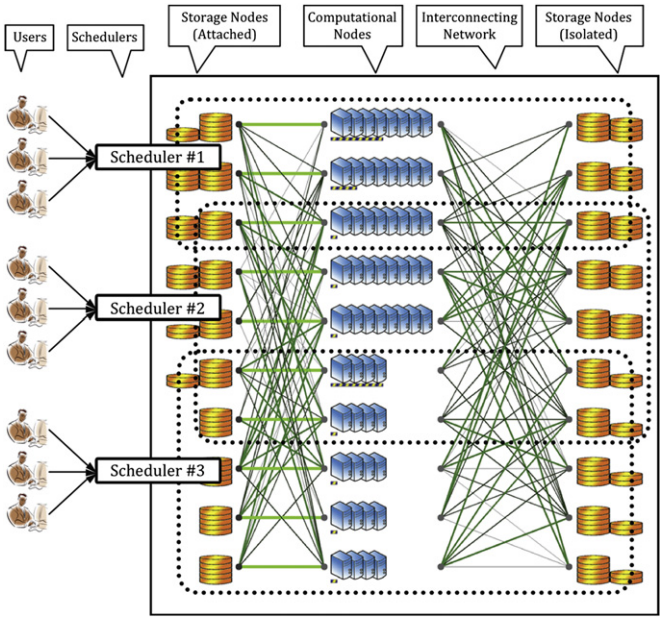

Fig. 1. Framework or our model.

them to requesting CNs. Attached SNs, on the other hand, are local storage capacities of CNs to host their local datafiles as well as to provide them to other CNs if requested. Although from the optimization point of view there is no difference between the two and they are treated equally in a grid system, isolated SNs usually have more capacity than the attached ones; whereas, attached SNs can upload datafiles to their associated CNs almost instantly.

3.3. Interconnection network

CNs and SNs are connected through an interconnection network that is comprised of individual links. Each link in this system has its own characteristics and is modeled using two parameters: delay and bandwidth. Delay is set based on the average waiting time for a datafile to start flowing from one side of the link to the other; bandwidth is set based on the average bandwidth between two sides of the link. Although the above formulation differs from reality in which delay and bandwidth among nodes significantly varies based on system traffic, our extensive simulation showed that this difference is negligible when the number of jobs and datafiles increases in a system. Furthermore, the simulation time is significantly decreased using the proposed simple link model as it has also been endorsed by other works, such as DIANA [6,8]. In our framework, we also assume that links between a CN/SN to other CNs/SNs are independent; and thus, a CN/SN can use the full capacity of each of its connected links and simultaneously download/upload datafiles to other CNs/SNs. We also assume that download and upload streams are independent and cannot delay each other, even for the same link.

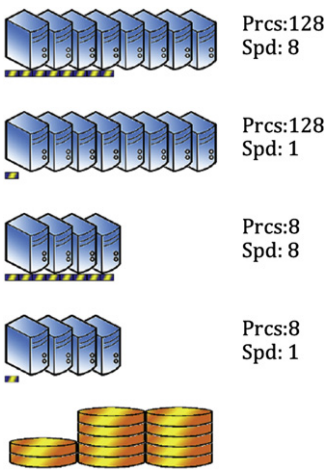

Fig. 2. A computer center example.

3.4. Schedulers

Schedulers are independent entities in the system that accept jobs and datafiles from users and schedule/assign/replicate them to relevant CNs and SNs. Schedulers are in fact the decision makers of the whole system that decide where each job and datafile should be executed or stored/replicated, respectively. Each individual scheduler can be connected to all CNs/SNs or only to a subset of them. Schedulers can be either sub-entities of CNs/SNs or individual job/datafile brokers that accept jobs and datafiles from users. In this work, to cover both cases, the more general case in which schedulers are treated as individual job/datafile brokers is assumed.

3.5. Users

Users generate jobs with specific characteristics. Each user is only connected to one scheduler to submit jobs. Although the majority of users only use pre-existing datafiles in a system, they also can generate their own datafiles should they want to.

3.6. Jobs

Jobs are generated by users and are submitted to schedulers to be executed by CNs. Each job is assumed to be moldable [28] and consists of several dependent tasks – described by a DAG – with specific characteristics, i.e., (1) execution time, and (2) number of processors. Execution time determines the number of seconds a particular task needs to be executed/finalized in the slowest CN in the system—the actual execution time of a task can be significantly reduced if it is assigned to a faster CN instead; number of processors determines a task's degree of parallelism. Using this factor, schedulers eliminate CNs that do not have enough processors to execute specific jobs. Jobs are generated with different shapes to reflect different classes of operations as outlined by TGFF [37] and have the following characteristics: (1) width, (2) height, (3) number of processors, (4) time to execute, (5) shape, and (6) a list of required datafiles. Width is the maximum number of tasks that can run concurrently inside a job; height is the number of levels/stages a job has; number of processors is the maximum number of processors any of the tasks in the job needs to be run; time to execute specifies the minimum time a job can

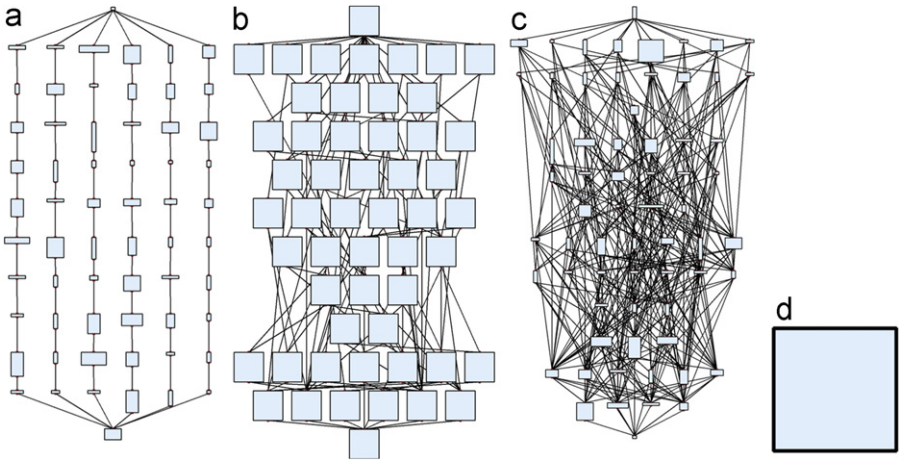

Fig. 3. Jobs' shapes: (a) serious-parallel, (b) homogenous-parallel, (c) heterogeneous-parallel, and (d) single-task.

Table 2
Tasks' characteristics for jobs is Fig. 3.

Shape	Width	Height	Num. of tasks	Num. of processors	Time to execute
Serious-parallel	6	12	62	7	491
Homogenous-parallel	7	12	53	8	260
Heterogeneous-parallel	9	14	65	6	470
Single-task	1	1	1	4	20

be run on the slowest CN in a system [6,8]; and list of required datafiles determines a list of datafiles a CN must download before executing this job. Data to execute each task is provided to it through (1) previously existed datafiles listed by the list of required datafiles and/or (2) output of the task's immediate predecessors in the DAG (either as local/temporary datafile or inter-processing messages). Based on TGFF [37], jobs' shapes are: (1) serious-parallel, (2) homogenous-parallel, (3) heterogeneous-parallel, and (4) single-task. Fig. 3 and Table 2 show sample jobs and their characteristics.

3.7. Datafiles

Datafiles are assumed to be owned by SNs and are allowed to have up to a predefined number of replicas in a system. Schedulers can only delete or move replicas; i.e., the original copies are always kept untouched.

4. Problem statement: Data Aware Job Scheduling

Data Aware Job Scheduling (DAJS) is a bi-objective optimization problem and is defined as assigning jobs to CNs and replicating datafiles on SNs to concurrently minimize (1) the overall makespan of executing a batch of jobs as well as (2) the transfer time of all datafiles to their dependent jobs. Here, it is assumed that the makespan of executing a batch of jobs does not include the transfer time of their requested datafiles. Because these two objectives are usually interdependent, and in many cases even conflicting, minimizing one objective usually results in compromising the other. For example, achieving lower makespans requires scheduling jobs to powerful CNs; whereas, achieving lower transfer times requires using powerful links with higher bandwidths in a system. Table 3 summarizes symbols we use to mathematically formulate the DAJS problem.

To formulate this problem, assume jobs are partitioned into several job-sets, $\{JSet_1, JSet_2, \ldots, JSet_{N_{CN}}\}$, to be executed by CNs, and datafiles are partitioned into several datafile-sets, $\{DSet_1, DSet_2, \ldots, DSet_{N_{SN}}\}$, to be replicated onto SNs. A partition of a set is defined as decomposition of a set into disjoint subsets whose union is the original set. For example, if $N_J = 9$ and $N_{CN} = 3$, then $JobSets = \{\{1,5,7\},\{2,4,8,9\},\{3,6\}\}$ means jobs $\{J_1, J_5, J_7\}$, $\{J_2, J_4, J_8, J_9\}$, and $\{J_3, J_6\}$ are assigned to CN_1, CN_2, and CN_3, respectively.

