Regular Paper

Parallel Branch and Bound Algorithm with the Hierarchical Master-Worker Paradigm on the Grid

Kento Aida,[†,☆] Yoshiaki Futakata[††] and Tomotaka Osumi[†††,☆☆]

This paper proposes a parallel branch and bound algorithm that efficiently runs on the Grid. The proposed algorithm is parallelized with the hierarchical master-worker paradigm in order to efficiently compute fine-grain tasks on the Grid. The hierarchical algorithm performs master-worker computing in two levels, computing among PC clusters on the Grid and that among computing nodes in each PC cluster, and reduces communication overhead by localizing frequent communication in tightly coupled computing resources, or a PC cluster. On each PC cluster, granularity of tasks dispatched to computing nodes is adaptively adjusted to obtain the best performance. The algorithm is implemented on the Grid testbed by using GridRPC middleware, Ninf-G and Ninf. In the implementation, communication among PC clusters is securely performed via Ninf-G using the Grid Security Infrastructure, and fast communication in each PC cluster is performed via Ninf. The experimental results showed that parallelization with the hierarchical master-worker paradigm using combination of Ninf-G and Ninf effectively utilized computing resources on the Grid in order to run a fine-grain application. The results also showed that the adaptive task granularity control automatically gave the same or better performance compared to performance with manual control.

1. Introduction

Grid computing is regarded as new computing technology that provides huge computational power by employing computing resources geographically distributed over the internet. A user of Grid computing can use the computational power securely, stably and easily. Thus, it has possibility not only to reduce execution time of applications currently computed on hi-end computing systems but also to expand applications of high-performance computing or the internet. However, on the current Grid infrastructures, applications that are effectively computed are limited. Some applications show unacceptable performance on the Grid because of the large overhead, e.g., the overhead caused by poor network performance, and that by Grid security service such as user authentication and secure communication.

An example of applications that show poor performance on the Grid is a fine-grain application. Performance of an application that consists of small tasks is significantly affected by relatively large overhead on the Grid. Thus, currently, applications effectively running on the Grid have enough task grain sizes that compensate for the overhead, dozens of seconds or hundreds seconds [1)~4)]. For instance, the work presented in Ref. 1) shows experimental results for the application, which solves the quadratic assignment problem, on the Grid; and its mean task grain size, or the mean execution time of the single task in the application, is 190 [sec]. The work in Ref. 3) also presents experimental results for the application, which solves the traveling salesman problem, on the Grid; and its mean task grain size is distributed from 177 [sec] through 430 [sec].

However, there exist finer-grain applications, where the mean task grain sizes are a few seconds or less, and developers/users of these applications give up running their applications on the Grid. Some of these applications might consist of a huge number of fine-grain tasks and require huge computational power. Thus, developing algorithms to efficiently run these fine-grain applications on the Grid contributes for expanding applications of Grid computing. An example of these fine-grain applications is a branch and bound application. A branch and bound algorithm is widely used to solve optimization problems in many engineering fields, e.g., operations research, control engineering, multiprocessor scheduling [5)~8)]. However, many applications using a branch and bound algorithm tend to be composed of a huge number of fine-grain tasks, i.e., they are fine-grain applications.

† Tokyo Institute of Technology/PRESTO, JST
†† University of Virginia
††† Tokyo Institute of Technology
☆ Presently with Tokyo Institute of Technology
☆☆ Presently with NTT Communications Corporation

This paper proposes a parallel branch and bound algorithm that efficiently runs on the Grid. The proposed algorithm consists of two techniques, parallelization of a branch and bound algorithm with the hierarchical master-worker paradigm and the adaptive task granularity control.

The proposed algorithm parallelizes a branch and bound algorithm with the hierarchical master-worker paradigm[9] in order to efficiently compute fine-grain tasks on the Grid. The hierarchical algorithm performs master-worker computing in two levels, computing among PC clusters on the Grid and that among computing nodes in each PC cluster. It avoids performance degradation, which is mainly caused by communication overhead between a master process and worker processes, by localizing frequent communication in tightly coupled computing resources, or a single PC cluster. In each PC cluster, granularity of tasks dispatched to worker processes affects performance of the computation. The proposed algorithm adaptively adjusts granularity of tasks during the application run, and gives the best performance.

While the hierarchical parallelization is becoming common on the Grid, there are problems in order to parallelize a branch and bound algorithm in the hierarchical way on the Grid. For instance, tasks should be efficiently dispatched to worker processes running on multiple sites, and the best upper bound needs to be efficiently shared among worker processes running on multiple sites. The proposed algorithm solves above problems.

The proposed algorithm is implemented in a Grid application by using GridRPC[10] middleware, Ninf-G[11] and Ninf[12]. GridRPC is a programming model based on client-server-type remote procedure calls on the Grid, and its model and APIs have been standardized in GGF[13]. In the implementation, communication among PC clusters is securely performed via Ninf-G, which uses the Grid Security Infrastructure (GSI) provided in the Globus Toolkit[14], and communication among computing nodes in each PC cluster is performed via Ninf, which has no mechanism to support Grid security service but enables fast invocation of remote computing routines.

While fine-grain applications on distributed systems have been discussed in literatures[15],[16], the detailed performance of a fine-grain parallel branch and bound application with GridRPC on the Grid has not been sufficiently discussed. The contribution of this paper is to propose an efficient parallelization scheme of the fine-grain parallel branch and bound algorithm, and to present its implementation and detailed performance on the Grid constructed with standard Grid technology[13],[14]. Furthermore, the proposed algorithm presented in this paper enhances the algorithm firstly presented in the conference paper[9] by adding the new idea of the task granularity control.

The experimental results showed that the proposed algorithm implemented using combination of Ninf-G and Ninf effectively utilized computing resources on the Grid testbed in order to run the fine-grain branch and bound application, where the average computation time of the single task was less than 1 [sec]. The results also showed that the adaptive task granularity control automatically gave the same or better performance compared to performance with manual control.

The rest of this paper is organized as follows: Section 2 gives the background and presents the proposed algorithm, and Section 3 presents implementation of the proposed algorithm. Section 4 presents experimental results on the Grid testbed. Section 5 describes related works, and Section 6 concludes the work presented in this paper and outlines future work.

