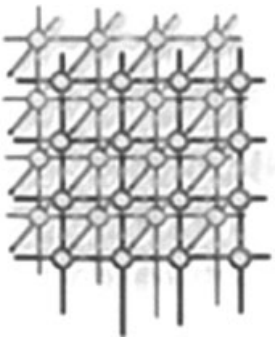
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **21**:2491–2517
Published online 27 October 2009 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1534



A strategy for scheduling tightly coupled parallel applications on clusters

H. A. Sanjay and Sathish S. Vadhiyar*, †

Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India

SUMMARY

Although various strategies have been developed for scheduling parallel applications with independent tasks, very little work exists for scheduling tightly coupled parallel applications on cluster environments. In this paper, we compare four different strategies based on performance models of tightly coupled parallel applications for scheduling the applications on clusters. In addition to algorithms based on existing popular optimization techniques, we also propose a new algorithm called Box Elimination that searches the space of performance model parameters to determine the best schedule of machines. By means of real and simulation experiments, we evaluated the algorithms on single cluster and multi-cluster setups. We show that our Box Elimination algorithm generates up to 80% more efficient schedules than other algorithms. We also show that the execution times of the schedules produced by our algorithm are more robust against the performance modeling errors. Copyright © 2009 John Wiley & Sons, Ltd.

Received 2 August 2008; Revised 27 August 2009; Accepted 27 August 2009

KEY WORDS: scheduling; tightly coupled parallel applications; clusters

1. INTRODUCTION

Various efforts exist for scheduling parallel applications on multiple resources [1–7]. Some parallel applications are loosely coupled [8–10] where the interactions among the parallel tasks are negligible whereas some parallel applications are tightly coupled [11–15].

We define tightly coupled parallel applications as those that involve frequent heavy communications among parallel tasks. Applications containing routines for solving linear systems and fast Fourier transforms (FFT) are typical examples of tightly coupled parallel applications [16–18]. We

*Correspondence to: Sathish S. Vadhiyar, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India.
†E-mail: vss@serc.iisc.ernet.in

Contract/grant sponsor: Department of Science and Technology, India; contract/grant number: SR/S3/EECE/59/2005/8.6.06

differentiate these from workflow applications [19,20] that have been successfully scheduled and executed on multiple resources [21]. While substantive data transfers take place along the edges of a workflow graph, the communications on an edge are not as frequent as the inter-task communications in tightly coupled applications and mostly occur once after the completion of an application component. Hence, while loosely coupled and workflow applications achieve good performance when executed across multiple clusters, tightly coupled applications exhibit poor performance due to low-speed network links between the clusters. Thus, tightly coupled applications are typically executed within a single cluster consisting of homogeneous machines.

In this work, we have devised, evaluated and compared algorithms for scheduling tightly coupled parallel applications on a non-dedicated cluster consisting of homogeneous machines. Our algorithms are also applicable for frameworks consisting of multiple clusters where machines within a cluster are homogeneous while machines from different clusters can be heterogeneous. When a tightly coupled parallel application is submitted to such a framework, our algorithms will be invoked simultaneously on multiple clusters and the schedule of machines from one of the clusters that gives the overall minimum execution time will be chosen for application execution. Our algorithms use performance models that predict the execution times of parallel applications, for evaluation of candidate schedules.

While many algorithms exist for scheduling loosely coupled parallel applications [8–10], very few research efforts have focused on scheduling tightly coupled parallel applications [5,6,22]. The existing algorithms for scheduling tightly coupled parallel applications are based on evolutionary techniques including simulated annealing and genetic algorithm [22,23]. In this work, we propose a novel algorithm called Box Elimination (BE) that searches the space of performance model parameters to determine efficient schedules. By eliminating large search space regions containing poorer solutions at each step and using Roulette wheel-based mechanism, our algorithm is able to generate efficient schedules within few seconds for even clusters of 512 processors. By means of a large number of real experiments and simulations, we compared our algorithm with two evolutionary algorithms, namely simulated annealing and genetic algorithm, and an incremental dynamic programming algorithm. We show that our algorithm generates up to 80% more efficient schedules than other algorithms and the resulting execution times are more robust [24] against performance modeling errors. The primary contributions of our work are development of a novel scheduling algorithm and evaluation of algorithms for scheduling tightly coupled parallel applications on non-dedicated clusters.

In the next section, we review the existing work on scheduling parallel applications. The performance models of the parallel applications used by our scheduling algorithms are explained in Section 3. In Section 4, we describe the various algorithms we use in this study. We also describe in detail our Box Elimination algorithm. In Section 5, we present our simulation experiments and show the comparison results. Section 6 gives the conclusions and Section 7 discusses the future plans.

2. RELATED WORK

Existing scheduling efforts can be classified based on the kinds of applications, the load environment on the systems, the execution model, and the objectives of scheduling. The following subsections discuss the existing work in different categories of scheduling.

DOI: 10.1002/cpe

2.1. Job vs application-level scheduling

Most of the efforts on scheduling parallel applications consider job scheduling where the focus is on reducing the average response time of a set of independent applications or jobs and in general, maximizing the throughput of a system [1,5,22,25]. The schedulers developed for batch systems [1] including PBS [26], and IBM Loadleveler [27] and the corresponding scheduling policies, namely FCFS, conservative and EASY backfilling [28] employ job scheduling. The work by Shmueli and Feitelson [2] considers a set of jobs in a batch queue and uses dynamic programming to pack the jobs in a given set of processors to improve processor utilization and reduce the mean response time and the mean slowdown of all jobs. The work by Kettimuthu *et al.* [25] proposes pre-emption strategies and analyzes the impact of pre-emption of executing jobs on the average response times of the jobs.

Application-level scheduling focuses on reducing the execution time of a single application. Some efforts on application-level scheduling utilize task graphs of the applications that express task precedence constraints and indicate the computation and communication requirements of different parts of the applications [3,29,30]. The nodes of the task graphs are then allocated to the processors based on heuristics such as assigning nodes of the longest path in the graph to processors with close proximity. Building task graphs expressing precedence constraints for complex parallel applications is non-trivial and hence cannot be adopted for a large number of parallel applications. Some efforts use performance models that the predict execution times for parallel applications [4,23,31]. These performance models are used by optimization algorithms that search the space of candidate schedules and choose the schedule with the minimum predicted execution time. Our work uses application performance models or scheduling the applications on a set of processors. Our work performs comprehensive analysis of different well-known optimization strategies and also proposes a new strategy that can effectively use the characteristics of the performance models.

2.2. Single-site vs multi-site scheduling

Various existing efforts deal with scheduling in environments consisting of multiple sites or systems in which a parallel job submitted from a system can be executed on any of the local and remote systems. These efforts can be generally classified into single-site and multi-site scheduling. In single-site scheduling, a job is executed entirely within a single local or remote system. A metascheduler, based on the load dynamics of the systems, chooses the system for executing the job. In this paradigm, multiple systems are essentially used for *job sharing*. Abawajy and Dandamudi [5] used a dynamic hierarchical scheduling policy to determine a cluster and the processors in the cluster for job allocation. He *et al.* [22] developed techniques for scheduling a set of parallel jobs on processors of a multi-cluster grid. They use a multi-tiered architecture comprising a metascheduler called MUSCLE for allocating parallel jobs to different clusters and a workload manager called TITAN at the single cluster level for scheduling in a cluster. TITAN employs genetic algorithm to choose schedules for the jobs to minimize response times of the jobs and idle times of processors and to meet job deadlines. Similar to our work, their algorithm evaluates candidate schedules using a performance prediction system called PACE [32]. Unlike our work, their scheduling architecture and performance models are intended for dedicated systems. Our work also evaluates various algorithms including genetic algorithm and shows that our Box Elimination (BE) algorithm

DOI: 10.1002/cpe

yields better schedules than the genetic algorithm. The efforts by Subramani *et al.* [33] and Sabin *et al.* [14] employ multiple simultaneous submission of a parallel job to different batch systems to improve the response time of the job. When the job starts executing in one of the clusters, the submissions to the other clusters are canceled. Sabin *et al.* [14] consider scheduling parallel jobs in a heterogeneous multi-site environment, where each site has homogeneous clusters. Thus, their scheduling environment is similar to ours. When a tightly coupled parallel application is submitted, the application is scheduled to an individual site using a scheduling algorithm. They initially used greedy algorithm and extended it to use multiple redundant requests for scheduling. However, in their model, each job specifies the processor requirements. In our work, we choose the set of processors for a parallel job, whereas, in their work, the individual clusters are space shared and local scheduling at the clusters uses backfilling with FCFS policy. Our work deals with both space-shared and non-dedicated clusters.

Some efforts deal with multi-site scheduling where an application execution spans across multiple systems. The efforts by Platt *et al.* [34] and Bal *et al.* [35] analyzed the impact of inter-cluster speeds on the performance of parallel applications when executed across wide-area clusters. The work by Ernemann *et al.* [36] analyzes the mean response times of synthetic applications when executed across multiple clusters for different ratios of execution times of applications when executed on a single local cluster and on multiple clusters. Their results show that multi-cluster computing can yield improved response times due to the decreased queue waiting times as long as the execution times due to multi-cluster computing do not increase more than 1.25 times the execution times on local clusters. The work by Li [6] also deals with scheduling a set of parallel jobs on multiple clusters. It uses a simplifying cost model that predicts the execution time across multiple clusters in terms of computation-to-communication ratio of parallel applications and the ratio of communication bandwidth within a parallel machine to the communication bandwidth of inter-cluster links. They propose a minimum effective execution time algorithm that optimizes their cost model by minimizing the number of clusters used for parallel application execution. The work performs simulations for various fixed values of application computation-to-communication ratios. In our work, we determine the characteristics of real parallel applications using performance modeling techniques. Bucur and Epema have extensively studied the benefits of co-allocation of processors from different clusters for job executions [37]. In their work, they analyze the impact of using different scheduling policies, component sizes and number of components on co-allocation. Using a large number of simulations with various workload logs, application characteristics and inter-cluster speeds, they show that execution of multi-component jobs across multiple clusters can reduce the mean response times of jobs and improve the processor utilization when the number of components and component sizes are restricted.