Based on this model, DAJS is defined as finding elements of job-sets and datafile-sets to minimize the following two objective functions:

$$\begin{cases} 1. & \text{MIN } \text{MAX}_{i=1}^{N_{CN}} JSet_i^{MS} \\ 2. & \text{MIN} \sum_{i=1}^{N_{CN}} JSet_i^{TT} \\ \text{s.t.} & \\ 1.\ JSet_i^{Prcs} \le CN_i^{Prcs}; & i = 1,\ldots,N_{CN} \\ 2.\ DSet_i^{size} \le SN_i^{size}; & i = 1,\ldots,N_{SN} \end{cases}$$

Table 3
Symbols summary.

N_{CN}, N_{SN}, N_J, N_D	Total number of CNs, SNs, jobs, and datafiles in the system.
CN_i^{spd}, CN_i^{prcs}	Relative speed and the total number of processors for the CN # i.
SN_i^{size}	Size of the SN # i.
J_i^w, J_i^h, J_i^{exe}, J_i^{prcs}	Width, height, time to execute, and number of processors required to execute job # i.
$J_i^{\tilde{T}}$, $J_i^{\tilde{D}}$	Set of tasks ($\tilde{T}$) composing job # i and its dependent set of datafiles ($\tilde{D}$) to execute.
J_i^{ST}, J_i^{EX}, J_i^{TT}	Start, execution time and transfer time for all datafiles for executing job # i.
$JSet_i^{MS}$, $JSet_i^{TT}$	Makespan and total transfer time of all datafiles addressed by a collection of jobs described in $JSet_i$ to be executed by CN_i.
$DSet_i^{size}$	Total size of a collection of datafiles addressed by $DSet_i$ to be hosted by SN_i.

here, if $JSet_i = \{J_1, J_2, \ldots, J_K\}$ contains K jobs scheduled to be executed by CN_i, then makespan and total transfer time of this job-set can be calculated as follows:

$$JSet_i^{MS} = \text{MAX}_{k=1}^{K}(J_k^{ST} + J_k^{EX})$$

and

$$JSet_i^{TT} = \sum_{k=1}^{K} J_k^{TT}$$

In the stated bi-objective formulation, the first constraint is to guarantee that all CNs are capable of executing their assigned jobs, while the second constraint is to guarantee that the total size of all datafiles each SN hosts is less than its total capacity. Overall makespan of executing a set of jobs greatly depends on each CN's local scheduling policy; extensive research however showed that the local scheduling policy of First-Come-First-Served with backfilling usually results in optimal deployment of CNs' resources when large number of jobs are submitted [38]. Therefore, we also adopt this policy as the local scheduling policy for CNs in our framework.

5. Bee Colony algorithm for solving the DAJS problem

This section first overviews the Bee Colony Optimization (BCO) method and then details how it is used to solve the DAJS problem.

5.1. Bee Colony Optimization (BCO)

The Bee Colony is an optimization procedure inspired by the behavior of nectar collecting honeybees. This biologically inspired approach has been employed to solve variety of optimization problems, including but not limited to: training neural networks [39], numerical function optimization [40], job shop scheduling [41], Internet server optimization [42], the travelling salesman problem [43], and many more. The algorithm is based on the simple fact that honeybees report their findings – amount and quality of their collected nectar as well as its collecting distance – upon their return to the hive. Such information is conveyed through a "waggle dance" on the hive's *dance floor*: the better the nectar, the higher the benefit, and consequently, the longer the dance. In turn, other bees observe these waggle dances and probabilistically follow one of these bees with respect to their waggling times; this procedure is repeated during the whole nectar collecting time. In the general form of BCO, there are three different bee types: scouts, dancers, and followers [44]. Scouts continuously search hive's neighborhood to discover new sources of nectars; dancers inform other bees of the current established food sources (through their waggling dances); and, followers

probabilistically choose one of the dancers and follow its lead to a food source.

5.2. Simultaneous job and data scheduling using Bee Colony Optimization

The original BCO procedure is meticulously modified in this work to match the special needs of the problem we address in this work. Here, as the number of available CNs is always predefined and limited, bees sometimes act in more than one role. Therefore, in our approach (JDS-BC), every bee chooses its collecting nectar source through two parallel procedures: (1) roughly searches its neighborhood as a scout, and (2) pays attention to dancing bees on the *dance floor* as a follower. At the end, upon choosing its food source and after collecting its nectar, it reports its benefit from its selection and may replace one of the dancing bees if it collects more benefit than it.

Fig. 4 shows the overall flowchart of our approach in which jobs and datafiles are disseminated within two correlated procedures. We decided to design two independent correlated procedures instead of one overall procedure as (1) it has been shown to be more effective [22,30], and (2) it significantly reduces the complexity of the problem and always results in a faster convergence. Based on these findings, the first procedure adopts a BCO-based approach to schedule jobs among CNs, considering location of datafiles; and, the second procedure is to replicate datafiles to provide faster uploads to their dependent jobs, considering their already assigned CNs. This two-stage optimization procedure is repeated for a limited number of iteration (*MaxItr*); the best answer found during these iterations is reported as the final answer. Below we explain some necessary terminology before we elaborate the overall optimization process.

5.2.1. Benefit

This is the overall amount a job can achieve through selecting a given CN. In JDS-BC, to simultaneously target both objectives of the DAJS problem, such benefit consists of two parts: execution benefit and transfer benefit. Execution benefit is designed to motivate jobs in selecting less loaded CNs so that computation load of all CNs in a system is effectively balanced. Transfer benefit is designed to motivate jobs in choosing CNs that can download their dependent datafiles faster. These objectives are defined as follows:

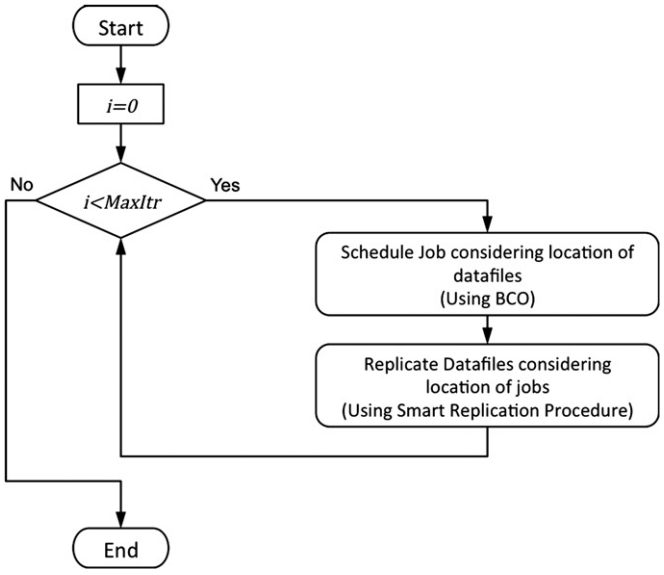

Fig. 4. JDS-BC flowchart.

$$Benefit(J_i,CN_j)=\left(CN_j^{FrPrcs}-\frac{J_i^{Exe}}{CN_j^{prcs}}\right)+\left(\alpha\times\exp\left(\frac{J_i^{TT}}{\beta}\right)\right)$$

where CN_j^{FrPrcs} is total number of free processors CN_j has up to the current makespan of the system; α and β are two arbitrary constants to adjust decreasing rate of the chosen exponential function for downloading datafiles. In JDS-BC, these constants are empirically set as: $\alpha=\beta=10$.

5.3. Dance floor

The *dance floor* in JDS-BC is designed to host a limited number of bees to represent heterogeneous CNs in a system. Here, because CNs can have vastly different characteristics, the number of bees to advertise each CN is designed to be relevant to its overall capacity. As a result, powerful CNs that will eventually execute a wider range of jobs will also have more bees to waggle in their favor; it is captured by the following:

$$\begin{cases} dance\,floor=\bigcup_{i=0}^{N_{CN}}\{B_i^1,B_i^2,\ldots,B_i^{M_i}\} \\ M_i=CN_i^{spd}\times CN_i^{prcs} \end{cases}$$

where '∪' is the union of sets, B_i is a dancing bee for CN_i, and M_i determined the maximum number of bees can waggle in favor of CN_i.

