IPSJ Transactions on Advanced Computing Systems Vol. 3 No. 1 88–100 (Mar. 2010)

Regular Paper

Interactive Application Scheduling with GridRPC

Hao Sun[†1] and Kento Aida[†1,†2]

GridRPC is known as an effective programming model to develop Grid applications. However, it is still difficult for non-expert users to apply it efficiently. For example, a GridRPC application user needs to select computational resources, monitor the resources and estimate the application performance on the resources. In this paper, we propose InterS, an interactive scheduling system for GridRPC applications. First, the automatic scheduling mechanism provides resource allocation plans, from which the user can choose the most suitable one. Second, the execution advice mechanism helps the user to improve the performance of the application at run time while overload or failure on the resource(s) is(are) detected. Third, the scheduling policy mechanism provides the user with an interface in ClassAd format to define the scheduling policy applied in InterS. This paper also presents experimental results to show the advantage of interactive scheduling and how they can be performed at run time.

1. Introduction

GridRPC[7] is known as an effective programming model for developing Grid applications. However, it is still difficult for non-expert users to apply it efficiently. The current implementation of GridRPC assumes that a user selects remote computational resources before run time; thus, it forces the user to do hard work requiring expert knowledge, e.g., monitoring remote computational resources and estimating the application performance on the selected computational resources. Additionally, computational resources on the grid are unstable. Loads of the resources fluctuate and some resources may fail. GridRPC users need to make their applications robust enough, so that they can accommodate the fluctuation and failure of remote computational resources.

Several mechanisms to reduce the complexity of running Grid applications have been proposed, and some of them focus on GridRPC applications. Condor[4] and Nimrod/G[5] focus on the high throughput and economic feature of Grid applications, respectively. Task Farming[3] and F-Omega[6] help GridRPC users by providing advanced mechanisms in terms of task scheduling and dynamic resource allocation at application run time. GridWay[1] meta-scheduler provides users with adaptive scheduling features such as automatic rescheduling when a task fails or a better resource is found. Some of the above mechanisms perform automatic scheduling, where the resource allocation algorithm is implemented in the scheduling software or is provided by the user through APIs. Although there are many successes in automatic scheduling, it sometimes fails to obtain a satisfying performance due to the fluctuation of resources or a complicated user requirement to run the application. Some scheduling software enables the user to configure the resource allocation algorithm before run time. However, it is still hard for non-expert users to completely configure the algorithm. In this case, interactions such as selection of execution plans and changing resource allocation with the user's decisions contributes to improving the efficiency and robustness of the application run and user's satisfaction.

In this paper, we propose InterS, an interactive scheduling system for GridRPC applications. Advantages of the proposed InterS are presented by the following scenarios: Some scheduling software enables the user to configure the resource allocation algorithm before run time. An expert user may give a complete configuration, which includes job migration when failure on computing resources happens. However, it is complex and hard work for a non-expert user to give the complete configuration before run time. InterS provides the user with multiple candidates of scheduling configurations (or plans) and enables the user to choose a suitable one. It particularly helps the non-expert user to run the application with a suitable resource allocation algorithm.

Another scenario is the situation that the scheduling system needs a user's decision to change the resource allocation at run time. For example, when a user runs an application (or jobs) with the limited budget, a computing resource executing the job fails. A sophisticated automatic scheduling system may move the job to another computing resource if the cost of running the application does not exceed the budget declared by the user before run time. However, when

†1 Tokyo Institution of Technology
†2 National Institute of Informatics

Table 1 Comparison of scheduling techniques in scheduling software.

	Condor	Nimrod/G	Task Farming	F-Omega	GridWay	InterS
automatic scheduling	Y	Y	Y	N	Y	Y
RAA[1] customization	N	N	N	N	Y	Y
execution advice runtime selection[2]	N	N	N	N	N	Y

1. RAA: short for resources allocation algorithm
2. Execution advice includes execution plans

moving the job increases the cost beyond the user's budget, the scheduling system needs the user's decision if it is to migrate the job at run time. InterS enables the user to give such a decision during application run time.

The interactive scheduling proposed in this paper is enabled by the cooperation of three mechanisms. The automatic scheduling mechanism provides resource allocation plans, from which the user can choose the most suitable one. The execution advice mechanism helps the user to improve the performance of the application at run time while overload or failure on the resource(s) is(are) detected. Some expert users may want to customize resource allocation algorithms for their applications. The scheduling policy mechanism provides the user with an interface in ClassAd format to define the resource allocation algorithms applied in InterS. It also enables users to change the currently running resource allocation algorithms to others during the run time.

The rest of this paper is organized as follows: in Section 2, comparison between InterS and related works are presented. In Section 3, design and implementation issues are discussed. Section 4 shows the experimental study, and Section 5 gives the conclusions.

2. Related Works

Condor and Nimrod/G are resource brokers, which dispatch user tasks to suitable computational resources. Both requirements from user applications and those from resources providers are specified in the ClassAds, and the matchmaking mechanism dispatches tasks to resources so as to satisfy both the requirements in Condor. Nimrod/G has a similar mechanism and it also enables task scheduling with budget constraints. Both Condor and Nimrod/G perform fully automatic scheduling. The user cannot change resource allocation during application run time.

Task Farming middleware provides a user APIs for task scheduling and a fault tolerant mechanism. F-Omega is a programming framework, which enables flexible grid application development and its execution. Although Task Farming performs automatic scheduling, the implemented allocation algorithm is a simple one and the user cannot change resource allocation algorithm at run time. F-Omega enables a user to change the resource allocation during application run time, however, the user needs to select an initial set of computational resources.

GridWay is a job submission framework on the Globus toolkit. It performs automatic scheduling and enables a user to change resource allocation during application run time.

The proposed InterS performs automatic scheduling and enables a user to change resources allocation during application run time. Generally, changing resource allocation requires the user to have expert knowledge about both the applications and resources on the grid. InterS helps the user by giving advice for changing resource allocation during application run time to solve this problem. Furthermore, InterS enables the user to change the resource allocation algorithm currently running during application run time. **Table 1** summarizes techniques enabled in scheduling software. To the best of our knowledge, there is no software that enables "interactive scheduling" mechanisms as InterS does.

3. Design and Implementation

Figure 1 shows the flow of interactive scheduling implemented in InterS. Interactive scheduling proposed in this paper is enabled by the cooperation of three mechanisms: the automatic scheduling mechanism, the execution advice mechanism and the scheduling policy mechanism. The automatic scheduling mechanism

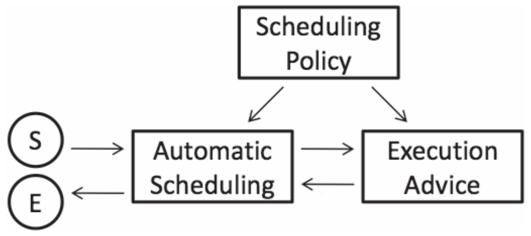


Fig. 1 Interactive scheduling flow.

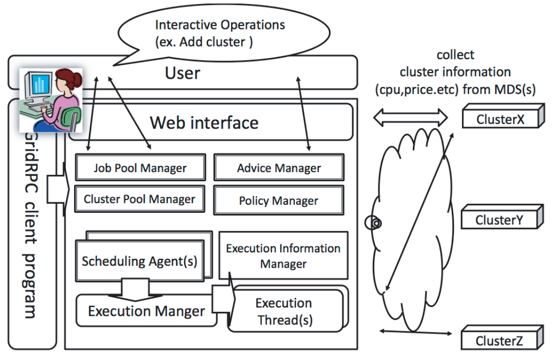


Fig. 2 InterS architecture overview.

provides initial resource allocation plans, and the user can change the plans and control the scheduling behavior at run time through execution advice and/or scheduling policy files.

Figure 2 illustrates the software architecture of InterS. A user of InterS first writes a GridRPC client program using the InterS client APIs. For example "addJob" method (in **Table 2**) creates GridRPC tasks first then stores them into the job pool through the job pool manager. Three ways of adding resources to InterS are supported: the API, the web interface and the policy file. GridRPC executables, or tasks, are executed on remote computational resources selected by InterS. At run time, the user starts the application and performs interactive scheduling through the web or policy file interface.

3.1 Client Interface

A user of InterS first writes a GridRPC client program using the InterS client APIs. The user gives information about the application, or remote GridRPC executables, through the APIs. Currently, APIs for Java & Groovy [11] are available. Table 2 summarizes the InterS Client APIs. **Figure 3** presents an example of the InterS client program in Groovy language. First, the user needs to decide which resource allocation algorithm to use for automatic scheduling, and then binds the remote executables with the scheduling agent by creating a RemoteFunction instance. This time "MasterWorker" is adopted for utilizing local clusters. It allocates every cluster once for task execution, and then it chooses the resources that finish task execution for the remaining tasks earlier. When the tasks finish their execution, the results are stored into "results", which is an array of String arrays (defined at line 3). After submitting the tasks to InterS, the user can call the waitAll method to block the client program until all tasks are finished. In this example the user decides to add a commercial resource later, which shows better performance. So the user changed the "MasterWorker" scheduling agent to "CostPrior" for the cost management, after waiting for the end of tasks specified in waitAnd. Finally, after all tasks finish execution, the user can access values stored in the "results" list.

3.2 Automatic Scheduling

The automatic scheduling mechanism selects computational resources that satisfy the requirements of the user application. Multiple resource allocation algorithms are implemented in InterS. Currently, the MasterWorker algorithm, the RoundRobin algorithm, the PerformancePrior and the CostPrior algorithm are available in InterS. The MasterWorker and RoundRobin are the simple heuristics, which allocate resources without any cost concern. On the other hand, the PerformancePrior and the CostPrior algorithm offer budget constraints resource allocation plans to the user. Various allocations as well as the cost and performance estimation are given in a plan produced by the automatic scheduling mechanism. The user then chooses the one suitable for his/her preference. Also, the user can define the customized resource allocation algorithm(s) through the scheduling policy mechanism (e.g., last three lines in Fig. 15). For example, the algorithm to generate PerformancePrior plans is shown in **Fig. 5**. First, InterS

Table 2 The list of APIs in InterS.

Task initiation APIs	
ScheduleAgent	A Java class implementing the scheduling agent(s) in InterS. It takes the agent type as argument in terms of MasterWorker, RoundRobin, PerformacePrior and CostPrior
RemoteFunction	A Java class implementing the GridRPC remote executable(s) in InterS. It takes the name of the executable and a scheduling agent, a default agent setting for this executable, as arguments.
Cluster	A java class, which stores the resource information(s).
Task submission APIs	
addJob()	Submitting user tasks to InterS. Tasks are stored in the job pool and InterS decides the resource allocation. A task id will be returned for further job handling.
addJobWith()	Same as addJob except that resources are selected by users.
waitAll()	Blocking until all tasks are finished.
waitFor()	Blocking until a certain task are finished. The only argument is the task id.
waitAnd()	Blocking until all tasks in a group are finished. The arguments are a list of task ids.
waitOr()	Blocking until one of the tasks in a group are finished. The arguments are a list of task ids.
Execution control APIs	
reschedule()	Rescheduling tasks, which do not start

```
 0.  // INITIALIZE CLIENT PROGRAM
 1.  def schedulingAgent =
       new ScheduleAgent( type:"MasterWorker" );
 2.  def remoteFunc = new RemoteFunction(
       name:"NPB/EP", agent:schedulingAgent );
 3.  def scheduler, results = [], taskids = [];
 4.  // SUBMIT TASKS
 5.  for(i=0;i<N;i++) {
 6.   def aResult = new String[];  results << aResult;
 7.   taskids <<
        scheduler.addJob(remoteFunc,arg1,arg2,aResult );
 8.  }
 9.  scheduler.waitAnd( ** subset of taskids **);
10.  // CHANGE SCHEDULING AGENT
11.  schedulingAgent =
        new ScheduleAgent( type:"CostPrior" );
12.  remoteFunc.agent = schedulingAgent;
13.  remoteFunc.save();
14.  scheduler.reschedule();
15.  scheduler.waitAll();
16.  // PROCESS RESULTS
17.  for(i=0;i<N;i++) { **use results[i]** }
```

Fig. 3 InterS client program.

ranks all the resources by job execution times in ascending order (through 1 to 7 lines) and stores the ranking to the list L_{rt}. Then, InterS gets the first resource R from the list L_{rt}, which has N_r cores, and assigns the first N_r jobs in the list L_j to it. InterS repeats this process until it assigns all the jobs (through 9 to 22 lines). Finally, InterS calculates the execution time and costs for each resource and creates the execution plans for the user (through 24 to 26 lines). The algorithm to generate CostPrior plans is almost the same, InterS ranks all the resources by the cost rate R_r (the cost to execute one job with one core in one second) in ascending order and then does the same as the PerformancePrior plan generation (through 9 to 26 lines).

The scheduling agent presented in **Fig. 4** performs scheduling. An instance of the scheduling agent is generated for each resource allocation algorithm, that is, four scheduling agent instances (MasterWorker, RoundRobin, PerformancePrior, CostPrior) are implemented in the default setting. The user can change the resource allocation algorithm at application run time by switching the scheduling agent instances. The scheduling agent submits tasks to the execution manager (illustrated in Fig. 2) following the resource allocation algorithm. The execution manager invokes tasks to remote computing resources through execution threads, where an execution thread is created for each task.

Information concerning remote computational resources, e.g., CPU specifications, available memory sizes and unit prices for computation, is required in

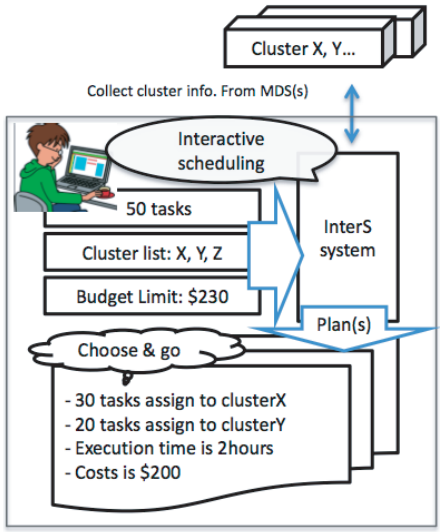


Fig. 4 An example of automatic scheduling.

```
    $L_r$ is the list of available resources now
    $L_{rt}$ = [], an empty list for ranking resources in $L_r$
 3: for all resource $R$ in $L_r$ do
      calculate the average of execution times $T_r$ for all finished jobs in resource $R$
      store the pair $(R, T_r)$ to list $L_{rt}$
 6: end for
    sort $L_{rt}$ by execution times in ascending order.

 9: $L_j$ is the list of jobs to be execute
    $L_{cr}$ = [], an empty list to store costs for the resources in $L_r$
    $L_{tr}$ = [], an empty list to store execution times for the resources in $L_r$
12: repeat
      for all pair $(R, T)$ in $L_{rt}$ do
        $N_r$ is number of available nodes in the resource $R$
15:       remove the first $N_r$ jobs from $L_j$
        get the execution cost rate $R_r$ for resource $R$
        {$R_r$ is the cost to execute one job with one core in one second}
18:       calculate the costs $C_r = C_r + N_r \cdot T \cdot R_r$
        calculate the execution time $T_r = T_r + T$
        store $C_r$ and $T_r$ in $L_{cr}$ and $L_{tr}$, respectively
21:   end for
    until The list $L_j$ is empty

24: for all resource $R$ in $L_r$ do
      make plan $P_r$ with $C_r$ and $T_r$ in $L_{cr}$ and $L_{tr}$, respectively
    end for
```

Fig. 5 The algorithm to generate PerformancePrior plan.

scheduling. InterS has two ways of obtaining the information. The first way is collecting resource information from the MDS [8]. The resource information is collected automatically and the user does not have to input the information. The second way is that the user provides the information.

The estimation of task execution time on remote computational resources is an important issue in making a better scheduling plan. InterS has a mechanism to estimate task execution time on remote computational resources by running test jobs. The user can configure the test, e.g., defining the number of test jobs to run, through the InterS interface. Running test jobs is not acceptable in some cases due to performance problems or budget constraints. In this case, the user can give an estimated task execution time to InterS.

The automatic scheduling mechanism takes care of job execution and makes sure that the jobs are finished successfully. When some jobs have failed and the migration cost does not exceed the user's budget declared before run time, the automatic scheduling mechanism moves jobs automatically. When the migration cost exceeds the budget, the automatic scheduling mechanism needs the user's direction, so InterS creates advice and waits for the user's instruction. In this situation, InterS does not move jobs unless it receive the user's instruction. The next section will show the details of execution advice.