2. Parallel Branch and Bound Algorithm

This section summarizes an overview of a parallel branch and bound algorithm and presents the proposed parallelization scheme.

2.1 Branch and Bound Algorithm

The main idea of a branch and bound algorithm is to find an optimal solution and to prove its optimality by successively partitioning the feasible set of the solution, or the original problem, into subproblems of smaller size. To this end, these subproblems are investigated by computing lower and upper bounds of the objective function. These lower and upper bounds are used to avoid exhaustive search of the solution space.

Procedures for the branch and bound algorithm are illustrated by a tree structure like an example in **Fig. 1**. In the figure, the root node on the tree denotes the original problem. The original problem is partitioned into two subproblems, which are depicted as child

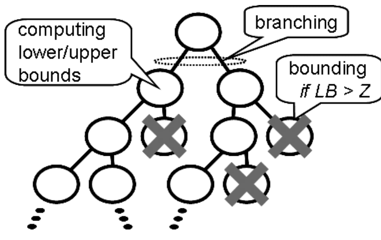

Fig. 1 An example of a search tree.

nodes of the root node. This partitioning process is called branching. After the branching, lower and upper bounds of the objective function are computed on each subproblem, and the best upper bound is computed. The best upper bound means the lowest upper bound among upper bounds currently computed on all subproblems☆. By continuing in this way, a tree structure called a search tree is obtained. Some subproblems, where their lower bounds (LB) are higher than the current best upper bound (Z), can be pruned, because further branching for these subproblems does not yield an optimal solution. This process is called pruning or bounding, and efficient pruning is effective to reduce computation time. Finally, an optimal solution is obtained, when the gap between the best upper bound and the lower bound becomes zero or the allowable error.

A branch and bound algorithm is able to be parallelized by distributing computation of subproblems on multiple computing nodes. Parallel branch and bound algorithms with the master-worker paradigm, where a single master process dispatches tasks to multiple worker processes, have been proposed in many literatures [1),3),17)].

2.2 Parallelization with Hierarchical Master-Worker Paradigm

In the proposed algorithm, a branch and bound algorithm is parallelized with the hierarchical master-worker paradigm [9)] to avoid performance degradation exhibited in the conventional master-worker paradigm on the Grid. In this paradigm, a single supervisor process controls multiple process sets, each of which is composed of a single master process and multiple worker processes. Distribution of tasks is performed in two phases: the distribution from the supervisor process to master processes and that from the master process to worker processes. Collection of computed results is performed in the reverse way. The hierarchical master-worker paradigm has advantages compared with the conventional master-worker paradigm. The first advantage is to reduce communication overhead by putting a set of a master process and worker processes, which frequently communicate with each other, in tightly coupled computing resources. The second advantage is to avoid the performance bottleneck due to a single heavily loaded master process by distributing the workload among multiple master processes.

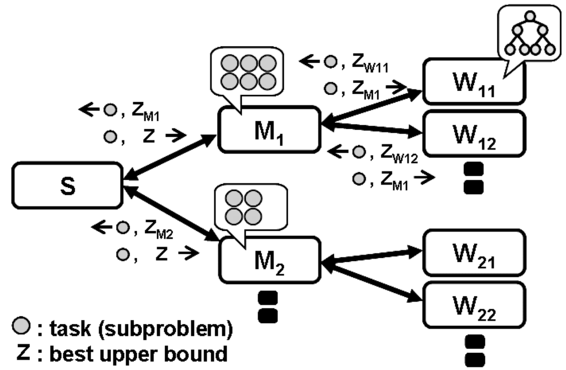

Fig. 2 The branch and bound algorithm with the hierarchical master-worker paradigm.

The branch and bound algorithm parallelized with the hierarchical master-worker paradigm performs parallel computation in the following way: A set of the master and worker processes performs a parallel branch and bound operation for a subset of the search tree, that is, the master process dispatches subproblems to multiple worker processes and receives computed results from these worker processes. The supervisor process performs load balancing among master processes and advertises the best upper bound of the objective function by communicating with master processes. The advertisement of the best upper bound is crucial to improve performance of applications, because it accelerates pruning. **Figure 2** shows an overview of the branch and bound algorithm with the hierarchical master-worker paradigm. Symbols in the figure, Z_{Wi}, Z_{Mj} and Z, denote the current upper bound of the objective function stored on the worker process W_i, the master process M_j and the supervisor process, respectively.

In each set of a master process and worker processes, the master process maintains a subset of the search tree. Un-computed subproblems are saved in the queue on the master pro-

☆ This paper assumes an optimization problem that minimizes the objective function.

cess. It dispatches subproblems, which correspond to leaf nodes on the search tree, to multiple worker processes and receives computed results from these worker processes. Simultaneously, the master process sends the best upper bound stored on itself to worker processes.

The worker process that received a subproblem from the master process performs branching, that is, it partitions the subproblem into multiple (sub-)subproblems and generates the sub-tree. Next, it computes lower and upper bounds for each subproblem on the sub-tree and performs bounding; that is, it prunes an unnecessary subproblem, where its lower bound exceeds the current best upper bound. Finally, the worker process returns computed results to the master process. The computed results contain lower and upper bounds computed on the worker process, the solution, and subproblems that have been generated by branching and have not been pruned on the worker process.

The size of the sub-tree generated on a worker process corresponds to granularity of the task dispatched by the master process. The proposed algorithm defines the task granularity as depth of the sub-tree. For instance, W_{11} in Fig. 2 generates the sub-tree with $depth = 2$. The performance of a branch and bound operation performed on a set of a master process and worker processes is affected by task granularity. The further discussion about the task granularity is presented in Section 2.3.

The supervisor process periodically queries master processes about their statuses, which include the number of un-computed subproblems and upper bounds stored on these master processes. When numbers of un-computed subproblems, or loads, on master processes are not well balanced, the supervisor process moves un-computed subproblems from highly loaded master processes to lightly loaded master processes. When the supervisor process finds the new best upper bound on the master process Mi, where $Z_{Mi} < Z$, the supervisor process updates the best upper bound stored on the supervisor process (Z) and advertises the updated Z to other master processes. Thus, a master process communicates both with its worker processes and with the supervisor process. Finally, the supervisor process terminates computation if the termination condition is satisfied.