5.4. Similarity

Despite the original BCO algorithm in which all bees are similar, jobs in our approach can have vastly different characteristics; therefore, only jobs with similar characteristics can expect achieving similar benefits if they follow each other. To define such similarity in JDS-BC, five attributes of each job that can roughly describe its characteristics are chosen to defined the similarity of two jobs as follows:

$$Sim(J_x,J_y)=Average\left\{\frac{\frac{\mathrm{MIN}(J_x^w,J_y^w)}{\mathrm{MAX}(J_x^w,J_y^w)},\frac{\mathrm{MIN}(J_x^h,J_y^h)}{\mathrm{MAX}(J_x^h,J_y^h)},\frac{\mathrm{MIN}(J_x^{prcs},J_y^{prcs})}{\mathrm{MAX}(J_x^{prcs},J_y^{prcs})},\frac{\mathrm{MIN}(J_x^{exe},J_y^{exe})}{\mathrm{MAX}(J_x^{exe},J_y^{exe})},SIZE(\cap(J_x^{\tilde{D}},J_y^{\tilde{D}}))}{\mathrm{MAX}(SIZE(J_x^{\tilde{D}}),SIZE(J_y^{\tilde{D}}))}\right\}$$

where '∩' is the intersection of sets, and $SIZE(\bullet)$ returns the size of a set of datafiles.

5.5. Estimated-benefit

Each follower bee estimates a benefit for following a dancing bee on the *dance floor*. It is worth mentioning that because the actual benefit each bee can receive cannot be determined before its actual trip to a flower, they estimate such benefits so that they can probabilistically follow one of the dancing bees. In JDS-BC, each bee estimates such benefit through two parallel procedures (scouting and following) and chooses their maximum as its estimated-benefit; it is calculated as follows:

$$\widehat{Benefit}(J_i,CN_j)=\mathrm{MAX}\{Benefit_{Scout}(J_i,CN_j),Benefit_{Follower}(J_i)\}$$

where

$$Benefit_{Scout}(J_i,CN_j)=\widehat{CN}_j^{FrPrcs}-\frac{J_i^{exe}}{CN_j^{prcs}}$$

and

$$Benefit_{Follower}(J_i)=\mathrm{MAX}\left\{\bigcup_{B_w\in DanceFloor}Sim(J_i,B_w)\times B_w^{bnft}\right\}$$

here, B_w^{bnft} is the received benefit for B_w; and, $\widehat{CN}_j^{FrPrcs}$ is the expected number of free processors for CN_j up to the expected

makespan of the whole system upon scheduling J_i. Because the actual makespan cannot be determined; it is estimated as follows:

$$\widehat{MkSpn} = \mathrm{MAX}_{x=1}^{N_{CN}} \left\{ CN_x^{MS} + \frac{J_i^{exe}}{CN_x^{spd}} \right\}$$

5.6. Schedule jobs

To schedule jobs onto CNs, each job/bee is trying to maximize its benefit by choosing the right CN/food-source. Here, as the benefit of choosing a CN is estimated not guaranteed, each job/bee probabilistically (thorough a Roulette Wheel) chooses its CN based on the acquired estimated benefits. The following procedure details this scheduling procedure:

Step 1: sort jobs according to a criterion.
Step 2: initialize the *dance floor*.
Step 3: for each job, J_x.
  Calculate the estimated makespan upon scheduling J_x, namely $\widehat{MkSpn}$.
  Calculate estimated benefit, $\widehat{Benefit}(J_i,CN_j)$, for all CNs and save results into an Array called *ArrEstBnfts*.
  Generate a Roulette Wheel with respect to *ArrEstBnfts*.
  Choose CN_y from the generated Roulette Wheel and schedule/assign J_x to it.
  Calculate the exact benefit of executing J_x on CN_y, i.e., $Benefit(J_i,CN_j)$.
  Update *dance floor*.
Step 4: repeat Step 3 until all jobs are scheduled.

In Step 1, jobs are sorted based on a criterion, such as Longest Jobs First, before allocations; in Step 2, a predefined number of positions are allocated for bees to advertise each CN; in part of Step 3, the newly allocated job/bee returns to the *dance floor* and replaces an older bee if its collected benefit is higher than it. However, to prevent biasing all bees to a particular bee type (job type), bees will be replaced if they have less than 80% similarity to already dancing bees in the *dance floor*. For bees with more than 80% similarity in advertising a unique CN, only the bee with the higher benefit is kept on the *dance floor*. As a result, different bee types representing different job types will be on the *dance floor* to advertise a CN for a more variety of jobs.

5.7. Replicate datafiles

In this procedure, jobs assumed scheduled to CNs and JDS-BC is trying to provide better access to their dependent jobs. The following procedure details such procedure:

Step 1: sort datafiles according to a criterion.
Step 2: for each datafile, D_x.
  Find the total upload time of D_x to all its dependent jobs if it is replicated on each SN; store these uploading times in an array called *ArrUpTimes*.
  Sort *ArrUpTimes* in ascending order.
  For $k=1$ to *MaxNumReplicas*.
  IF $\left(\frac{ArrUpTimes(k)}{MinUpTime(D_x)} < 2\right)$ THEN, replicate D_x onto SN_k.
  Next k.
Step 3: repeat Step 2 until all datafiles are replicated.

In the above procedure, Step 1 sorts datafiles with respect to a criterion – such as Largest datafiles first – to prioritize replication of datafiles with respect to each other; in Step 2, *MaxNumReplicas* is the maximum number of replicas each datafile can have in the system; i.e., up to *MaxNumReplicas*+1 instances of each datafile can exist in a system: one non-removable original copy that is hosted by a specific SN, and up to *MaxNumReplicas* removable replicas on other SNs. The condition of '$ArrUpTimes(k)/MinUpTime(D_x) < 2$' prevents replicating datafiles that probably will not help reducing the total transfer time of a datafile to its dependent jobs; $MinUpTime(D_x)$ returns the minimum uploading time of D_x replicated on any SN.

6. Simulation

To simulate the performance of JDS-BC three artificial grids are generated by using exclusively designed simulator. These grids are generated based on the direct observations from [2,3,6,8] where other similar algorithms in this area, such as DIANA, are also made for. Tables 4 and 5 show the general and specific characteristics of these systems where jobs and datafiles are assumed generated by an arbitrary number of 100–1000 users. Fig. 5 shows the overall structure of the smallest test-grid in our system; overall shape of the other two can be judiciously predicted.

Table 4
Overall system characteristics.

System entity	Attribute	Mean	St. dev.	Min	Max
CN	Number of processors	32	32	8	128
	Processors' speed	4	4	1	10
	Attached storage size (GByte)	5	5	1	20
SN	Storage size (GByte)	100	100	20	500
Users	Job generation	100	100	10	1000
Datafile	Size (MByte)	50	50	1	200
Network links	Link latency (s)	0.2	0.2	0	1
	Bandwidth (Mbit/s)	1.024	1.024	0.128	12.8
Jobs: tasks	Execution time (s)	30	30	10	1000
	Processors' dependency	2	2	1	10
Jobs: shape	Serious-parallel (30%) Homogenous-parallel (30%) Heterogeneous-parallel (20%) Single-task (20%)				
	Width	4	4	1	10
	Height	10	4	1	15
Jobs: datafiles	Datafiles' dependency	5	5	0	20

6.1. Comparing Algorithms

As was clearly mentioned in Section 2, most algorithms that are intended to schedule jobs and replicate datafiles do not support bulk scheduling; and thus, cannot be compared against JDS-BC. However, a limited number of approaches could either support both online and batch/bulk modes, or, could be easily modified to support the batch mode. DIANA [6,8], Chameleon/FLOP [45], MinTrans [9,22,24,34], and MinExe [9,22] are four other approaches that also support the batch mode scheduling; and thus, are selected to test the performance JDS-BC.

In summary, DIANA [6,8] categorizes the submitted jobs as either computationally intensive or data intensive. For a computationally intensive job, DIANA migrates it to a CN with the lowest execution time; and for a data intensive jobs DIANA either migrates the job to a CN with the fastest datafile download time, or, replicates the datafiles to SNs with higher upload times. Chameleon [45], also known as FLOP, targets CNs that can start executing jobs straight away; i.e., it always migrates jobs to CNs that can start executing them faster than others. Although Chameleon/FLOP does not initially consider datafiles download times during it scheduling process, it always replicates datafiles upon scheduling jobs to provide highest upload times to them.

MinTrans represent a collection of approaches that schedule jobs to CNs with already cached datafiles; including: JobDataPresent in [22], Data-Present in [9], TLSS+TLRS in [34], and an extension made to SAMGrid using Condor-G in [24]. The above approaches are motivated by the fact that obtaining datafiles are usually the costlier portion of executing a job; and thus, if jobs are sent to CNs with already cached datafiles, the overall performance of a system must improve. MinExe represent another group of approaches that schedule jobs to CNs that can execute them faster; including: JobLeastLoaded in [22] and Shortest-Turnaround-Time in [9]. Such approaches are motivated by the fact that cache repositories of powerful CNs are gradually enriched as more types of jobs are scheduled on them; and thus, it is the running portion of jobs that would eventually dominate the overall performance of a system. Achieving lower makespans and transfer-times are the second priority in MinTrans and MinExe, respectively. Because MinTrans prioritize transfer-time to makespan, we hypothetically assume that it can probably achieve the lowest possible transfer-time of any algorithm, and thus, its results are used as a benchmark to gauge performance of other algorithms in this work; same argument is applied to MinExe and using its makespan/resource-utilization as the benchmark.