3.3 Execution Advice

The execution advice mechanism gives advice for changing resource allocation, or scheduling plan, during application run time, when it finds a better plan. InterS monitors the execution status of each task in each cluster, so it can give advice for job execution failure and performance degradation on the allocated resource(s) at application run time. The user can use the new resources, avoiding performance degradation caused by execution failure and/or external tasks submission, through the InterS interface if the user accepts the advice. **Figure 6** shows the structure of execution advice and **Fig. 7** represents the algorithm for generating execution advice. To generate advice, InterS stores all of the abnormal jobs in the job list L of each execution advice. For example, when a job J fails its execution in a resource R_1, InterS adds the job J into a list L of an exe-

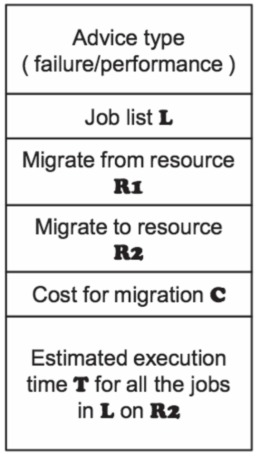

Advice type
(failure/performance)
Job list **L**
Migrate from resource **R1**
Migrate to resource **R2**
Cost for migration **C**
Estimated execution time **T** for all the jobs in **L** on **R2**

Fig. 6 Structure of the execution advice.

cution failure advice. To generate migration plans for all the jobs in list L, InterS uses the resource allocation algorithm A such as CostPrior, PerformancePrior or customized algorithms to decide the resource R_2 next to use, and calculates the estimated execution time T and cost C. The user may decide which algorithm to use by the policy file (line three of Fig. 9). If not, InterS uses the algorithm of the previous plan. InterS only shows the advice that contributes to performance improvement to the user. For example, if InterS detects a job fails its execution in the resource R_1, the execution time of the job in the resource R_1 is considered to be infinite, so any advice to move the job to a new resource R_2 is considered to be good. When the migration costs become higher or exceeds the user's budget if any, InterS shows the advice to the user.

Figure 8 presents an example of the execution advice. The numbers 3 and 6 are the advice ids. The user can handle these ids to accept the advice through the web interface to the advice manager (illustrated in Fig. 2). The execution advice shows the number of migration tasks, cluster names, cost and performance changes. In the example of Fig. 8, the advice 3 recommends the user to migrate 10 tasks from the resource *gk* to the resource *kuruwa*. The advice also shows that the migration costs $30 more but makes execution time 100 seconds shorter.

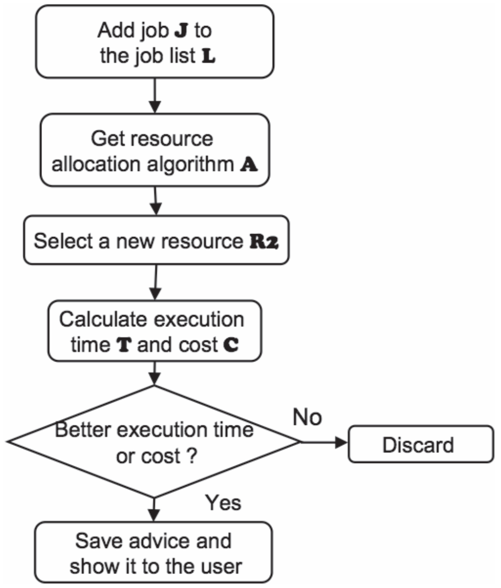
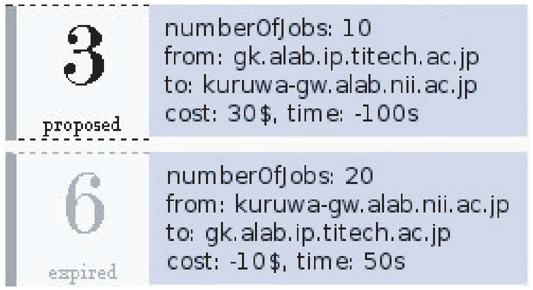

Add job **J** to the job list **L**
Get resource allocation algorithm **A**
Select a new resource **R2**
Calculate execution time **T** and cost **C**
Better execution time or cost ?
No
Discard
Yes
Save advice and show it to the user

Fig. 7 The algorithm for generating execution advices.



3 proposed
```
numberOfJobs: 10
from: gk.alab.ip.titech.ac.jp
to: kuruwa-gw.alab.nii.ac.jp
cost: 30$, time: -100s
```
6 expired
```
numberOfJobs: 20
from: kuruwa-gw.alab.nii.ac.jp
to: gk.alab.ip.titech.ac.jp
cost: -10$, time: 50s
```

Fig. 8 An example of execution advice.

Execution advice are managed in three statuses: proposed, used and expired. Advice 6 is in the expired status because less than 20 tasks are available in cluster *kuruwa*.

The advice manager presented in Fig. 2 performs the execution advice. The advice manager periodically communicates with the execution information manager, the policy manager and the cluster pool manager presented in Fig. 2. The execution information manager collects past task execution records, which include errors, performances and costs to run tasks on remote computational resources. When a new resource becomes available, the cluster pool manager notifies the advice manager concerning the information of the new resource.

3.4 Scheduling Policy

The scheduling policy mechanism enables the user to give the customized resource allocation algorithm. Two interfaces to give the user's policy, the web interface and the policy file in ClassAd format are available in InterS.

The policy manager presented in Fig. 2 actuates changing the user's policy to scheduling at application run time. It includes creating new scheduling plans, adding new computational resources, changing budget constraints and the configuration of test jobs. The user needs to write all the candidate resources in the policy file.

Figure 9 shows an example of the policy file. Every change is detected and handled by the policy manager. "CostLimitation" stands for the limitation of the entire application run. "TestJobCostLimitation" specifies the total cost upper bound of test jobs. The user may change these options to affect the behavior of the PerformancePrior and CostPrior scheduling agents. Cluster information is written by ClassAd expressions; for example, REMOTE_NODE_NUMOFCORE is assigned to corePerCPU, if InterS can retrieve the number of cores in "gs" from MDS.

```
costLimitation = 17;   // Budget constraints
testJobCostLimitation = 3;   // Test job cost constraints
planTypeForFaultTolerant = "performancePrior";
cluster2 = [   // Cluster definition
    name = "gs.alab.ip.titech.ac.jp";
    price = 0.002;
    cpuInfo = (REMOTE_NODE_CPUINFO is undefined)? 1263.475 : REMOTE_NODE_CPUINFO;
    memInfo = (REMOTE_NODE_MEMINFO is undefined)? 1010 : REMOTE_NODE_MEMINFO;
    numOfNodes = (REMOTE_NODE_NUMBER is undefined)? 4 : REMOTE_NODE_NUMBER;
    corePerCPU = (REMOTE_NODE_NUMOFCORE is undefined)? 2 : REMOTE_NODE_NUMOFCORE;
    onFailReduceRatio = numOfJobsPerCallReduceRatio;
];
```

Fig. 9 A policy file example.

4. Experimental Study

This section presents experiments to see the effect of interactive scheduling provided by InterS. We implemented InterS and conducted experiments using PC clusters located in two sites. In the experiments, we verify the effect of interactive scheduling by InterS using three scenarios.

4.1 Experimental Setting

Table 3 shows PC clusters used in the experiments. Two clusters, *davinchi* and *kuruwa*, are located in Yokohama and Tokyo, respectively. The execution time in the table shows the average benchmark execution time of each core in each PC cluster. The EP benchmark (Class A) in NAS Parallel Benchmarks[9] is used here. We assume that a unit price to run computation on the PC cluster, price in Table 3, is announced from resources providers.

In this experiment, we used three scenarios, which present interactions between InterS and the user. Using the scenarios, we show how interactive scheduling by InterS works and how to verify the effect of InterS.

4.2 Automatic Scheduling

Figure 10 presents an interaction scenario for the automatic scheduling mechanism. In the scenario, the user first submits the application, which consists of 42 parallel jobs to run the NAS Parallel Benchmark EP (Class A). InterS then presents resource allocation plans for the submitted jobs. The user chooses the most suitable one for his/her requirements, and InterS starts the execution of the

Table 3 The experimental environment.

	DRM ⋆1	CPU (vendor/MHz)	nodes × cores	OS	exec. time (s) ⋆2	price ($/(*core* · *s*))
davinchi	Torque	Xeon/2392	8 × 2	Ubuntu 7.10	120	0.00052
kuruwa	Torque	AMD/2412	10 × 4	CentOS 5.0	70	0.0010

1. DRM: distributed resource manager, such as SGE, PBS, Condor
2. Average execution time of EP (Class A) benchmark per core in each cluster

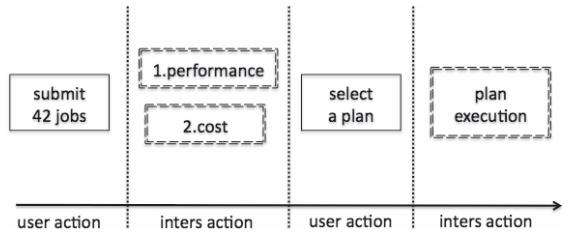


Fig. 10 An interaction for the automatic scheduling mechanism.

Table 4 The Evaluation of Automatic Scheduling.

PerformancePrior Plan (ranking by execution time)			
Cluster	number of jobs	the plan (time, cost)	evaluation results (time, cost)
davinchi	2	120 [sec]	142 [sec]
kuruwa	40	\$2.84	\$3.65

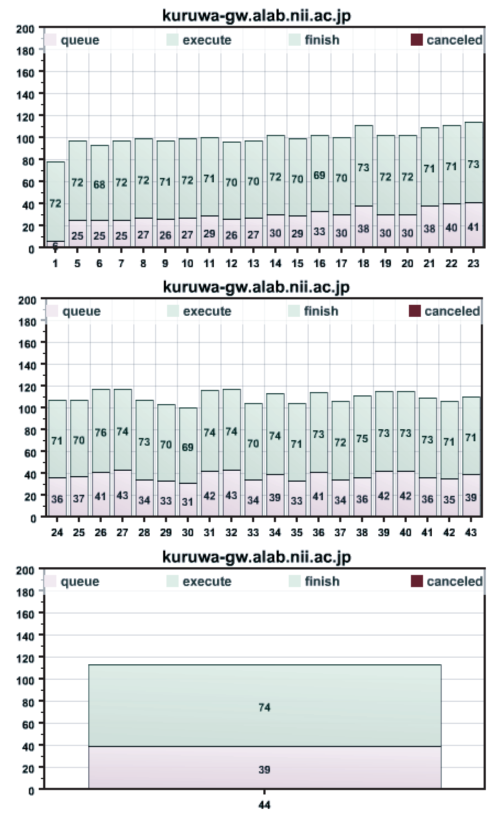
CostPrior Plan (ranking by fixed price)			
Cluster	number of jobs	the plan (time, cost)	evaluation results (time, cost)
davinchi	16	120 [sec]	176 [sec]
kuruwa	26	\$2.76	\$3.00

1. the definition of price is the same as in Table 3

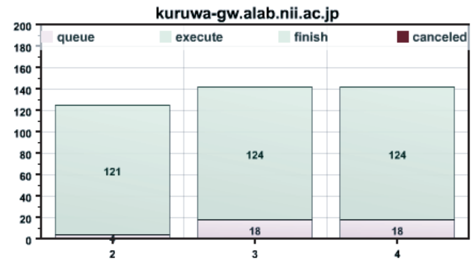
jobs. In Fig. 10, InterS presents two resource allocation plans such as CostPrior and PerformancePrior plans. Besides the above plans, InterS can also present RoundRobin and MasterWorker plans, which do not consider the costs of running the users application. The CostPrior plan and the PerformancePrior plan make ranking of clusters by cost and performance, respectively. The CostPrior plan chooses computing resources that have a lower cost first, and PerformancePrior chooses resources, which show shorter a execution time first.

In this scenario, what the user needs to do is to choose a suitable plan for him/her, and the user does not have to configure details of the plan. We believe that this mechanism particularly helps non-expert users.

Table 4 shows the results of resource allocation, execution times and costs of running the application where the user chooses each of the two resource allocation plans. For example, when the user choose the PerformancePrior plan, 40 CPU cores on *kuruwa* were allocated to the application, (or 40 jobs were assigned to



(a) monitoring results in *kuruwa*



(b) monitoring results in *gs*

Fig. 11 Execution results of PerformancePrior plan.

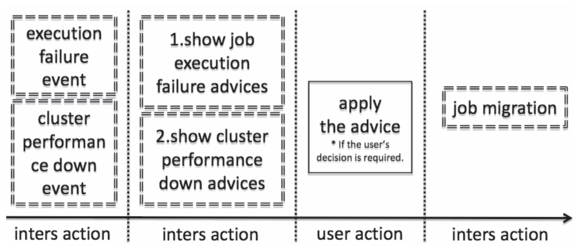

execution failure event
cluster performan ce down event
1.show job execution failure advices
2.show cluster performance down advices
apply the advice
* If the user's decision is required.
job migration
inters action
inters action
user action
inters action

Fig. 12 An interaction for the execution advice mechanism.

kuruwa) and two CPU cores on *davinchi* were allocated to the application. The PerformancePrior plan allocates resources to minimize the application execution time. Thus, InterS allocated all CPU cores (40 cores) on *kuruwa* with higher performance and two CPU cores on *davinchi*. InterS estimated the application would finish in 120 [sec] with a cost of \$2.84; however, the actual execution time and the cost were 142 [sec] and \$4.39.

The gap between the estimated execution time and the actual time is due to the waiting time in the batch queue, or Torque. InterS does not estimate the queuing time in the current implementation. We will leave the queuing time estimation in InterS as our future work. When the user chose the CostPrior plan, InterS allocated 16 CPU cores on *davinchi* with a lower price and 26 CPU cores on *kuruwa*.

InterS also presents monitoring results at application run time. **Figure 11** shows a snapshot of the monitoring results in *kuruwa* and *gs* respectively, when the user chooses the PerformancePrior plan. The X-axis indicates the jobs. Note that Fig. 11 includes one test job on each cluster for estimating the execution time. The Y-axis shows the execution time in seconds. It shows that first two jobs run on the *davinchi* and the remaining 40 jobs run on the *kuruwa*.

4.3 Execution Advice

Figure 13 shows a snapshot of the monitoring results by InterS. In this snapshot, job 26 and job 27 are terminated due to the failure on the computing nodes. InterS automatically detects the failure of job execution during the run time and give the user through the web interface. It also has an option to detect performance degradation of computing nodes due to the high load caused by external jobs and informs the user.

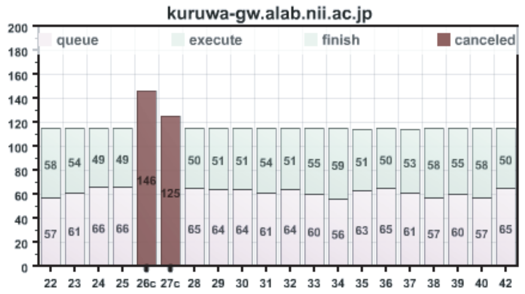
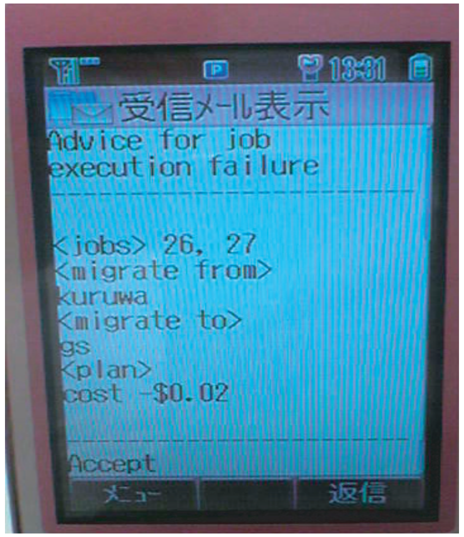


Fig. 13 Output of InterS when job execution failed.


受信メール表示
Advice for job
execution failure

<jobs> 26, 27
<migrate from>
kuruwa
<migrate to>
gs
<plan>
cost -\$0.02

Accept

Fig. 14 Screen capture of advices on cell phone.

The scenario in **Fig. 12** presents interactions between InterS and the user using the execution advice mechanism to solve problems due to the failure or the performance degradation. When InterS detects events of the failure or the performance degradation on computing nodes, it makes a plan to move jobs running on the computing nodes to other nodes and gives the user advice of the job mi-

Table 5 The evaluation of execution advice.

(a) Advice for Job Execution Failure

Cluster	#initial cores[1]	#failed jobs	#migrated jobs	#final cores[2]	change of cost[3]
davinchi	2	0	+2	4	\$0.12
kuruwa	40	2	−2	38	−\$0.14

(b1) Advice for Cluster Performance Degradation (Automatic Migration)

Cluster	#initial cores[1]	#migrated jobs	#final cores[2]	change of cost
davinchi	2	+10	12	\$0.62
kuruwa	40	−10	30	−\$0.70

(b2) Advice for Cluster Performance Degradation (With the User's Permission)

Cluster	#initial cores[1]	#migrated jobs	#final cores[2]	change of cost[3]
davinchi	16	−14	2	−\$0.87
kuruwa	26	+14	40	\$0.98

(c) External Jobs Running on Kuruwa and Davinchi Clusters[4]

Cluster	#cores	normal exec. time (s)	burden exec. time (s)[5]	performance down[6]
kuruwa	40	70	308	23%
gs	16	120	313	38%

1. CPU cores at job allocation 2. CPU cores after job migration 3. Cost changes because of job migration
4. Details of external jobs: two parallel execution of skampi_coll.ski bechmark set
5. The execution time of the PerformancePrior plan with external jobs.
6. Percentage of performance compared to PerformancePrior plan execution only.

gration. When the migration cost becomes higher than the previous plan, the automatic scheduling mechanism needs the user's decisions. In this situation, InterS does not move jobs unless the the user provides interaction. Otherwise, InterS executes the migration automatically for the user.

In the current implementation, there are two options to give advice to the user, the web interface and the user's cell phone. It is not realistic to assume a user sits in front of the user's PC for monitoring the long-term application. The user instructs InterS when he/she accepts the advice or chooses one piece of advice from the multiple pieces of advices given by InterS. The user can give instructions through the web interface or the cell phone. Then InterS moves jobs following the user's instructions. **Figure 14** shows a snapshot of the advice given by InterS through the cell phone. It tells the user to migrate jobs 26 and 27 from *kuruwa* to *davichi* (the gateway node is named gs), because InterS detects the above jobs cannot finish normally in *kuruwa*. And it also shows that the migration can save the user \$0.02. The user just needs to choose the options: accept or deny at the end of the advice, then a reply email is created. After the user has sent the email back, InterS can explain the email and apply the advice if the user chose the accept option.

Table 5 shows the results of job migration using the execution advice mechanism. The application used in the experiment is the same as the scenario of Fig. 10. For the results in Table 5 (a), we simulated failure of nodes on *kuruwa* by removing jobs in the batch queue on *kuruwa*. InterS detected failure of two jobs in the queue and calculated the costs of moving the jobs to *davinchi*. The costs are less than the previous plan, so two jobs were moved from *kuruwa* to *davinchi* automatically. For the results in Table 5 (b1), we simulated the performance degradation on *kuruwa* by running external jobs presented in Table 5 (c).

```
 1.  // Test job cost constraints
 2.  testJobCostLimitation = 3;
 3.  planTypeForFaultTolerant = "performancePrior";
 4.  cluster1 = [   // Kuruwa cluster definition
 5.      name = "kuruwa-gw.alab.ip.titech.ac.jp";
 6.      price = 0.0001;
 7.      cpuInfo = (REMOTE_NODE_CPUINFO is undefined)? 2412 : REMOTE_NODE_CPUINFO;
 8.      memInfo = (REMOTE_NODE_MEMINFO is undefined)? 4021 : REMOTE_NODE_MEMINFO;
 9.      numOfNodes = (REMOTE_NODE_NUMBER is undefined)? 10 : REMOTE_NODE_NUMBER;
10.      corePerCPU = (REMOTE_NODE_NUMOFCORE is undefined)? 4 : REMOTE_NODE_NUMOFCORE;
11.      onFailReduceRatio = numOfJobsPerCallReduceRatio;
12.  ];
13.  cluster2 = [   // Davinchi cluster definition
14.      name = "gs.alab.ip.titech.ac.jp";
15.      price = 0.00052;
16.      cpuInfo = (REMOTE_NODE_CPUINFO is undefined)? 2392 : REMOTE_NODE_CPUINFO;
17.      memInfo = (REMOTE_NODE_MEMINFO is undefined)? 1010 : REMOTE_NODE_MEMINFO;
18.      numOfNodes = (REMOTE_NODE_NUMBER is undefined)? 8 : REMOTE_NODE_NUMBER;
19.      corePerCPU = (REMOTE_NODE_NUMOFCORE is undefined)? 2 : REMOTE_NODE_NUMOFCORE;
20.      onFailReduceRatio = numOfJobsPerCallReduceRatio;
21.  ];
22.  scheduling_MyRanking = [  // Definition of scheduling_"PlanName" will add a "PlanName" plan
23.      cluster_ranking_data = { cluster2, cluster2, cluster1 }  // Using clusters in the list by the index order
24.  ];
```

Fig. 15 The policy file used in this experimental study.

Because the migration saves cost for the user this time, 10 jobs were moved from highly loaded *kuruwa* to lightly loaded *davinchi* automatically. The results in Table 5 (b2) show that the migration costs more than the previous plan, then InterS asks for the user's decision. From the results, we can confirm that jobs migrated successfully when problems occurred. The advice can achieve a better performance, and the user only needs to choose and apply the advice. Thus, the execution advice mechanism with interactive operation works robustly with the changing nature of grid environment.

4.4 Scheduling Policy

Figure 15 shows the policy file used in this experiment. Many behaviors can be customized through the scheduling policy file, such as cost limitation (line 1), cluster configuration (through 4 to 21 lines) and default to use plans when job failure (line 3). **Figure 16** shows the usage of scheduling policy. The user creates a new automatic scheduling plan by adding a "scheduling_MyRanking" definition to the policy file. MyRanking (through 22 to 24 lines) utilizes resources in the order: *davinchi*, *davinchi*, *kuruwa*. As described in Section 4.2, the automatic scheduling mechanism creates plans by cost or performance order, so it left the problem that user cannot customize the order by hand. The user would like to rank clusters by him/herself, for instance, when some crucial data sets are gathered in the local clusters (such as *davinchi*), so he/she has to deal with the data in the dedicated clusters and submit enough jobs to process the data sets.

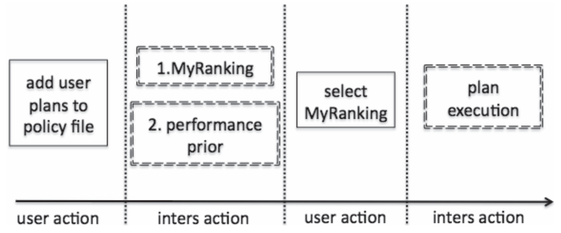


Fig. 16 An interaction for the scheduling policy mechanism.

Table 6 shows the results of the MyRanking plan. InterS allocates 32 jobs to *davinchi* and the execution time is raised to 240 [sec]. The remaining 10 jobs

Table 6 The evaluation of scheduling policy.

Resource Allocation Results of MyRanking Plan			
Cluster	#initial cores[1]	the plan (time, costs)	results (time, costs)
davinchi	32	240 [sec]	310 [sec]
kuruwa	10	\$2.7	\$3.49

1. Job allocation with "MyRanking" plan.

are allocated to *kuruwa* and the execution is finished successfully. Thus, the scheduling policy mechanism provides users with an interface to customize the scheduling behavior and make their own plans to meet special requirements.

5. Conclusions

This paper proposed InterS, an interactive scheduling system for GridRPC applications. Interactive scheduling is enabled by the cooperation of three mechanisms: automatic scheduling mechanism, execution advice mechanism and scheduling policy mechanism. We implemented the interactive scheduler on the testbed and evaluated the effectiveness of the interactive scheduling using the application scenarios. The experiments presented in this paper are limited to those with a simple application scenario. We plan to evaluate the advantage of InterS through experiments with more scenarios.

Acknowledgments A part of this work is supported by Japan Society for the Promotion of Science (JSPS) within the framework of Global COE Program "Photonics Integration-Core Electronics".

References

1) Huedo, E., Montero, R.S. and Llorente, I.M.: A Framework for Adaptive Execution on Grids, *Intl. J. of Software — Practice and Experience* (*SPE*) (2004).
2) Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T. and Matsuoka, S.: Ninf-g: A reference implementation of rpc-based programming middleware for grid computing, *Journal of Grid Computing* (2003).
3) Tanimura, Y., Nakada, H., Tanaka, Y. and Sekiguchi, S.: Implementation of A Task Farming API over GridRPC Framework, *IPSJ SIG Technical Reports, HPC-103* (2005) (in Japanese).
4) Raman, R. jesh, Livny, M. and Solomon, M.: Matchmaking: Distributed Resource Management for High Throughput Computing, *Proc. Seventh IEEE International Symposium on High Performance Distributed Computing*, July 28-31, Chicago, IL (1998).
5) Buyya, R., Abramson, D. and Giddy, J.: Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid, *The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region*, hpc, p.283 (2000).
6) Watanabe, H., Hirasawa, S. and Honda, H.: F-Omega: A Framework for GridRPC Application with Adaptive Server Use, *IPSJ SIG Technical Reports, HOKKE-2007* (2007) (in Japanese).
7) Seymour, K., Nakada, H., et al.: Overview of GridRPC: A Remote Procedure Call API for Grid Computing, *GRID COMPUTING, GRID 2002*, LNCS 2536 (2002).
8) Czajkowski, K., Fitzgerald, S., Foster, I. and Kesselman, C.: Grid Information Services for Distributed Resource Sharing, *Proc. 10th IEEE Int. Symp. on High Performance Distributed Computing*, San Francisco, CA, USA (2001).
9) Bailey, D., Barton, J., Lasinski, T. and Simon, H. (Eds.): The NAS Parallel Benchmarks, NAS Technical Report RNR-91-002, NASA Ames Research Center, Moffett Field, CA (1991).
10) SKaMPI 5 is a benchmark for MPI implementations. http://liinwww.ira.uka.de/~skampi/
11) An dynamic language for Java.

(Received July 24, 2009)
(Accepted December 2, 2009)

Hao Sun received his B.E. and M.E. degrees from Tokyo Institute of Technology University in 2006, 2008, respectively. He is now a doctor course student at the Department of Information Processing, Tokyo Institute of Technology from 2008. His research interests are parallel and distributed computing and e-science. He is a member of IPSJ.

Kento Aida received his B.E., M.E. and Dr.Eng. degrees from Waseda University in 1990, 1992, 1997, respectively. He became a research associate at Waseda University in 1992. He joined the Tokyo Institute of Technology and became a research scientist at the Department of Mathematical and Computing Sciences in 1997, an assistant professor at the Department of Computational Intelligence and Systems Science in 1999, and an associate professor at the Department of Information Processing in 2003, respectively. He was a researcher at PRESTO, Japan Science and Technology Agency (JST) from 2001 through 2005. He was a research scholar at the Information and Computer Sciences Department, University of Hawaii in 2007. He is now a professor at the National Institute of Informatics and a visiting professor at the Department of Information Processing, Tokyo Institute of Technology from 2007. His research interests are parallel and distributed computing and e-science. He is a member of IEICE, IEEJ, ACM and IEEE-CS.