J. Parallel Distrib. Comput. 69 (2009) 307–314

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

A general distributed scalable grid scheduler for independent tasks

Cong Liu, Sanjeev Baskiyar *

Department of Computer Science and Software Engineering, Shelby Technology Center, Auburn University, Auburn, AL, 36849, United States

ARTICLE INFO

Article history:
Received 4 February 2008
Received in revised form
30 October 2008
Accepted 10 November 2008
Available online 24 November 2008

Keywords:
Distributed scheduling
Grid computing
Successful schedulable ratio
Peer to peer scheduler
Priority
Deadline
Shuffling

ABSTRACT

We consider non-preemptively scheduling a bag of independent mixed tasks (hard, firm and soft) in computational grids. Based upon task type, we construct a novel generalized distributed scheduler (GDS) for scheduling tasks with different priorities and deadlines. GDS is scalable and does not require knowledge of the global state of the system. It is composed of several phases: a multiple attribute ranking phase, a shuffling phase, and a task-resource matched peer to peer dispatching phase. Results of exhaustive simulation demonstrate that with respect to the number of high-priority tasks meeting deadlines, GDS outperforms existing approaches by 10%–25% without degrading schedulability of other tasks. Indeed, with respect to the total number of schedulable tasks meeting deadlines, GDS is slightly better. Thus, GDS not only maximizes the number of mission-critical tasks meeting deadlines, but it does so without degrading the overall performance. The results have been further confirmed by examining each component phase of GDS. Given that fully known global information is time intensive to obtain, the performance of GDS is significant. GDS is highly scalable both in terms of processors and number of tasks—indeed it provides superior performance over existing algorithms as the number of tasks increase. Also, GDS incorporates a shuffle phase that moves hard tasks ahead improving their temporal fault tolerance. Furthermore, since GDS can handle mixed task types, it paves the way to open the grid to make it amenable for commercialization. The complexity of GDS is $O(n^2m)$ where n is the number of tasks and m the number of machines.

© 2008 Elsevier Inc. All rights reserved.

1. Introduction

A major motivation of grid computing [6,7] is to aggregate the power of widely distributed resources to provide services to users. Application scheduling plays a vital role in achieving such services. A number of deadline based scheduling algorithms already exist. However, in this paper we address the problem of scheduling a bag of independent mixed tasks in computational grids. We consider three types of tasks: hard, firm and soft [14]. It is reasonable for a grid scheduler to prioritize mission-critical tasks while maximizing the total number of tasks meeting deadlines. Doing so may make the grid more commercially viable as it opens it up for all classes of users.

To the best of our knowledge, GDS is the first attempt at prioritizing tasks according to task types as well as considering deadlines and dispatch times. It also matches tasks to appropriate computational and link (bandwidth) resources. Additionally, GDS consists of a unique shuffle phase that reschedules mission-critical tasks as early as possible to provide temporal fault tolerance. Furthermore, GDS is highly scalable as it does not require a full knowledge of the state of all nodes of the grid as many other algorithms do. For GDS's peer to peer dispatch, knowledge of peer site capacities is sufficient. One must consider that obtaining full knowledge of the state of the grid is difficult and/or temporally intensive.

The remainder of this paper is organized as follows. A review of recent related works has been given in Section 2. In Section 3, we outline the task taxonomy used in this work. Section 4 describes the grid model. Section 4.1 presents the detailed design of GDS. Section 5 presents a comprehensive set of simulations that evaluate the performance of GDS. Conclusions and suggestions for future work appear in Section 6.

2. Problem statement

We consider three types of tasks: hard, firm and soft. GDS uses such a task taxonomy which considers the consequence of missing deadlines, and the importance of task property. Hard tasks are mission critical, in that the consequences of failure are catastrophic. For firm tasks a few missed deadlines will not lead to total failure. However, more missed deadlines may lead to total failure. For soft tasks, failures will only result in degraded performance.

An example of mission-critical application is the computation of the orbit of a moving satellite to make real-time defending

* Corresponding author.
E-mail addresses: liucong@auburn.edu (C. Liu), baskisa@auburn.edu (S. Baskiyar).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.11.003

decisions [21]. As one would expect, catastrophic consequences may occur if such an operation fails to meet its deadline. An example of a firm task with deadline is of the Network for Earthquake Engineering Simulation (NEES) [22], which seeks to lessen the impact of earthquake and tsunami related disasters by providing capabilities for earthquake engineering research. Such applications do have deadlines; however, since some computational tasks are not real-time, consequences of missing them are not that catastrophic. Applications which fall in the category of soft tasks include coarse-grained task-parallel computations arising from parameter sweeps, Monte Carlo simulations, and data parallelism. Such applications generally involve large-scale computation to search, optimize, statistically characterize products, solutions, and design spaces normally do not have hard real-time deadlines.

2.1. Task model

We consider scheduling Bag-of-Tasks (BoT) applications, which are those parallel applications whose tasks are independent of one another. BoT applications are used in a variety of scenarios, including parameter sweeps, computational biology [28], computer imaging [26,27], data mining, fractal calculations and simulations. Furthermore, because of the independence of tasks, BoT applications can be successfully executed over geographically distributed computational grids, as demonstrated by SETI@home [1]. In fact, one can argue that BoT applications are the applications most suited for computational grids [5], where communication can easily become a bottleneck for tightly-coupled parallel applications.

We assume that the average computation time (avg_comp) of each task on each machine is known based on user-supplied information, task profiling and analytical benchmarking. The assumption of such *avg_comp* information is a common practice in scheduling research (e.g. [9–12,15,25,31]). The *avg_comp* (i, j) is the estimated execution time of task i on machine j. These estimated values may differ from actual times, e.g., actual times may depend on input data. Therefore, for simulation studies, the actual computation time (*act_comp*) values are calculated using the avg_comp values as the mean. The details of the simulation environment and the calculation of the *act_comp* values have been presented in Section 5.

2.2. Grid model

In our grid model, as shown in Fig. 1, geographically distributed sites interconnect through WAN. We define a site as a location that contains many computing resources of different processing capabilities. Heterogeneity and dynamicity cause resources in grids to be distributed hierarchically or in clusters rather than uniformly. At each site, there is a main server and several supplemental servers, which are in charge of collecting information from all machines within that site. If the main server fails, a supplemental server will take over. Intra-site communication cost is usually negligible as compared to inter-site communication.

3. Related work

Several effective scheduling algorithms such as EDF [16], Sufferage [18], *Min–Min* [20], *Max–Min* [18], and *Max–Max* [18] have been proposed in previous works. The rationale behind Sufferage is to allocate a site to a task that would "suffer" most in completion time if the task is not allocated to that site. For each task, its sufferage value is defined as the difference between its lowest completion time and its second-lowest completion time. The complexity of the conventional Sufferage algorithm, which is applied to a single cluster system, is $O(mn^2)$, where m is the number of machines and n the number of incoming tasks. If Sufferage were to be applied in a multi-cluster grid system, its complexity would become $O(msn^2)$ where s is the number of clusters within the grid. The *Min–Min* heuristic begins with computing the set of minimum completion time for each task on all machines. Among all tasks, the one that has the overall minimum completion time is chosen and allocated to the tagged resource. The goal of *Min–Min* is to attempt to complete as many tasks as possible. The *Max–Min* heuristic is very similar to *Min–Min*. It also begins with computing the set of minimum completion time for each task on all machines. Then the machine that has the overall maximum completion time is selected. The *Max–Max* heuristic begins with computing the set of maximum completion time for each task. Then the one that has the overall maximum completion time is first chosen and mapped to the corresponding resource. The complexity of these three algorithms is $O(msn^2)$, when applied in a grid system.

Little research exists on scheduling algorithms taking into account both the task types and deadlines in grids. An only deadline based scheduling algorithm appears in [29] for multi-client and multi-server within a single resource site. The algorithm aims at minimizing deadline misses by using load correction and fallback mechanisms. It dispatches each task in FIFO order to the server that provides a completion time earlier than but closest to task's deadline. Since it uses estimations for data transfer times and computation times, it may be possible that once the input data has reached the server, it is actually unable to complete the task by its deadline. In such a case, the server will notify the client to resubmit the task to another server. Although the algorithm is pretty simple, its complexity is $O(m.s.n)$ if applied in a grid system. Based on this work, in [3], a deadline scheduling algorithm with priority concern appropriate for multi-client, multi-server within a *single* resource site has been proposed. It schedules tasks by priority on a server that can satisfy the deadline. Its complexity is $O(nm^2s^2)$ when applied on grid systems. Since preemption is allowed, it leaves open the possibility that tasks with lower priority but early deadlines may miss their deadlines. In [13], a number of classical heuristics (e.g. *Max–Min*, *Max–Max*, Percentage Best [19], Relative Cost [30]) are revised to map tasks with priorities and deadlines in a single cluster environment. However, these revised algorithms do not provide guarantee to complete mission-critical tasks before deadlines. Moreover, since the target hardware platform is a single cluster, they have not taken the data transfer requirements into consideration. Also the issue of scalability has not been addressed.

Several research works consider the data staging problem when making scheduling decisions. Casanova [4] describes an adaptive scheduling algorithm for a bag of tasks in Grid environment that takes data storage issues into consideration. However, they make scheduling decisions centrally, assuming *full knowledge* of current loads, network conditions and topology of all sites in the grid. Ranganathan and Foster [24] consider dynamic task scheduling along with data staging requirements. Data replication is used to suppress communication and avoid data access hotspots. Park and Kim [23] describe a scheduling model that considers both the amount of computational resources and data availability in a data grid environment.

The aforementioned algorithms do not consider all of the following criteria: task types, dispatch times, deadlines, scalability and distributed scheduling. Furthermore, they require a full knowledge of the state of the grid which is difficult and/or expensive to maintain.

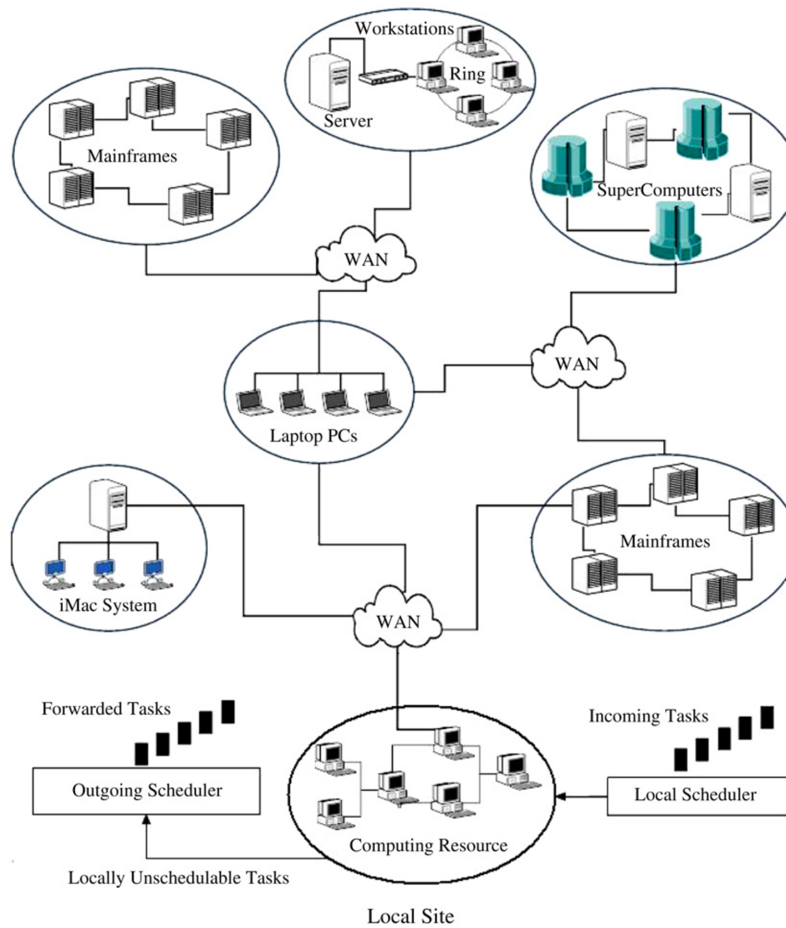

Fig. 1. Grid model.

4. Scheduling algorithm

The following are the design goals of GDS:

- Maximize number of mission-critical tasks meeting their deadlines
- Maximize total number of tasks meeting their deadlines
- Provide temporal fault tolerance to the execution of mission-critical tasks
- Provide Scalability

Since neither EDF nor using priorities alone can achieve the above goals, we proposed GDS [17]. GDS consists of three phases. First incoming tasks at each site are ranked. Second, a shuffling based algorithm is used to assign each task to a specific resource on a site, and finally those tasks that are unable to be scheduled are dispatched to remote sites where the same shuffling based algorithm is used to make scheduling decisions. The pseudo-code of GDS's main function is shown in Fig. 2.

4.1. Notations

The following notations have been used in this paper.

- t_i: task i
- e_{ijk}: estimated execution time of t_i on $machine_k$ at $site_j$
- c_{ij}: estimated transmission time of t_i (i.e. time taken to transmit a task's executable code, input and output data) from current site to $site_j$
- l_{ijk}: latest start time of tasks t_i on $machine_k$ at $site_j$
- e_i: instruction size of t_i

```
GDS
// Q is a task queue in site S
    Sort Q by decreasing priority then by decreasing CCR-type
    then by increasing deadline
        Schedule
    If unscheduled tasks remain in Q
        Send message to each m ∈ S to execute Shuffle
        Schedule
    endif
    If unscheduled tasks remain in Q
        Dispatch
    endif
end GDS
```

Fig. 2. GDS.

- d_i: deadline of t_i
- CCR_i: communication to computation ratio of $task_i$
- n_j: number of machines within $site_j$
- cc_{jk}: computing capacity of $machine_k$ at $site_j$
- S_{pkj}: start time of the pth slack on m_k at s_j
- E_{pkj}: end time of the pth slack on m_k at s_j
- CC_j: average computing capacity of $site_j$
- Ave_CC_i: average computing capacity of all the neighboring sites of $site_i$
- Ave_C_i: estimated average transmission time of t_i from $site_i$ to all the neighbors.

A task is composed of execution code, input and output data, priority, deadline, and CCR. Tasks are assigned one of the priorities: high, normal, or low, which correspond to mission-critical, firm,

and soft tasks. A task's CCR-type is decided by its Communication to Computation Ratio (CCR), which represents the relationship between the transmission time and execution time of a task. It can be defined as:

$$CCR_i = Ave_C_i/(e_i/Ave_CC_i). \quad (1)$$

If $\mathrm{CCR}_{ijk} \gg 1$, we assign a CCR-type of communication-intensive to task t_i. If $\mathrm{CCR}_{ijk} \ll 1$, we assign a CCR-type of computation-intensive to t_i. If CCR_{ijk} is comparable to 1, we assign a CCR-type of neutral to t_i. In estimating CCR, we assume that users can estimate the size of output data. This can be a valid assumption under many situations particularly when the size of input–output data are related.

Each site contains a number of machines. The average computing capacity of $site_j$ is defined as:

$$CC_j = \sum_{k=1}^{n_j} cc_{jk}/n_j. \quad (2)$$

4.2. Multi-attribute ranking

At each site, various users may submit a number of tasks with different priorities and deadlines. Our ranking strategy takes task priority, deadline and CCR-type into consideration. The scheduler at each site puts all incoming tasks into a task queue. First, tasks are sorted by decreasing priority, then by decreasing CCR-type and then by increasing deadline. Sorting by decreasing CCR-type allows executing most communication-intensive tasks locally. If we were to dispatch such tasks to a remote site, the transfer time may be negative to performance. As we will see later, experimental results show that sorting by CCR-type gives us good performance.

4.3. Scheduling tasks within slacks

To schedule task t_i on a site s_j, each machine m_k at s_j will check if t_i can be assigned to meet its deadline. If tasks have already been assigned to m_k, slacks of varying length will be available on m_k. If no task has been assigned, slacks do not exist so that:

$$S_{pkj} = 0 \quad \&\& \quad E_{pkj} = \infty. \quad (3)$$

The scheduler checks whether t_i may be inserted into any slack while meeting the deadline. The slack search starts from the last to the first. The criteria to find a feasible slack for t_i are:

$$e_{ijk} + \max(S_{pkj,c_{ij}}) <= E_{pkj} \quad \&\& \quad e_{ijk} + \max(S_{pkj,c_{ij}}) <= d_i. \quad (4)$$

If the above conditions are satisfied, we schedule t_i to the pth slack on m_k at s_j, and set its start time to:

$$l_{ijk} = \min(d_i, E_{pkj}) - e_{ijk}. \quad (5)$$

Setting tasks start time to their latest start times creates large slacks, enabling other tasks to be scheduled within such slacks. Also, if s_j is the local site for t_i, the transmission time is ignorable; in other words, $c_{ijk} = 0$. The pseudo-code of *Schedule* is shown in Fig. 3.

4.4. Shuffle

If after executing *Schedule*, unscheduled tasks remain, a shuffling procedure is executed on each machine of the site. *Shuffle* tries to move all mission-critical tasks as early as possible. Next, it moves other tasks as close as possible to their earliest start times. In doing so, *Shuffle* creates larger slacks for possible use by unscheduled tasks. The pseudo-code of *Shuffle* is shown in Fig. 4. The advantages of shuffling are two-fold:

- Longer slacks may be obtained by packing tasks.
- Executing mission-critical tasks early provides temporal fault tolerance.

```
Schedule
  for each unscheduled task t ∈ Q
    dofor each machine m ∈ S //visit in random order to
balance load
      Visit slacks from latest to earliest
      If t fits in slack
        Schedule t on m at the latest time within the
slack
        Mark t scheduled
        Update count of unscheduled tasks in Q
      endif
    until t is scheduled
  endfor
end Schedule
```

Fig. 3. Procedure schedule.

```
Shuffle
  for each mission-critical task t
    Schedule t to the earliest available slack
  endfor
  for each task t
    Reschedule t to the earliest available slack
  endfor
end Shuffle
```

Fig. 4. Procedure shuffle.

4.5. Peer to peer dispatch

Each task is assigned a ticket [2], which is a very small file that contains certain attributes of a task. A ticket has several fields: ID, priority, deadline, CCR-type, instruction size, input data size, output data size, schedulable flag and route information. Since tickets are small they are dispatched in scheduling decisions, rather than the tasks themselves. If a task cannot be scheduled locally, its ticket is dispatched to a remote site to find a suitable resource.

In dispatching, previous works have selected a remote site randomly or used a single characteristic, such as computing capacity, bandwidth, or load. GDS uses both the computing capacity and bandwidth in dispatching. Furthermore, GDS helps decrease communication overhead since each site only needs to maintain its immediate neighbors' (i.e. neighbors that are one-hop apart) basic information such as bandwidth and average computing capacity.

Every site always maintains three dispatching lists which are used for the three CCR-typed tasks. In each list, immediate neighbors are sorted according to different attributes. The order of neighbors represents the preference of choosing a target neighboring site for dispatch. For computation-intensive tasks, the corresponding list has neighboring sites sorted by decreasing average computing capacity. For communication-intensive tasks, neighboring sites are sorted by decreasing bandwidth. For neutral-CCR tasks, neighboring sites are sorted by decreasing rank. The rank of $site_j$, a neighbor of $site_i$, is defined as:

$$\mathrm{Rank}_{ji} = CC_j \Big/ \sum_{k=1}^{r} CC_k + BW_{ij} \Big/ \sum_{k=1}^{r} BW_{ik} \quad (6)$$

where r is the number of neighbors of $site_i$, and BW_{ij} is the network bandwidth between $site_i$ and $site_j$. The three lists are available at each site and are periodically updated. A site will check whether any of its neighbors can consume a task within deadline or not. Neighbors are checked breadth-first. If none can, the most favorite neighbor will search its neighbors. This process continues until

```
Dispatch
  for each unscheduled task t ∈ Q
     for each neighbor N of S
     // visit neighbors in order depending upon CCR-type of t
        Send t's ticket to N
        if N can successfully schedule t
           Send t to N
           Mark t scheduled
      endfor
  endfor
end Dispatch
```

Fig. 5. Procedure dispatch.

Task	Priority	Exec. Time	Deadline
1	Mission-critical	1	3
2	Mission-critical	1.5	7
3	Mission-critical	1	11
4	Firm	0.5	1
5	Firm	2	14
6	Firm	3.5	14.5
7	Soft	1	4.5
8	Soft	1.5	9
9	Soft	2.5	11

Ranked Tasks at a Resource Site

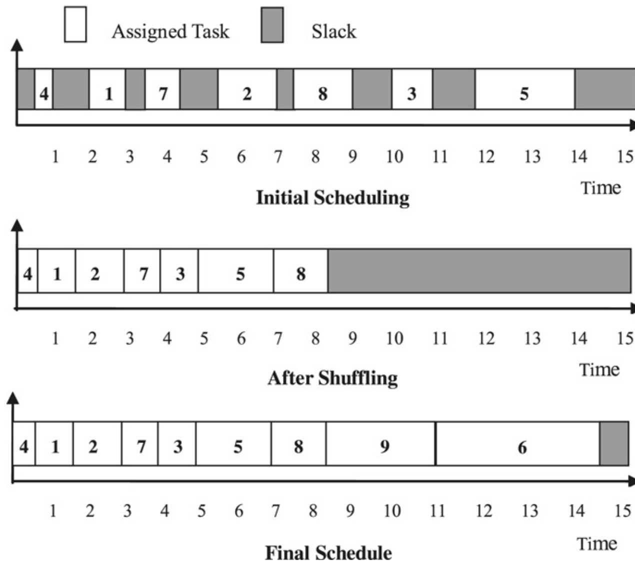

Fig. 6. An example of a schedule by GDS.

suitable remote resource has been found, or all sites have been visited. The pseudo-code of *Dispatch* is shown in Fig. 5. An example of GDS has been shown in Fig. 6.

4.6. Complexity

Let n be the number of incoming tasks, m the number of machines within each site, and s the number of sites. Then, the complexity of *Shuffle* is $O(n)$, of *Schedule* is $O(n^2m)$ and of *Dispatch* is $O(ns)$. The complexity of GDS's ranking phase is $O(n \log n)$. Therefore, the complexity of GDS is $O(n^2m)$, assuming $s < nm$. If in *Schedule*, the slacks within each machine were to be evaluated in parallel by each machine in a non-blocking fashion, the complexity of GDS would be $O(n^2)$. We note that the complexities of Sufferage and *Min–Min* are $O(n^2ms)$.

5. Performance

We conducted extensive simulations to evaluate GDS. The goals of simulations were: (i) to compare GDS against other heuristics, and (ii) to evaluate the merits of each component of GDS.

The *Critical Successful Schedulable Ratio* (*Critical* SSR) and the *Overall* SSR have been used as the main metrics of evaluation. The algorithm that produces the highest *Critical* SSR and *Overall* SSR is the best algorithm with respect to performance. They are defined as:

$$\textit{Critical}\ \text{SSR} = \frac{\text{number of mission-critical tasks meeting deadlines}}{\text{total number of mission-critical tasks}}$$

$$\textit{Overall}\ \text{SSR} = \frac{\text{number of tasks meeting deadlines}}{\text{total number of tasks}}.$$

5.1. Parameter initialization

The simulations were performed by creating a set of random independent tasks which were input to the heuristics. We varied the instruction size, size of input and output datasets, bandwidth between sites, and each machine's processing capability. The following input parameters are used to create the task set:

- Communication-to-computation ratio, CCR. If the task has a low CCR, it is considered to be computation-intensive; if high, communication-intensive.
- Average computation cost of a task, avg_comp. Computation costs are generated randomly from a uniform distribution with mean value equal to avg_comp. Therefore, the average communication cost is calculated as CCR* avg_comp.
- Range percentage of computation costs on processors, β. A high β causes a wide variance between a task's computation across the processors. A low β causes a task's computation time on all processors to be almost equal. Let w be the average computation cost of task t_i selected randomly from a uniform distribution with mean value equal to avg_comp. The computation cost of t_i on any machine m_j will then be randomly selected from the range $[w^*(1 - \beta/2)]$ to $[w^*(1 + \beta/2)]$.

A set of random tasks was generated as the study test bed. The input parameters described above were varied with the following values:

- CCR = {0.05, 0.1, 0.5, 1.0, 5.0, 10.0, 20.0}
- β = {0.1, 0, 25, 0.5, 0.75, 1.0}
- ave_comp = {100}.

These values produce 35 different combinations of parameters. Since we generated up to 10,000 random tasks for each combination, the total number of tasks in the study is around 350,000. We varied other parameters to understand their impact on different algorithms. We randomly assigned priority values to tasks. *The deadlines and other parameters were chosen such that the grid system is close to its breaking point where tasks start to miss deadlines.*

5.2. Evaluation

In order to study the performance of GDS, we compared it with the extended versions of three other classical scheduling heuristics: Earliest Deadline First (EDF) [7], *Min–Min*, and Sufferage. Since they were proposed to solve the scheduling problem mainly in single cluster system, we revised them to fit into the grid model. Since all of them are centralized schemes, any global information is assumed to be known, i.e., the machine with the highest processing capacity within grid is known.

(i) GridEDF: GridEDF first ranks tasks by increasing deadline. Then, it finds the machine with the minimum completion time for each task.
(ii) GridMinMin: Part of the GridMinMin heuristic also is based

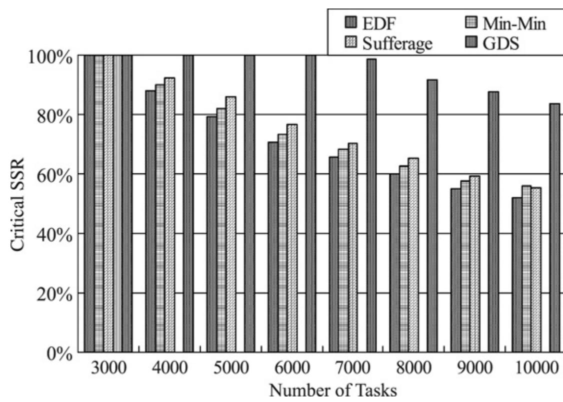

Fig. 7. *Critical* SSR of different algorithms as number of tasks increase.

on the *Min–Min* algorithm proposed in [10]. This scheme first finds the machine with the minimum completion time for each task. From these task-machine pairs, it selects the pair that has the minimum completion time. Then, it schedules the selected task and updates machine available time if deadline is met.

5.2.1. GridSuff

The GridSuff method is based on the algorithm Sufferage proposed in [18]. This scheme always assigns a machine to a task that would "suffer" most in terms of expected completion time if that particular machine is not assigned to it. Let the sufferage value of a task t_i be the difference between its second earliest completion time (on some machine m_y) and its earliest completion time (on some machine m_x). Sufferage can be summarized by the following procedure, which starts when a new task arrives.

(i) Create a task list that includes all the incoming tasks.
(ii) For each task in the list calculate the sufferage value.
(iii) Rank tasks by decreasing sufferage value.
(iv) For each task, if its minimum completion time machine can meet the task's deadline, assign it to that machine. Remove this task from the list and update the machine available times.
(v) Repeat steps (iii)–(iv) until all the tasks are mapped.

5.3. Results

The first experiment set was to evaluate the performance against other algorithms. We compared GDS against three other heuristics: EDF, *Min–Min*, and *Sufferage*.

For *Critical* SSR, from Fig. 7, we observe that GDS yields 10%–25% better performance on average than others especially when the number of tasks is high. GDS always schedules mission-critical tasks first, which guarantees to complete as many mission-critical tasks as possible. The other three heuristics do not consider task priority, which results in a number of unscheduled mission-critical tasks. We note that with increasing number of tasks, GDS performs better, thus offering better scalability.

With respect to *Overall* SSR, as shown in Fig. 8, the performance difference among the five heuristics diminishes. Although EDF, *Min–Min* and Sufferage do not consider priorities of tasks, overall they are very effective. But, the fact that GDS is no worse (indeed, it still slightly outperforms them on the average, albeit by a small margin) is important. Thus, GDS not only maximizes the number of mission-critical tasks meeting deadlines, but it does so without degrading the *Overall* SSR. Especially given that fully known global information is an assumption for the other heuristics, it makes the superior performance of GDS even more significant.

5.4. Impact of dispatching strategy

In this experiment set, we compare the performance of the dispatching strategy of GDS against other dispatching strategies,

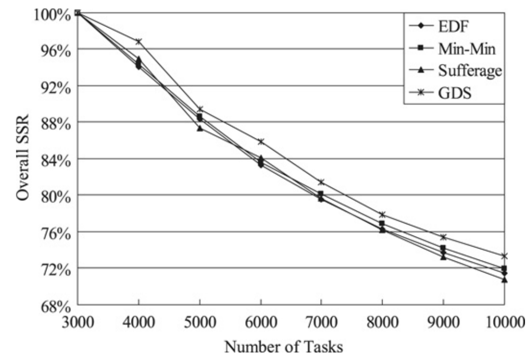

Fig. 8. *Overall* SSR for the different algorithms as number of tasks increase.

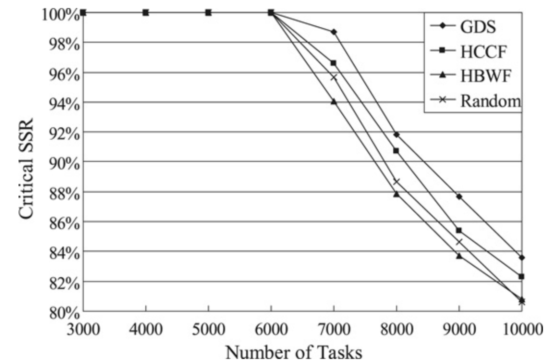

Fig. 9. Impact of dispatching strategies on *Critical* SSR.

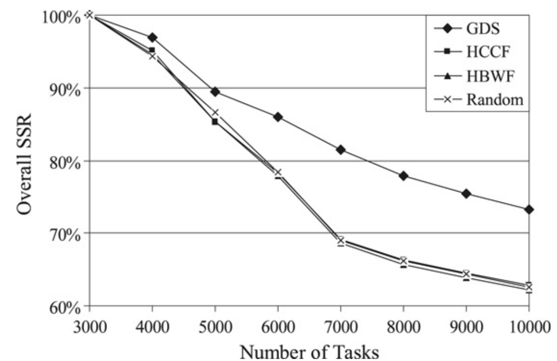

Fig. 10. *Overall* SSR with different dispatching strategies.

namely *HCCF*, *HBWF* and *Random*. *HCCF* is the highest computing capacity first dispatching strategy. In *HCCF*, an unscheduled task is first dispatched to the neighbor with the highest computing capacity. The next unscheduled task is dispatched to a neighbor with the next highest computing capacity. *HBWF* is the highest bandwidth first dispatching strategy. In *Random*, tasks are dispatched to randomly selected neighbors.

From Fig. 9 we observe that dispatching strategy of GDS yields better *Critical* SSR than other dispatching strategies for mission-critical tasks. This is because of the task to resource matching based dispatching strategy.

From Fig. 10, the dispatching strategy of GDS achieves better *Overall* SSR than other three methods, particularly for large number of tasks. When the number of tasks is very large, the *Overall* SSR of using the GDS dispatching strategy is better than those of *HCCF*, *HBWF* and *Random* by more than 10% on average.

5.5. Impact of shuffling

In this experiment, we investigate the use of the shuffling component of GDS. To do so, we use GDS_2, which is the scheduler

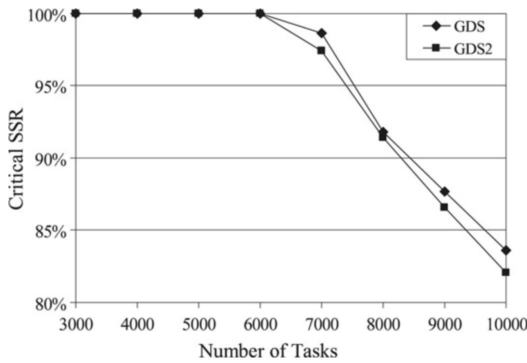

Fig. 11. *Critical* SSR for the shuffling component of GDS.

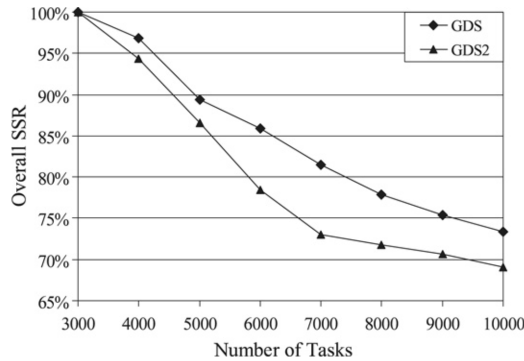

Fig. 12. *Overall* SSR for the shuffling component of GDS.

obtained upon removing the shuffling portion from GDS. From Fig. 11, we see that GDS's *Critical* SSR is almost identical to GDS_2.

However, From Fig. 12 we observe that GDS's *Overall* SSR is higher than GDS_2 by 5%. In other words, *Shuffle* schedules more tasks with firm and soft deadlines while maximizing the number of mission-critical tasks that meet deadlines. It also provides temporal fault tolerance to mission-critical tasks by re-scheduling them earlier.

5.6. Earliest versus latest start times

In GDS we set the task start time to its latest possible start time within the slack whereas other traditional scheduling algorithms (e.g. MCT [8]) use the earliest time. We now study whether using earliest or latest start time within slack is better. To do so, we use GDS_3, which is the same as GDS except that in GDS_3each assigned task's start time is set to the earliest time within the slack. As shown in Figs. 13 and 14, GDS's *Critical* SSR is almost identical to GDS_3. This is due to the fact that setting different task start times do not affect mission-critical tasks much because they are scheduled first. However, the *overall* SSR is better by about 10%. Thus using the latest start time rather than the earliest start time increases the number of firm and soft tasks meeting deadlines. Since we schedule mission-critical tasks first, it causes many firm and soft tasks with short deadlines to be miss deadlines while mission-critical tasks with very long deadlines can be successfully scheduled, if we set every task's start time to be the earliest time. By using the latest start time, we are able to create slacks into which firm and soft tasks can be inserted.

5.7. Performance when ranking by CCR-type

In GDS, in order to understand the merits of ranking by CCR-type, we used another algorithm named GDS_1 which is the same as GDS except that in the ranking phase GDS_1 ranks tasks only according to priority and deadline, but not CCR-type. From Fig. 15 we observe that GDS's *Critical* SSR is slightly better than GDS_1 by 3% on average.

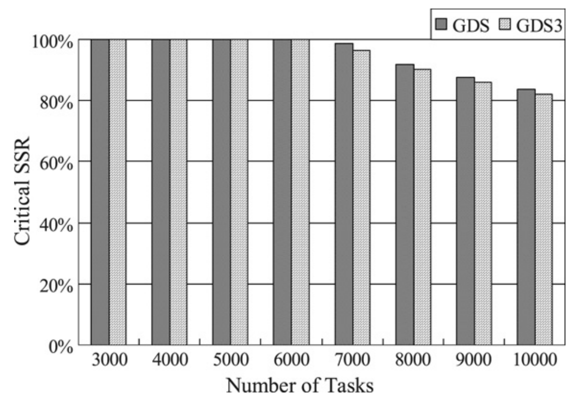

Fig. 13. *Critical* SSRs by earliest or latest start times approach within slack.

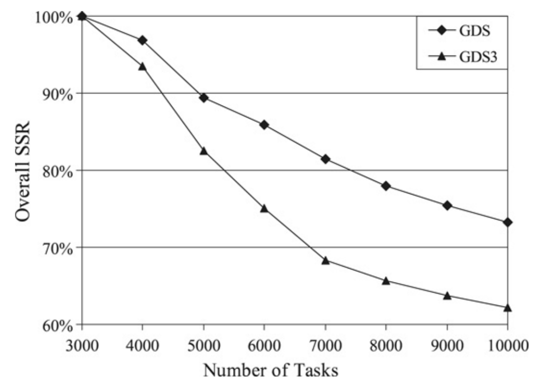

Fig. 14. *Overall* SSR by earliest or latest start times approach within slack.

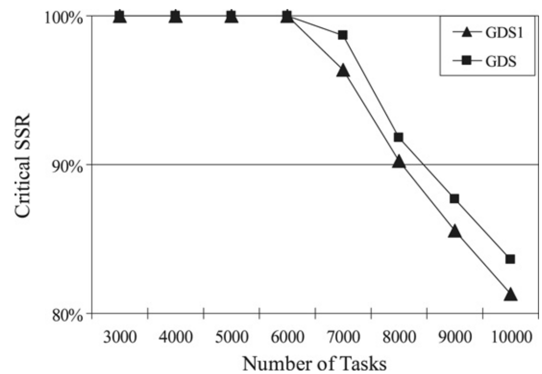

Fig. 15. *Critical* SSR with and without using CCR type in ranking.

Also, as shown in Fig. 16, with respect to *Overall SSR,* GDS yields better performance than GDS_1 by 5% on average. The better performance of GDS is brought by considering CCR-type in the ranking phase. Ranking tasks by decreasing CCR gives preference to communication-intensive for local execution. Executing communication-intensive tasks locally and dispatching computation-intensive tasks to other sites bring benefits to GDS. Since communication-intensive tasks typically have small computation size but large communication size, more tasks can meet deadlines if they are executed locally. If they are dispatched to remote sites, long transfer times will cause many of them to miss deadlines.

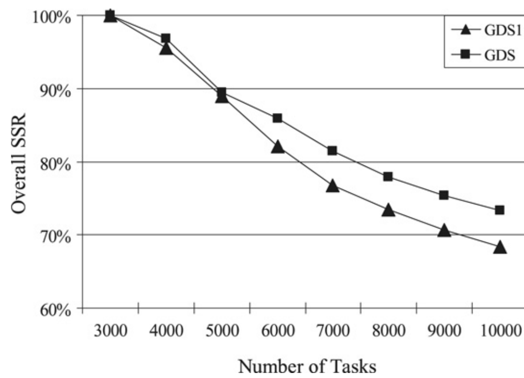

Fig. 16. *Overall* SSR with and without using CCR type in ranking.

6. Conclusion

In this paper, we proposed a novel algorithm to schedule mixed independent real-time tasks in heterogeneous grid systems. This is the first work that schedules mixed tasks (hard, firm and soft) while considering their priorities and deadlines on a heterogeneous grid. GDS is highly scalable as (i) it does not need to know the global state of the grid (which otherwise may be time intensive) as result of using peer to peer dispatch and (ii) its performance benefits increase as the number of tasks increase. Exhaustive simulations demonstrate that GDS is able to successfully schedule 10%–25% more hard tasks than existing approaches without degrading schedulability of firm and soft tasks. Furthermore, a unique shuffle phase packs the tasks in the timeline moving hard tasks ahead, enhancing their temporal fault tolerance. Thus GDS paves the way in making the grid simultaneously usable for hard and soft tasks thereby increasing the possibilities of the commercial use of the grid.

Acknowledgments

The second author's work was partly supported by NSF grants # 0408136 and 0411540.

References

[1] D. Abramson, J. Giddy, L. Kotler, High performance parametric modeling with Nimrod/G: Killer application for the global grid, in: Proc. Fourteenth IPDPS, Cancun, 2000, pp. 520–528.
[2] S. Baskiyar, N. Meghanathan, Scheduling and load balancing in mobile computing using tickets, in: Proc. Thirty Ninth SE-ACM Conference, Athens, GA, 2001.
[3] E. Caron, P.K. Chouhan, F. Desprez, Deadline scheduling with priority for client-server systems on the grid, in: Proc. Fifth International Workshop on Grid Computing, 2004, pp. 410–414.
[4] H. Casanova, et al., The AppLeS parameter sweep template: User-level middleware for the grid, in: Proc. Thirteenth International Conference for High Performance Computing, Networking, Storage and Analysis, 2000, pp. 111–126.
[5] W. Cirne, et al., Running bag-of-tasks applications on computational grids: The MyGrid approach, ICPP (2003) 407–416.
[6] I. Foster, C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, 1998.
[7] I. Foster, C. Kesselman, The Grid2, Morgan Kauffmann Publishers, 2003.
[8] R.F. Freund, et al., Scheduling resources in multi-user, heterogeneous computing environments with SmartNet, in: Seventh IEEE Heterogeneous Computing Workshop, HCW '98, Mar. 1998, pp. 184–199.
[9] A. Ghafoor, J. Yang, A distributed heterogeneous supercomputing management system, IEEE Comput. 26 (6) (1993) 78–86.
[10] O.H. Ibarra, C.E. Kim, Heuristic algorithms for scheduling independent tasks on non-identical processors, Journal of the ACM 24 (2) (1977) 280–289.
[11] M. Kafil, I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems, IEEE Concurrency 6 (3) (1998) 280–289.
[12] A. Khokhar, et al., Heterogeneous computing: Challenges and opportunities, IEEE Comput. 26 (6) (1993) 18–27.
[13] J.K. Kim, et al., Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment, Journal of Parallel and Distributed Computing 67 (2) (2007) 154–169.
[14] P.A. Laplante, Real-Time Systems Design and Analysis, Wiley-IEEE Press, 2004.
[15] C. Leangsuksun, J. Potter, S. Scott, Dynamic task mapping algorithms for a distributed heterogeneous computing environment, in: Fourth IEEE Heterogeneous Computing Workshop, HCW '95, 1995, pp. 30–34.
[16] C. Liu, J. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the ACM 20 (1) (1973) 46–61.
[17] C. Liu, S. Baskiyar, S. Li, A general distributed scalable peer to peer scheduler for mixed tasks in grids, in: Proc. Fourteenth HIPC, Goa, 2007, pp. 320–330.
[18] M. Maheswaran, et al., Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems, in: Proc. Eighth Heterogeneous Computing Workshop, 1999, pp. 30–44.
[19] M. Maheswaran, T.D. Braun, H.J. Siegel, Heterogeneous distributed computing, in: J.G. Webster (Ed.), in: Encyclopedia of Electrical and Electronics Engineering, vol. 8, Wiley, 1999, pp. 679–690.
[20] D. Menasce, D. Saha, S. Porto, Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures, Journal of Parallel and Distributed Computing 28 (1995) 1–18.
[21] National Aeronautics and Space Admin. [Online]. Available: http://liftoff.msfc.nasa.gov/academy/rocket_sci/satellites. [Accessed May. 8, 2007].
[22] Network for Earthquake Engineering Simulation. [Online]. Available: http://www.nees.org. [Accessed July. 10, 2007].
[23] S. Park, J. Kim, Chameleon: A resource scheduler in a data grid environment, in: Proc. Third IEEE International Symposium on Cluster Computing and the Grid, 2003, pp. 258–265.
[24] K. Ranganathan, I. Foster, Identifying dynamic replication strategies for a high performance data grid, in: International Workshop on Grid Computing, 2001, pp. 75–86.
[25] H. Singh, A. Youssef, Mapping and scheduling heterogeneous task graphs using genetic algorithms, in: Fifth IEEE Heterogeneous Computing Workshop, 1996, pp. 86–97.
[26] S. Smallen, et al., Combining workstations and supercomputers to support grid applications: The parallel tomography experience, in: Proc. Heterogeneous Computing Workshop, 2000, pp. 241–252.
[27] S. Smallen, H. Casanova, F. Berman, Applying scheduling and tuning to on-line parallel tomography, in: Proc. Supercomputing, Denver, CO, Nov. 2001, pp. 46.
[28] J.R. Stiles, et al., Monte Carlo simulation of neuromuscular transmitter release using M-Cell, a general simulator of cellular physiological processes, Computational Neuroscience (1998) 279–284.
[29] A. Takefusa, et al., A study of deadline scheduling for client-server systems on the computational grid, in: Proc. Tenth IEEE Symposium on High Performance and Distributed Computing, 2001, pp. 406–415.
[30] M.-Y. Wu, W. Shu, A high-performance mapping algorithm for heterogeneous computing systems, in: International Parallel and Distributed Processing Symposium, April 2001, pp. 74–79.
[31] D. Xu, K. Nahrstedt, D. Wichadakul, QoS and contention-aware multi resource reservation, Cluster Comput. 4 (2) (2001) 95–107.

Cong Liu received an M.S. in Software Engineering from Auburn University in 2007. He received a B.S. degree from Wuhan University of Technology, China with high honors. He is currently working towards a doctorate in the department of Computer Science at Univ. of North Carolina at Chapel Hill. His research interests are in the area of Grid Computing.

Sanjeev Baskiyar is an Associate Professor in the department of Computer Science and Software Engineering at Auburn University, Auburn, Alabama. He received the Ph.D. and MSEE degrees from the *University of Minnesota*, Minneapolis, B.E. degree in Electronics and Communications from the *Indian Institute of Science*, Bangalore and a B.Sc. degree in Physics with honors and distinction in Mathematics from *St. Xaviers's College*, India. He has taught courses in Computer Architecture, Real-time and Embedded Computing, Operating Systems, Microprocessor Programming and Interfacing and VLSI Design. His research interests are in the areas of Computer Architecture and Cluster Computing. His experience includes working as an Assistant Professor at Western Michigan University, a Senior Software Engineer in Unisys and a Computer Engineer in Tata Motors.