In our work, we deal with single-site or single-system scheduling of parallel applications to achieve job sharing. Our work does not consider multi-site scheduling or co-allocation since our work is intended for tightly coupled parallel applications whose performance degrade when executed across multiple clusters.

2.3. Dedicated vs non-dedicated execution

Many existing scheduling strategies consider dedicated environments for application execution [1,25,29]. In these environments, scheduling decisions consider only the computation and

DOI: 10.1002/cpe

communication characteristics of the application and do not consider the system and network loads. Few efforts consider scheduling of applications in non-dedicated environments [7,23,38]. Grid Harvest system [38] considers the arrival rate of jobs, machine utilization, service time and machine capacity for scheduling applications. However, it does not consider the application's communication and synchronization requirements, and hence is only suitable for applications with independent tasks. The work by Yarkhan and Dongarra [23] uses the simulated annealing-based approach for scheduling tightly coupled parallel applications on non-dedicated grids. Our work compares different algorithms including simulated annealing for non-dedicated environments and shows by means of experiments that simulated annealing-based techniques give inefficient schedules in many cases.

2.4. Type of parallel applications

Most of the existing efforts in scheduling parallel applications deal with independent tasks [8–10] that do not consider the communication characteristics of the application and the communication capacities and loads of the links in the system.

Recently, workflow applications have been widely deployed and scheduled on multi-cluster environments [39–42]. The work by Yu *et al.* [39] provides a taxonomy and a description of the various workflow scheduling algorithms and compares the algorithms both in terms of time complexities and ability to meet QoS constraints using the GridSim simulator. The work by Wieczorek *et al.* [40] proposes a dynamic constraint algorithm for meeting two important criteria/objectives of minimizing time and cost for execution. As described in Section 1, the communication requirements of the workflow components are less frequent and less intensive when compared with tightly coupled applications. Hence, scheduling strategies that have been developed for allocating the workflow application components on different clusters cannot be used for tightly coupled applications due to the higher frequency of inter-task communications in these applications.

There have also been efforts at scheduling a set of tightly coupled parallel applications on a set of multiprocessor clusters [22,23,29]. Various effective algorithms exist for scheduling applications with dependent tasks on a set of machines [29,43,44]. The input for these algorithms is a directed acyclic graph (DAG) that represents the dependencies between the tasks of an application. As mentioned earlier, constructing a DAG for a large-scale parallel application is non-trivial when compared with using performance model equations for the applications. These equations can either be supplied by the application developer or can be constructed automatically using profiling runs.

3. APPLICATION PERFORMANCE MODELS

In our previous work [45], we developed performance modeling strategies for predicting the execution times of tightly coupled parallel applications on non-dedicated homogeneous resources. We calculated the time taken for the execution of a parallel application as:

$$T(N, P, minAvgAvailCPU, minAvgAvailBW) = \frac{f_{comp}(N)}{f_{cpu}(minAvgAvailCPU) \cdot f_{Pcomp}(P)} + \frac{f_{comm}(N)}{f_{bw}(minAvgAvailBW) \cdot f_{Pcomm}(P)} \quad (1)$$

DOI: 10.1002/cpe

where

- N: problem size or data size; P: number of processors;
- *minAvgAvailCPU*, *minAvgAvailBW*: represent the transient CPU and network characteristics, respectively.
- f_{comp}, f_{comm}: indicate the computational and communication complexity, respectively, of the application in terms of problem size;
- f_{cpu}: function to indicate the effect of processor loads on computations;
- f_{Pcomp}: used along with computational complexity to indicate the computational speedup or the amount of parallelism in computations;
- f_{bw}: function to indicate the effect of network loads on communications;
- f_{Pcomm}: used along with communication complexity to indicate the communication speedup or the amount of parallelism in communications.

The formula shown in Equation (1) splits the execution time of a parallel application into two parts, f_{comp} and f_{comm}, for representing computation and communication aspects, respectively, of the parallel application. This representation is useful for scheduling purposes since a scheduler can allocate the appropriate CPU and network resources for the application based on the computation and communication requirements, respectively, of the application. The scalability of the computational and communication times with the increasing number of processors is represented by f_{Pcomp} and f_{Pcomm}, respectively. Since, in most parallel applications, the execution times decrease with the increase in number of processors, these functions are contained in the denominators. The increase in CPU and network loads on non-dedicated systems increases the computation and communication times, respectively. *minAvgAvailCPU* and *minAvgAvailBW* represent the inverse of the CPU and network loads, respectively. Hence, the corresponding functions, namely, f_{cpu} and f_{bw}, are contained in the denominators.

The formula shown in Equation (1) generalizes the parallel runtime equations of many parallel numerical drivers that deal with memory-resident data [16,17]. For example, the parallel runtime equation for an FFT application using binary exchange algorithm on a dedicated homogeneous system is $T = t_c(N/P)\log N + t_m(N/P)\log P$, where t_c is the time for a floating point operation and t_m is the transfer time for a unit message. Thus, the various functions of Equation (1) for the parallel FFT application are: $f_{comp} = N\log N$, $f_{comm} = N$, $f_{Pcomp} = P$ and $f_{Pcomm} = P/\log P$. Since Equation (1) represents the parallel runtime for non-dedicated systems, it includes the effects of CPU and network loads in f_{cpu} and f_{bw}, respectively. The values for t_c and t_m are determined as model coefficients using a multi-phase linear regression procedure.

In order to calculate *minAvgAvailCPU* and *minAvgAvailBW*, we measure *AvailCPU* and *AvailBW*. *AvailCPU* is a fraction of the CPU that can be used for the application and is inversely proportional to the amount of CPU load. *AvailBW* of a link is the bandwidth on the link available to an application. *AvailBW* is usually lesser than the link capacity and is inversely proportional to the network load on the link. Network Weather Service (NWS) [46,47], a tool for forecasting system parameters, was used for obtaining these values. During the training of the model functions for an application, we measure *AvailCPU*s and *AvailBW*s on all processors and links involved in application execution at periodic intervals of time from the beginning to the end of the application execution. We then calculate for each processor and link, *AvgAvailCPU* and *AvgAvailBW*, respectively. These are the averages of the periodic *AvailCPU*s and *AvailBW*s collected during the application execution.

DOI: 10.1002/cpe

Finally, we calculate *minAvgAvailCPU* and *minAvgAvailBW* values by finding the minimum of *AvgAvailCPU* and *AvgAvailBW* values, respectively, on all processors and links. By considering *minAvgAvailCPU* and *minAvgAvailBW*, we assume that the slowest processor and link used by the application affect the overall execution time. This is true for many of the tightly coupled regular parallel applications executing on homogeneous nodes in a cluster [16,17,48].

By using a multi-phase procedure, we evaluate various candidate functions for f_{comp}, f_{comm}, f_{cpu}, f_{bw}, f_{Pcomp} and f_{Pcomm} using linear regression and choose a combination of functions that give the minimum average percentage prediction error. By means of a large number of experiments with different clusters and applications, we showed that our performance models gave highly accurate predictions of execution times with a less than 30% average percentage prediction errors for all cases. We also devised a strategy for scaling the coefficients of computational complexities for predicting the execution time of a parallel application on a cluster using the performance models developed for the application in another cluster. The details of the performance models, the modeling procedure and the results can be found in our previous work [45].

The modeling techniques are intended for simple parallel application kernels, which are invoked in complex parallel applications. These parallel kernels have single phase of uniform computations and communications and are integral to many scientific applications. Equation (1) represents a coarse-level model and does not include fine-level performance behavior of the applications including computation–communication overlap, memory access stride and range, cache misses and the corresponding system parameters including cache configurations and memory bandwidth.

For our problem of scheduling tightly coupled parallel applications on a non-dedicated or dedicated homogeneous cluster, we use the performance model of Equation (1) for comparing various candidate schedules and choosing the best schedule with the minimum predicted execution time for the application. The *minimum execution-time multiprocessor scheduling problem* is known to be NP-hard in its generalized form, and is NP-hard even in some restricted forms [49]. A brute-force approach of evaluating all possible candidate schedules to choose the best schedule will require nearly 2^N evaluations for a cluster of N machines. Thus the brute-force approach becomes intractable even for moderately sized clusters. Hence, we propose heuristic algorithms for determining the best schedule.

4. ALGORITHMS

In this section, we describe four algorithms for finding schedules of machines in a cluster for execution of a parallel application with a given problem size. The first three algorithms are based on popular optimization techniques while the last algorithm, the Box Elimination algorithm, is our contribution. All of these algorithms use Equation (1) shown in Section 3 for evaluation of candidate schedules in terms of the predicted execution times of the schedules. Timers are set in all the algorithms by means of the *alarm*() function to make the algorithms time-tunable. This allows the user of our scheduling codes to specify the duration of scheduling. The algorithms, at the point of expiry of the timer, exit from their current operations and return the best schedules at that point. In all our algorithms, max_procs denotes the total number of available cores/processing units in the cluster, out of which a set of machines is returned as the best schedule. In a cluster consisting of a set of multi-core systems, our algorithms consider each core or a processing unit as a separate machine.

DOI: 10.1002/cpe

Such a cluster has network heterogeneity since the communication speeds between the cores in a system are greater than the communication speeds between the cores in different systems. The performance of a tightly coupling application on such a cluster of multi-core systems is dictated by the slowest link connecting the cores in different systems. Since our performance model considers the minimum bandwidth corresponding to the slowest link using the *minAvgAvailBW* parameter, considering a single core as a separate machine is adequate for our problem of scheduling tightly coupled parallel applications. In a multi-cluster setup, the algorithms are invoked for each of the clusters and the cluster that contains the overall best schedule with the minimum execution time is chosen for executing the application.

4.1. Simulated annealing

Yarkhan and Dongarra [23] developed a simulated annealing-based algorithm for choosing a set of machines for execution of a tightly coupled parallel application on a grid. This algorithm initially generates a random schedule and perturbs the schedule at various steps. Poorer schedules with higher execution times are accepted probabilistically based on a temperature value. We show the algorithm in Figure 1. The algorithm by Yarkhan *et al.* used a simulator for the parallel application as an objective function for search space exploration. We have modified the algorithm to use our performance model of Equation (1) both as an objective function for comparison of schedules and for generating the initial schedule of machines (lines 2–7).

4.2. Genetic algorithm

Genetic algorithm has been used in a number of scheduling problems [22,50,51]. In this algorithm, a population of chromosomes is initially generated and the chromosomes undergo cross-over and mutations across various generations. The chromosomes having high fitness values are retained over successive generations. In our problem, a chromosome corresponds to a schedule or a set of machines. We calculate the fitness of a chromosome as the reciprocal of the execution time of the corresponding schedule. The algorithm is shown in Figure 2. As can be seen, we not only use the performance model equation for evaluating the fitness of the chromosomes, but also during mutations to generate better mutated chromosomes (lines 10–22).

4.3. Dynamic programming

Dynamic programming finds a solution to a problem in terms of solutions to a set of subproblems. The overall solution is expressed as a recursive equation containing solutions to the smaller subproblems. For our problem of scheduling tightly coupled parallel application, we apply dynamic programming to find the best schedule of n machines, $sched_n$, in terms of the best schedule of $n-1$ machines, $sched_{n-1}$. The recursive formulation we use for dynamic programming is shown in Equation (2). t denotes the execution time for a schedule determined using the performance model equation and *minMc* is the machine corresponding to the minimum of execution times of all schedules constructed out of machines in $sched_{n-1}$ and another machine. The algorithm is shown in Figure 3. The schedule is constructed incrementally by starting with a schedule of one machine most suitable for the application and adding machines, that give the best predicted execution time, to

DOI: 10.1002/cpe

```
1  Algorithm:Simulated Annealing (SA)
2  for each machine i do
3      Evaluate execTime[i] of machine i using Eq. 1 ;
       /* For bandwidth in Eq. 1, use average bandwidth of links to i          */
4  end
5  sortedList = machines in ascending order of execTime ;
6  for procs = max procs to 2 do
7      curSched = top procs machines in sortedList ; curExecTime = execution time of curSched
       using Eq. 1;
8      for temperature=100 to 1 in steps of 0.8 do
9          num accepts = 0 ;
10         for steps = 1 to 80 do
               /* Perturb schedule randomly                                    */
11             newSched = Randomly replace a machine in curSched ;
12             newExecTime = execution time of of newSched using Eq. 1 ;
13             if newExecTime < curExecTime then
14                 prevExecTime = curExecTime ; curSched = newSched ;
                   curExecTime = newExecTime ;
15                 num accepts = num accepts + 1 ;
16             end
17             else
18                 r = random number ;
19                 if r < e^((prevExecTime − curExecTime)/temperature) then
20                     prevExecTime = curExecTime; curSched = newSched;
                       curExecTime = newExecTime;
21                     num accepts = num accepts + 1;
22                 end
23             end
24             if curExecTime > prevExecTime or num accepts > 20 break ;
25         end
26         if curExecTime > prevExecTime break ;
27         break;
28     end
29 end
```

Figure 1. Simulated annealing (SA).

the schedule. At each stage of the algorithm, we also consider the second best schedule of machines for the addition of machines in the next step (line 18). This is done to reduce the effects of local minima.

$$\begin{aligned} sched_n &= \{sched_{n-1}, minMc\} \text{ s.t.} \\ t(\{sched_{n-1}, minMc\}) &= Min_{i \notin sched_{n-1}} t(\{sched_{n-1}, i\}) \end{aligned} \tag{2}$$

4.4. Box Elimination

The Box Elimination algorithm is based on the observation that the predicted execution time of an application with a given problem size on a set of machines is expressed in Equation (1) in terms of *minAvgAvailCPU*[‡] (maximum CPU load) of the machines in the set, *minAvgAvailBW* (maximum

[‡]For description of the terms *minAvgAvailCPU*, *minAvgAvailBW*, *AvgAvailCPU* and *AvgAvailBW* used in this section, please refer Section 3.

DOI: 10.1002/cpe

```
1  Algorithm:Genetic Algorithm (GA)
2  for procs = max procs to 2 do
3      Randomly generate initial population of 60 chromosomes of size procs;
4      Evaluate the fitness of chromosomes using Eq. 1 ;
5      for generations = 1 to 50 do
           /* CrossOver                                                    */
6          Divide chromosomes into pairs ;
7          for each pair of chromosome do
8              Pick a random chromosome position ; Cross-over chromosome segments from that
               position ;
9          end
           /* After cross-over, there are 120 chromosomes in the population */
           /* Mutation                                                     */
10         for each of 60 newly generated chromosomes do
11             for mutation steps = 1 to 20% of chromosome length do
12                 Randomly choose a machine X in a chromosome to mutate ; topfitness = 0;
13                 for each machine Y not in chromosome do
14                     Replace machine X with Y to form mutated chromosome ;
15                     Evaluate fitness of mutated chromosome using Eq. 1 ;
16                     if fitness > topfitness then
17                         topfitness = fitness ; topmc = Y ;
18                     end
19                 end
20                 Replace machine X with topmc in the chromosome ;
21             end
22         end
23         Evaluate the fitness of 120 chromosomes using Eq. 1 ;
           /* Selection                                                    */
24         Select the best 60 ranking individuals for next generation;
25     end
26 end
```

Figure 2. Genetic algorithm (GA).

network load) of the links between the machines and number of machines in the set. Unlike other algorithms, which search through a space of schedules of machines, the Box Elimination algorithm searches through a space of hypothetical points, where each point corresponds to a (*minAvgAvailCPU*, *minAvgAvailBW*, number of processors) tuple. Each hypothetical point is then mapped to a schedule of machines whose *minAvgAvailCPU* and *minAvgAvailBW* values are greater than and closest to the corresponding values in the hypothetical point. The strength of the algorithm is in its ability to eliminate search space regions of hypothetical points based on a hypothetical point that was searched. The elimination of the search space points is based on the characteristic of the performance model function used in Equation (1) and is not possible in other algorithms where the search space points are schedules or lists of machines[§].

The algorithm works on a 3-D box of grid points with each grid point corresponding to a (*minAvgAvailCPU*, *minAvgAvailBW*, number of processors) tuple. The box is bounded on the x-axis

[§]The results for the other algorithms presented in this work should be considered as evaluations of implementations of the algorithms that use the commonly used search space of schedules [23,52] and not as evaluations of the inherent characteristics of the algorithms themselves.

DOI: 10.1002/cpe

```
1  Algorithm:Dynamic Programming (DP)
2  for each machine i do
3      Evaluate execTime[i] of machine i using Eq. 1 ;
       /* For bandwidth in Eq. 1, use average bandwidth of links to i          */
4  end
5  curSchedule = {Machine with min. execTime};
6  curScheduleList = {curSchedule} ;
7  curMachineCount = 1 ;
8  while curMachineCount < max_procs do
9      newMachineCount = curMachineCount + 1 ;
10     newScheduleList = φ ;
11     for curSchedule ∈ curScheduleList do
12         Form schedules with newMachineCount from machines in curSchedule and a single
           machine not in curSchedule ;
13         Evaluate execution times of schedules using Eq. 1 ;
14         Pick schedules with the minimum and second minimum execution times to form
           bestNewSchedule and nextBestNewSchedule ;
15         newSchedule = {bestNewSchedule, nextBestNewSchedule} ;
16         newScheduleList = newScheduleList + newSchedule ;
17     end
18     curScheduleList = top two schedules from newScheduleList with minimum execution times ;
19     curMachineCount = newMachineCount ;
20 end
```

Figure 3. Dynamic programming (DP).

by (cpu_{min}, cpu_{max}), the minimum and maximum, respectively, of *AvgAvailCPU* of *all* machines in the cluster, on the y-axis by (bw_{min}, bw_{max}), the minimum and maximum, respectively, of *AvgAvailBW* of all the links, and on the z-axis by ($P_{min} = 1$, $P_{max} = max\_procs$). The x-axis values are incremented in steps of 0.1. For y-axis values, we use different increment steps based on the maximum bandwidth of the links. We use increment steps of 10 Mbps for maximum bandwidths of 100 Mbps and 1 Gbps, 50 Mbps for maximum bandwidth of 5 Gbps and 100 Mbps for maximum bandwidth of 10 Gbps.

The algorithm, shown in Figure 4, begins by finding the center grid point, (cpu_c, bw_c, P_c) of the 3-D box. This point is then mapped to a schedule of machines whose *minAvgAvailCPU* and *minAvgAvailBW* values are greater than and closest to cpu_c and bw_c, respectively (line 4). The schedule is then mapped back to a corresponding hypothetical point (cpu_g, bw_g, P_g) in the 3-D box (line 6). Eight 3-D sub-boxes, SB_1–SB_8, are formed in the 3-D box with reference to (cpu_g, bw_g, P_g) as shown in Figure 5. The algorithm then repeatedly generates a random point in the uncovered region of the 3-D box, finds closest schedule of machines and maps the schedule to a hypothetical point in the box (lines 15–18). For a hypothetical grid point (cpu_h, bw_h, P_h) two sub-boxes corresponding to two regions of search space are eliminated:

1. Sub-box bounded by (cpu_{min}, bw_{min}, P_{min}) and (cpu_h, bw_h, P_h). This elimination is based on the observation that there exists no point ($cpu_p \leq cpu_h$, $bw_p \leq bw_h$, $P_p \leq P_h$) for which the predicted execution time by Equation (1) is less than the predicted execution time for (cpu_h, bw_h and P_h). Thus, the sub-box corresponds to a search space of poorer solutions.
2. Sub-box bounded by (cpu_h, bw_h, P_h) and (cpu_{max}, bw_{max}, P_{max}). The elimination of this sub-box is made possible by our mapping function, *FindScheduleWithConstraints*, shown in Figure 6, that maps a hypothetical grid point, (cpu_{grid}, bw_{grid}, P_{grid}) to a schedule of

DOI: 10.1002/cpe

machines defined by ($cpu_{real} \geq cpu_{grid}$, $bw_{real} \geq bw_{grid}$, P_{real}). This schedule is then mapped to the closest grid point (cpu_h, bw_h, P_h). The mapping function uses heuristics and attempts to maximize the number of processors with values of *minAvgAvailCPU* and *minAvgAvailBW* at least equal to cpu_{grid} and bw_{grid}, respectively. Thus, the mapping function attempts to minimize the number of schedules of machines in the sub-box that represents schedules with *minAvgAvailCPU* and *minAvgAvailBW* and number of processors higher than (cpu_h, bw_h, P_h).

The objective of the mapping function is to determine a maximal set of machines whose *minAvgAvailCPU* and *minAvgAvailBW* values, denoted by cpu_{real} and bw_{real}, respectively, are greater than cpu_{grid} and bw_{grid}, respectively, such that the sub-box containing higher values of *minAvgAvailCPU*, *minAvgAvailBW* and number of processors will contain only few schedules and hence can be eliminated. The mapping procedure has two phases. In the first phase (lines 5–17), a set of machines, *initSchedule*, whose *AvgAvailCPU* values are $\geq cpu_{grid}$, is formed. However, the *minAvgAvailBW* value of this set can be $< bw_{grid}$. Hence, the machines are successively removed from the set in a number of steps until the minimum of the *AvgAvailBW* values of the links connecting the machines in the set is $\geq bw_{grid}$. In each step of removal, the machine with the greatest number of links whose *AvgAvailBW* values are less than bw_{grid} is removed. The resulting set formed in the first phase consists of machines whose *minAvgAvailCPU* and *minAvgAvailBW* values are greater than cpu_{grid} and bw_{grid}, respectively. However, this may not be the maximal set due to the heuristic followed in choosing a machine for removal from *initSchedule* in a given step of the first phase. Hence, in the second phase of the mapping procedure (lines 18–23), we incrementally add machines that were removed in the first phase. In each step of addition, we choose a machine such that the minimum of the *AvgAvailBW* of the links connecting the machine to the machines already in the set is $\geq bw_{grid}$. The final schedule of machines corresponds to (cpu_{real}, bw_{real}, P_{real}) and are mapped to the closest grid point in the 3-D box, (cpu_h, bw_h, P_h).

initSchedule represents a schedule with maximal number of machines whose *minAvgAvailCPU* is at least equal to cpu_{grid}. By choosing the machine with the greatest number of links whose *AvgAvailBW* values are less than bw_{grid} for removal in the first phase, our mapping function attempts to remove minimal number of machines from the maximal schedule, *initSchedule*. The mapping function also attempts to increase the number of machines in the second phase by adding some of the machines removed in the first phase. Hence, the region bounded by (cpu_h, bw_h, P_h) and (cpu_{max}, bw_{max} and P_{max}) that represent schedules with larger number of machines will have only few valid schedules. Moreover, since cpu_{real} and bw_{real} are lower bounds for a schedule of machines, increasing any one of the lower bounds will only a lead to schedules with smaller number of machines. Hence, for practical purposes and to explore a large number of valid good schedules within a certain time, the sub-box denoting values larger than (cpu_h, bw_h, P_h) can be eliminated. In Figure 5, the two eliminated sub-boxes, SB_1 and SB_8, corresponding to the grid-point (cpu_g, bw_g, P_g) are darkly shaded.

To generate random hypothetical points in the 3-D box, we use a Roulette-wheel-based mechanism where the Roulette-wheel is initially formed of six equal-sized sectors corresponding to the six sub-boxes, SB_2–SB_7, with reference to the grid point (cpu_g, bw_g, P_g) shown in Figure 5. When a generated random point in the 3-D box results in a schedule better than any of the previously determined schedules, the sub-box containing the random point is determined, and the corresponding sector in the Roulette wheel is enlarged. Thus, the probability of generating random points

DOI: 10.1002/cpe

```
1  Algorithm:Box Elimination (BE)
2  UnExploredGridProints = {all grid points} ;
3  Find the center grid point of the box, cpu_c, bw_c, P_c ;
4  (currentSchedule, cpu_r, bw_r, P_r) = FindScheduleWithConstraints (cpu_c, bw_c, P_c) ;
5  currentExecTime = Evaluate currentSchedule using Equation 1 ;
6  (cpu_g, bw_g, P_g) = FindClosestGridPoint(cpu_r, bw_r, P_r) ;
7  UnExploredGridProints = UnExploredGridProints− (cpu_g, bw_g, P_g) ;
8  FormEightSubBoxes(cpu_g, bw_g, P_g) ;
9  EliminateTwoSubBoxes(cpu_g, bw_g, P_g) ;
10 UnExploredGridProints = UnExploredGridProints− eliminated sub boxes ;
11 Initialize counters for six sub boxes to 1 ;
12 while UnExploredGridPoints ≠ φ do
13     Form a Roulette wheel consisting of six sectors for six sub boxes based on counter values;
14     Probabilistically choose a sub box, b_i, based on the Roulette wheel ;
15     Choose a random point, cpu_new, bw_new, P_new in the sub box ;
16     (newSchedule, cpu_newr, bw_newr, P_newr) = FindScheduleWithConstraints (cpu_new, bw_new,
       P_new) ;
17     newExecTime = Evaluate newSchedule using Equation 1 ;
18     (cpu_newg, bw_newg, P_newg) = FindClosestGridPoint(cpu_newr, bw_newr, P_newr) ;
19     UnExploredGridProints = UnExploredGridProints− (cpu_newg, bw_newg, P_newg) ;
20     FormEightSubBoxes(cpu_newg, bw_newg, P_newg) ;
21     EliminateTwoSubBoxes(cpu_newg, bw_newg, P_newg) ;
22     UnExploredGridProints = UnExploredGridProints− eliminated sub boxes ;
23     if newExecTime < currentExecTime then
24         Increment counter value for sub box, b_i ;
25         currentExecTime = newExecTime ;
26         currentSchedule = newSchedule ;
27     end
28 end
```

Figure 4. Box elimination (BE).

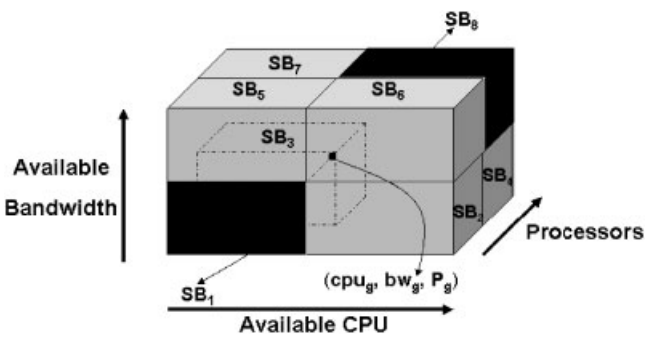


Figure 5. 3-D box of (cpu, bandwidth, processors) tuples for box elimination.

in the sub-box or regions containing good schedules or solutions increases over time. When the search space in a sub-box is completely explored, its corresponding sector is eliminated in the Roulette-wheel and the remaining sector sizes are adjusted so that the sum of the probabilities of generating random points in the sectors equals 1.

Compared with other algorithms, the Box Elimination (BE) algorithm performs several steps in processing a schedule due to generation of a random point of the 3-D box, mapping the point to a

DOI: 10.1002/cpe

```
1  Algorithm:FindScheduleWithConstraints()
   input  : cpu_grid, bw_grid, P_grid
   output: schedule, cpu_real, bw_real, P_real
2  initSchedule = list of machines whose available CPU ≥ cpu_grid ;
3  currentSchedule = initSchedule ;
4  min_bw = minimum of bandwidths of links connecting machines in currentSchedule ;
5  while min_bw < bw_grid do
6  |  for all machines m ∈ currentSchedule do
7  |  |  violation_count[m] = 0 ;
8  |  |  avg_bw[m] = 0 ;
9  |  end
10 |  for each machine m ∈ currentSchedule do
11 |  |  bw_violation_count[m] = number of links between m and other machines in
   |  |  currentSchedule whose bandwidths are < bw_grid ;
12 |  |  avg_bw[m] = average of bandwidths of links connecting m and other machines in
   |  |  currentSchedule ;
13 |  end
14 |  top_m = machine with maximum bw_violation_count ;
   |  /* If machines have equal bw_violation_count, machine with minimum avg_bw is
   |     chosen                                                                  */
15 |  currentSchedule = currentSchedule − top_m ;
16 |  min_bw = minimum of bandwidths of links connecting machines in currentSchedule ;
17 end
18 for each machine m ∈ initSchedule and ∉ currentSchedule do
19 |  min_bw = minimum of bandwidths on links connecting m and machines in currentSchedule ;
20 |  if min_bw > bw_grid then
21 |  |  currentSchedule = currentSchedule + m ;
22 |  end
23 end
24 schedule = currentSchedule ;
25 P_real = number of machines in schedule ;
26 cpu_real = minimum of available CPUs of machines in schedule ;
27 bw_real = minimum of bandwidths of links connecting machines in schedule ;
28 return (schedule, cpu_real, bw_real, P_real) ;
```

Figure 6. FindScheduleWithConstraints().

schedule using the iterative mapping procedure, mapping the schedule back to a grid point in the box, elimination of sub-boxes and adjusting the Roulette wheel. However, the number of processed schedules is less than the other algorithms due to the elimination mechanism followed.

5. EXPERIMENTS AND RESULTS

We compared the various scheduling algorithms using both real experiments on 104 cores of an Intel Xeon cluster and simulation experiments for larger number of cores or processors.

5.1. Real experiments

We evaluated our algorithms using the Molecular Dynamics (MD) application with 2400 molecules on a 128-core cluster, consisting of 16 nodes, with each node consisting of two Intel Quad core Xeon E5440 CPUs running at 2.83 GHz. Each node has 16 GB RAM, 500 GB total hard disk

DOI: 10.1002/cpe

capacity and runs Debian Etch (version 4.0) with Linux kernel 2.6.24. The nodes are connected by Gigabit Ethernet through a 24-port Gigabit Ethernet switch. We used 104 cores and 13 nodes for our experiments[¶].

For each experiment corresponding to an execution of the MD application, we introduced synthetic CPU and network loads in the system by continuously executing synthetic CPU and network loading programs on the cores in the background and maintained the loads for the duration of the experiment. For CPU loading for an experiment, a set of cores was randomly chosen out of the available 104 cores in the system and synthetic loading programs were run on the cores in the set. The Unix command, *taskset*, was used to set the affinity of a CPU loading process to a particular core. The amount of loading on each core was randomly varied by running a random number of loading programs on the core such that the available CPU value of the core is varied between 6.5 and 72% of the total CPU. Small available CPU percentages imply large loading of the core. For network loading, we used a loading program to introduce synthetic network loads on the links of the system and to reduce the available bandwidths of the links. We introduced synthetic network loads on the links connecting the cores of different nodes. For an experiment, a random number (between 1 and 8) of source–destination pairs was chosen out of all possible source–destination pairs in the system where the source and destination represent cores of different CPUs. Random amounts of network loads were introduced on the links between the source–destination pairs by running the synthetic network program so as to vary the available bandwidths of the links between the cores from 2 to 80% of the total bandwidth capacities of the links. For executing a synthetic network load between two cores of two different nodes, a 2-process MPI loading program was executed on the CPUs of the nodes containing the two cores.

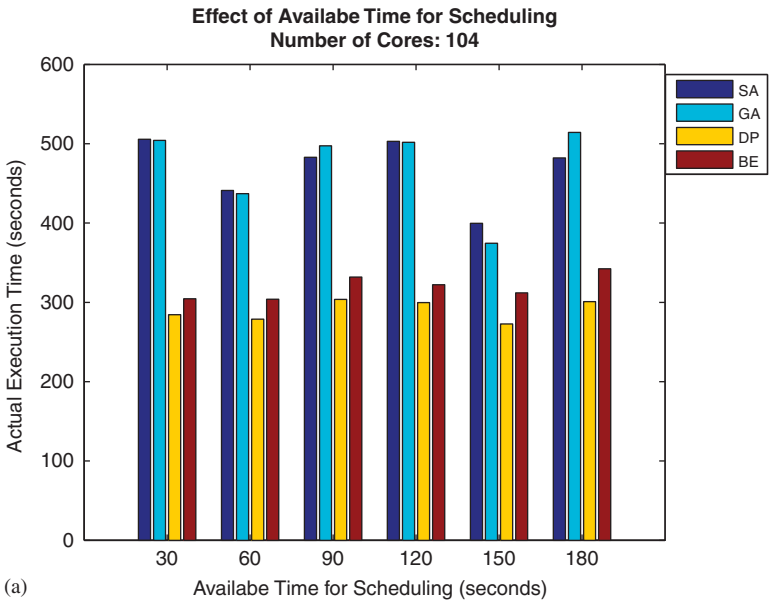
After loading the CPUs and network links for an experiment, we observed the available CPUs and bandwidths of the processors and links, respectively, using NWS [46]. Each of the scheduling algorithms was then invoked with these available CPUs and bandwidths along with the problem size and the resulting schedules of machines determined by the algorithms were obtained. The schedules determined by the algorithms were then compared by executing the MD application on each of the schedules and observing the actual execution times on the schedules.

Figure 7(a) shows the actual execution times of the MD application with 2400 molecules for different available times for scheduling. Figure 7(b) shows the corresponding predicted execution times obtained from the performance models of MD application for the same schedules, problem size and resource loads. Each bar in the figures corresponds to an average of 10 experiments.

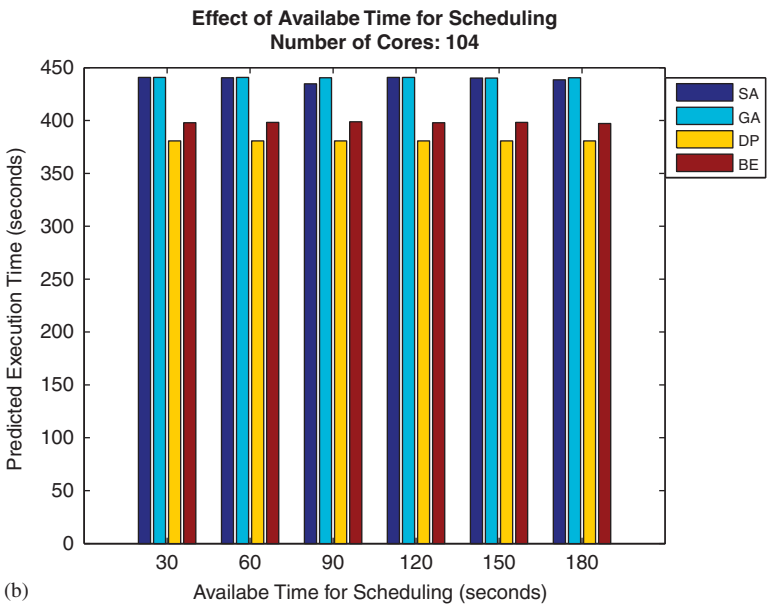
A few observations can be made from the figures. We find that in all cases the dynamic algorithm (DP) gives the best schedules with minimum execution times. The execution times corresponding to the DP algorithm are about 8.5% less than the execution times corresponding to the box elimination (BE) algorithm in all cases. This is because the DP algorithm is able to incrementally evaluate different schedules and determine a good schedule within the allotted times for 104 cores or processors. Our BE algorithm involves some randomness in the generation and evaluation of the schedules. Hence, the schedules determined by the BE algorithm, though competent with the schedules by the DP algorithm, result in slightly larger execution times. The random strategies

[¶] We used only MD application for the real experiments due to the limited system time we obtained on the 128-core system. We have shown results with MD and three more applications in our simulation experiments of Section 5.2.

DOI: 10.1002/cpe



(a)



(b)

Figure 7. Comparison of Algorithms in Terms of Actual and Predicted Execution Times for Different Times Available for Scheduling on 104 Cores of Intel Xeon Cluster. Application: MD with 2400 molecules: (a) actual execution times and (b) predicted execution times.

DOI: 10.1002/cpe

employed by the simulated annealing (SA) and genetic algorithm (GA) were not able to converge on good schedules in the time allotted to the algorithms and hence their execution times were significantly higher than the DB and BE algorithms. We also find that contrary to our expectations, the execution times do not decrease with the time available for scheduling. This is because all the algorithms are able to converge to good schedules within 30 s for 104 cores. The algorithms are not able to find better schedules with the availability of more time for scheduling.

The results also show that the predicted execution times shown in Figure 7(b) are highly accurate and are within 10–30% of the actual execution times shown in Figure 7(a). This confirms the conclusions in our previous work [45] regarding the accuracy of the performance models. We also find that the relative differences between the predicted execution times of the schedules for the different algorithms, as shown in Figure 7(b), match the relative differences between the actual execution times shown in Figure 7(a). The predictions also show that the dynamic programming algorithm gives the best schedules, the schedules by the box elimination algorithm are competent with the schedules by the DP algorithm, the SA and GA algorithms give the worst performance, and the execution times do not decrease with the availability of more time for scheduling. Thus, the predicted execution times by the performance models can be used to adequately compare the different algorithms for larger configurations.

The results in this section showed that the DP algorithm consistently gave the best performance. However, we expect and show in the later experiments that for a large number of cores or algorithms, the DP algorithm will give poorer schedules since the incremental evaluation of schedules followed in the DP algorithm will not adequately evaluate schedules containing larger number of processors within the allotted time for scheduling. The results in this section also showed that the availability of more time for scheduling does not improve the quality of the schedules determined by the algorithms. We claim that for larger number of processors, the algorithms will take more time to determine good schedules and hence the execution times for the schedules will decrease with the increasing times available for scheduling.

We verify these claims using simulation experiments to evaluate and compare the various algorithms in terms of the predicted execution times of the schedules generated by the algorithms for larger number of processors in the following subsections. We also investigate the impact of errors in the predicted execution times on the quality of the schedules generated by the algorithms. The simulation experiments are explained in the following subsections.

5.2. Simulation setup

We used the performance models of four parallel applications, namely, Molecular Dynamics application (MD), Eigenvalue solver (eigen), Symmetric Successive Over-Relaxation (SSOR) and Integer Sort (IS), for comparison of the scheduling algorithms. The performance model equations for the applications were obtained for a 8-processor Intel Pentium IV cluster with each processor having 2.8 GHz CPU and the processors were connected by a 100 Mbps switched Ethernet. In our experiments, the available CPU values ranged from 0.1 to 1.0 where a value of 1.0 indicates an unloaded processor. Thus, the available CPU of 0.75 for a processor in our simulation setup represents an Intel processor that is one-fourth loaded.

We compared the performance of the algorithms for different clusters containing power-of-2 number of processors sizes ranging from 32 to 1024 processors. For a simulation experiment

DOI: 10.1002/cpe

with a given cluster, we randomly chose the maximum available bandwidth of links in the cluster to be one of 100 Mbps, 1 Gbps, 5 Gbps and 10 Gbps. We then randomly varied the available bandwidth of each link to be within 20–80% of the maximum available bandwidth. For each experiment, we also varied *load percentage*, the ratio of *lightly*, *medium* and *heavily* loaded machines. We randomly varied the available CPU to be within 0.701–1.0 for *lightly* loaded machines, 0.351–0.7 for *medium* loaded machines and 0.05–0.35 for *heavily loaded machines*. For each experiment, the percentage of machines in a particular load category is randomly varied between 10 and 80% such that the sum of all the percentages equals 100. Since our algorithms are time-tunable, we also experimented with different times needed for running the scheduling heuristics. For a given cluster size, scheduling time and load percentage, we executed 50 experiments corresponding to different CPU loads and report average performance of the scheduling algorithms for the experiments.

5.3. Results

In the following subsections, we present various simulation results corresponding to the effects of various resource parameters on the performance of scheduling. For brevity, we report the results obtained with the parallel MD application[‖].

5.3.1. Different resource parameters

Figure 8 shows the scalability of the scheduling algorithms with the increasing number of processors. We find that for 32 processors, all the algorithms were able to determine equivalent schedules in the 25 s allotted for scheduling. However, as the number of processors increases, we find that dynamic programming (DP), with its intelligent incremental addition of nodes, and box elimination (BE) method gave schedules with smaller execution times than the other algorithms.

We find that similar to the results for real experiments in Section 5.1, the DP algorithm gives good schedules for small number of processors and scales well up to 256 processors where it reaches saturation. For greater than 256 processors, its performance degrades with increasing number of processors. This is due to the incremental addition of nodes to schedules in the DP algorithm where the best schedules for $i + 1$ number of processors are formed from the best schedules for i number of processors. The number of candidate schedules evaluated to determine a best schedule with i processors from a total of N processors is $C(N, i)$ and increases with N for a given i number of processors. As N increases, the DP algorithm spends more time in finding the best schedule with i number of processors. Thus, for a large number of processors, N, and for a given available time for scheduling, the DP algorithm will not be able to explore sufficient candidate schedules with larger number (greater than i) of processors and yields schedules with small number of processors. We found in our experiments that for greater than 256 total number of processors, the DP algorithm returned schedules with less than 256 processors and hence the execution times of the schedules for 512 and 1024 total number of processors were greater than for 256 processors as shown in Figure 8. We find that our BE algorithm scales very well with the increasing number of processors

[‖]We found that for all applications, the BE algorithm gave the best or near-best schedules.

DOI: 10.1002/cpe

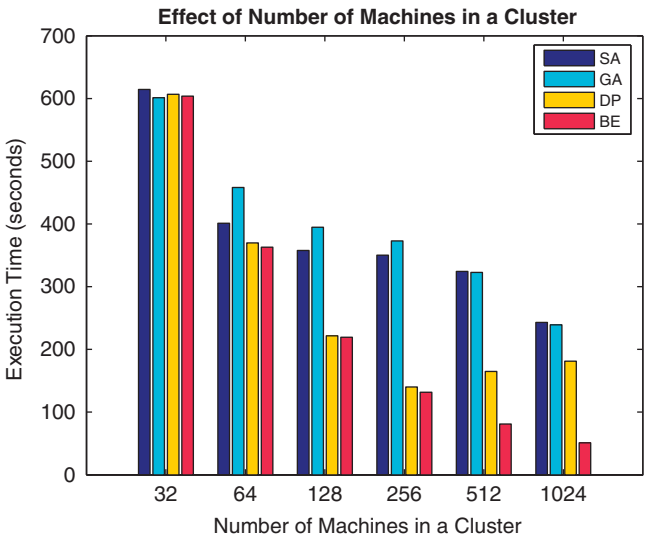


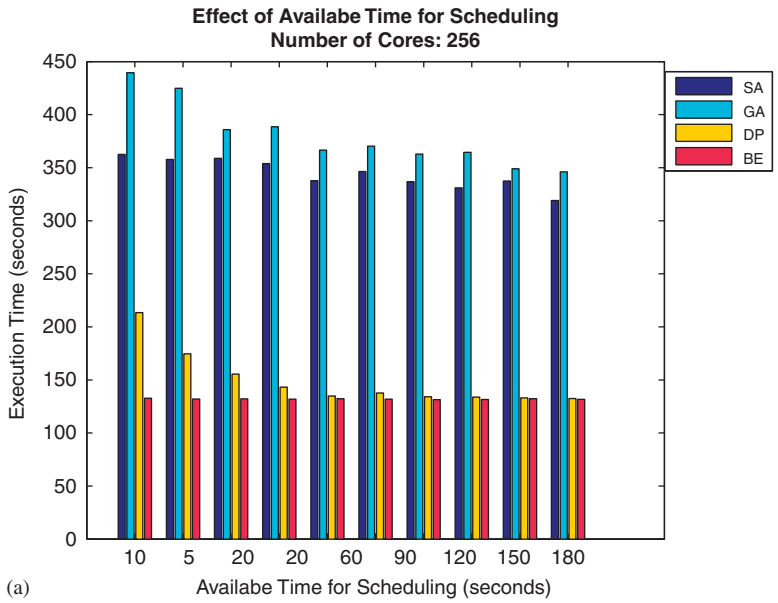
Figure 8. Scalability with processors. MD with 2048 molecules, 30% Lightly, 40% Medium and 30% Highly Loaded. Scheduling Time: 25 s.

and clearly outperforms the DP algorithm by about 70% for larger number of processors. This is because the BE algorithms eliminate large numbers of search space regions corresponding to poor schedules. Thus, as claimed in Section 5.1, the DP algorithm can give efficient schedules only for small number of processors.

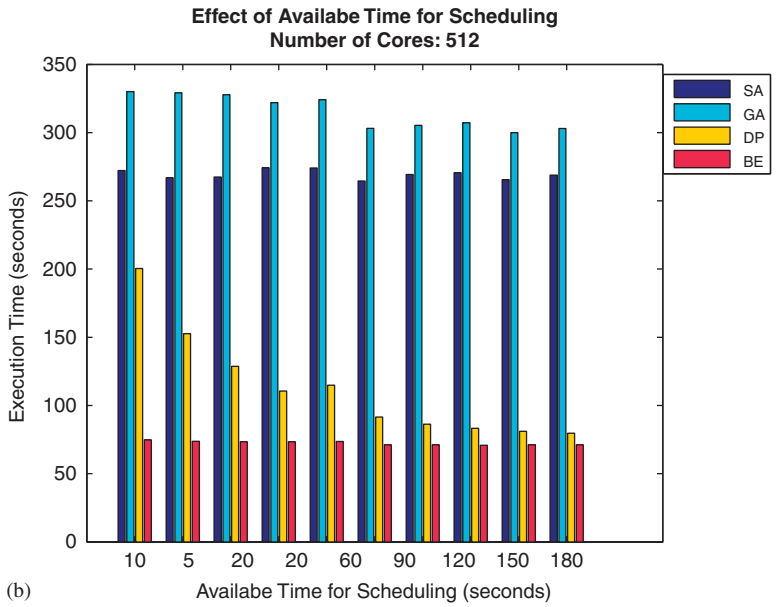
5.3.2. Times needed for scheduling

Figures 9(a) and (b) compare the algorithms when different times are allotted for scheduling. We find that unlike the results for real experiments shown in Section 5.1, the execution times of the schedules decrease with the increasing times available for execution. Thus, as claimed in Section 5.1, the time available for scheduling impacts the quality of the schedules for higher number of processors. The SA and GA algorithms, due to randomness in schedule generation, were not able to generate significantly better schedules even with the increase in the times allotted for scheduling. We find that our BE algorithm is able to generate highly efficient schedules even with 2 s. This is due to its procedure of finding a center point in its 3-D grid and the associated eliminations of large search space regions. We find that the execution times of the schedules by DP algorithm decrease with increased times allotted for scheduling and approach the performance of our BE algorithm especially on 256 processors. However, we find that on 256 processors, the DP algorithm reaches saturation at 90 s and does not perform better than our BE algorithm even for higher scheduling times. We find in Figure 9(b) that the schedules generated by our BE algorithm in 2 s are more efficient than the schedules generated by the DP algorithm in 3 min.

DOI: 10.1002/cpe



(a)



(b)

Figure 9. Performance of algorithms with different times available for scheduling for 256 and 512 processors. Application: MD with 2048 molecules, 30% Lightly, 40% Medium and 30% Highly Loaded: (a) 256 Processors and (b) 512 Processors.

DOI: 10.1002/cpe

5.3.3. *Robustness against prediction errors*

We also compared the algorithms in terms of the robustness [24] or sensitivities of the execution times for their schedules against the errors due to performance models. We modified our algorithms such that while evaluating the schedules by using performance model equations, we perturbed the predicted execution times by a random percentage in the interval $[-p, +p]$, where p is the maximum percentage error. We then compared the schedules generated by the modified algorithms with the schedules generated by the unmodified algorithms in terms of the unperturbed predicted execution times using our performance models.

Figures 10(a) and (b) show the execution times and the percentage increase in the execution times for various values of prediction error intervals. We observe that the execution times of the schedules for the SA and DP algorithms are highly sensitive to prediction errors since their percentages in increase of execution times increase with the amount of prediction errors. Although the GA algorithm is robust against performance modeling errors, it gives poor schedules irrespective of the prediction errors. Our BE algorithm shows a marginal increase in the execution times while generating high quality schedules. We also make an interesting observation for the SA algorithm in certain cases. The schedules of the SA algorithm in these cases were found to be better with errors in the performance modeling-based predictions. This happens when an algorithm using accurate predictions has missed evaluating several good quality solutions in the search space. These points in the search space are evaluated by chance when the algorithm uses perturbed predicted values.

5.3.4. *Different applications*

Figure 11 shows the performance of the different scheduling algorithms for different applications with different performance model equations. We observe a very interesting phenomenon in the graph. We find that the relative performance of the different scheduling algorithms depends on the application that is scheduled. For example, we find that the DP algorithm gives the worst schedules for eigenvalue problems although it gives good schedules for the other applications. This is because the eigenvalue application is highly scalable with processors. The DP algorithm that incrementally adds nodes to the schedule spends most of its efforts in schedules with smaller number of processors. We also find that the SA algorithm gives significantly better schedules than the GA for the SSOR application when compared with other applications. This graph shows that the existing optimization techniques, that have been evaluated for different number of processors and loads, will have to be reevaluated for different objective functions. We find that our BE algorithm performs better than the other algorithms for all applications except for the eigenvalue problem, where the schedules generated by it have slightly higher execution times than the execution times corresponding to the SA and GA algorithms.

5.3.5. *Multi-cluster experiment*

In this experiment, we randomly generated 1100 multi-cluster setups and compared the performance of the algorithms for each setup. For each multi-cluster setup, we invoked each algorithms on each cluster. For a given algorithm, we chose the schedule generated by the algorithm for a cluster that gave the overall minimum execution time of all schedules generated by the algorithm in all clusters

DOI: 10.1002/cpe

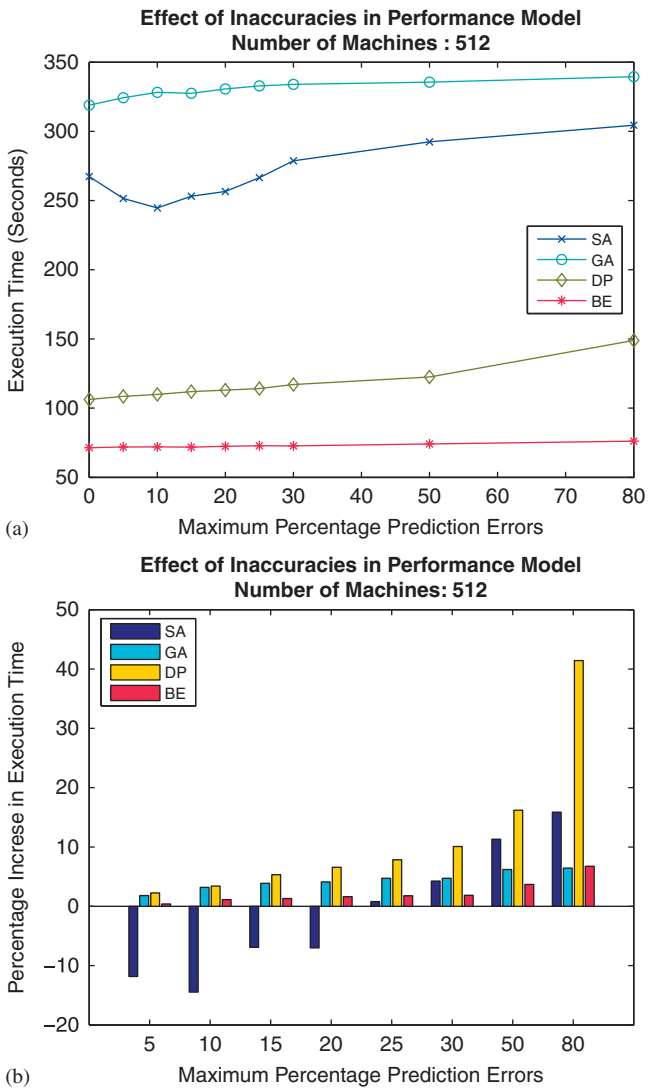


Figure 10. Effect of Prediction Errors on Scheduling for 512 processors. Application: MD with 2048 molecules, 30% Lightly, 40% Medium and 30% Highly Loaded, Time for Scheduling: 25 s. (a) Effect of prediction errors on execution times and (b) effect of prediction errors on percentage increase in execution times.

of the setup. We then compared the algorithms on the basis of the overall minimum execution times. For these experiments, we used the performance modeling equation of Molecular Dynamics (MD) application. We use $U[x, y]$ to denote uniform probability distribution in the interval (x, y). For randomly generating each multi-cluster setup, we generated $U[4, 15]$ number of clusters and $U[32, 1024]$ number of processors in each cluster. In order to simulate heterogeneity of processors

DOI: 10.1002/cpe

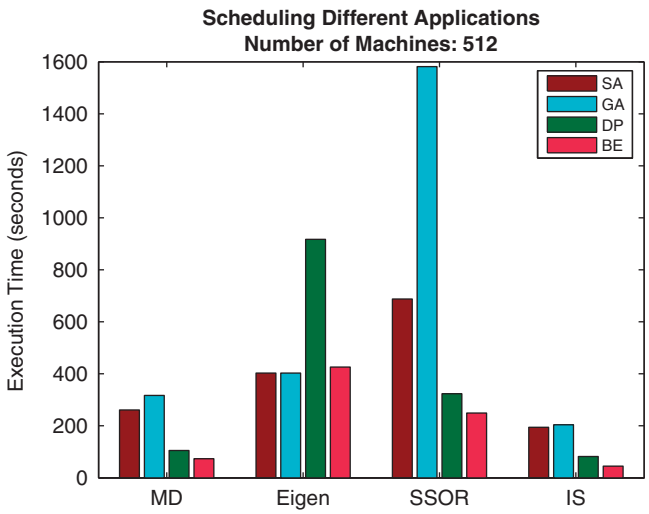


Figure 11. Scheduling different applications on 512 processors. Problem sizes: 2048 (MD), 10 000 (Eigen), 3328 (SSOR), 400 000 (IS). 30% Lightly, 40% Medium and 30% Highly Loaded. Time for Scheduling: 25 s.

in different clusters, we used a random *cpu scaling factor* from the set (0.6, 0.8, 1.0, 1.2, 1.4)** for each cluster, and multiplied the coefficients of computational complexity of the performance model equation with the scaling factor. We also chose the maximum bandwidth of links in a cluster to be one of 100 Mbps, 1 Gbps, 5 Gbps and 10 Gbps. Finally, for each cluster, we also randomly fixed the percentages of lightly, medium and heavily loaded machines in the cluster. We used scheduling time of 25 s for these experiments.

Table I summarizes the performance of difference algorithms for the multi-cluster experiments. We find that the average execution time of schedules generated by our BE algorithm is much smaller than the execution times of the other algorithms in terms of both arithmetic and geometric means. For each multi-cluster setup, we also compared the best of the other algorithms and our BE algorithm in terms of execution times of the schedules. The average difference in the execution times between our BE algorithm and the best of the other algorithms was 43% and the minimum and maximum differences were −4 and 82%, respectively. Out of the 1100 experiments, our BE algorithm did not give the minimum execution times only in 12 cases in which the DP algorithm gave the minimum times. In these 12 cases, the maximum difference in execution times of DP and BE algorithms was only 4%.

In order to determine the effect of heterogeneity of different clusters in a multi-cluster setup on the performance of the algorithm, we calculated the *weight* of a cluster as a product of number of machines, the cpu scaling factor and the percentage of lightly and medium loaded machines in the cluster. Clusters with larger weights are expected to yield better schedules. We then calculated the ratio of the maximum and the minimum weight for a multi-cluster setup. Higher values of this

**The scaling factors are chosen from a small range since the CPU speeds of modern processors vary by small amounts.

DOI: 10.1002/cpe

Table I. Multi-cluster setup.

Algorithm	Arithmetic Mean (s)	Std. Dev. (s)	Geometric Mean (s)
SA	188.42	44.908	183.34
GA	217.41	49.46	212.19
DP	124.53	21.45	122.9
BE	71.29	27.22	66.96

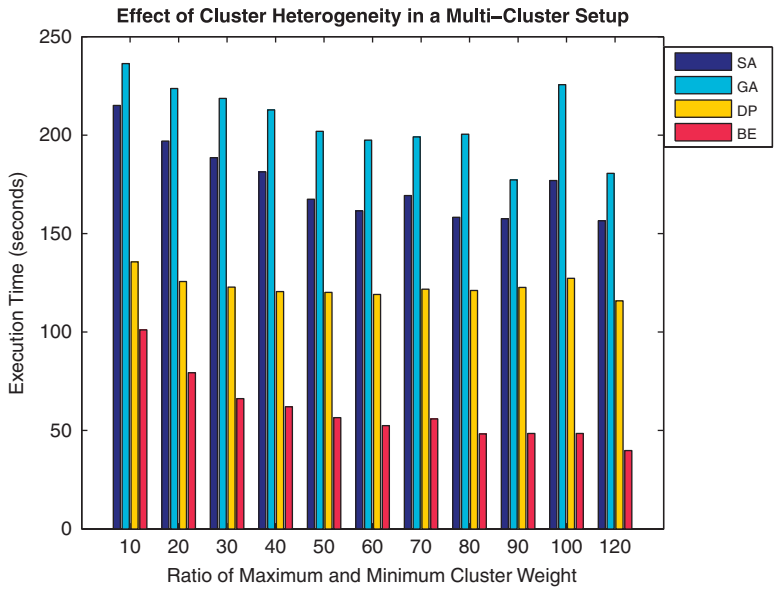


Figure 12. Effect of cluster heterogeneity in a multi-cluster setup.

ratio indicate greater heterogeneity between the clusters. We divided the 1100 multi-cluster setups into different groups such that setups in the same group have the same ratio. Figure 12 shows the average performance of the algorithms for different groups of ratio values. We find that with the increasing ratio of maximum to minimum cluster weight or with increasing heterogeneity, our BE algorithm gives schedules of the decreasing execution times. This behavior is expected because with increasing differences in the capacities of the clusters, an efficient scheduling algorithm should be able to demarcate the lower and higher capacity clusters better and hence choose schedules from the clusters of higher capacities or large weight values. However, this behavior is not observed for other algorithms since these algorithms sometimes choose schedules from clusters of lower capacities.

6. CONCLUSIONS

In this work, we have devised a novel scheduling algorithm called Box Elimination that uses the performance model of a tightly coupled parallel application to schedule the applicationon a single

DOI: 10.1002/cpe

or a multi-cluster setup. By treating the search space as a set of performance model parameters and eliminating regions containing poor solutions, our algorithm is able to generate efficient schedules with minimum execution times for the application. By means of a large number of real and simulation experiments, we have shown that our algorithm generates schedules with up to 80% less execution times than the other popular algorithms. We have also shown that performance predictions errors have the least effect on the quality of the schedules generated by our algorithm.

7. FUTURE WORK

We plan to extend our scheduling algorithms to dedicated batch systems by considering the effects of different queue waiting times for schedules with different number of processors. We also plan to devise job scheduling algorithms to schedule a set of parallel applications on multiple clusters for simultaneous executions with the objective of minimizing the average response times of the applications. We also plan to build performance modeling-based rescheduling techniques for multi-phase tightly coupled applications.

ACKNOWLEDGEMENTS

We thank the High Performance Computing Lab of Prof. R. Govindarajan, Supercomputer Education Research Centre, Indian Institute of Science, for providing their 128-core cluster for us to perform real experiments with our scheduling algorithms. We also thank the anonymous reviewers for their useful comments that helped us to significantly improve the quality of the paper. This work is supported by Department of Science and Technology, India. project ref no. SR/S3/EECE/59/2005/8.6.06.

REFERENCES

1. Feitelson D. A survey of scheduling in multiprogrammed parallel systems. *Technical Report Research Report RC 19790 (87657)*, IBM T. J. Watson Research Center, October 1994.
2. Shmueli E, Feitelson D. Backfilling with lookahead to optimize the packing of parallel jobs. *Journal of Parallel and Distributed Computing* 2005; **65**(9):1090–1107.
3. Boyer W, Hura G. Non-evolutionary algorithm for scheduling dependent tasks in distributed heterogeneous computing environments. *Journal of Parallel and Distributed Computing* 2005; **65**(9):1035–1046.
4. Nascimento A, Sena A, Boeres C, Rebello V. Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation*: *Practice and Experience* 2007; **19**(14):1955–1974.
5. Abawajy J, Dandamudi S. Parallel job scheduling on multicluster computing systems. *Proceedings of the IEEE International Conference on Cluster Computing*, Kowloon, Hong Kong, China, 2003; 11–18.
6. Li K. Job scheduling and processor allocation for grid computing on metacomputers. *Journal of Parallel and Distributed Computing* 2005; **65**(11):1406–1418.
7. Mnaouer A, Al-Riyami B. Effective scheduling of local interactive processes and parallel processes in a non-dedicated cluster environment. *Journal of Parallel and Distributed Computing* 2005; **65**(6):755–766.
8. Beaumont O, Carter L, Ferrante J, Legrand A, Marchal L, Robert Y. Centralized versus distributed schedulers for multiple bag-of-task applications. *Twentieth International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, 2006; 10.
9. van der Raadt K, Yang Y, Casanova H. Practical divisible load scheduling on grid platforms with APST-DV. *IPDPS '05*: *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, Denver, CO, U.S.A., 2005; 29.2.
10. Yang Y, van der Raadt K, Casanova H. Multiround algorithms for scheduling divisible loads. *IEEE Transactions on Parallel and Distributed Systems* 2005; **16**(11):1092–1102.

11. Petitet A, Blackford S, Dongarra J, Ellis B, Fagg G, Roche K, Vadhiyar S. Numerical libraries and the grid: The GrADS experiments with ScaLAPACK. *Journal of High Performance Applications and Supercomputing*, *Winter* 2001; **15**(4):359–374.
12. Allen G, Dramlitsch T, Foster I, Karonis N, Ripeanu M, Seidel E, Toonen B. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. *Supercomputing '01*: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (*CDROM*), Denver, CO, U.S.A., 2001; 52.
13. Chen L, Zhu Q, Agrawal G. Supporting dynamic migration in tightly coupled grid applications. *Supercomputing '06*: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006; 117.
14. Sabin G, Kettimuthu R, Rajan A, Sadayappan P. Scheduling of parallel jobs in a heterogeneous multi-site environment. *Proceedings of 9th Workshop on Job Scheduling Strategies for Parallel Processing* (*JSSPP 2003*), Seattle, WA, U.S.A., 2003; 87–104.
15. Bouteiller A, Bouziane HL, Herault T, Lemarinier P, Cappello F. Hybrid preemptive scheduling of message passing interface applications on grids. *International Journal of High Performance Computing Applications* 2006; **20**(1):77–90.
16. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC. *ScaLAPACK Users' Guide*. SIAM: Philadelphia, PA, 1997.
17. Frigo M, Johnson S. The design and implementation of FFTW3. *Proceedings of the IEEE* 2005; **93**(2):216–231. Special Issue on Program Generation, Optimization, and Platform Adaptation.
18. Drake J, Foster I, Hack J, Michalakes J, Semeraro B, Toonen B, Williamson D, Worley P. PCCM2: A GCM adapted for scalable parallel computers. *Proceedings of the Fifth Global Change Symposium*, American Meteorological Society, 1994; 91–98.
19. Hastings S, Kurc T, Langella S, Catalyurek U, Pan T, Saltz J. Image processing on the grid: A toolkit or building grid-enabled image processing applications. *CCGRID '03*: *Proceedings of the 3st International Symposium on Cluster Computing and the Grid* 2003; 36.
20. Ludtke S, Baldwin P, Chiu W. EMAN: Semiautomated software for high-resolution single-particle reconstructions. *Journal of Structural Biology* 1999; **128**:82–97.
21. Deelman E, Singh G, Su M-H, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman GB, Good J, Laity AC, Jocob JC, Katz DS. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 2005; **13**(3):219–237.
22. He L, Jarvis S, Spooner D, Chen X, Nudd G. Dynamic scheduling of parallel jobs with QoS demands in multiclusters and grids. *GRID '04*: *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, U.S.A., 2004; 402–409.
23. Yarkhan A, Dongarra J. Experiments with scheduling using simulated annealing in a grid environment. *Grid Computing—GRID 2002*, *Third International Workshop* (*Lecture Notes in Computer Science*, vol. 2536). Springer: Berlin, 2002; 232–242.
24. Sugavanam P, Siegel HJ, Maciejewski AA, Oltikar M, Mehta AM, Pichel R, Horiuchi A, Shestak V, Al-Otaibi M, Krishnamurthy YG, Ali SA, Zhang J, Aydin M, Lee P, Guru K, Raskey M, Pippin AJ. Robust static allocation of resources for independent tasks under makespan and dollar cost constraints. *Journal of Parallel and Distributed Computing* 2007; **67**(4):400–416.
25. Kettimuthu R, Subramani V, Srinivasan S, Gopalasamy T, Panda D, Sadayappan P. Selective preemption strategies for parallel job scheduling. *ICPP '02*: *Proceedings of the 2002 International Conference on Parallel Processing* (*ICPP'02*), Vancouver, BC, Canada, 2002; 602.
26. OpenPBS. Available at: http://www.openpbs.org.
27. Ibm corporation. *Ibm Loadleveler*. Available at: http://www.mppmu.mpg.de/computing/AIXuser/loadl.
28. Feitelson D, Weil A. Utilization and predictability in scheduling the IBM SP2 with backfilling. *Twelfth International Parallel Processing Symposium* (*IPPS*), Orlando, FL, U.S.A., 1998; 542–546.
29. Kwok YK, Ahmad I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys* 1999; **31**(4):406–471.
30. Corrêa R, Ferreira A, Rebreyend P. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Transactions on Parallel and Distributed Systems* 1999; **10**(8):825–837.
31. Berman F, Wolski R, Figueira S, Schopf J, Shao G. Application-level scheduling on distributed heterogeneous networks. *Supercomputing '96*: *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing* (*CDROM*), Pittsburg, U.S.A., 1996; 39.
32. Nudd G, Kerbysin D, Papaefstathiou E, Perry S, Harper J, Wilcox D. PACE—A toolset for the performance prediction of parallel and distributed systems. *International Journal of High Performance Computing Applications* 2000; **14**(3):228–251.
33. Subramani V, Kettimuthu R, Srinivasan S, Sadayappan P. Distributed job scheduling on computational grids using multiple simultaneous requests. *HPDC '02*: *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, U.K., 2002; 359.
34. Plaat A, Bal HE, Hofman R, Kielmann T. Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. *Future Generation Computer Systems* 2001; **17**(6):769–782.

DOI: 10.1002/cpe

35. Bal H, Plaat A, Bakker M, Dozy P, Hofman R. Optimizing parallel applications for wide-area clusters. *Proceedings of the International Parallel Processing Symposium*, Orlando, FL, U.S.A., 1998; 784–790.
36. Ernemann C, Hamscher V, Schwiegelshohn U, Yahyapour R, Streit A. On Advantages of grid computing for parallel job scheduling. *CCGRID '02*: *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Germany, 2002; 39.
37. Bucur A, Epema D. Scheduling policies for processor coallocation in multicluster systems. *IEEE Transactions on Parallel and Distributed Systems* 2007; **18**(7):958–972.
38. Wu M, Sun XH. Grid harvest service: A performance system of grid computing. *Journal of Parallel and Distributed Computing* 2006; **66**(10):1322–1337.
39. Yu J, Buyya R, Ramamohanarao K. Workflow scheduling algorithms for grid computing. In *Metaheuristics for Scheduling in Distributed Computing Environments*, Xhafa F, Abraham A (eds.). Springer: Berlin, Germany, 2008. SBN: 978-3-540-69260-7.
40. Wieczorek M, Podlipnig S, Prodan R, Fahringer T. Bi-criteria scheduling of scientific workflows for the grid. *CCGRID '08*: *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid* (*CCGRID*), Lyon, France, 2008; 9–16.
41. Blythe J, Jain S, Deelman E, Gil Y, Vahi K, Mandal A, Kennedy K. Task scheduling strategies for workflow-based applications in grids. *CCGRID '05*: *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid* (*CCGrid'05*), vol. 2, 2005; 759–767.
42. Mandal A, Kennedy K, Koelbel C, Marin G, Mellor-Crummey J, Liu B, Johnsson L. Scheduling strategies for mapping application workflows onto the grid. *HPDC '05*: *Proceedings of the High Performance Distributed Computing*, *2005. HPDC-14. Proceedings. 14th IEEE International Symposium*, Research Triangle Park, NC, U.S.A., 2005; 125–134.
43. Barbosa J, Morais C, Nobrega R, Monteiro A. Static scheduling of dependent parallel tasks on heterogeneous clusters. *IEEE International Conference on Cluster Computing*, Boston, MA, U.S.A., 2005; 1–8.
44. Jin S, Schiavone G, Turgut D. A performance study of multiprocessor task scheduling algorithms. *Journal of Supercomputing* 2008; **43**(1):77–97.
45. Sanjay H, Vadhiyar S. Performance modeling of parallel applications for grid scheduling. *Journal of Parallel and Distributed Computing* 2008; **68**(8):1135–1145.
46. Wolski R, Spring N, Hayes J. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems* 1999; **15**(5–6):757–768.
47. Wolski R. Dynamically forecasting network performance using the network weather service. *Journal of Cluster Computing* 1998; **1**(1):119–132.
48. Bailey D, Harris T, Saphir W, van der Wijngaart R, Woo A, Yarrow M. The NAS parallel benchmarks 2.0. *Technical Report NAS-95-020*, Nasa Ames Research Center, December 1995.
49. Ullman J. NP-complete scheduling problems. *Journal of Computer and System Sciences* 1975; **10**:384–393.
50. Kim S, Weissman J. A genetic algorithm based approach for scheduling decomposable data grid applications. *International Conference on Parallel Processing*, Montreal, Quebec, Canada, 2004; 406–413.
51. Gao Y, Rong H, Huang J. Adaptive grid job scheduling with genetic algorithms. *Future Generation Computer Systems* 2005; **21**(1):151–161.
52. Braun TD, Siegel HJ, Beck N, Bölöni L, Maheswaran M, Reuther AI, Robertson J, Theys MD, Yao B, Hensgen DA, Freund RF. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. *HCW '99*: *Proceedings of the Eighth Heterogeneous Computing Workshop*, San Juan, Puerto Rico, 1999; 15.

DOI: 10.1002/cpe