Table 5
Detailed characteristics of tailor-made grids.

Attributes	Test-Grid-Small	Test-Grid-Medium	Test-Grid-Large
Number of CNs	5	10	20
Number of SNs	10	20	40
Total number of computing elements in all CNs	184	312	624
Total storage size in all SNs (GB)	12,672	79,872	127,872
Number of users	100	500	1000
Number of jobs	1068	5023	9652
Total number of tasks in all Jobs	26,427	119,042	231,360
Number of datafiles	70	435	731

6.2. Measurement criteria

Several measurement criteria are already proposed in the literature to measure performance of such systems; the most common ones are: (1) turnaround, (2) throughput, (3) slowdown, (4) fairness, (5) makespan, (6) transfer-time, and (7) resource-utilization. The first four criteria are intended to study a system's behavior/dynamic during the reception of jobs, while the latter three are used to study the overall performance of a system. Although all criteria are applicable for online approaches, only the latter three are meaningful for bulk scheduling; mainly because, jobs are submitted as a bulk and the system will not report any result unless all jobs are finalized. Therefore, only the latter three criteria are used to compare algorithms in this work. In summary, makespan reflects the latest time all CNs finish their allocated/scheduled jobs; transfer-time represents the total amount of uploading time to deliver all datafiles to their relevant jobs; and, resource-utilization represents how well CNs are deployed. Table 6 shows how resource-utilization is calculated for the heterogeneous environment of our framework. It is also worthwhile mentioning, similar to other approaches [6,8,9,22,24,26,30,31,33–35,45], we also assumed that if several jobs in a CN require the same datafile, the requested datafile will be downloaded only once and then stored in a local repository (cache) for further local requests.

6.3. Test grids

6.3.1. Test-Grid-Small

This is a small-sized grid with very limited number of jobs and datafiles. Fig. 5 shows the overall configuration of this network with 1068 jobs – consists of 26,427 tasks – and 70 datafiles.

Table 6
A sample calculation for resource-utilization.

Item	Processors	Makespan	Num of used processors up to time: makespan	All available processors
CN1	16	95	1520	$16\times105=1680$
CN2	8	100	800	$8\times105=840$
CN3	4	105	420	$4\times105=420$
$Res-util=\frac{1520+800+420}{1680+840+420}=93.2\%$				

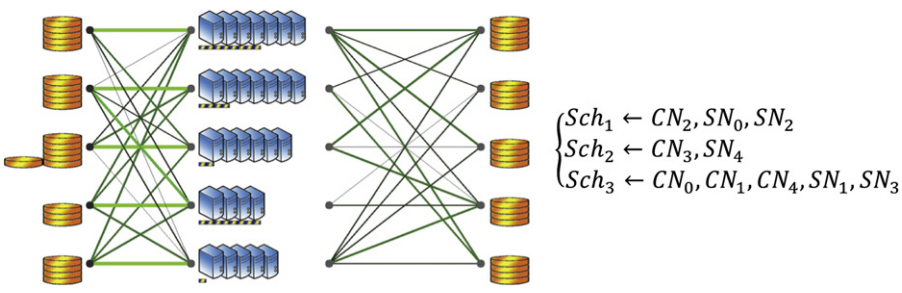

Fig. 5. Test-Grid-Small.

Table 7
Results for Test-Grid-Small for job sorting policy of LJF (a and b), SJF (c and d), and MIX (e and f), and datafile sorting policy of *SIZE* (a, c, and e) and POP (b, d, and f).

SIZE				POP			
LJF **(a)**	Makespan	Trans-time	*Res-util* (%)	**(b)**	Makespan	Trans-time	*Res-util* (%)
JDS-BC	2677		94.2	JDS-BC	2670	1290	93.2
DIANA	4202	1100	56.3	DIANA	3873	1190	64.5
FLOP	4365	1180	60.6	FLOP	4365	1320	60.6
MinTrans	4739	80	35.2	MinTrans	4748	100	35.4
MinExe	2625	1310	95.6	MinExe	2625	1470	95.6
SJF **(c)**	Makespan	Trans-time	*Res-util* (%)	**(d)**	Makespan	Trans-time	*Res-util* (%)
JDS-BC	2857	1260	87.3	JDS-BC	2863	1210	87.4
DIANA	3847	1170	65.7	DIANA	3294	1290	75.9
FLOP	4846	1140	55.4	FLOP	4846	1010	55.4
MinTrans	4977	80	32.5	MinTrans	4975	90	32.5
MinExe	2812	1160	87.2	MinExe	2812	1110	87.2
MIX **(e)**	Makespan	Trans-time	*Res-util* (%)	**(f)**	Makespan	Trans-time	*Res-util* (%)
JDS-BC	2846	1180	88.5	JDS-BC	2746	1070	91.0
DIANA	3607	1440	56.0	DIANA	3810	1130	53.7
FLOP	3258	1250	78.8	FLOP	3258	1220	78.8
MinTrans	4820	80	33.1	MinTrans	4805	90	34.1
MinExe	2715	1120	91.7	MinExe	2715	1180	91.7

Table 8
Results for Test-Grid-Medium for job sorting policy of LJF (a and b), SJF (c and d), and MIX (e and f), and datafile sorting policy of *SIZE* (a, c, and e) and POP (b, d, and f).

SIZE				POP			
LJF **(a)**	Makespan	Trans-time	*Res-util* (%)	**(b)**	Makespan	Trans-time	*Res-util* (%)
JDS-BC	6824	5845	86.3	JDS-BC	6584	5235	90.0
DIANA	11,234	5220	53.7	DIANA	10,642	4610	58.2
FLOP	10,921	5140	58.1	FLOP	10,921	4355	58.1
MinTrans	19,075	555	26.1	MinTrans	10,999	155	42.3
MinExe	6358	6355	96.0	MinExe	6358	5055	96.0
SJF **(c)**	Makespan	Trans-time	*Res-util* (%)	**(d)**	Makespan	trans-time	*Res-util* (%)
JDS-BC	7513	4625	76.5	JDS-BC	7296	3945	78.0
DIANA	10,360	5340	60.4	DIANA	17,035	4630	35.3
FLOP	13,023	5555	49.9	FLOP	13,023	4895	49.9
MinTrans	21,509	525	20.6	MinTrans	11,487	55	38.6
MinExe	6621	5795	89.8	MinExe	6621	5035	89.8
MIX **(e)**	Makespan	Trans-time	*Res-util* (%)	**(f)**	Makespan	Trans-time	*Res-util* (%)
JDS-BC	7269	4725	79.1	JDS-BC	7174	4135	80.3
DIANA	10,498	2435	43.1	DIANA	10,752	1740	42.0
FLOP	10,813	5730	58.5	FLOP	10,813	4705	58.5
MinTrans	21,134	480	21.2	MinTrans	11,181	245	40.7
MinExe	6405	7700	94.1	MinExe	6405	6065	94.1

Table 7 presents the performance of different algorithms for scheduling jobs and replicating datafiles in this environment. Here, jobs are sorted based on LJF: Longest Jobs First, SJF: Shortest Jobs First, or MIX: mixture of them; and, datafiles are sorted based on *SIZE*: size, or POP: popularity—datafiles are more popular if more jobs are dependent on them.

6.3.2. Test-Grid-Medium

This test grid is to represent medium-sized grids with moderate number of jobs and datafiles. Table 8 shows performance of different algorithm on this system with 5023 jobs – consists of 11,9042 tasks – and 435 datafiles.

6.3.3. Test-Grid-Large

This test grid represents large-sized grids with very high number of jobs and datafiles. Table 9 shows performance of different algorithm on this system with 9652 jobs – consists of 231,360 tasks – and 731 datafiles.

7. Discussion and analysis

Tables 7–9 show the results of applying the algorithms to minimize the overall makespan as well as transfer-time of the whole system for these tailor-made grids. The following sections explain more about different aspects of these results.

Table 9
Results for Test-Grid-Large for job sorting policy of LJF (a and b), SJF (c and d), and MIX (e and f), and datafile sorting policy of *SIZE* (a, c, and e) and POP (b, d, and f).

SIZE				POP			
LJF **(a)**	Makespan	Trans-time	*Res-util* (%)	**(b)**	Makespan	Trans-time	*Res-util* (%)
JDS-BC	6561	9966	90.7	JDS-BC	6594	8656	90.2
DIANA	17,202	8344	34.9	DIANA	15,482	6906	36.6
FLOP	11,971	8988	51.4	FLOP	11,971	8178	51.4
MinTrans	19,566	738	33.9	MinTrans	20,647	748	30.6
MinExe	6215	10,236	96.1	MinExe	6215	9064	96.1
SJF **(c)**	Makespan	Trans-time	*Res-util* (%)	**(d)**	Makespan	Trans-time	*Res-util* (%)
JDS-BC	6856	9436	85.5	JDS-BC	6878	8346	85.9
DIANA	15,262	7628	36.8	DIANA	15,192	6592	36.4
FLOP	11,731	9698	52.7	FLOP	11,731	8500	52.7
MinTrans	30,706	1042	21.6	MinTrans	22,281	664	26.3
MinExe	6642	10,294	88.9	MinExe	6642	8988	88.9
MIX **(e)**	Makespan	Trans-time	*Res-util* (%)	**(f)**	Makespan	Trans-time	*Res-util* (%)
JDS-BC	6797	9508	87.1	JDS-BC	6673	8178	89.4
DIANA	27,074	5070	17.4	DIANA	30,999	4268	14.8
FLOP	8913	9724	68.4	FLOP	8913	8614	68.4
MinTrans	18,460	754	30.3	MinTrans	21,147	712	29.1
MinExe	6333	12,542	94.1	MinExe	6333	10,886	94.1

7.1. Makespan

This criterion represents the time the latest CN in the system finalizes its assigned jobs, which is one of the two objectives that must be minimized in a system. For the small-sized grid (Table 7), JDS-BC's makespan was always 1–5% above the benchmark (MinExe's makespan) for all job sorting and datafile sorting policies. DIANA's makespan was more dynamic for different setups. For different job sorting policies, it achieves 33–60% and 17–48% above the benchmark for *SIZE* and POP datafile sorting methods, respectively. FLOP's makespan was always 20–72% above the benchmark for all situations. MinTrans' makespans were always 77–81% above the benchmark. For the medium-sized grid (Table 8) where more jobs and datafiles exist in the system, performance differences between these algorithms become more distinguishable. Here, JDS-BC could still continue its near optimal performance and achieves makespans only 4–13% above the benchmark. DIANA's makespan became worse compare to its previous deployment; it achieved makespans, 56–157% above the benchmark. FLOP also slightly worsen its performance and achieved 69–97% above the benchmark. MinTrans' makespans were 73–230% above the benchmark; almost comparable to DIANA and FLOP sometimes. For the large-sized grid (Table 9) where these algorithms are pushed to their limits, their true performances are become very distinct. Here, JDS-BC kept its superb performance and achieved makespans only 3–7% above the benchmark. DIANA's makespan became almost unacceptable as it achieved 29–389% above the benchmark. FLOP, however, could retain its performance and achieved makespans only 41–93% above the benchmark. Not surprisingly, MinTrans' makespan were the absolute worse with 191–362% above the benchmark. In summary, JDS-BC's, DIANA's and FLOP's makespans were always 1–13%, 17–389%, and 20–93% above the benchmark (at all times), respectively.

7.2. Transfer time

This criterion reflects the quality of the replication policy for each of the aforementioned techniques. For the small-sized grid (Table 7) the total transfer time for all techniques is almost similar for all scheduling and replication policies; mainly because not enough datafiles exist in the system to challenge these techniques. MinTrans has managed to deliver all datafiles in this system almost without any delay as the low number of datafiles in the system allowed replication of all datafiles onto CNs' attached SNs to provide instant access to all jobs. For the medium-sized grid (Table 8), DIANA shows its marginal better performance in comparison to JDS-BC and FLOP; DIANA managed to deliver all datafiles in almost 83% of the time the other two algorithms could. For the large-sized grid (Table 9), however, DIANA's performance is more noticeable where it managed to deliver all datafiles in only 71% of the time the other three algorithms were able. For all systems, MinTrans was able to deliver all datafiles in almost 1–10% of the times the others could. This shows that although enough space is available in SNs to replicate most datafiles, they cannot be used; mainly because, it would result in unreasonably under resource utilization of CNs in such systems.

7.3. Recourse Utilization

This criterion is to measure the microscopic behavior of scheduling techniques in disseminating jobs. Despite the macroscopic criterion of makespan that only measures the overall outcome of a system, resource-utilization reflects the exact percentage of unused computing units during the whole process of executing a bag-of-jobs. Resource-utilization has a direct relation to the makespan of a system where lower makespan usually means better resource-utilization and vice versa. For the small-sized grid (Table 7), JDS-BC's recourse-utilization was only 0.1–4% less than the benchmark (MinExe); DIANA's and FLOP's recourse-utilization were 11–40% and 13–35% less than the benchmark, respectively. For the medium-size grid (Table 8), differences between these algorithms become more distinguishable. Here, JDS-BC almost kept it performance and managed to utilize the system only 5–15% less than the benchmark. DIANA and FLOP, however, showed more undesirable performance where their resource-utilizations were dropped by 30–52% and 36–40% than the benchmark, respectively. For the large-sized grid (Table 9) where these algorithms are pushed to their limits,

JDS-BC still managed to utilize the system only by 3–9% lower than the benchmark. DIANA and FLOP, in this case, showed absolutely unacceptable performances where their resource-utilization severely dropped to 53–77% and 26–45%, lower than the benchmark, respectively. In summary, JDS-BC's resource-utilization was never worse than 15% of the benchmark; whereas, DIANA and FLOP could sometimes under utilize a system for up to 77% and 45%, respectively.

7.4. Performance analysis of JDS-BC

Further analysis of Tables 7–9 shows that JDS-BC implicitly adopts different strategies in solving the DAJS problem under different conditions. For small-sized systems (Table 7) where transfer time of datafiles is almost half of the jobs' execution times (makespan), JDS-BC leans more toward faster execution of jobs and maximum utilization of CNs. A logical explanation for such behavior relates to existence of a fairly small number of CNs as well as datafiles in such systems. Therefore, all CNs are able to either replicate the requested datafiles in their attached SNs or cache them upon the first download. As a result, local depository of all CNs are enriched very soon and the transfer time portion of the stated DAJS become almost ineffective in further scheduling decisions. For medium-size grids (Table 8), on the other hand, existence of a larger number of datafiles does not allow all CNs to replicate their requested datafiles or cache them. Here, JDS-BC changed its strategy and tries to balance the transfer time and makespan of such systems. This usually results in slight under utilization of CNs (where better scheduling is still possible). Such under utilization is however smart enough to have a very positive impact in reducing transfer time of the system. In other words, the amount of transfer time is saved as a result of such under utilization is much better than the extra time JDS-BC spends to execute jobs (makespan). For large-sized systems (Table 9) where total size of datafiles is beyond capacity of any CN, JDS-BC is more inclined toward reducing datafiles' transfer time. Here, JDS-BC implicitly acts like algorithms designed for data-intensive applications (such as DIANA). This imitation is however smart enough to never allow heavy under utilization of a system at all times. As a result, although JDS-BC is focused on reducing the transfer time, it never allows recourse utilization below 85%.

In summary, JDS-BC's definition of benefit that consists of both objectives of the stated problem allows it to efficiently adapt to both computational-intensive and data-intensive applications as well as any other scenarios in between. In all situations, JDC-BC can detect the dominant objective of a system and automatically allocates more weight to minimizing it. Nevertheless, JDS-BC's optimization process is balanced enough to never allow a heavy negative impact from any of the stated two objectives. As a result, the amount JDS-BC loses by relaxing one of its objectives (e.g., transfer time) is always much lower than the amount it gains through optimizing the other one (e.g., makespan).

7.5. Comparing all the methods

Each of the methods has its own pros and cons. The two extreme approaches, MinTrans and MinExe, demonstrate the true nature of this bi-objective optimization problem and confirmed that reducing makespan and transfer-time cannot be concurrently achieved. As it could be seen in Tables 7–9, MinTrans and MinExe always result in the lowest transfer-time and makespan for all cases, respectively. They however showed unacceptable performance in reducing the objective they were not initially targeted. For example, MinTrans's makespan was always 4–5 times than that of MinExe's and MinExe's transfer-time was always much higher than that of MinTrans. Figs. 6–8 show a better neck-to-neck comparison of all these techniques for these test grids under different simulation setups. As can be seen in all these figures, JDS-BC always achieved the lowest makespan plus transfer-time, DIANA – the best approach in the literature – achieved the second overall performance, and FLOP was the worse.

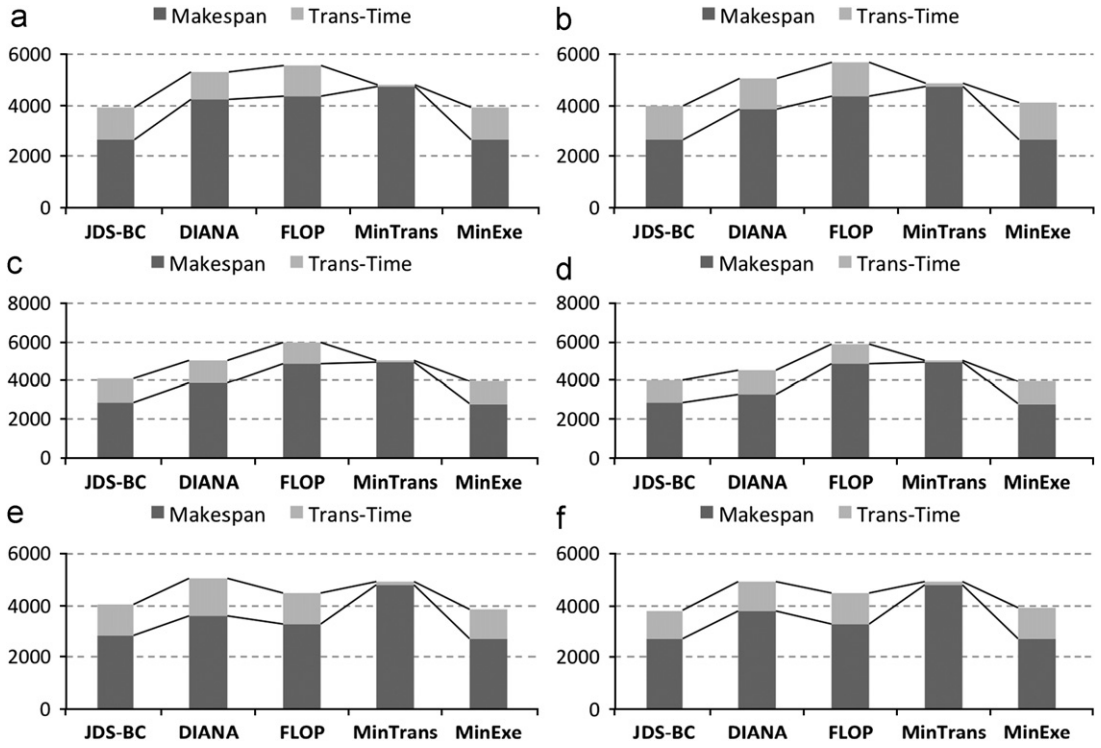

Fig. 6. Overall execution time on Test-Grid-Small with job sorting policy of LJF (a and b), SJF (c and d), and MIX (e and f), and datafile sorting policy of *SIZE* (a, c, and e) and POP (b, d, and f).

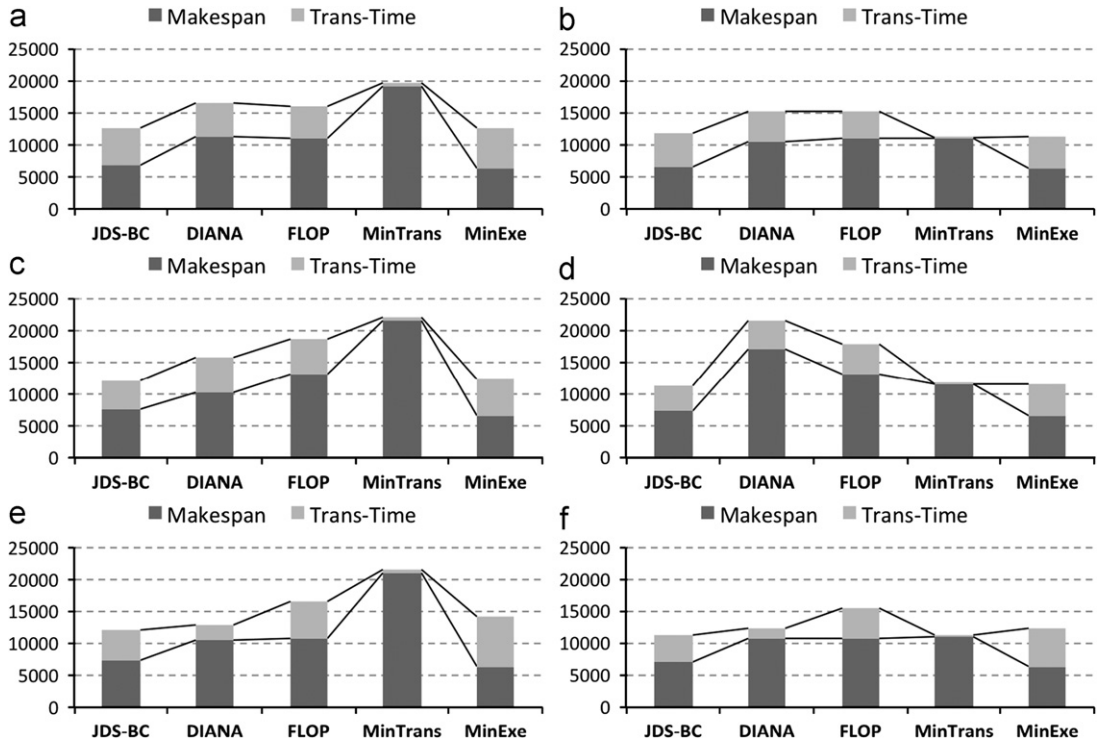

Fig. 7. Overall execution time on Test-Grid-Medium with job sorting policy of LJF (a and b), SJF (c and d), and MIX (e and f), and datafile sorting policy of *SIZE* (a, c, and e) and POP (b, d, and f).

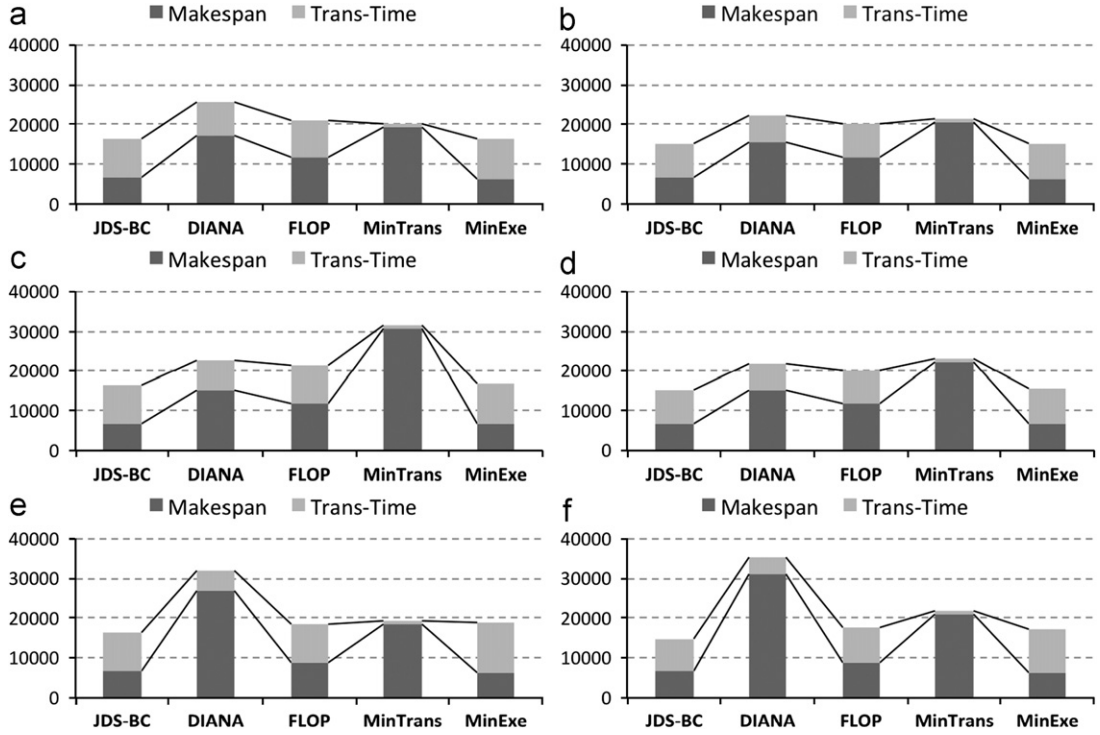

Fig. 8. Overall execution time on Test-Grid-Large with job sorting policy of LJF (a and b), SJF (c and d), and MIX (e and f), and datafile sorting policy of *SIZE* (a, c, and e) and POP (b, d, and f).

In summary, FLOP with its greedy approach, which starts to execute jobs without considering the location of the relevant datafiles ranked the least favorite. DIANA shows reasonable results in reducing transfer-time, while its makespan and resource-utilization proves its incompetency for deployment in medium- to large-sized systems. DIANA's overall performance proves that (1) migrating jobs to CN with better access to datafiles or (2) preventing replication of datafiles only because they have large sizes usually results in heavy system's under utilizations. JDS-BC showed the best performance among the three for all systems. Its makespan and resource-utilization was always closest to the benchmark, while its transfer-time was always less than 33% worse than

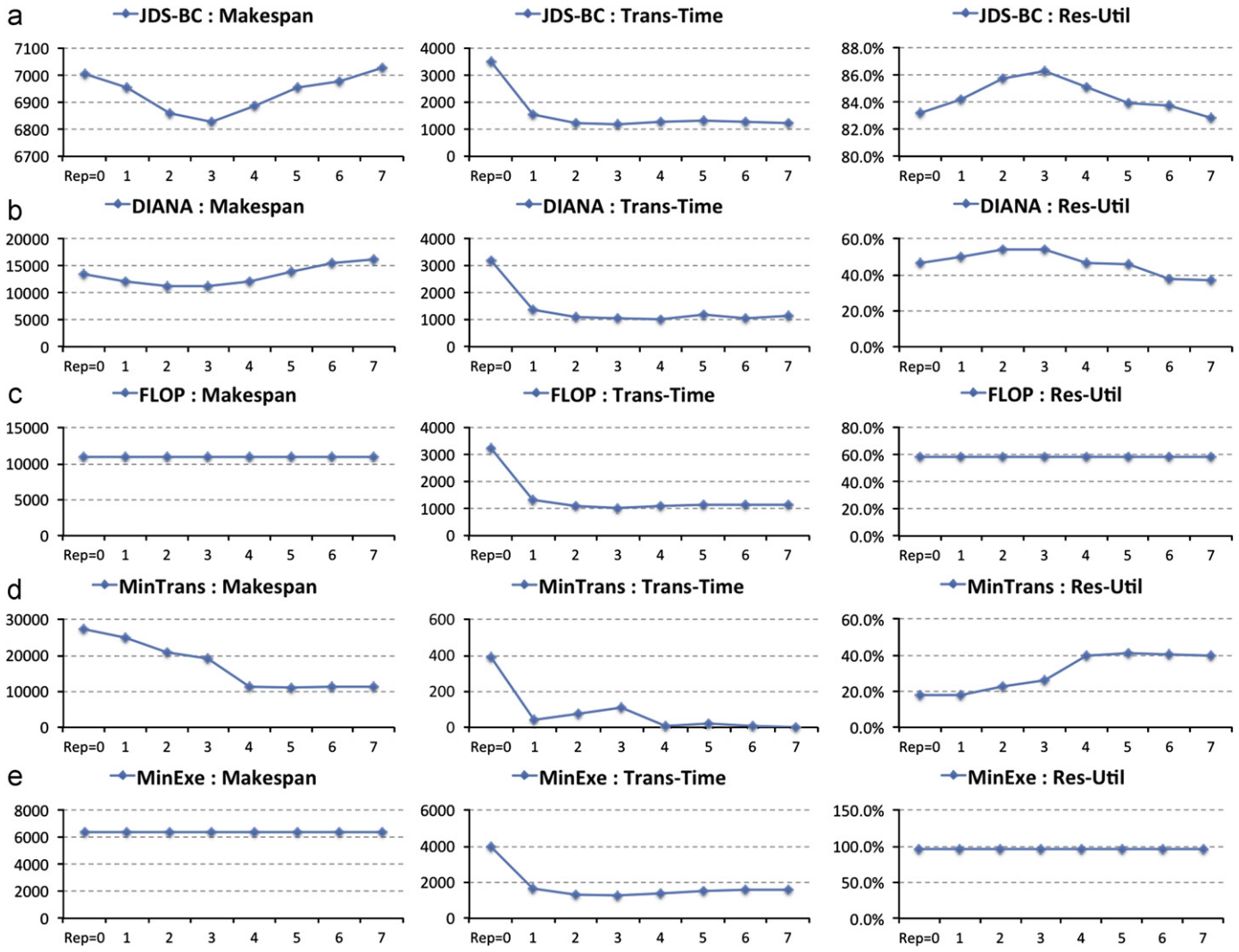

Fig. 9. Effect of the number of replicas on performance of JDS-BC (a), DIANA (b), FLOP (c), MinTrans (d), and MinExe (e).

DIANA. JDS-BC's strategy in scheduling jobs and replicating datafiles proves the fact that sending similar jobs to a CN while replicating their similar dependent datafiles on a SN is most probably the best strategy to concurrently reduce both equally important objectives of the stated DAJS problem. In other words, approaches that try to group similar jobs/datafiles to be scheduled/replicated on a CN/SN would probably result in better system utilizations as well as transfer-times; Gang scheduling [16] also hypothesized this fact in scheduling its jobs and datafiles.

7.6. The effect of number of replicas

For all simulations, datafiles were limited to have at most four instances in a system (one original and three replicas). In this section, this limitation is relaxed to study the effect of number of replicas on the overall performance of the aforementioned techniques. To this end, Test-Grid-Medium is selected again and performance of different techniques for up to 0–7 number of replicas – i.e., up to 1–8 instances for each datafile – is measured. Fig. 9 shows the overall makespan, transfer-time and resource-utilization when jobs and datafiles are prioritized based on their executing time (LJF) and size (*SIZE*), respectively. Results of the other two test grids with different combination of job and datafile sorting were not reflected here as they were very similar to the presented one. Fig. 9 shows that despite the general impression that more replicas should result in reducing the overall transfer-time and consequently better utilization of a system, more replicas can in fact waste some precious resources in a system and result in under utilization sometimes! This figure also showed that high and low number of replicas always result in system under utilization; therefore, setting a proper number for replicating datafiles can be as important as efficiently scheduling jobs and/or replicating datafiles.

7.7. Convergence time

Table 10 shows the convergence time, for a single iteration of each algorithm. The actual number of iterations that each of these algorithms needs to converge to a solution greatly depends on the problem size as well as the capability of the computer running the algorithm. Nevertheless, just to provide an overview of their convergence speeds for the presented test grids in this work, all algorithms managed to converge to their final solution in less than five iteration at all times on a dual core 2.1 MHz desktop with 4G RAM running Windows XP. This table clearly shows that JDC-BC requires the least amount of time to converge to a solution; whereas, DIANA, FLOP, and MinExe usually needed 4–14 times more computing time to converge; MinTrans needs 2–6 times more computing time to converge.

Table 10
Convergence speed for one iteration of each algorithm.

	JDS-BC	DIANA	FLOP	MinTrans	MinExe
Test-Grid-Small	18 (00 h:00 min:18 s)	74 (00 h:01 min:14 s)	73 (00 h:01 min:13 s)	41 (00 h:00 min:41 s)	69 (00 h:01 min:09 s)
Test-Grid-Medium	255 (00 h:04 min:15 s)	1787 (00 h:29 min:47 s)	1673 (00 h:27 min:53 s)	822 (00 h:13 min:42 s)	1302 (00 h:21 min:42 s)
Test-Grid-Large	295 (00 h:04 min:55 s)	4036 (01 h:07 min:16 s)	3405 (00 h:56 min:45 s)	1845 (00 h:30 min:45 s)	3219 (00 h:53 min:39 s)

8. Conclusion and future works

This paper presented a novel BCO based approach, namely JDS-BC, to schedule jobs to CNs and replicate datafiles in SNs. JDS-BC focuses on the matchmaking process where two independent and in many cases even conflicting objectives in such systems (makespan and total transfer time) must be concurrently minimized. Three tailor-made test grids varying from small to large are used to study the performance of JDS-BC and compare it with other algorithms in the field. Results showed that JDS-BC is able to efficiently adapt itself to both computational and data-intensive grids as well as other systems in between. JDS-BC can automatically favor the optimization of the dominant objective and implicitly prioritizes it in its optimization process. JDS-BC also showed a balanced decision making behavior, where it sometimes relaxes one of its objectives (e.g., transfer time) to gain more from optimizing the other one (e.g., makespan).

The results presented here also pave the path to new research that can be targeted in the future. For example, the DAJS problem in this work is defined as a bi-objective problem in which both makespan and transfer time of a batch of jobs (bulk) must be concurrently minimized. Recent studies however show that other criteria such as energy efficiency of grid resources or the cost of system maintenance can become important factors for future systems; and thus, a natural extension to our work would be adding extra criteria to the DAJS problem to address these issues as well. Another avenue to extent result of our work is to modify the proposed solutions to clouds where major service providers need to concurrently minimize execution of their jobs as well as downloading/accessing time of datafiles by users. For clouds in particular, security issues and jobs priority as well as other criteria must be added to the DAJS problem so that service providers can satisfy many Service Level Agreements of the submitted jobs through maximizing system utilization.

Acknowledgements

Professor A.Y. Zomaya's work is supported by an Australian Research Council Grant LP0884070.

Professor H.J. Siegel's work is supported by the United States National Science Foundation (NSF) Grant CNS-0905399, and by the Colorado State University George T. Abell Endowment.

References

[1] Berman F, Fox G, Hey AJG. Grid computing: making the global infrastructure a reality. New York: John Wiley and Sons; 2003.
[2] CERN. Compact muon solenoid (CMS). 〈http://public.web.cern.ch/public/en/lhc/CMS-en.html〉; 2011 [visited].
[3] CERN. Large hadron collider (LHC). 〈http://public.web.cern.ch/public/en/lhc/lhc-en.html〉; 2011 [visited].
[4] Holtman K. CMS data grid system overview and requirements. The compact muon solenoid (CMS) experiment note 2001/037. Switzerland: CERN; 2001.
[5] Holtman K. HEPGRID2001: a model of a virtual data grid application. In: Hertzberger LO, Hoekstra AG, Williams R, editors. HPCN Europe 2001: Proceedings of the ninth international conference on high-performance computing and networking. London, UK: Springer-Verlag; 2001, p. 711–20.
[6] Anjum A, McClatchey R, Ali A, Willers I. Bulk scheduling with the DIANA scheduler. IEEE Transactions on Nuclear Science 2006;53:3818–29.
[7] Subrata R, Zomaya AY, Landfeldt B. Cooperative power-aware scheduling in grid computing environments. Journal of Parallel and Distributed Computing 2010;70:84–91.
[8] McClatchey R, Anjum A, Stockinger H, Ali A, Willers I, Thomas M. Data intensive and network aware (DIANA) grid scheduling. Journal of Grid Computing 2007;5:43–64.
[9] Tang M, Lee B-S, Tang X, Yeo C-K. The impact of data replication on job scheduling performance in the data grid. Future Generation Computer Systems 2006;22:254–68.
[10] Frey J, Tannenbaum T, Livny M, Foster I, Tuecke S. Condor-G: a computation management agent for multi-institutional grids. Cluster Computing 2002;5:237–46.
[11] Andreetto P, Borgia S, Dorigo A, Gianelle A, Mordacchini M, Sgaravatto M et al. Practical approaches to grid workload and resource management in the EGEE project. In: Proceedings of the conference on computing in high energy and nuclear physics (CHEP'04); 2004. p. 899–902.
[12] Jin H, Shi X, Qiang W, Zou D. An adaptive meta-scheduler for data-intensive applications. International Journal of Grid and Utility Computing 2005;1:32–7.
[13] Kosar T, Livny M. A framework for reliable and efficient data placement in distributed computing systems. Journal of Parallel and Distributed Computing 2005;65:1146–57.
[14] Thain D, Bent J, Arpaci-Dusseau A, Arpaci-Dusseau R, Livny M. Gathering at the well: creating communities for grid I/O. In: Proceedings of the 2001 ACM/IEEE conference on supercomputing; 2001. p. 58–78.
[15] Basney J, Livny M, Mazzanti P. Utilizing widely distributed computational resources efficiently with execution domains. Computer Physics Communications 2001;140:246–52.
[16] Bode B, Halstead DM, Kendall R, Lei Z, Jackson D. The portable batch scheduler and the maui scheduler on linux clusters. In: Proceedings of the fourth annual showcase and conference (LINUX-00); 2000. p. 217–24.
[17] Cirne W, Da Silva DP, Costa L, Santos-Neto E, Brasileiro FV, Sauve J et al. Running bag-of-tasks applications on computational grids: the MyGrid approach. In: Proceedings of the 2003 international conference on parallel processing (ICPP'03); 2003. p. 407–16.
[18] Huedo E, Montero ReS, Llorente IMi. The GridWay framework for adaptive scheduling and execution on grids. Scalable Computing: Practice and Experience 2005;6:1–8.
[19] Strazdins PE Uhlmann J. A comparison of local and gang scheduling on a beowulf cluster. In: Proceedings of 2004 IEEE international conference on cluster computing (CLUSTER'04); 2004. p. 55–62.
[20] Maheswaran M, Ali S, Siegel HJ, Hensgen D, Freund RF. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. Journal of Parallel and Distributed Computing, Special Issue on Software Support for Distributed Computing 1999;59:107–31.
[21] Casanova H, Zagorodnov D, Berman F, Legrand A. Heuristics for scheduling parameter sweep applications in grid environments. In: Proceedings of the ninth heterogeneous computing workshop; 2000. p. 349–63.
[22] Ranganathan K Foster I. Decoupling computation and data scheduling in distributed data-intensive applications. In: Proceedings of 11th IEEE international symposium on high performance distributed computing (HPDC'02); 2002. p. 352–8.
[23] GriPhyN. Grid physics network in ATLAS. 〈http://www.usatlas.bnl.gov/computing/grid/griphyn/〉; 2011 [visited].
[24] Hoschek W, Jaen-Martinez J, Samar A, Stockinger H, Stockinger K. Data management in an international data grid project. In: Buyya R, Baker M, editors. Grid computing, vol. 1971. New York: Springer-Verlag; 2000. p. 77–90.
[25] SAM. 〈http://projects.fnal.gov/samgrid/〉; 2011 [visited].
[26] Chakrabarti A, Sengupta S. Scalable and distributed mechanisms for integrated scheduling and replication in data grids. In: Rao S, Chatterjee M, Jayanti P, Murthy C, Saha S, editors. Distributed computing and networking, vol. 4904. Berlin/Heidelberg: Springer; 2008. p. 227–38.
[27] GILDA. 〈https://gilda.ct.infn.it/〉; 2011 [visited].
[28] Feitelson D, Rudolph L, Schwiegelshohn U, Sevcik K, Wong P. Theory and practice in parallel job scheduling. In: Feitelson D, Rudolph L, editors. Job scheduling strategies for parallel processing, vol. 1291. Berlin/Heidelberg: Springer; 1997. p. 1–34.

[29] Mohamed HH, Epema DHJ. An evaluation of the close-to-files processor and data co-allocation policy in multiclusters. In: Proceedings of 2004 IEEE international conference on cluster computing; 2004. p. 287–98.
[30] Chakrabarti A, Dheepak R, Sengupta S. Integration of scheduling and replication in data grids. In: Bougé L, Prasanna V, editors. High performance computing—HiPC, vol. 3296. Berlin/Heidelberg: Springer; 2004. p. 85–101. In: Bougé L, Prasanna V, editors. High performance computing—HiPC, vol. 3296. Berlin/Heidelberg: Springer; 2005. p. 85–101.
[31] Chang R-S, Chang J-S, Lin S-Y. Job scheduling and data replication on data grids. Future Generation Computer Systems 2007;23:846–60.
[32] UniGrid. Taiwan unigrid environment. 〈http://www.unigrid.org.tw〉; 2011 [visited].
[33] Dang NN, Lim SB. Combination of replication and scheduling in data grids. International Journal of Computer Science and Network Security (IJCSNS) 2007;7:304–8.
[34] Abdi S, Mohamadi S. Two level job scheduling and data replication in data grid. International Journal of Grid Computing and Applications (IJGCA) 2010;1:23–37.
[35] Bell WH, Cameron DG, Capozza L, Millar AP, Stockinger K, Zini F. OptorSim: a grid simulator for studying dynamic data replication strategies. The International Journal of High Performance Computing Applications 2003;17:403–16.
[36] Subrata R, Zomaya AY, Landfeldt B. A cooperative game framework for QoS guided job allocation schemes in grids. IEEE Transactions on Computers 2008;57:1413–22.
[37] Dick RP, Rhodes DL, Wolf W. TGFF: task graphs for free. In: Proceedings of the sixth international workshop on hardware/software codesign; 1998. p. 97–101.
[38] Hongzhang S, Oliker L, Biswas R. Job superscheduler architecture and performance in computational grid environments. In: Proceedings of the ACM/IEEE SC2003 conference (SC'03); 2003. p. 44–58.
[39] Karaboga D, Akay B, Ozturk C. Artificial bee colony (ABC) optimization algorithm for training feed-forward neural networks. In: Proceedings of the fourth international conference on modeling decisions for artificial intelligence; 2007. p. 318–29.
[40] Karaboga D, Basturk B. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. Journal of Global Optimization 2007;39:459–71.
[41] Chin Soon C, Hean L Malcolm Yoke, Iyer S Appa, Leng G Kheng. A bee colony optimization algorithm to Job shop scheduling. In: Proceedings of the winter simulation conference (WSC 06); 2006. p. 1954–61.
[42] Tovey C. The honey bee algorithm: a biological inspired approach to internet server optimization. The alumni magazine for ISyE at Georgia Institute of technology; 2004. p. 13–5.
[43] Wong L-P, Low MYH, Chong CS. Bee colony optimization with local search for traveling salesman problem. International Journal on Artificial Intelligence Tools 2010;19:305–34.
[44] Bitam S, Batouche M, Talbi EG. A survey on bee colony algorithms. In: Proceedings of 2010 IEEE international symposium on parallel and distributed processing, workshops and Ph.D. forum (IPDPSW); 2010. p. 1–8.
[45] Sang-Min P, Jair-Hoom K. Chameleon: a resource scheduler in a data grid environment. In: Proceedings of third IEEE international symposium on cluster computing and the grid (CCGRID'03); 2003. p. 258–65.