2.3 Adaptive Task Granularity Control

Performance of computation performed in each set of a master process and worker processes, or in a PC cluster, is affected both by communication overhead and an interval between advertisements of the best upper bound. The communication overhead is caused by communication between a master process and worker processes, and reducing the overhead improves the performance. On the other hand, the new best upper bound is advertised to worker processes by a master process, and reducing the interval improves the performance. Note that bounding is performed on a worker process using the best upper bound, which is available on the worker process. If a worker process finds the new best upper bound, quickly advertising the new best upper bound to other worker processes helps other worker processes to efficiently perform bounding.

There is a tradeoff between reducing the communication overhead and reducing the interval between the advertisements. In order to reduce the communication overhead, granularity of a task should be relatively large compared to the overhead. However, in order to reduce the interval, task granularity should be small so that a master process dispatches a task with the new best upper bound to worker processes more frequently.

The proposed algorithm adaptively adjusts granularity of tasks to achieve both low communication overhead and efficient advertisement of the best upper bound. The idea of the scheme is:

- Increasing task granularity to reduce communication overhead, when we cannot expect effect by advertising the new best upper bound frequently, or the gap between the newly computed best upper bound and the current one is small,
- Decreasing task granularity to reduce the interval between the advertisements, when we can expect substantial effect to frequently advertise the new best upper bound, or the gap between the new best upper bound and the current one is large.

Whenever a master process (j) receives the new best upper bound (Z_{Wi}) from the worker process (i), the master process compare Z_{Wi} and the best upper bound stored on the master process (Z_{Mj}). If $Z_{Wi} < Z_{Mj}$, the master process computes the gap between Z_{Wi} and Z_{Mj},

and updates Z_{Mj} to the value of Z_{Wi}. Then, if the gap is smaller than a certain threshold, the master process increases granularity of the task that is dispatched in the next turn, or it increases depth of the sub-tree that is generated from the task. If the gap is larger than or equal to the threshold, the master process reduces granularity of the task. When the master process dispatches a new task, it notifies a worker process of the task granularity, or the depth of the sub-tree.

The threshold, θ, is computed as follows:

$$\theta = a \times \Delta Z \quad (1)$$

Here, ΔZ means the gap computed when the Z_{Mj} is updated for the first time since the application starts, and the variable, a ($0 \leq a \leq 1$), indicates a constant parameter. The preliminary experiment shows that the gap computed for the first time exhibits the maximum gap in most cases; thus, the proposed scheme computes the threshold using ΔZ.

A worker process receives a task from the master process, and generates a sub-tree with depth notified by the master process. Whenever a worker process finds the new best upper bound during computation of the task, it immediately reports the new best upper bound to the master process.

3. Implementation

A Grid testbed considered in this paper consists of multiple PC clusters that are connected to the internet and are administrated in multiple domains. In order to efficiently run an application parallelized with the hierarchical master-worker paradigm on the Grid testbed, mapping of processes on computing resources and communication methods among these processes are crucial. Particularly, to run a fine-grain application on the Grid testbed, implementation to reduce the communication overhead is necessary, because performance of a fine-grain application is significantly affected by the overhead.

3.1 Process Mapping

Figure 3 illustrates mapping of processes in the application on the Grid testbed. In the figure, multiple PC clusters, which are depicted by squares with dotted lines, are distributed on the internet. Symbols in the figure, S, M and W, denote a supervisor process, a master process and a worker process, respectively. The symbol, C, denotes a process that runs with the master process on the same computing node, which is depicted by the square with solid lines. Processes, M and C, are invoked by the supervisor process via GridRPC, and examples of codes are presented in Section 3.3. The process M communicates with the process C via the inter-process communication mechanism provided by the System V message queue.

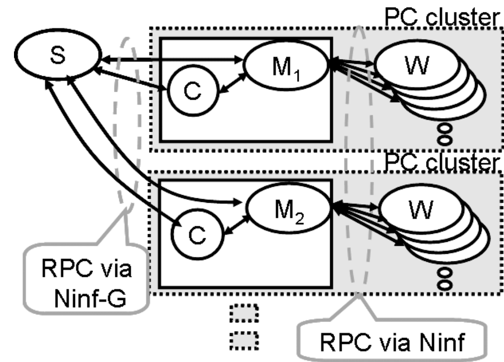

Fig. 3 Process mapping.

The process, C, relay operations between the supervisor process and the master process. These relay operations consist of the following operations:

- initializing the queue on the master process
- querying about a status of the master process
- updating the new best upper bound saved on the master process
- stealing subproblems from the master process
- assigning subproblems to the master process
- notifying the master process to stop computation

As described in Section 2.2, a master process communicates both with its worker processes and with the supervisor process. The former communication is performed for computation of subproblems, or dispatching subproblems to worker processes and receiving computed results. The process C relays operations requested by the supervisor process so that computation on master processes will not be blocked by the supervisor process.

A set of the master process (M and C) and worker processes (W) are mapped on computing nodes in a single PC cluster, where computing nodes are connected via a dedicated high-speed network. This mapping is effective to reduce communication overhead, because the amount of data transferred between the supervisor process and master processes is much

smaller than that between a master process and worker processes. The supervisor process is mapped on a computing node on the Grid testbed.

3.2 Communication among Processes

On the Grid testbed, communication between the supervisor process and master processes is performed among different domains via the internet, while that between a master process and worker processes is performed in a single PC cluster. Thus, the former communication needs to be securely performed using Grid security service, e.g., user authentication over different domains, secure communication and etc., even if it causes additional overhead. The latter communication needs to be fast performed without the Grid security service, because communication inside a PC cluster does not require user authentication and secure communication.

In the implementation, communication between the supervisor process and master processes is performed by Grid RPC middleware Ninf-G[11)], which uses the Grid Security Infrastructure (GSI) provided in the Globus Toolkit[14)]. Also, communication between a master process and worker processes is performed by Ninf[12)], which has no mechanism to support Grid security service but enables fast invocation of remote computing routines.

3.3 Implementation with GridRPC

Ninf-G[11)] is reference implementation of GridRPC API. The client program is able to invoke server programs, or executables, on remote computing resources using the Ninf-G client API. Ninf-G is implemented on the Globus Toolkit[14)]. When the client program starts its execution, it accesses MDS to obtain interface information to invoke the remote executable. Next, the client program requests GRAM to invoke the remote executable. In this phase, authentication is performed using GSI. After the invocation, the remote executable connects back to the client to establish connection. Finally, the client program dynamically encodes its arguments according to the interface information, and transfers them using Globus I/O and GASS. Ninf[12)] has been developed as an initial product of Ninf-G. Ninf provides a client program almost same API as Ninf-G. Ninf is implemented as a standalone software system, and has no mechanism to support Grid security service; however, it enables fast invocation of remote computing routines with low overhead.

The supervisor process is firstly initiated when a user starts the application. Next, it initiates a master process on the designate node for each PC cluster using Ninf-G. The example of the program code with the Ninf-G API on the supervisor process is as follows:

```
for(i = 0; i < nMaster; i++){
  grpc_function_handle_init(&ex[i],···,
  "Master");
}

for(i = 0; i < nMaster; i++){
  pid[i] = grpc_call_async(&ex[i],···);
}
```

Here, *nMaster* denotes the number of master processes, which is equal to the number of PC clusters employed to run the application. The API, **grpc_function_handle_init()**, is called to initialize the function handle to invoke the remote executable, or the master process. Its arguments include a hostname of the remote computing node, a port number and a path for the executable. The API, **grpc_call_async()**, is called to invoke the remote executable indicated by the function handle in its argument.

A master process initiates worker processes on computing nodes in the same PC cluster and dispatches subproblems to idle worker processes using Ninf. The example of the program code with the Ninf API on the master process is as follows:

```
for(i = 0; i < nWorker; i++){
  sprintf(ninfURL[i], NINF_URL_LENGTH,
  "ninf://%s/Worker", workerList[i]);
  exs[i] =
    Ninf_get_executable(ninfURL[i]);
}

while (1) {
  id = Ninf_wait_any();
  for (i = 0; i < nWorker; i++)
  if (ids[i] == id) break;
  :
  ids[i] =
    Ninf_call_executable_async(exs[i],
···);
}
```

Here, *nWorker* denotes the number of worker processes. The API, **Ninf_get_executable()**, is called to initialize the function handle to invoke the worker process. Its arguments include the same information as those for **grpc_function_handle_init()**. The API, **Ninf_wait_any()**, blocks execution of the client program until one of invoked exe-

cutables finishes its task, that is, one of worker processes becomes idle. The API, **Ninf_call_executable_async()**, is called to dispatch a subproblem to an idle worker process.

On ordinary RPC systems, all input data for a remote computing routine need to be transferred to the remote computing node whenever the remote routine is invoked. This data transfer might cause redundant communication for some applications, where input data for a remote computing routine are same for every invocation. The proposed algorithm avoids the redundant communication by re-using constant input data transferred at the first invocation. When a master process dispatches the first subproblem to a worker process, the master process transfers all input data to the worker process. At this time, the worker process stores the constant input data on the local memory. Since the second invocation, the master process does not transfer the constant data, and the worker process computes subproblems using the stored constant data.

Load balancing and advertisement of the best upper bound are performed by the supervisor process invoking remote executables using Ninf-G. The supervisor process queries statuses of master processes by invoking Ninf-G executables on computing nodes where master processes are running. The invoked executable, which is presented as the process C in Fig. 3, obtains the number of un-computed subproblems and the best upper bound by communicating with the master process via inter-process communication. Then, the executable returns results to the supervisor process. Other operations, stealing/assigning subproblems from/to master processes and advertising the updated best upper bound, are performed in the same way.

3.4 Advantage of Implementation with GridRPC

There are implementation methods other than GridRPC, e.g., MPI [18),19)], a hybrid method of GridRPC and MPI [20)], on the Grid. The motivation that the authors implemented the application program by GridRPC is that GridRPC is suitable to implement a master-worker application on the Grid.

The RPC programming model is suitable to implement an application program parallelized by the master-worker paradigm. In the implementation, a worker process is implemented as a subroutine, and a master process invokes the subroutine, or the worker process, via RPC. Parallelization with the hierarchical master-worker paradigm is also implemented by cascading RPCs. Furthermore, in the authors' original (sequential) application program, computation of a worker process, e.g., computation of lower and upper bounds, is implemented as a subroutine. Thus, the subroutine call is easily modified to the GridRPC call.

There is room for discussion about the performance issue, or performance comparison among implementation methods. However, this discussion is beyond the scope of this paper.

4. Experimental Results

The Grid testbed used in the experiment consists of four PC clusters and a client PC distributed over four cities in Japan [21)]. **Table 1** shows resources on the testbed. Four PC clusters in the testbed, **Blade**, **PrestoIII**, **Sdpa** and **Mp**, are installed in Tokyo Institute of Technology (Yokohama), Tokyo Institute of Technology (Tokyo), Tokyo Denki University (Saitama), and The University of Tokushima (Tokushima), respectively. The client PC and **Blade** are installed in the same site. The column, RTT, on the table indicates round trip time measured by the ping command between the client PC and PC clusters. The supervisor process runs on the client PC, and a set of a master process and worker processes runs on each PC cluster. Certificates for users/hosts on the testbed are issued from the AIST GTRC CA [22)].

The benchmark application in this experiment is the Bilinear Matrix Inequality Eigenvalue Problem (BMI-EP). The objective of the

Table 1 The Grid testbed.

-	spec/node	Grid software	RTT [ms]
client PC	PIII 1.0 GHz, 256 MB mem., 100BASE-T NIC	GTK 2.4, Ninf-G 2.2	
Blade	PIII 1.4 GHz x2, 512 MB mem., 100BASE-T NIC	GTK 2.4, Ninf-G 2.2	0.03
PrestoIII	Athlon 1.6 GHz x2, 768 MB mem., 100BASE-T NIC	GTK 2.4, Ninf-G 2.2	6
Sdpa	Athlon 2.0 GHz x2, 1024 MB mem., 1000BASE-T NIC	GTK 2.4, Ninf-G 2.2	12
Mp	Athlon 2.0 GHz x2, 512 MB mem., 100BASE-T NIC	GTK 2.4, Ninf-G 2.2	28

problem is to find an optimal solution, x_i and y_i, which minimizes the greatest eigenvalue of the following bilinear matrix function with given constant matrices ($F_{ij} = F_{ij}^T \in \mathcal{R}^{m\times m}$ $(i=0,\cdots,n_x,\ j=0,\cdots,n_y)$).

$$F(\mathbf{x},\mathbf{y}) := F_{00} + \sum_{i=1}^{n_x} x_i F_{i0} + \sum_{j=1}^{n_y} y_j F_{0j} + \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} x_i y_j F_{ij} \quad (2)$$

$$\text{where } F \ : \ \mathcal{R}^{n_x} \times \mathcal{R}^{n_y} \to \mathcal{R}^{m\times m}$$
$$\mathbf{x} := (x_1,\cdots,x_{n_x})^T \quad (3)$$
$$\mathbf{y} := (y_1,\cdots,y_{n_y})^T$$

The BMI-EP is recognized as a general framework for analysis and synthesis of the output feedback control systems in a variety of industrial applications, such as position control of a helicopter and control of robot arms. However, it is known that the BMI-EP is hard to solve due to the huge computational cost. Thus, speedup of the computation is expected in the control engineering community in order to enable analysis and synthesis of large scale control systems [6]. Also, in the operations research community, it is an academic grand challenge to solve the large scale BMI-EP that has never been solved [7].

4.1 Results on Grid Testbed

Figure 4 shows the execution time of five benchmark problems (P1-P5), where their problem sizes are same ($n_x = 6$, $n_y = 6$, $m = 24$) but their given constant matrices (F_{ij}) are different, on the Grid testbed. For the experiment, 412 CPUs on four PC clusters, 73 CPUs (one for the master process and 72 CPUs for worker processes) on **Blade**, 129 CPUs on **PrestoIII**, 81 CPUs on **Sdpa** and 129 CPUs on **Mp**, are employed to solve problems. In the figure, **seq** denotes sequential execution time on the single computing node of **Blade**; **cluster** means execution time on the single cluster (**Blade**), where the application is parallelized by the conventional master-worker paradigm with Ninf; finally **grid** indicates execution time on the Grid testbed, where the application is parallelized by the hierarchical master-worker paradigm with Ninf-G and Ninf. The value on the right hand side of the bar diagram indicate the execution time [sec]. **Figure 5** shows the speedup to the sequential execution time on PC clusters and the Grid testbed.

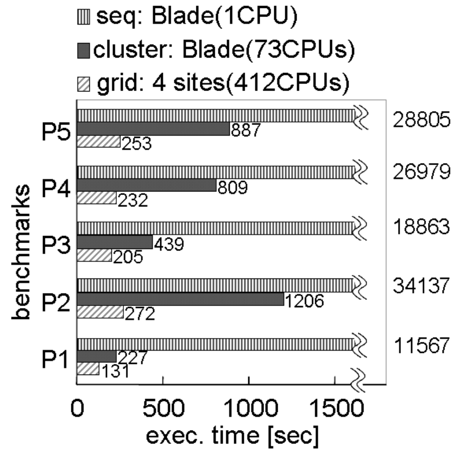

Fig. 4 The execution time on the Grid testbed.

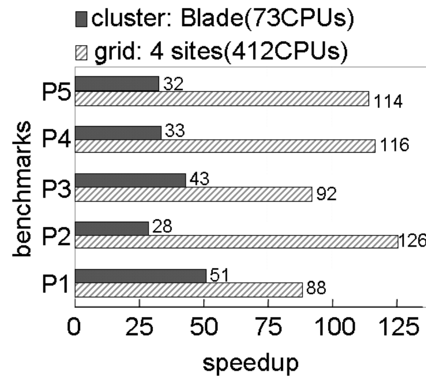

Fig. 5 The speedup to sequential execution time on the Grid testbed.

The results show that the execution time of the benchmark problems is effectively reduced by parallelization on the single PC cluster, **Blade**, compared with the sequential execution time. Also, the execution time is further reduced by employing four PC clusters distributed on the Grid testbed. The best performance is observed for the benchmark problem P2. It is solved for 4.5 minutes on the Grid testbed, while it requires nine hours and half on the single CPU.

Figure 6 shows the breakdown of the execution time for the benchmark problems on the Grid testbed. In the figure, **init**, **compt** and **fin** mean the overhead to initialize Ninf-G processes, the computation time to solve problems, and the overhead to finalize Ninf-G processes, respectively. The results in Fig. 6 indicate that the overhead to finalize Ninf-G processes significantly affects the overall performance. It might be one of reasons why the performance

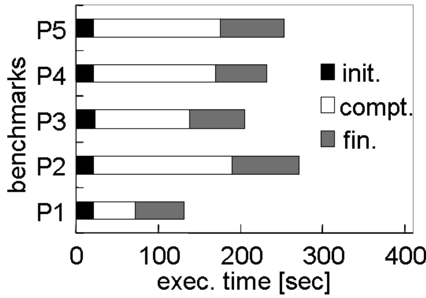

Fig. 6 The breakdown of execution time.

for P1 on the Grid testbed is not well improved compared with that on the single PC cluster. However, in the implementation, a user of the application can obtain computed results before the finalization phase. Thus, from the user's point of view, the necessary time to obtain the optimal solution is shorter than the execution time in Fig. 6, e.g., the execution time without the finalization phase is 72 [sec] for P1.

The benchmark problem solved in this experiment is a fine-grain problem. The average execution time of the single task, or computation dispatched by a master process to a worker process, is less than 1 [sec]. The conventional master-worker paradigm on the Grid might show unacceptable performance because of the overhead to dispatch fine-grain tasks via the internet. For instance, the authors' preliminary experiment using a smaller testbed shows that the execution time for P1 with the hierarchical master worker paradigm is 639 [sec] while the execution time with the conventional master worker paradigm is more than one hour.☆ The results show that the hierarchical master-worker paradigm using combination of Ninf-G and Ninf effectively utilizes computing resources on the Grid testbed in order to efficiently run the fine-grain application.

4.2 Load Balancing

The performance of the application might be affected by load balancing strategies among master processes, or PC clusters. The load balancing strategy implemented in this experiment tries to assign un-computed subproblems to master processes, or PC clusters, proportionally to their measured performance. Whenever the supervisor process finds an idle PC cluster, the supervisor process steals/assigns un-computed subproblems from/to master processes so that un-computed subproblems are distributed among master processes proportionally to performances of master processes. Here, an idle PC cluster means a PC cluster with no un-computed subproblems in the queue of the master process.

☆ Detailed discussion about the performance degradation of the application implemented with the conventional master-worker paradigm on WAN is presented in Ref. 9).

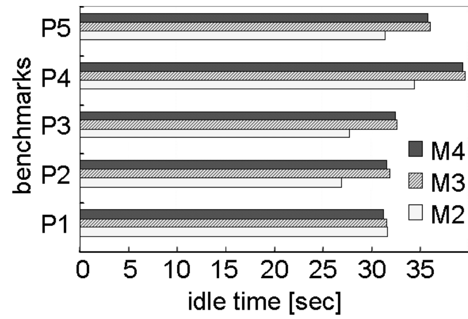

Fig. 7 The idle time on PC clusters in the first period.

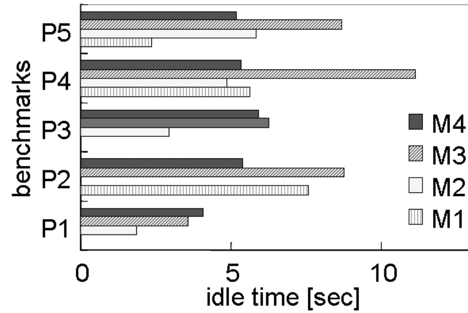

Fig. 8 The idle time on PC clusters in the second period.

Idle time on PC clusters is one of metrics to indicate performance of load balancing strategies. **Figures 7** and **8** show the idle time on PC clusters during a single application run. In the figure, **M1**, **M2**, **M3** and **M4** denote idle time on master processes on **Blade**, **PrestoIII**, **Sdpa** and **Mp**, respectively. The idle time is divided into two periods. In the first period, three PC clusters among four PC clusters are idle because of low parallelism. Tasks, or subproblems, are generated by branching during the execution; thus, only one PC cluster is busy in the first period because there are not enough tasks to make all PC clusters busy. The idle time in the first period cannot be reduced by load balancing strategies. In the second period, there are enough subproblems to make all PC cluster busy; thus, load balancing strategies, which move tasks from highly loaded PC

clusters to lightly loaded PC clusters, can reduce the idle time.

Figure 7 shows the idle time on PC clusters in the first period. When a user runs the application, the first subproblem is generated and computed on **Blade**. Thus, no idle time is observed on **M1**. Idle time is observed on three PC clusters, **M2**, **M3** and **M4**. In other words, the idle time indicated in Fig. 7 shows elapsed time that a master process waits for the first subproblem to be dispatched.

Figure 8 shows the idle time on PC clusters in the second period. No idle time is observed on some master processes, or **M1** for P1 and P3, and **M2** for P2. The results show that idle time on master processes is not much observed in the second period. For instance, ratio of the idle time to the overall computational time, which is indicated in Fig. 6, is 8% or less for all benchmark problems. It means that the load balancing strategy performs well in this experiment.

4.3 Results on Emulated Grid Testbed

The performance of the application might be affected by communication performance between the supervisor process and master processes. **Figure 9** shows execution time for P1 on the emulated Grid testbed illustrated in **Fig. 10**, where communication latency between the supervisor process and master processes is emulated from 0 [msec] through 100 [msec]. For instance, the 100 [ms] latency corresponds to the one way latency between US and Japan. The emulated Grid testbed includes four groups of computing nodes, each of which has one computing node (P4 2.4 GHz, 512 MB mem.) for a master process and four computing nodes (PIII 1.4 GHz x2, 512 MB mem.) for worker processes. Communication between the supervisor process and master processes is routed via the PC router (P4 2.4 GHz, 512 MB mem.), which emulates communication latency on wide area network by the software, NIST Net [23].

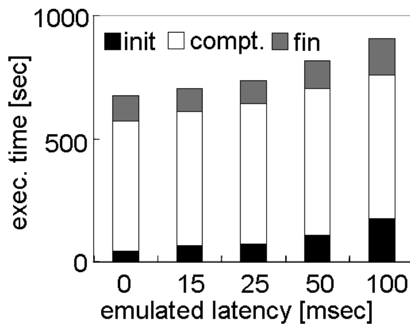

Fig. 9 Effects of communication latency.

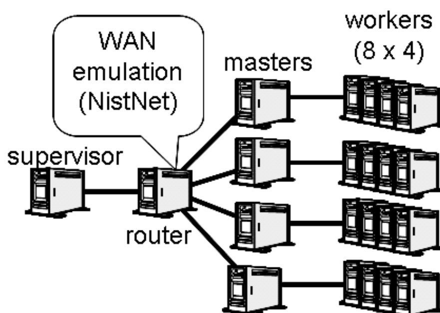

Fig. 10 The emulated Grid testbed.

The results in Fig. 9 show that performance degradation is observed when emulated latency is high. However, the performance degradation is mainly caused by increase of the overhead to initialize Ninf-G processes, and the computation time is not much affected by the latency. In the initialization phase of Ninf-G processes, a supervisor process communicates with each PC cluster to invoke multiple Ninf-G processes on the remote PC cluster. In the current implementation described in Section 3.1, the supervisor process invokes seven Ninf-G processes, one for a master process and six for relay operations, on remote PC clusters. The communication overhead is affected by latency between a node, on which the supervisor process runs, and remote PC clusters.

The results indicate that the hierarchical master-worker paradigm with GridRPC works efficiently enough on the Grid testbed with high communication latency.

4.4 Task Granularity Control

Figure 11 illustrates the execution time of the application on **Blade** (33 CPUs), where the task granularity control is performed. Here, the application is parallelized by the master-

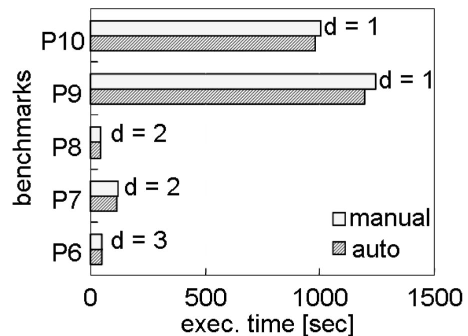

Fig. 11 The performance of the task granularity control.

worker paradigm with Ninf. Different benchmark problems, P6-P10, are selected to see performance of the task granularity control for benchmarks with different problem sizes. The problem size of P6 is $n_x = 10$, $n_y = 2$, $m = 8$; the size of P7-P8 is $n_x = 5$, $n_y = 5$, $m = 20$; and the size of P9-P10 is $n_x = 6$, $n_y = 6$, $m = 24$. Particularly, the size of m defines computational complexity of a subproblem in the BMI-EP. In the figure, **manual** means execution time where the task granularity is manually adjusted. Here, the task granularity, or the depth of the sub-tree generated on a worker process, is set by the authors before the application run, and the granularity is fixed during the application run; i.e., the authors ran the application multiple times using different setting of the task granularity, and chose the setting with the best performance. The symbol, **d**, denotes the depth of the sub-tree in the best setting. On the other hand, **auto** indicates the execution time where the task granularity is automatically adjusted by the adaptive task granularity control presented in Section 2.3. The constant parameter, a in the formula (1), is empirically defined and is set to 0.5 in the experiment, and the maximum task granularity is set to $depth = 5$.

The results show that the adaptive task granularity control appropriately adjusts the task granularity during the application run and exhibits the same or better performance compared to the manual control. Note that the objective of the task granularity control is to run the application without tuning of task granularity setting. The manual control requires multiple times of preliminary application runs to investigate the best setting.

Furthermore, if task granularity is inappropriately set, it could degrade performance. **Figure 12** shows execution time for P6, where task granularity is manually set to *depth*. Figure 12 indicates that the execution time is smallest where $depth = 3$, and the execution time increases when *depth* increases/decreases. The results show that too small/large task granularity significantly degrades performance of the application. For instance, the execution time increases by 50% when $depth = 1$ compared to the best setting, and the execution time increases by 30% when $depth = 5$. The proposed adaptive task granularity control algorithm automatically adjusts the task granularity, or *depth*, so that it gives the best performance without tedious preliminary runs.

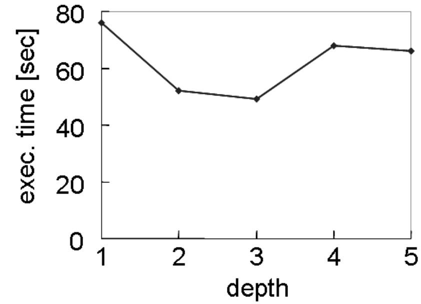

Fig. 12 Effects of task granularity.

The parameter, a, also affects the performance. In the formula (1), ΔZ exhibits the maximum gap between the new best upper bound and previous one in most cases. Thus, if a is set to 1, task granularity increases to the maximum ($depth = 5$) and preserves the maximum granularity. For instance, the performance might be close to the performance where task granularity manually set to $depth = 5$ in Fig. 12. On the other hand, if a is set to 0, task granularity decreases to the minimum ($depth = 1$) and preserves the minimum granularity; that is the performance might be close to the performance where task granularity manually set to $depth = 1$ in Fig. 12.

In the experiments, the authors set $a = 0.5$ to avoid significant performance degradation as shown in Fig. 12. The preliminary experiments show that this setting, $a = 0.5$, gives acceptable performance. However, there is room for further investigation for the value for a, and the development of the method to define the optimal a is the authors' future work.

4.5 User Interface

The authors developed the user interface to run the branch and bound application using the proposed algorithm. A user of the application can operate through the web interface as illustrated in **Fig. 13** and can observe interim results of the computation. The upper window on the interface depicts the convergence of lower and upper bounds currently computed on the Grid, and the lower window shows the number of un-computed subproblems. The interim information is useful for the user to find the best parameter for the user's problem. The user can restart the computation with other parameters through the web, if he/she finds unsatisfactory behavior in the interim information.

5. Related Work

Fine-grain applications on distributed sys-

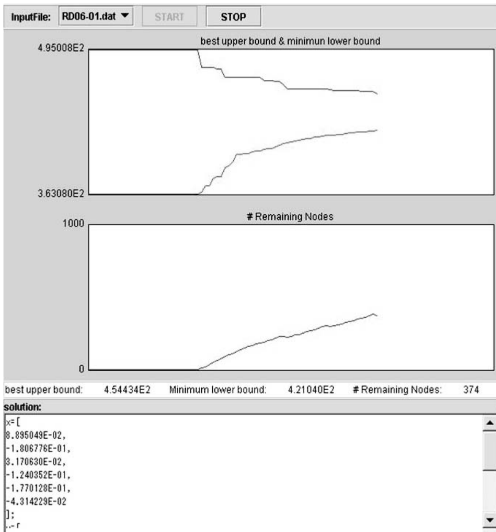

Fig. 13 An example of the user interface.

tems have been discussed in literatures [15),16)]. The work presented in Ref. 15) discusses performance of applications on multiple PC clusters connected via slow network. The experimental results show an impact on performance by a gap between fast network and slow network for six benchmark applications. The work also discusses optimization techniques, which includes communication in a hierarchical manner, to improve the performance. Furthermore, the experiment for fine-grain divide-and-conquer applications on the Grid is reported in Ref. 16). It shows the performance of the divide-and-conquer Java applications, which is parallelized in a hierarchical manner, on Satin/Ibis, Java based Grid programming environment.

The work presented in Ref. 24) discusses load balancing strategies on distributed systems, where applications are parallelized in a hierarchical manner. The work reports experimental results for various load balancing strategies on multiple PC clusters with simulated WAN setting. The idea behind the load balancing strategy in the hierarchical master-worker paradigm, which is presented in this paper, is similar to that of CLS [24)] in the view that load balancing is performed in a hierarchical way via designated nodes on PC clusters.

The work presented in this paper is extended from the work in Refs. 9), 21). Reference 9) proposes the idea of the hierarchical master-worker paradigm, and Ref. 21) presents the implementation and the experimental results on the Grid. The proposed algorithm in this paper enhances the original algorithm [9)] by adding the new idea of task granularity control. This paper also presents new experimental results on the larger Grid testbed.

6. Conclusions

This paper proposed a parallel branch and bound algorithm that efficiently ran on the Grid. The proposed algorithm is parallelized with the hierarchical master-worker paradigm in order to efficiently compute fine-grain tasks on the Grid. The hierarchical approach effectively reduces communication overhead on WAN by localizing frequent communication in tightly coupled computing resources, or a PC cluster. On each PC cluster, granularity of tasks is adaptively adjusted to obtain the best performance.

The application is implemented on the Grid testbed by using two GridRPC middleware, Ninf-G and Ninf, where secure communication among PC clusters is performed via Ninf-G and fast communication among computing nodes in each PC cluster is performed via Ninf. The experimental results showed that the implementation with the hierarchical master-worker paradigm using combination of Ninf-G and Ninf effectively utilized computing resources on the Grid testbed in order to efficiently run the fine-grain application, where the average computation time of the single task was less than 1 [sec]. The results also showed that the adaptive task granularity control automatically gave the same or better performance compared with manual control that required preliminary application runs.

There is room to improve the load balancing strategy for the application. Experiments on the actual testbed are not suitable for comparison of multiple strategies, because the testbed does not exhibit reproducible results. The authors plan to perform experiments to compare various load balancing strategies, including the conventional load balancing strategies proposed in the distributed computing community, on the emulated Grid testbed. Also, the current emulation model in the emulated Grid testbed is too simple to emulate realistic behavior of the internet. The development of the more sophisticated Grid emulation model is the future work. There is also room for further investigation for the constant parameter a in the adaptive task granularity control. Further investigation and development of the method to define an optimal

a is the authors' future work.

Acknowledgments The authors would like to thank members of the Ninf project for their insightful comments. This research is partially supported by Research and Development for Applying Advanced Computational Science and Technology (ACT-JST), Japan Science and Technology Agency.

References

1) Goux, J., Kulkarni, S., Linderoth, J. and Yoder, M.: An enabling framework for master-worker applications on the computational Grid, *Proc. 9th IEEE Symposium on High Performance Distributed Computing* (*HPDC9*) (2000).
2) Heymann, E., Senar, M.A., Luque, E. and Livny, M.: Adaptive scheduling for master-worker applications on the computational Grid, *Proc. 1st IEEE/ACM International Workshop on Grid Computing* (*Grid2000*) (2000).
3) Neary, M.O. and Cappello, P.: Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing, *Proc. 2002 joint ACM-ISCOPE conference on Java Grande* (2002).
4) Takemiya, H., Shudo, K., Tanaka, Y. and Sekiguchi, S.: Development of Grid applications on standard Grid middleware, *Proc. GGF8 Workshop on Grid Applications and Programming Tools* (2003).
5) Horst, R., Pardalos, P.M. and Thoai, N.V. (Eds.): *Introduction to Global Optimization*, Kluwer Acadimic Publishers (1995).
6) Goh, K.C., Safonov, M.G. and Papavassilopoulos, G.P.: A global optimization approach for the BMI problem, *Proc. 33rd IEEE Conrerence on Decision and Control*, pp.2009–2014 (1994).
7) Fukuda, M. and Kojima, M.: Branch-and-cut algorithms for the bilinear matrix inequality eigenvalue problem, *Computational Optimization and Applications*, Vol.19, No.1, pp.79–105 (2001).
8) Kasahara, H. and Narita, S.: Practical multiprocessor scheduling algorithms for efficient parallel processing, *IEEE Trans. Comput.*, Vol.C-33, No.11, pp.1023–1029 (1984).
9) Aida, K., Natsume, W. and Futakata, Y.: Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm, *Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid* (*CCGrid 2003*) (2003).
10) Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C. and Casanova, H.: Overview of GridRPC: A remote procedure call API for Grid computing, *Proc. Grid Computing — Grid 2002*, *LNCS2536*, pp.274–278 (2002).
11) Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T. and Matsuoka, S.: Ninf-G: A reference implementation of RPC-based programming middleware for Grid computing, *J. of Grid Computing*, Vol.1, No.1, pp.41–51 (2003).
12) Matsuoka, S., Nakada, H., Sato, M. and Sekiguchi, S.: Design issues of network enabled server systems for the Grid, *Proc. Grid Computing — Grid 2000*, *LNCS1971*, pp.4–17 (2000).
13) Nakada, H., Matsuoka, S., Seymour, K., Dongarra, J., Lee, C. and Casanova, H.: A GridRPC model and API for end-user applications, *GGF Document*, GFD-R.052 (2005).
14) Foster, I. and Kesselman, C.: Globus: A metacomputing infrastructure toolkit, *Int. J. of Supercomputing Applications*, Vol.11, No.2, pp.115–128 (1997).
15) Plaat, A., Bal, H.E. and Hofman, R.F.: Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects, *Porc. High Performance Computer Architecture* (*HPCA-5*), pp.244–253 (1999).
16) van Nieuwpoort, R., Massen, J., Kielmann, T. and Bal, H.E.: Satin: Simple and efficient java-based Grid programming, *Proc. Workshop on Adaptive Grid Middleware* (*AGridM 2003*) (2003).
17) Tanaka, Y., Sato, M., Hirano, M., Nakada, H. and Sekiguchi, S.: Performance evaluation of a firewall compliant Globus-based wide-area cluster system, *Proc. 9th IEEE Symposium on High-Performance Distributed Computing* (2000).
18) GridMPI. http://www.gridmpi.org/
19) Karonis, N., Toonen, B. and Foster, I.: MPICH-G2: A Grid-enabled implementation of the message passing interface, *Journal of Parallel and Distributed Computing*, Vol.63, No.5, pp.551–563 (2003).
20) Takemiya, H., Tanaka, Y., Nakada, H. and Sekiguchi, S.: Development and execution of large scale Grid applications using MPI and GridRPC: Hybrid QM/MD simulation (in japanese), *IPSJ Transaction on Advanced Computing Systems*, