An SCP-based Heuristic Approach for Scheduling Distributed Data-Intensive Applications on Global Grids

Srikumar Venugopal * and Rajkumar Buyya

Grid Computing and Distributed Systems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, VIC 3010, Australia.

Abstract

Data-intensive Grid applications need access to large datasets that may each be replicated on different resources. Minimizing the overhead of transferring these datasets to the resources where the applications are executed requires that appropriate computational and data resources be selected. In this paper, we consider the problem of scheduling an application composed of a set of independent tasks, each of which requires multiple datasets that are each replicated on multiple resources. We break this problem into two parts: one, to match each task (or job) to one compute resource for executing the job and one storage resource each for accessing each dataset required by the job and two, to assign the set of tasks to the selected resources. We model the first part as an instance of the well-known Set Covering Problem (SCP) and apply a known heuristic for SCP to match jobs to resources. The second part is tackled by extending existing MinMin and Sufferage algorithms to schedule the set of distributed data-intensive tasks. Through simulation, we experimentally compare the SCP-based matching heuristic to others in conjunction with the task scheduling algorithms and present the results.

Key words: Grid Computing, Data-Intensive Applications, Task Mapping

* Corresponding author.
Email addresses: srikumar@csse.unimelb.edu.au (Srikumar Venugopal), raj@csse.unimelb.edu.au (Rajkumar Buyya).

(c) Elsevier 2008. J. of Parallel and Distributed Computing, Vol. 68, No. 4, pp: 471-487, ISSN: 0743-7315

1 Introduction

Grids [1] aggregate computational, storage and network resources to provide pervasive access to their combined capabilities. In addition, Data Grids [2,3] provide services such as low latency transport protocols and data replication mechanisms to distributed data-intensive applications that need to access, process and transfer large datasets stored in distributed repositories. Such applications are commonly used by communities of researchers in domains such as high-energy physics, astronomy and biology.

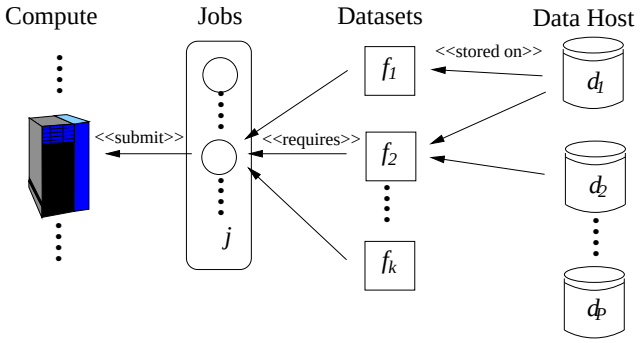


Fig. 1. Mapping Problem.

The work in this paper is concerned with scheduling data-intensive applications that can be considered as a collection of independent tasks, each of which requires multiple datasets, onto a set of Grid resources. An astronomy image-processing application following this model is described by Yamamoto, et al. [4]. Each task is translated into a job that is scheduled on to a computational resource and requests datasets from the storage resources (or *data hosts*). Each of these datasets may be replicated at several locations that are connected to each other and to the computational sites (or *compute resources*) through networks of varying capability. This scenario is illustrated in Figure 1. This paper makes two contributions: first, it introduces the problem of matching a task to a set of resources that consists of one compute resource for executing the job and a data host each to access each dataset required for the job and models this problem as an instance of the well-known Set Covering Problem (SCP) and second, it applies a known heuristic for SCP to perform the matching and evaluates it against other strategies in conjunction with MinMin and Sufferage task scheduling algorithms through extensive simulations.

The rest of the paper is structured as follows: the next section presents a detailed resource model and the application model that is targeted in the research presented in this paper. The mapping heuristic is presented in the following section and is succeeded by details of experimental evaluation and the consequent results. Finally, the related work is presented and the paper is

concluded.

2 Model

The target data-intensive computing environment is modeled based on existing production Grid testbeds such as the European DataGrid testbed [3] or the United States Grid3 testbed [5]. As an example, Figure 2 shows a subset of European DataGrid Testbed 1 derived from Bell, et. al [6]. The resources in the figure are spread across 7 countries and belong to different autonomous administrative domains.

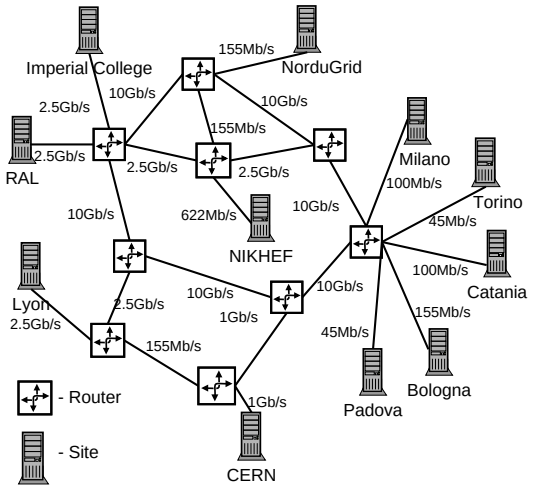


Fig. 2. European Data Grid Testbed 1 [6].

A data-intensive computing environment can be considered to consist of a set of M *compute resources*, $R = \{r_1, r_2, \ldots, r_M\}$ and a set of P *data hosts*, $D = \{d_1, d_2, \ldots, d_P\}$. Within production Grids, a compute resource is commonly a high performance computing platform such as a cluster consisting of processing nodes that are connected in a private local area network and are managed by a batch job submission system hosted at the "head" or "front-end" node connected to the public Internet. Jobs submitted to a cluster are assigned to processing nodes by the batch system or are held in queues. Such queues may have a limited capacity counted as the number of "slots" that can be filled by jobs. If all the slots in a queue are filled, further job submissions will not be allowed. A *data host* can be a dedicated storage resource such as a Mass Storage Facility connected to the Internet. At the very least, it may be a storage device attached to a compute resource in which case it inherits the network properties of the latter. It is important to note that even in the second case, the data host is considered separate from the compute resource. Figure 3 shows a simplified data-intensive computing environment consisting

of four compute resources and an equal number of data hosts connected by links of different bandwidths.

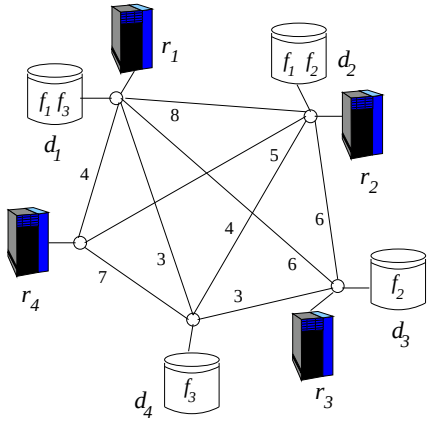


Fig. 3. A data-intensive environment.

The physical network between the resources consists of entities such as routers, switches, links and hubs. However, the model presented here abstracts the physical network to consider the logical network topology wherein each compute resource is connected to every data host by a distinct network link as shown in Figure 3. This logical link is denoted by $Link(r_m, d_p), r_m \in R, d_p \in D$. The bandwidth of the logical link between two resources is the bottleneck bandwidth of the actual physical network between the resources and is given by $BW(Link(r_m, d_p))$. This information may be obtained from various information sources such as the Network Weather Service [7]. The numbers alongside the links in Figure 3 depict the bandwidths of the various logical links in the network. The time taken by a compute resource to access data through the Internet is assumed to be an order of magnitude higher than that taken for it to access data on a storage resource at the same site (either a separate machine or a simple disk storage). Therefore, only remote access times are taken into account in the model and datasets on local storage have zero access times.

Data is organised in the form of datasets. A dataset can be an aggregated set of files, a set of records or even a part of a large file. Datasets are replicated on the data hosts by a separate replication process that follows a strategy such as one of those described by Bell, et al. [6] which takes into consideration various factors such as locality of access, load on the data host and available storage space. Information about the datasets and their location is available through a catalog such as the Storage Resource Broker Metadata Catalog [8].

The application is composed of a set of N atomic (indivisible) and non-interdependent jobs, $J = \{j_1, j_2, \ldots, j_N\}$ (Note: Since each task is translated into a job, tasks and jobs are used interchangeably). Typically, $N \gg M$, the number of compute resources. Each job, $j \in J$, requires a set of K datasets,

denoted by F^j, that are distributed on a subset of D. Specifically, for a dataset $f \in F^j$, $D_f \subseteq D$ is the set of data hosts (each denoted by d_f) on which f is replicated and from which it is available. Also, D_{f_1} and D_{f_2} need not be pairwise disjoint for every $f_1, f_2 \in F$. In other words, a data host can serve multiple datasets at a time.

Each job requires one processor in a compute resource for executing the job and one data host each for accessing each of the K datasets required by the job. The compute resource and the data hosts thus selected are collectively referred to as the *resource set* associated with the job and is denoted by $S^j = \{R^j, D^j\}$ where $R^j = \{r\}, r \in R$ represents the compute resource selected for executing the job and $D^j \subseteq \bigcup_{f \in F^j} D_f$ is the set of data hosts chosen for accessing the datasets required by the job.

The job execution time model followed here is extended from that presented by Maheswaran, et. al [9]. Consider a job j that has been submitted for execution to a compute resource r. The time spent in waiting in the queue on the compute resource is denoted by $T_w(j, r)$ and the expected execution time of the job is given by $T_e(j, r)$. T_w increases with increasing load on the resource. Likewise, T_e is the time spent in purely computational operations and depends on the processing speed of the individual nodes within the compute resource. For each dataset $f \in F^j$, the time required to transfer f from d_f to r is given by

$$T_t(f, d_f, r) = Response\_time(d_f) + Size(f)/BW(Link(d_f, r))$$

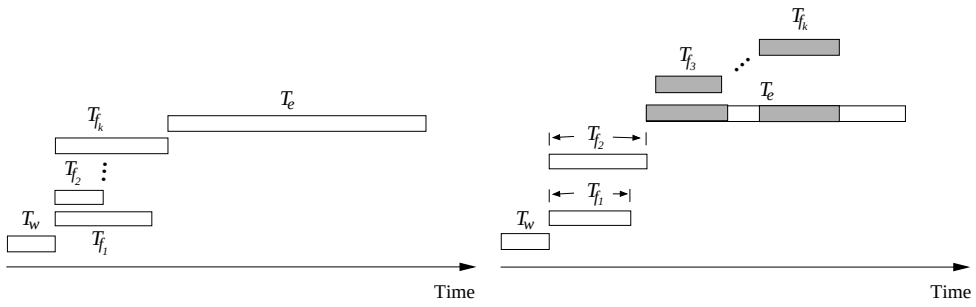
$Response\_time(d_f)$ is the difference between the time when the request was made to d_f and the time when the first byte of the dataset f is received at r. This is an increasing function of the load on the data host. The *estimated completion time* for the job, $T_{ct}(j)$, is the wallclock time taken for the job from submission till eventual completion and is a function of these three times. Figure 4 shows two examples of data-intensive jobs with times involved in various stages shown along a horizontal time-axis. In this figure, for convenience, the time for transferring $f_1, f_2, \ldots, f_k$ is denoted by $T_{f_1}, T_{f_2}, \ldots, T_{f_k}$ respectively.

The impact of the transfer time of the datasets is dependent on the manner in which the dataset is processed by the job. For example, Figure 4(a) shows a common scenario in which Grid applications request and receive the required datasets in parallel before starting computation. In this case,

$$T_{ct}(j) = T_w(j, r) + \max_{f \in F^j}(T_t(f, d_f, r)) + T_e(j, r)$$

However, the number of simultaneous transfers on a link determines the bandwidth available for each transfer and therefore, the T_t.

Figure 4(b) shows a more generic data processing approach in which some of the datasets are transferred completely prior to execution and the rest are



(a) (b)

Fig. 4. Job Execution Stages and Times (Gray areas denote overlaps between the computation and data operations).

accessed as streams during the execution. The grey areas show the overlap of computation and communication. In this case, the transfer time of the streamed data is masked by the computation time of the application. However, data access still affects the performance of the application. If there is a latency associated with accessing the data, the application may still have to wait until the first byte of the data is received at the compute resource.

This paper focuses on the application models of the first type, that is, applications that require all the datasets to be transferred to the actual compute resource (or its associated data host) before execution. This is the most common model followed by data-intensive applications [10]. Also, the impact of data transfer time is the highest in this model. However, it is possible that lessons learnt from scheduling these type of applications may also be applicable to the other types of data-intensive applications.

2.1 A Generic Scheduling Algorithm

The scheduling paradigm followed is that of *offline* or *batch mode* scheduling of a set of independent tasks [9]. The general problem of creating a schedule for a set of jobs to run on distributed resources is called *list scheduling* and is considered to be *NP*-complete [11] in the general case. Many approximate heuristics have been devised for this problem and a short survey of these have been presented by Braun, et al. [11]. Algorithm 1 outlines a generic strategy for batch mode scheduling of a set of jobs based on the skeleton presented by Casanova, et al. [12].

The scheduler forms a part of a larger application execution framework such as a Grid resource broker (e.g.[13],[14]). The resource broker is able to identify resources that meet minimum requirements of the application such as architecture (instruction set), operating system, storage threshold and data access

Algorithm 1: A Generic Scheduling Algorithm.

```
1  while there exists unsubmitted jobs do
2      Update the resource performance data based on job scheduled in previous
       intervals
3      Update network data between resources based on current conditions
4      foreach unsubmitted job do
5          Match the job to a resource set to satisfy the objective function at the
           job level
6          Order the jobs depending on the overall objective
7      end
8      repeat
9          Assign mapped jobs to each compute resource heuristically
10     until all jobs are submitted or no more jobs can be submitted
11     Wait until the next scheduling event
12 end
```

permissions and these are provided as suitable candidates for job execution to the scheduler. The scheduling is carried out at time intervals called *scheduling events* [15]. These events can be determined to either run at regular intervals (*poll-based*) or in response to certain conditions (*event-based*). There are two parts in a scheduling strategy: mapping and dispatching. The jobs have to be *matched* to a set of resources and ordered depending on the objective function (*mapping*) and then sent to remote resources for execution (*dispatching*). Each of the parts can be implemented independently and therefore, many strategies are possible.

The sections that follow concentrate on matching jobs to distributed resources where the selection of computational and data resources are interdependent. The aim of the matching heuristic is to select a resource set that produces the Minimum Completion Time (MCT) for a job. The general strategy adopted here is to find a resource set with the least number of data hosts required to access the datasets required for a job and then, find a suitable compute resource to execute it. The goal here is to maximise the local access of datasets and thus, reduce the data transfer times.

3 A Graph-based Approach to the Matching Problem

For a job $j \in J$, consider a graph $G^j = (V, E)$ where $V = (\bigcup_{f \in F^j} \{D_f\}) \cup F^j$ and E is the set of all directed edges $\{d, f\}$ such that $d \in D_f$. Figure 5(a) shows an example of a job j that requires 3 datasets f_1, f_2 and f_3 that are replicated on data host sets $\{d_1, d_2\}$, $\{d_2, d_3\}$ and $\{d_1, d_4\}$ respectively. The graph of data sets and data resources for job j is shown in Figure 5(b).

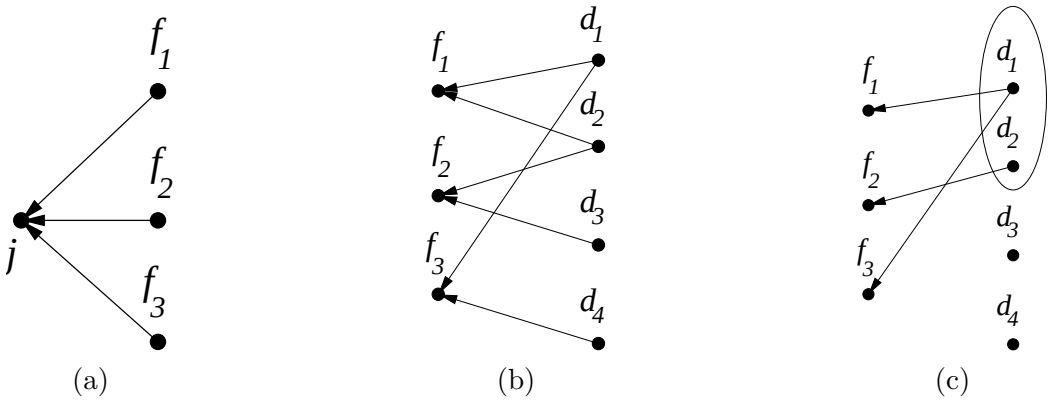


Fig. 5. Graph-based approach to the matching problem. (a)Job j dependent on 3 datasets. (b) Directed graph of data resources and data sets for job j. (c) A possible minimal set for the data graph.

The intuition followed in this work is to assign a job to a compute resource that is "closest" (in network terms) to a set of data hosts that contain the datasets required by the job. The selection of the compute resource, however, should not only be based on the proximity of the data but also on its availability and performance as well. In terms of the graph model presented, the resource set should therefore contain a set H of data hosts such that there exists atleast one edge from a member of H to f for every $f \in F^j$ in G^j so that all the datasets required for the job can be accessed. Figure 5(c) shows a possible set for the graph of datasets and data hosts for job j shown in Figure 5(b). There may be upto P^K possible sets of data hosts (as there may be upto P data hosts for each of the K datasets associated with a job) that can be combined with each compute resource in R to produce MP^K resource sets (where M is the number of compute resources) with different values of total completion time. Out of these, a combination of a set of data hosts and a compute resource is to be selected such that the total completion time for j is minimised. This problem is defined and referred to hereafter as the **M**inimum **R**esource **S**et (MRS) problem.

3.1 Modelling the Minimum Resource Set as a Set Cover

For a graph G^j such as that shown in Figure 5(b), a reduced adjacency matrix $A = [a_{ik}], 1 \leq i \leq P, 1 \leq k \leq K$ can be constructed wherein $a_{ik} = 1$ if data host $d_i \in D_{f_k}$ for a dataset f_k. Such an adjacency matrix is shown in Figure 6(a). The rows that contain a 1 in a particular column are said to "cover" the column. The problem of finding the minimal set of data hosts to access all datasets in G^j is now equivalent to finding the set of the least number of rows such that every column is covered, that is, every column contains an entry of 1 in at least one of the rows. In other words, if each data host can

be considered as a set of datasets, then finding the minimal set of data hosts is equivalent to finding the least number of such sets of datasets such that all datasets are covered. This problem has been studied extensively as the *Set Covering Problem (SCP)* [16].

$$\begin{array}{c} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \end{array} \begin{array}{c} \begin{array}{ccc} f_1 & f_2 & f_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

(a)

$$\begin{array}{c} \\ d_1 \\ d_2 \\ - \\ d_2 \\ d_3 \\ - \\ d_1 \\ d_4 \end{array} \begin{array}{c} \begin{array}{ccc} f_1 & f_2 & f_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ - & - & - \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ - & - & - \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

(b)

Fig. 6. (a) Adjacency Matrix for the job example. (b) Tableau.

The SCP is an *NP-complete* problem and the most common approximation algorithm applied to the SCP is the greedy strategy [17]. It is possible to derive a set cover for the datasets by following the greedy strategy as outlined below:
Step 1. Repeat until all the datasets have been covered.
Step 2. $\hookrightarrow$ Pick the data host that has the maximum number of uncovered datasets and add it to the current candidate set.
It can be seen that such a greedy strategy will produce only one of the possible set covers. This, however, excludes the other candidate sets from consideration. It is also possible to arrive at an exhaustive search procedure that generates all the possible covers. However, this is bound to be computationally-intensive.

The next subsection details a heuristic for matching jobs to resources based on the approximate tree search algorithm provided by Christofides [18] for the SCP. This heuristic restricts the search to a region where the solution is most likely to be found. But, before applying that algorithm, it is possible to reduce the size of the problem by taking advantage of the nature of the SCP. These reductions are as follows:

(1) If a dataset required for a job is present on only one data host, then that data host is part of any solution. Therefore, the problem can be reduced by assigning the dataset to that data host and removing the dataset from later consideration.
(2) For $f_1, f_2 \in F^j$, if $D_{f_1} \subseteq D_{f_2}$, then f_2 can be removed from consideration as any solution that covers f_1 must also cover f_2.

3.2 The SCP Tree Search Heuristic

Algorithm 2 outlines the SCP tree search heuristic that consists of three distinct phases: initialisation, execution and termination. These are described in the following paragraphs. Figure 7 shows an example of the heuristic in action based on the job shown in Figure 5 using the tableau in Figure 6(b) and the platform in Figure 3. At the bottom of each step, the candidate resource set arrived at is depicted as well.

Algorithm 2: SCP Tree Search Matching Heuristic.

Begin Main
1 For a job j, create the adjacency matrix A with data hosts forming the rows and datasets forming the columns
2 Sort the rows of A in the descending order of the number of 1's in a row
3 Create the tableau T from sorted A and begin with initial solution set $B_{final} = \phi$, $B = \phi$, $E = \phi$ and $z = \infty$
4 `Search(`B_{final}, B, T, E, z`)`
5 $S^j \leftarrow \{\{r\}, B_{final}\}$ where $r \in R$ such that S^j produces `MCT` (B_{final})
End Main

`Search(`B_{final}, B, T, E, z`)`
6 Find the minimum k, such that $f_k \notin E$. Let T_k be the block of rows in T corresponding to f_k. Set a pointer q to the top of T_k.
7 **while** *q does not reach the end of T_k* **do**
8 $F_T \leftarrow \{f_i | t_{qi} = 1, 1 \leq i \leq K\}$
9 $B \leftarrow B \cup \{d_q^k\}, E \leftarrow E \cup F_T$
10 **if** $E = F^j$ **then**
11 **if** $z >$ `MCT` *(B)* **then**
12 $B_{final} \leftarrow B, z \leftarrow$ `MCT` (B)
13 **else** `Search(`B_{final}, B, T, E, z`)`
14 $B \leftarrow B - \{d_q^k\}, E \leftarrow E - F_T$
15 Increment q
16 **end**

`MCT(`B`)`
17 Find $r \in R$ such that the completion time is minimum for the resource set $S^j = \{\{r\}, B\}$ and return value

Initialisation (Lines 1-3)

The initialisation starts off with the creation of the adjacency matrix A for a job. The rows of this matrix (that is, the data hosts) are then sorted in the descending order of number of 1's per column (or, the number of datasets contained). This sorted matrix is used to create an augmented matrix that is henceforth referred to as the *tableau* and is shown in Figure 6(b). The tableau

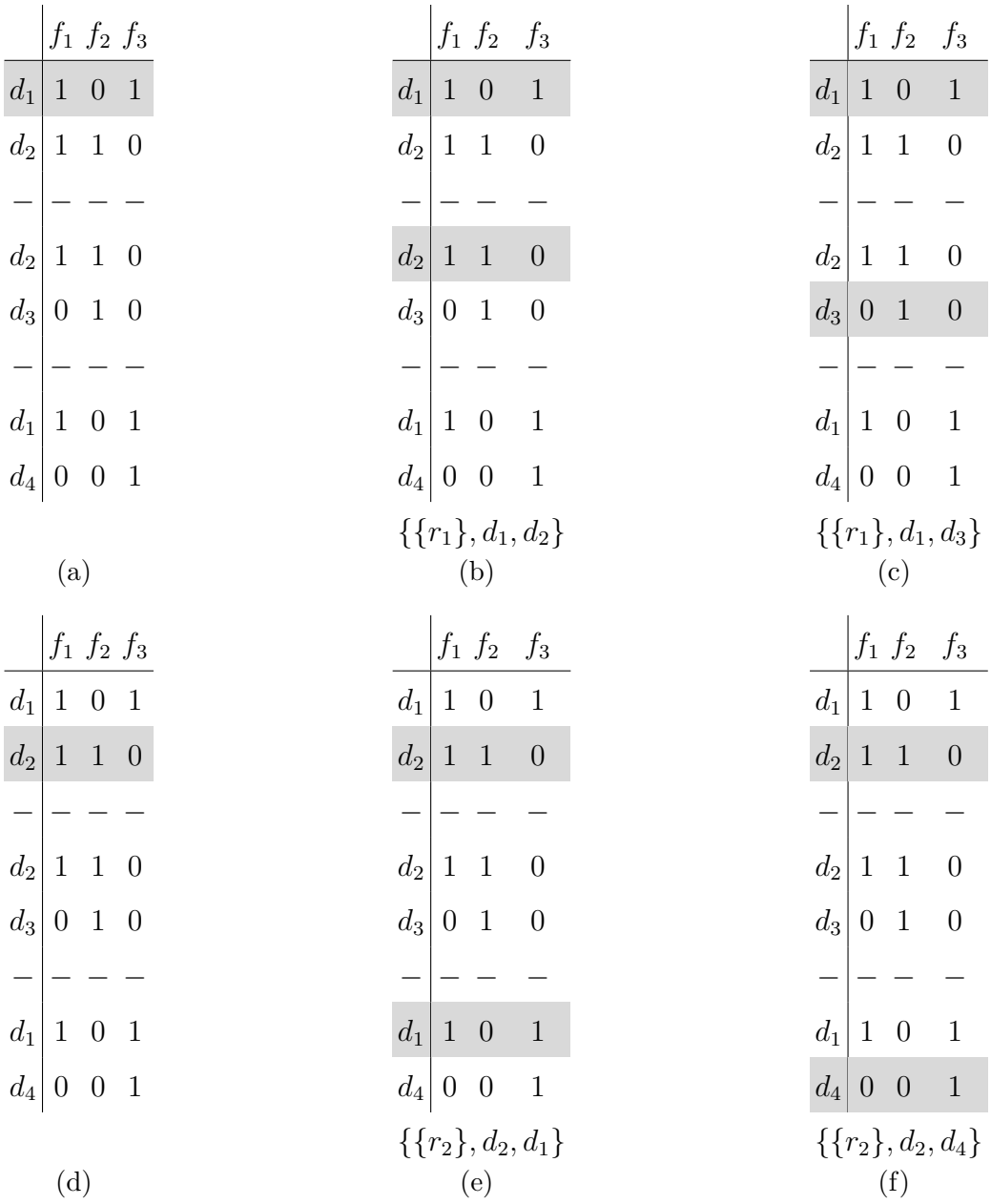


Fig. 7. Example of the SCP Tree Search Heuristic in action.

T consists of K blocks of rows (delineated by dashes in Figure 6(b)), where K is the size of F^j and the $k^{th}(1 \leq k \leq K)$ block consists of rows corresponding to data hosts that contain $f_k, f_k \in F^j$. The tableau is constructed in such a manner that the rows within each block are in the same sorted order as the rows in the sorted adjacency matrix. At any stage of execution, the set of data hosts B keeps track of the current solution set of datahosts, the set E contains the datasets already covered by the solution set and the variable z keeps track of the value of the completion time offered by the current solution set. The final solution set is stored in B_{final}. The procedure begins with the partial solution set $B = \phi$, $E = \phi$, $z = \infty$.

Execution (Lines 6-16)

During execution, the blocks are searched sequentially starting from the k^{th} block in T where k is the smallest index, $1 \leq k \leq K$ such that $f_k \notin E$. Within the k^{th} block, let d_q^k mark the data host under consideration where q is a row pointer within block k. The data host d_q^k is added to B and all the datasets for which the corresponding row contains 1 are added to E as they are already covered by d_q^k. These datasets are removed from consideration and the process then moves to the next uncovered block until $E = F^j$, that is, all the datasets have been covered. At this point, B represents the corresponding candidate set of data hosts that covers all the datasets. The function `MCT`(B) computes the expected value of the completion time for each compute resource combined with B and returns with the minimum of the values so found. If this is lower than the existing value in z, then the solution set is replaced with the current candidate set and z is assigned the returned value.

Whenever the heuristic enters a block that is not yet covered, it branches out within the block by a recursive call that passes along the incomplete solution set (line 13). The final solution set is returned in the variable B_{final} through normal pass-by-reference methods. At the end of each loop, the heuristic backtracks to try the next data host in the block and repeat the branching with that host (line 14).

Illustrating this using Figure 7, in the first step (Figure 7(a)), the heuristic starts with the first block in the tableau. As f_1 and f_3 are covered by choosing d_1, the heuristic moves to the second block to cover f_2. This gives us $B = \{d_1, d_2\}$ to cover all the datasets. For this B, r_1 produces the lowest value of MCT and therefore, the candidate resource set (or the resource set matched to the job) at this moment is $\{\{r_1\}, d_1, d_2\}$(Figure 7(b)). The heuristic then backtracks and moves to the next row in the same block to produce $B = \{d_1, d_3\}$ and the consequent candidate resource set $\{\{r_1\}, d_1, d_2\}$(Figure 7(c)). The latter is then compared to the previous resource set and the B with lowest MCT is selected for the next iteration. In a similar fashion (Figures 7(d)- 7(f)), the rest of the resource sets are discovered and a final resource set selected.

Termination (Line 5)

Through the recursive procedure outlined in the listing, the heuristic then backtracks and discovers other candidate sets. The solution set that guarantees minimum makespan is then chosen as the final solution set. The compute resource that provides the MCT is then combined with the solution set to obtain the resource set for the job.

To reduce the scope of the tree traversal, the heuristic terminates when the first block is exhausted. The data hosts with the maximum number of datasets appear at the top of the tableau due to the initialisation process. Therefore,

most of the candidate sets will be covered by the search function by starting at the rows in the first block.

4 Other Approaches to the Matching Problem

Algorithm 3: The Compute-First Matching Heuristic.

1. **foreach** $j \in J$ **do**
2. Let $S^j \leftarrow \{R^j, D^j\},\ R^j \leftarrow \phi,\ D^j \leftarrow \phi$
3. Let $R^j \leftarrow \{r_{final}\}$ such that $T_e(j, r_{final})$ is minimum for all $r \in R$
4. **foreach** $f \in F^j$ **do**
5. $D^j \leftarrow D^j \cup \{d_f\}$ where $T_t(f, d_f, r_{final})$ is minimum for all $d_f \in D_f$
6. **end**
7. **end**

Compute-First - In this mapping strategy, listed in Algorithm 3, the compute resource that provides the least execution time is selected first. This step is followed by choosing data hosts that have the highest bandwidths (and therefore, the lowest transfer times) to the selected compute resource. The running time of this heuristic is $O(MKP)$.

Algorithm 4: The Exhaustive Search Matching Heuristic.

1. **foreach** $j \in J$ **do**
2. Let $S^j \leftarrow \{R^j, D^j\},\ R^j \leftarrow \phi,\ D^j \leftarrow \phi$
3. Let $U \leftarrow R \times D_{f_1} \times D_{f_2} \times \ldots \times D_{f_K}$ where $f_1, f_2, \ldots, f_K \in F^j$
4. Find $u \in U$ such that $T_{ct}(j)$ is minimum
5. **end**

Exhaustive Search - Algorithm 4 lists the exhaustive search strategy wherein all the possible resource sets for a particular job are generated and the one guaranteeing the least completion time is chosen for the job. While this heuristic guarantees that the resource set selected will be the best for the job, it searches through MP^K resource sets at a time. This leads to unreasonably large search spaces for higher values of K. For example, for a job requiring 5 datasets with 20 possible data hosts and 20 available compute resources, the search space will consist of $(20 * 20^5) = 64 * 10^6$ resource sets.

Greedy Selection - This strategy, listed in Algorithm 5, builds the resource set by iterating through the list of datasets and making a greedy choice for the data host for accessing each dataset, followed by choosing the best compute resource for that data host. At the end of each iteration, it checks whether the compute resource so selected is better than the one selected in previous iteration when the data hosts selected in previous iterations are considered.

Algorithm 5: Greedy Selection Strategy.

1. **foreach** $j \in J$ **do**
2. Let $S^j \leftarrow \{R^j, D^j\},\ R^j \leftarrow \phi,\ D^j \leftarrow \phi$
3. Let $R^j_{temp} \leftarrow \phi$ `// A temporary variable`
4. **foreach** $f \in F^j$ **do**
5. Let $U \leftarrow \{(d_f, r)\}_{d_f \in D_f}$ where r is the first element of ordered set R_{d_f}
6. Find (d_f, r) such that $T_t(f, d_f, r) + T_e(j, r)$ is minimum over U
7. **if** $S^j = \{\phi, \phi\}$ **then**
8. $R^j \leftarrow \{r\}, D^j \leftarrow \{d_f\}, R^j_{temp} \leftarrow \{r\}$
9. **else**
10. $R^j \leftarrow \{r\}, D^j \cup \{d_f\}$
11. $S^j \leftarrow \min\{\{R^j, D^j\}, \{R^j_{temp}, D^j\}\}$
12. $R^j_{temp} \leftarrow R^j$
13. **end**
14. **end**

This heuristic was presented by the authors [19] for deadline and budget constrained cost and time minimisation scheduling of distributed data-intensive applications. The running time of this heuristic is $O(MKP)$.

5 Scheduling Heuristics

While the mapping heuristic finds a resource set for a single job, the overall objective is to minimize the total *makespan* [9], the total time from the start of the scheduling to the completion of the last job, of the application consisting of N such data-intensive jobs. To that end, we apply the well-known MinMin and Sufferage heuristics, proposed by Maheswaran,et.al [9], for dynamic scheduling of jobs on heterogeneous computing resources. These have been extended to take into account the distributed data requirements of the target application model.

Algorithm 6 outlines the extended MinMin scheduling heuristic. The basic idea of this heuristic is to find the job that has the least value of completion time among all the jobs and allocate it to the resource set that achieves it. The intuition behind this is that such an allocation over all the jobs will minimize the overall completion time. The term J_U denotes the set of jobs that have not been allocated to any resource set yet. In the beginning, it matches all the jobs to a resource set that guarantees the MCT for that job (line 4). This is produced through matching heuristics such as the SCP Tree Search, Greedy Selection, Compute-First or Exhaustive Search, that have been presented in previous sections. Then, the job with the MCT in the present

Algorithm 6: The MinMin Scheduling heuristic extended for distributed data-intensive applications.

1. **repeat**
   Begin Mapping
2. **repeat**
3. **foreach** $j \in J_U$ **do**
4. Find the resource set that achieves the MCT for j
5. **end**
6. Find the job $j \in J_U$ with the least value of $T_{ct}(j)$
7. Assign j to its selected resource set and remove it from J_U
8. Update the resource availability based on the allocation performed in the previous step
9. **until** J_U *is empty*
   End Mapping
10. Dispatch the mapped jobs to the selected resources such that the job allocation limit of each resource is not exceeded
11. Wait until the next scheduling event
12. **foreach** *job completed in the previous interval* **do**
13. For each dataset that has been transferred from a remote data host for the job, add its eventual destination (compute resource) as a future source of the dataset for the jobs remaining in J_U
14. **end**
15. For each resource, revise its capability estimates (job allocation limit or available queue slots) depending on various information sources such as external performance monitors or the jobs completed in the previous interval
16. **until** *all jobs are completed*

allocation is assigned to the compute resource in its chosen resource set (line 7). This job is then removed from the unallocated job set. As job assignment changes the availability of the compute resource with respect to the number of available slots/processors, the resource information is updated and the process is repeated until all the jobs in J_U have been allocated to some resource set.

For each compute resource, the dispatching function (line 10) submits the jobs mapped to it to the remote job management system (or the job queue) until all the slots on the queue have been filled or the jobs exhausted. The remaining jobs that were assigned to the compute resource but were not able to be allocated to the remote queues, are returned back to the unallocated jobs list. The scheduler then waits until the next scheduling event to resume.

When a job is scheduled for execution on a compute resource, all the datasets that are required for the job and are not available local to the resource, are transferred to the resource prior to execution. These datasets become replicas that can be used by following jobs. Here, this is taken into account by registering the compute resource in question (or its associated data host) as

Algorithm 7: The Sufferage heuristic extended for distributed data-intensive applications.

```
    Begin Mapping
1.  repeat
2.      foreach j ∈ J_U do
3.          Find the resource set that achieves the MCT for j
4.          Find the second best completion time for j
5.          sufferage value = second best value - best value
6.      end
7.      Find the job j ∈ J_U with the maximum sufferage value
8.      Assign j to its selected resource set and remove j from J_U
9.      Update the resource availability based on the allocation performed in
        the previous step
10. until J_U is empty
    End Mapping
```

a source of the transferred datasets for succeeding allocation loops (line 13). This enables exploiting both temporal and spatial locality of data access.

The motivation behind the Sufferage heuristic (listed in Figure 7) is to allocate a resource set to a job that would be disadvantaged the most (or "suffer" the most) if that resource set were not allocated to it. This is determined through a sufferage value computed as the difference between the second best and the best value of the completion time for the job.

For each job, the resource set that offers the least value of the completion time is determined through the same mechanisms as that in MinMin. Then, the compute resource in that resource set is removed from consideration and the matching function is rerun to provide another minimal resource set with the next best value for the completion time. The selection of the compute resource determines both the execution time (T_e) and the data transfer times (T_t) . Therefore, removing it from consideration will produce the maximum impact on the value of the completion time. After determining the sufferage value for each job, the job with the largest sufferage value is then selected and assigned to its chosen resource set. The rest of the heuristic including dispatching and updating of compute resource and data host information proceeds in the same manner as MinMin.

6 Evaluation of Scheduling Algorithms

Effective evaluation of scheduling algorithms requires the study of their performance under different scenarios such as different user inputs and varying resource conditions. Within Grid environments, resource loads and the number

of users vary continuously and the spread of resources among different administrative domains makes it nearly impossible to control the environment to provide a stable configuration for evaluation. Furthermore, the network plays a large role in the performance of scheduling algorithms for data-intensive applications and it is impossible to create consistent conditions over public networks. The scale of the evaluation is also limited by the number of Grid resources that can be accessed.

Therefore, it was decided to evaluate the performance of algorithms on a simulated Grid environment to ensure a stable and repeatable configuration. Simulation has been used extensively for modelling and evaluation of distributed computing systems and the popularity of this methodology for evaluation of Grid scheduling algorithms have led to the availability of several Grid simulation packages [20]. Some of the simulation systems available for data-intensive computing environments such as Data Grids include GridSim [21], MONARC simulator [22], OptorSim [6], ChicSim [23] and SimGrid [24]. GridSim enables modelling and simulation of heterogeneous Grid resources with time-shared and space-shared node allocation and different economic costs; Grid networks with different routing topologies and QoS classes [25]; and Data Grid replica catalogs that can be connected in different configurations [26]. Also, it presents itself as a toolkit that allows creation of different applications such as resource brokers having scheduling algorithms with different objectives. Hence, GridSim was used as the simulation system for evaluating the scheduling algorithms for distributed data-intensive applications. Evaluation of the scheduling algorithms in GridSim required modelling of Grid resources, their interconnections and the data-intensive applications. The sections that follow describe in detail how each of these were modeled.

6.1 Simulated Resources

The testbed modelled in this evaluation is shown in Figure 2. The modelled testbed contains 11 resources spread across 6 countries connected via high capacity network links. Each resource, except the one at CERN (Geneva), was used both as a compute resource and as a data host. The resource at CERN was used as a pure data source (data host) in the evaluation and therefore, no jobs were submitted to it for execution. The resources in the actual testbed have gone through several configuration changes, not all of which are publicly available, and hence it was impossible to model their layout and CPU capability accurately. Instead, it was decided to create a configuration for each resource such that the modelled testbed, in whole, would reflect the heterogeneity of platforms and capabilities that is normally the characteristic of Grids. All the resources were simulated as clusters of single CPU nodes or Processing Elements (PEs) with a batch job management system using

Table 1
Resources within EDG testbed used for evaluation.

Resource Name (Location)	**No. of Nodes**	**Single PE Rating (MIPS)**	**Storage (TB)**	**Mean Load**
RAL (UK)	41	1140	2.75	0.9
Imperial College (UK)	52	1330	1.80	0.95
NorduGrid (Norway)	17	1176	1.00	0.9
NIKHEF (Netherlands)	18	1166	0.50	0.9
Lyon (France)	12	1320	1.35	0.8
CERN (Switzerland)	–	–	12	–
Milano (Italy)	7	1000	0.35	0.5
Torino (Italy)	4	1330	0.10	0.5
Catania (Italy)	5	1200	0.25	0.6
Padova (Italy)	13	1000	0.05	0.4
Bologna (Italy)	20	1140	5.00	0.8

space-shared policy. This modelled real world Grid resources that are generally high performance clusters in which each job is allocated to a processing node through a job submission queue. The processing capabilities of the PEs were rated in terms of Million Instructions Per Sec (MIPS) so that the application requirements can be modelled in Million Instructions (MI). The configuration assigned to the resources in the testbed for the simulation are listed in Table 1.

To model resource contention caused by multiple users submitting jobs simultaneously and the resultant variation in resource availability, a *load factor* was associated with each resource. The load factor is simply the ratio of the number of PEs that are occupied to the total number of PEs available in a resource. During simulation, the instantaneous load (or number of PEs occupied) for each resource was derived from a Gaussian distribution centered around its mean load factor shown in Table 1.

Storage at the resources was modelled as the total disk capacity available at the site. Site access latencies such as disk read time were ignored as these are less than the network delays by an order of magnitude. The network between the resources were modelled as the set of routers and links shown in Figure 2. Variations of the available network bandwidth are simulated by associating a link load factor, which is the ratio of the available bandwidth to the total

bandwidth for a network link. During simulation, the instantaneous measure of the link load is derived from another Gaussian distribution centered around a mean load assigned at random, at the start of the simulation, to each of the links.

It was possible to keep track of the various load variations through information services built into the simulation entities. For example, it was possible to query the instantaneous bandwidth of the network link between any two resources. It was also possible to determine resource availability information by querying the resource for its instantaneous load and number of PEs available.

6.2 Distribution of Data

A universal set of 1000 datasets was used for this evaluation. Studies of similar environments [27] have shown that the size of the datasets follow a heavy-tailed distribution in which there are larger numbers of smaller size datasets and vice versa. Therefore, the set of datasets are generated with sizes distributed according to the logarithmic distribution in the interval $[1GB, 6GB]$. The distribution of datasets in a Data Grid depends on many factors including variations in popularity, the replication strategy employed and the nature of the Grid fabric. To model this distribution, at the start of the simulation, each of the datasets were replicated on one or more of the data hosts according to a preset pattern of dataset distribution. Two common patterns of data distribution considered in this evaluation are given below:

- *Uniform :* Here, the distribution of datasets is modelled on a uniform distribution. Here, each dataset is equally likely to be replicated at any site.
- *Zipf :* Zipf-like distributions follow a power law model in which the probability of occurrence of the i^{th} ranked dataset in a list of datasets is inversely proportional to i^{-a} where $a \leq 1$. In other words, a few datasets are distributed widely whereas most of datasets are found in one or two places. This models a scenario where the datasets are replicated on the basis of popularity. It has been shown that Zipf-like distributions holds true in cases such as requests for pages in World Wide Web where a few of the sites are visited the most [28]. This scenario has been evaluated for a Data Grid environment in related publications [29].

Henceforth, the distribution applied is described by the variable *Dist*. The distribution of datasets was also controlled through a parameter called the *degree of replication* which is the maximum possible number of replicas of any dataset present in the Data Grid at the beginning of the simulation. For example, a degree of replication of 3 means there can be up to 3 copies of

any dataset on the Grid resources. However, not all datasets are replicated to the limit of the degree of replication. In a uniform distribution, a higher percentage of the datasets are replicated up to the maximum limit than in the Zipf distribution. The degree of replication in this evaluation is 5.

6.3 Application and Jobs

The simulated application models a Bag-of-Task application that can be converted into a set of independent jobs. The size of the application was determined by the number of jobs in the set (or N). Each job translates to a Gridlet object which is the smallest unit of execution in GridSim. The computational size of a job or the job length, described by the term $Size$, is expressed in terms of the time taken to run the job on a standard PE with a MIPS rating of 1000. That is, a job with length 100,000 MI runs for 100 seconds on a standard resource. Each job requires as input, a pre-determined number of datasets (or K datasets) selected at random from the universal set of datasets. For the purpose of comparison, K is kept a constant among all the jobs in a set although this is not a condition imposed on the heuristic itself.

An experiment is an execution of the all the heuristics for an application while keeping the values for these parameters constant, and is therefore described by the tuple $(N, K, Size, Dist)$. At the beginning of each experiment, the set of datasets, their distribution among the resources, and the set of jobs are generated. This configuration is then kept constant while each of the scheduling heuristics are evaluated in turn. To keep the resource and network conditions repeatable among evaluations, a random number generator is used with a constant seed. The evaluation is conducted with different values for $N, K, Size$ and $Dist$ to study the performance under different input conditions.

7 Experimental Results

7.1 Comparison between the Matching Heuristics

The performances of the matching heuristics discussed in the previous section were compared with each other by pairing each of them with the MinMin heuristic and conducting 50 simulation experiments with different values for N, K, $Size$ and $Dist$. Throughout this section, *SCP* and *Greedy* refer to the SCP Tree Search and the Greedy Selection heuristics presented in the previous section respectively. The objective of this evaluation was to reduce

the *makespan* [9] of the application which is the total wallclock time between the submission of the first job to the completion of the last job in the set.

Table 2
Summary of Simulation Results.

Mapping Heuristic	**Geometric Mean**	**Avg. deg. (SD)**	**Avg. rank (SD)**
Compute-First	37593.71	69.01 (19.4)	3.63 (0.48)
Greedy	36927.44	71.86 (50.55)	3.23 (0.71)
SCP	24011.17	7.68 (10.42)	1.67 (0.6)
Exhaustive Search	23218.49	3.87 (6.46)	1.47 (0.58)

The results of the experiments are summarised in Table 2 and are based on the methodology provided by Casanova, et. al [12]. For each matching heuristic, the table contains three values:

(1) *Geometric Mean* of the makespans: The geometric mean is used as the makespans vary in orders of magnitude depending on parameters such as number of jobs per application set, number of files per job and the size of each job. The lower the geometric mean, the better the performance of the heuristic.
(2) Average degradation (*Avg. deg.*) from the best heuristic: In an experiment, the degradation of a heuristic is the difference between its makespan and the makespan of the best heuristic for that experiment and is expressed as a percentage of the latter measure. The average degradation is computed as an arithmetic mean over all experiments and the standard deviation of the population is given in the parentheses next to the means in the table. This is a measure of how far a heuristic is away from the best heuristic for an experiment. A lower number for a heuristic certainly means that on an average that heuristic is better than the others.
(3) Average rank (*Avg. rank*) of each heuristic in an experiment: The ranking is in the ascending order of makespans produced by the heuristics for each experiment, that is, the lower the makespan, the lower the rank of the heuristic. The average rank is calculated over all the experiments and the standard deviation is provided alongside the averages in parantheses.

The three values together provide a consolidated view of the performance of each heuristic. For example, it can be seen that on average Compute-First and Greedy both perform worse than either SCP or Exhaustive Search. However, the standard deviation of the population is much higher in the case of Greedy than that of Compute-First. Therefore, Compute-First can be expected to perform as the worst heuristic most of time. Indeed, in a few of the experiments, Greedy performed as good or even better than SCP while Compute-First never

came close to the performance of the other heuristics.

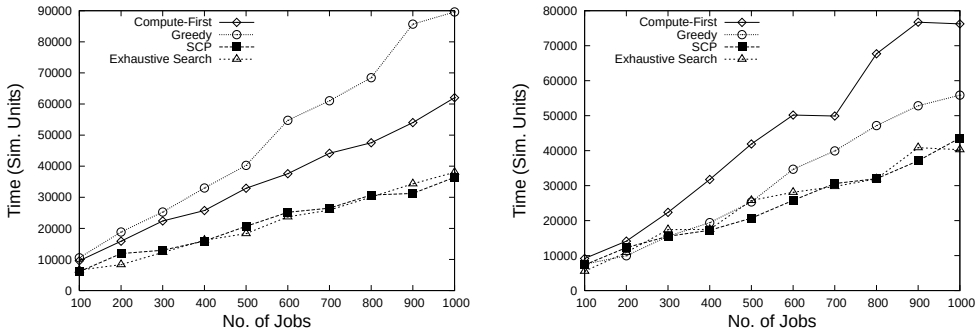
As is expected, between SCP and Exhaustive Search, the latter provides the better results by having a consistently lower score than the former. However, the nature of Exhaustive Search means that as the number of datasets per job increases, the number of resource sets that need to be considered by the heuristic increases dramatically. The geometric mean and average rank of SCP is close to that of Exhaustive Search heuristic. The average rank is less than 2 for both heuristics which implies that in many scenarios, SCP provides a better performance than Exhaustive Search.

7.1.1 Impact of Data Transfer on Performance

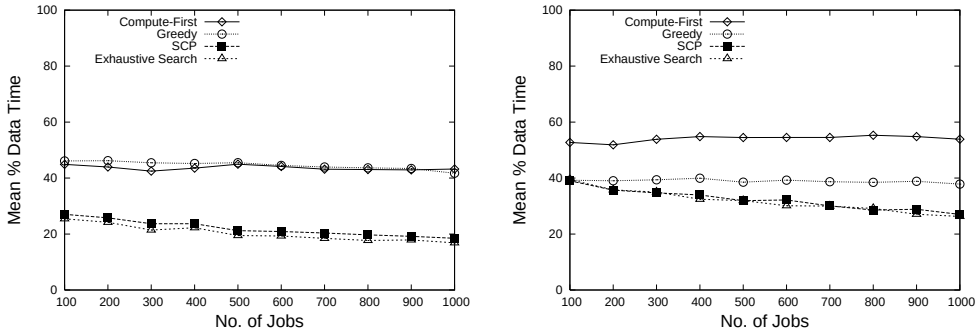
Figures 8-10 show a more fine-grained view of the experimental evaluation by showing the effect of varying one of the variables (N, K, $Size$, $Dist$), all others kept constant. Essentially, these are snapshots of the experimental results that contributed to the summary data in Table 2. Along with the makespan, two more measures of performance are considered within these figures. These are:

(1) *Mean percentage of data time*: For each job in an experiment, the share of the data transfer time is calculated as a percentage of the total execution time for that job. The average of this measure over all the jobs then represents the mean impact of the data transfer time on the set of jobs or the application as a whole. A lower number is better as one of the aims of the scheduling algorithms presented so far has been to reduce the data transfer time.
(2) *Mean locality of access*: For each job, the ratio of the number of datasets accessed from the local disk storage of the compute resource to the total number of datasets accessed by the job from all resources is calculated as a percentage of the latter and is termed as the *local access ratio*. The average of the local access ratio over all the jobs becomes a measure of locality exploited by each of the algorithms. In this case, a higher number is better as increased local access decreases the impact of remote data transfer on the performance.

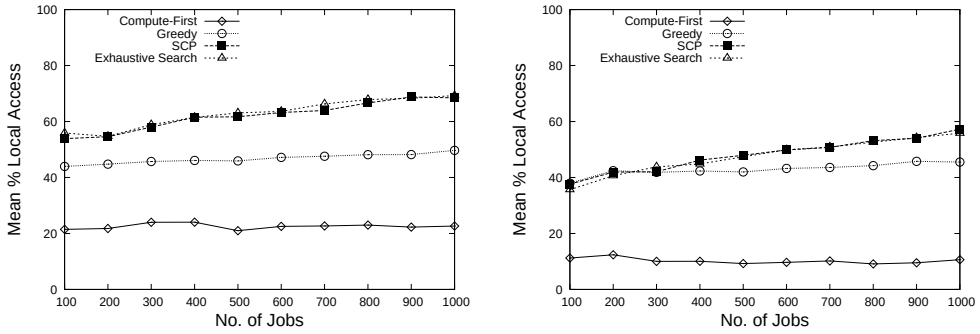
These two measures represent two slightly different perspectives on the data access performed by the jobs. Consider a job that requires one dataset of size 6 GB and two datasets of size 1 GB each. The job may be scheduled such that the larger-sized dataset is accessed locally, whereas the smaller-sized datasets may be accessed from remote data hosts. In this case, the data transfer component is small but the locality of access is low as well. However, when the sizes of the datasets are more or less equal, the locality of access becomes an important factor. These two measures, therefore, give an indication of the importance given by the algorithms to the location of data. These can be correlated with



(a) Makespan vs. No. of Jobs



(b) Data Time vs. No. of Jobs

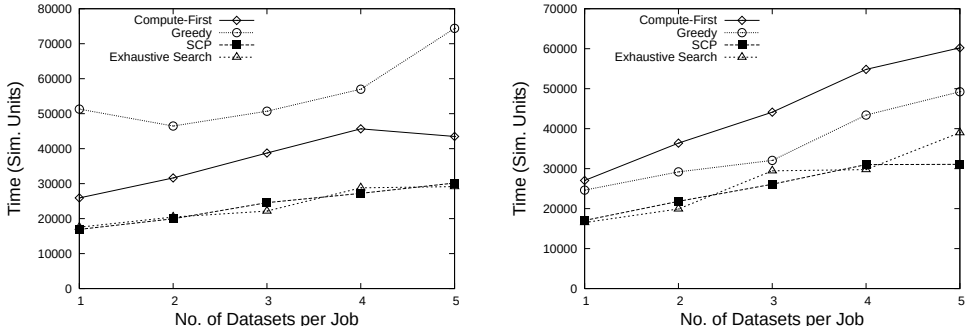


(c) Locality vs. No. of Jobs

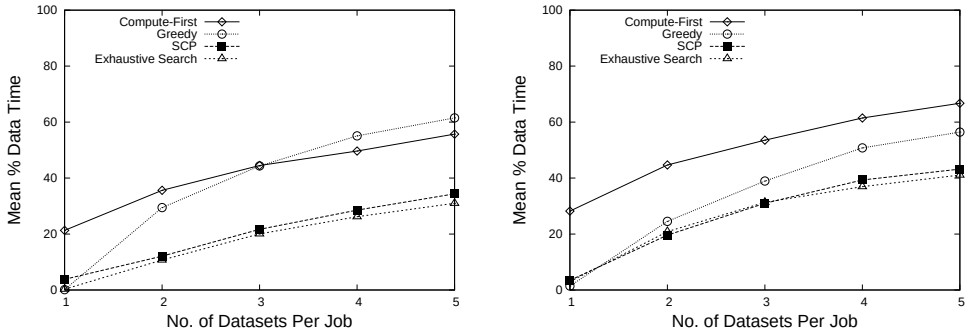
Fig. 8. Evaluation with increasing number of jobs ($Size$=300000 MI, K=3, Left: $Dist$=Uniform, Right: $Dist$=Zipf).

the makespan to judge the impact of the selection made by an algorithm on its performance.

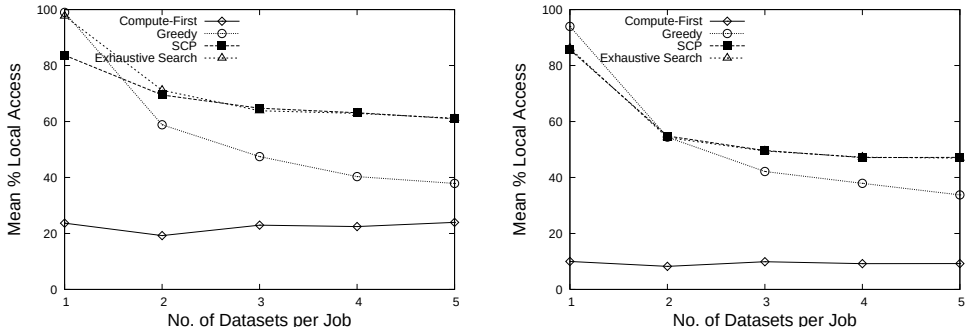
Figure 8 shows the impact of the number of jobs on the performance of the algorithm. It can be seen that as the number of jobs increases, the makespan of Compute-First and Greedy heuristic rise more steeply than the other two. The impact of data time is lower for SCP and Exhaustive Search than it is for Compute First and is a factor in their improved performance. Locality of access is also higher for the former two algorithms and it increases as the number of jobs in the set increases. This is because the probability of datasets being shared increases with more jobs accessing the same global set of datasets



(a) Makespan vs. No. of Jobs



(b) Data Time vs. No. of Jobs

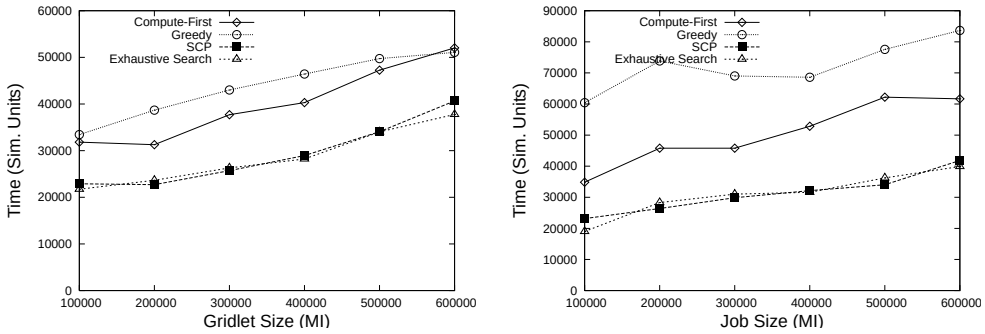


(c) Locality vs. No. of Jobs

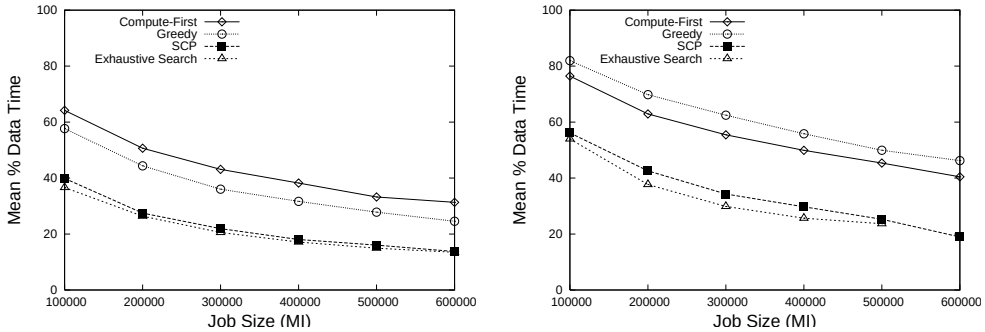
Fig. 9. Evaluation with increasing number of datasets per job (N=600, $Size$=300000 MI, Left: $Dist$=Uniform, Right: $Dist$=Zipf).

as was the case in this evaluation. This means that there is a greater chance for transferred datasets to be reused with a higher number of jobs. In case of Zipf distribution (right column), the locality is lower than in the case of Uniform distribution which means that a job submitted to a compute resource is less likely to find its required datasets locally. This can be attributed to the rarer availability of datasets in Zipf distribution than in the Uniform distribution.

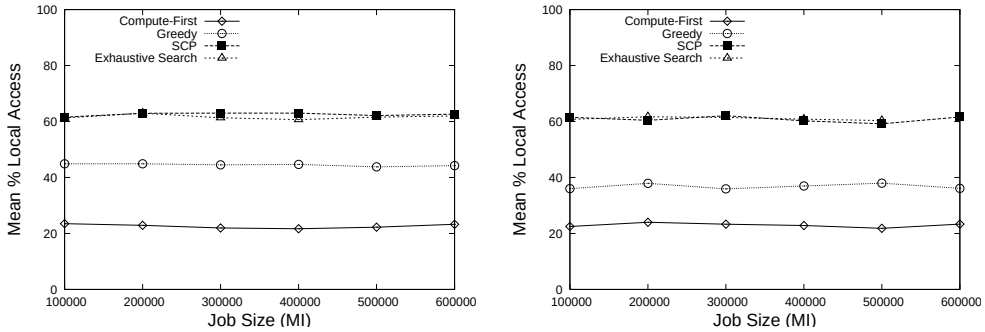
An interesting result here is that even with a high locality of access, the Greedy heuristic performs significantly worse than Compute-First for Uniform distribution (left column) while it performs better than the latter when the datasets are replicated according to Zipf distribution. In the second case, there



(a) Makespan vs. No. of Jobs



(b) Data Time vs. No. of Jobs



(c) Locality vs. No. of Jobs

Fig. 10. Evaluation with increasing computational size (N=600, $Dist$=Uniform, Left: K=3, Right: K=5).

is a lower number of choices than in the first and thus, the greedy strategy has a better probability of forming good resource sets. In this case, it can be seen that the performance of Greedy comes close to or in some cases, becomes as competitive as SCP mirroring the results of Table 2. With a higher number of choices, the greedy strategy has a lower probability of arriving at the best compute resource for a job and its performance is degraded.

Figure 9 shows the impact of changing only the number of datasets per job. Some of the trends in the previous graphs are also reflected here. With only one dataset per job, all algorithms except for Compute-First are able to produce schedules with zero data time and full locality of access. With the jobs per

dataset increasing, the impact of data transfer time increases at a faster rate for Greedy than for SCP and Exhaustive Search. Also, the locality reduces more steeply in the Zipf distribution than in the Uniform distribution, because there are fewer data hosts for each dataset. Finally, Figure 10 shows the impact of the computation time on the performance of data-oriented scheduling algorithms. The locality remains almost constant throughout the experiments. However, as expected, the impact of data transfer is steadily reduced with increasing size of computation.

Another interesting result here is that the performance of Exhaustive Search is worse than that of SCP in certain cases. This runs contrary to expectations that Exhaustive Search will produce the best results in every case. This is due to the fact that MinMin itself is not guaranteed to give the best schedules in every situation [9]. The assignment of resources to a job impacts the selection of resources for jobs that are yet to be assigned. This leads to variations in performance of all the algorithms.

7.2 Comparison between MinMin and Sufferage

Table 3
Summary of Comparison between MinMin and Sufferage.

Heuristic	**Geometric Mean**	**Avg. deg**	**Avg. rank**
MinMin			
Compute-First	19604.73	18.7 (12.84)	4.93 (1.0)
Greedy	25782.28	57.93 (28.51)	6.33 (1.45)
SCP	17353.87	5.2 (13.58)	1.73 (1.44)
Exhaustive Search	18481.26	11.83 (11.39)	3.47 (1.41)
Sufferage			
Compute-First	60631.56	269.31 (57.81)	8.0 (0)
Greedy	18558.61	12.06 (8.45)	4.2 (1.72)
SCP	17353.87	5.2 (13.58)	1.73 (1.44)
Exhaustive Search	18584.88	12.47 (11.53)	3.67 (1.53)

Each of the matching heuristics were paired with both MinMin and Sufferage scheduling algorithms and evaluated to determine if the latter provided a better performance than the former. The results of the experiments carried

out within this evaluation is summarised using the same metrics as in the previous section and are listed in Table 3. It can be seen that there is little difference in the performance of both SCP and Exhaustive Search heuristics when coupled with either MinMin or Sufferage scheduling algorithms. Also, there is only a slight improvement in the performance for Greedy when coupled with the Sufferage algorithm. However, the performance for Compute-First is significantly degraded by coupling it with the Sufferage algorithm. On average, it is about 2 1/2 times as worse as the best heuristic in any experiment. Also, the Compute-First-Sufferage pair is ranked 8th in terms of performance in all experiments (standard deviation is zero). In other words, it gives the worst performance in every case.

8 Related Work

There has been a lot of work in scheduling interdependent tasks with communication dependencies and an overview of strategies static allocation of such tasks can be found in a survey published by Kwok and Ahmad [30]. One such strategy proposed by Kafil and Ahmad [31] adapts the well-known A* search algorithm to search the entire space of possible task-processor mappings to identify one that provides optimal allocation with respect to processor load and task intercommunication. As mentioned previously, the work in this paper deals with a different model consisting of data-intensive independent (non-communicating) tasks where the tasks are not only mapped to processors but to storage resources as well. The matching algorithms also perform a bounded search within the task-resources (both compute and storage) mapping space but on a per task basis. Considering the entire space of all task to compute and data resource mappings will make the problem computationally intractable and hence, we have opted for a 2 stage mapping process.

Previous publications in scheduling distributed data-intensive applications on Grids [6][32][10] have tackled the problem of replicating the data for a single job depending on the site where the job is scheduled. However, the application model applied here is closer to that of Casanova, et.al [12] who investigate scheduling algorithms for a set of independent tasks that share files. They extend the MinMin and Sufferage algorithms to consider data requirements of the tasks and introduce the XSufferage algorithm to take advantage of file locality. However, in their article the source of all the files for the tasks is the resource that dispatches the jobs. This work is extended by Giersch, et. al [33] to consider the general problem of scheduling tasks that share multiple files, each available from multiple sources. They focus on developing routing algorithms for staging the input files through the network links on to data resources, close to the selected compute resources, such that the total execution

time is minimised. Khanna, et al. [34] propose a hypergraph-based approach for scheduling a set of independent tasks with a view to minimise the I/O overhead by considering the sharing of files between the tasks. However, they do not take into account the aspect of data replication as the files have only a single source.

The scheduling model considered in this paper is distinct from those mentioned previously because it considers: a) the problem of selecting a resource set for a job requiring multiple datasets in an environment where the data is available from multiple sources due to prior replication and b) the selection of computational and data resources in such a resource set to be interconnected. This paper also extends MinMin and Sufferage algorithms similar to that done by Casanova, et al. [12] and Giersch, et al. [33]. However, in the algorithms presented in this paper, the focus of the effort remains on matching or selection of resources which has not been given adequate weightage in related work. The matching algorithms aim to select a resource set such that both the computational and data transfer components of the execution time are reduced simultaneously. This is different from the approach, followed by most of the Data Grid scheduling algorithms of scheduling the jobs onto a compute resource based on minimum execution time and then replicating the data to minimise the access time. The latter approach was generalised and extended to support the multiple datasets model in the previous sections, and was evaluated as the Compute-First heuristic. Simulation results show that Compute-First produces worse schedules when compared to a strategy giving weightage to both computational and data factors such as the SCP Tree Search algorithm.

Mohamed and Epema [35] present a Close-to-Files algorithm for a similar application model, though restricted to one dataset per job, that searches the entire solution space for a combination of computational and storage resources to minimise execution time. This strategy, extended to support multiple datasets per job and evaluated as Exhaustive Search in the previous section, produces good schedules but becomes unmanageable for large solution spaces that occur when more than one dataset is considered per job.

Jain, et al. [36] proposed a set of heuristics for scheduling I/O operations so as to avoid transfer bottlenecks in parallel systems. However, these heuristics do not consider the problem of scheduling computational operations and also, the problem of selecting data sources in case of data replication. Other publications in parallel I/O optimisation [37][38][39] pay attention to improving performance through techniques such as interleaving and disk striping. However, such optimisation techniques are not the focus of this paper.

9 Conclusion and Future Work

This paper presents the problem of mapping an application with a collection of jobs that require multiple datasets that are each multiply replicated, to compute resources and data hosts in a Grid. It models the problem of matching the jobs as an instance of the SCP and proposes a tree-search heuristic based on a solution to the SCP. This is then combined with the MinMin and Sufferage algorithms for scheduling sets of independent jobs and evaluated through simulation against other matching heuristics such as Compute-First, Greedy Selection and Exhaustive Search. Experiments show that the SCP and the Exhaustive Search heuristics provide the best performance among all the four heuristics mainly because they exploit the locality of datasets, and thereby reduce the amount of data transferred during execution. However, the high computational complexity of Exhaustive Search means that it will search through large spaces that may become infeasible for jobs requiring large number of datasets. Also, the experimental results show that there is no gain in performance by applying the Sufferage heuristic in place of MinMin for scheduling the entire set of jobs.

As part of immediate future work, it is planned to evaluate the SCP mapping heuristic using other task scheduling algorithms such as Max-min and Genetic Algorithms. It would also be interesting to explore scheduling of distributed data-intensive tasks where they are interdependent. In this case, the overall mapping should not only take into account the location of distributed data but also the communication between the tasks. In present-day Grids, scientific applications are increasingly being composed as data-intensive workflows involving tasks that process, share and manage large, distributed datasets [40]. These workflows are generally modeled as Directed Acyclic Graphs (DAGs) and many scheduling strategies for interdependent tasks have been applied for mapping workflows on to Grid resources [41][42]. A possible extension to the work presented in this paper would be to use the SCP search heuristic with a known DAG scheduling algorithm such as the Dynamic Critical Path (DCP) [43] to schedule workflows with distributed data-intensive tasks.

The scheduling strategies in this paper are considered to be conventional in the sense that the compute resources and data hosts serve all requests and accept all jobs regardless of their source, and the scheduling is driven by the need to improve traditional parameters of performance such as application throughput. However, the emerging economy-based model of Grids [44] considers resource providers to be independent agents that are incentivized by profit motives to contribute resources to a Grid. A consumer in this environment would have a limited budget and would therefore, aim to execute her application at resources that provide her the best service within her budget. In such an environment, both resource providers and consumers aim to improve their

utility that may depend on non-system-centric metrics. Recent publications by Kwok, Song and Hwang [45], and Khan and Ahmed [46] model interactions between the participants in an economy-based Grid as games and analyse the behaviour of different agents under different game-theoretic strategies. These have been performed from a computational Grid perspective. Extending this model to distributed data-intensive applications is also a possible future work.

References

[1] I. Foster, C. Kesselman, The Grid: Blueprint for a Future Computing Infrastructure, Morgan Kaufmann Publishers, San Francisco, USA, 1999.

[2] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, S. Tuecke, The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets, Journal of Network and Computer Applications 23 (3) (2000) 187–200.

[3] W. Hoschek, F. J. Jaen-Martinez, A. Samar, H. Stockinger, K. Stockinger, Data Management in an International Data Grid Project, in: Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (GRID '00), Springer-Verlag, Berlin, Germany, Bangalore, India, 2000.

[4] N. Yamamoto, O. Tatebe, S. Sekiguchi, Parallel and Distributed Astronomical Data Analysis on Grid Datafarm, in: Proceedings of 5th IEEE/ACM International Workshop on Grid Computing (Grid 2004), IEEE CS Press,Los Alamitos, CA, USA, Pittsburgh, USA, 2004.

[5] R. Gardner, et al., The Grid2003 Production Grid: Principles and Practice, in: Proceedings of the 13th Symposium on High Performance Distributed Computing (HPDC 13), IEEE CS Press, Los Alamitos, CA, USA, Honolulu, HI, USA, 2004.

[6] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, F. Zini, Simulation of Dynamic Grid Replication Strategies in OptorSim, in: Proceedings of the 3rd International Workshop on Grid Computing(GRID 02), Springer-Verlag, Berlin, Germany, Baltimore,MD,USA, 2002, pp. 46–57.

[7] R. Wolski, N. Spring, J. Hayes, The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, Journal of Future Generation Computing Systems 15 (1999) 757–768.

[8] A. Rajasekar, M. Wan, R. Moore, MySRB & SRB: Components of a Data Grid, in: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), IEEE CS Press,Los Alamitos, CA, USA, Edinburgh, UK, 2002.

[9] 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 (1999) 107–131.

[10] K. Ranganathan, I. Foster, Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications, in: Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC), IEEE CS Press, Los Alamitos, CA, USA, Edinburgh, UK, 2002.

[11] 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.

[12] H. Casanova, A. Legrand, D. Zagorodnov, F. Berman, Heuristics for Scheduling Parameter Sweep Applications in Grid environments, in: Proceedings of the 9th Heterogeneous Computing Systems Workshop (HCW 2000), IEEE CS Press, Los Alamitos, CA, USA, Cancun, Mexico, 2000.

[13] E. Seidel, G. Allen, A. Merzky, J. Nabrzyski, GridLab: a grid application toolkit and testbed, Future Gener. Comput. Syst. 18 (8) (2002) 1143–1153.

[14] S. Venugopal, R. Buyya, L. Winton, A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids, in: Proceedings of the 2nd Workshop on Middleware in Grid Computing (MGC 04), ACM Press, New York, USA, Toronto, Canada, 2004.

[15] M. Maheshwaran, S. Ali, H. J. Siegel, D. Hengsen, R. F. Freund, Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems, in: 8th Heterogeneous Computing Systems Workshop (HCW '99), San Juan, Puerto Rico, 1999.

[16] E. Balas, M. W. Padberg, On the Set-Covering Problem, Operations Research 20 (6) (1972) 1152–1161.

[17] T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson, Introduction to Algorithms, McGraw-Hill Higher Education, 2001.

[18] N. Christofides, Graph Theory: An Algorithmic Approach, Academic Publishers, London, UK, 1975, Ch. Independent and Dominating Sets – The Set Covering Problem, pp. 30 – 57, iSBN 012 1743350 0.

[19] S. Venugopal, R. Buyya, A Deadline and Budget Constrained Scheduling Algorithm for e-Science Applications on Data Grids, in: Proceedings of the 6th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2005), Vol. 3719 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, Melbourne, Australia., 2005.

[20] A. Sulistio, C. S. Yeo, R. Buyya, A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools, Software: Practice and Experience (SPE) 34 (7) (2004) 653–673.

[21] R. Buyya, M. Murshed, GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing, Concurrency and Computation: Practice and Experience (CCPE) 14 (13-15) (2002) 1175–1220.

[22] I. C. Legrand, H. B. Newman, The MONARC toolset for simulating large network-distributed processing systems, in: Proceedings of the 32nd Winter Simulation Conference (WSC '00), Society for Computer Simulation International, San Diego, CA, Orlando, FL, 2000.

[23] K. Ranganathan, I. Foster, Simulation studies of computation and data scheduling algorithms for data grids, Journal of Grid Computing 1 (1) (2003) 53–62.

[24] H. Casanova, Simgrid: A Toolkit for the Simulation of Application Scheduling, in: Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID '01), IEEE CS Press , Los Alamitos, CA, USA, Brisbane, Australia, 2001.

[25] A. Sulistio, G. Poduval, R. Buyya, C.-K. Tham, On Incorporating Differentiated Network Service into GridSim, Tech. Rep. GRIDS-TR-2006-5, The University of Melbourne, Australia (Mar. 2006).

[26] A. Sulistio, U. Cibej, B. Robic, R. Buyya, A Tool for Modelling and Simulation of Data Grids with Integration of Data Storage, Replication and Analysis, Tech. Rep. GRIDS-TR-2005-13, University of Melbourne, Australia (Nov. 2005).

[27] K. Park, G. Kim, M. Crovella, On the relationship between file sizes, transport protocols, and self-similar network traffic, in: Proceedings of the 1996 International Conference on Network Protocols (ICNP '96), IEEE CS Press, Atlanta, GA, USA, 1996.

[28] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and zipf-like distributions: evidence and implications, in: Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '99.), New York, NY, USA, 1999.

[29] D. G. Cameron, R. Carvajal-Schiaffino, A. P. Millar, C. Nicholson, K. Stockinger, F. Zini, Evaluating Scheduling and Replica Optimisation Strategies in OptorSim, in: Proceedings of the 4th International Workshop on Grid Computing (Grid2003), IEEE CS Press, Los Alamitos, CA, USA, Phoenix, AZ, USA, 2003.

[30] Y.-K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Computing Surveys 31 (4) (1999) 406–471.

[31] M. Kafil, I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems, IEEE Concurrency 6 (3) (1998) 42–50.

[32] S.-M. Park, J.-H. Kim, Chameleon: A Resource Scheduler in a Data Grid Environment, in: Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), IEEE CS Press, Los Alamitos, CA, USA, Tokyo, Japan, 2003.

[33] A. Giersch, Y. Robert, F. Vivien, Scheduling tasks sharing files from distributed repositories, in: Proceedings of the 10th International Euro-Par Conference (EuroPar '04), Springer-Verlag, Berlin, Germany, Pisa, Italy, 2004.

[34] G. Khanna, N. Vydyanathan, T. Kurc, U. Catalyurek, P. Wyckoff, J. Saltz, P. Sadayappan, A hypergraph partitioning-based approach for scheduling of tasks with batch-shared I/O, in: Proceedings of the 2005 IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005), IEEE CS Press, Cardiff, UK, 2005.

[35] H. Mohamed, D. Epema, An evaluation of the close-to-files processor and data co-allocation policy in multiclusters, in: Proceedings of the 2004 IEEE International Conference on Cluster Computing, IEEE CS Press,Los Alamitos, CA, USA, San Diego, CA, USA, 2004.

[36] R. Jain, K. Somalwar, J. Werth, J. C. Browne, Heuristics for Scheduling I/O Operations, IEEE Transactions on Parallel and Distributed Systems 8 (3) (1997) 310–320.

[37] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, A. Sussman, Tuning the performance of i/o-intensive parallel applications, in: Proceedings of the fourth workshop on I/O in parallel and distributed systems (IOPADS '96), ACM Press, Philadelphia, PA, USA, 1996.

[38] K. Salem, H. Garcia-Molina, Disk striping, in: Proceedings of the Second International Conference on Data Engineering (ICDE-86), IEEE CS Press , Los Alamitos, CA, USA, Los Angeles, USA, 1986.

[39] R. Thakur, A. Choudhary, R. Bordawekar, S. More, S. Kuditipudi, Passion: Optimized I/O for Parallel Applications, Computer 29 (6) (1996) 70–78.

[40] E. Deelman, et al., Mapping abstract complex workflows onto grid environments, Journal of Grid Computing 1 (1) (2003) 25–39.

[41] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, K. Kennedy, Task scheduling strategies for workflow-based applications in grids, in: Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005), Cardiff, UK, IEEE CS Press, Los Alamitos, CA, USA, 2005.

[42] Z. Shi, J. J. Dongarra, Scheduling workflow applications on processors with different capabilities, Future Generation Computer Systems 22 (6) (2006) 665–675.

[43] Y.-K. Kwok, I. Ahmad, Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors, IEEE Trans. Parallel Distrib. Syst. 7 (5) (1996) 506–521.

[44] R. Buyya, Economic-based Distributed Resource Management and Scheduling for Grid Computing, Ph.D. thesis, Monash University, Australia (2002).

[45] Y.-K. Kwok, S. Song, K. Hwang, Selfish grid computing: game-theoretic modeling and nas performance results, in: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05), IEEE Computer Society, Washington, DC, USA, 2005, pp. 1143–1150.

[46] S. Khan, I. Ahmad, Non-cooperative, semi-cooperative, and cooperative games-based grid resource allocation, in: Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Rhodes island, Greece, IEEE CS Press, Los Alamitos, CA, USA, 2006.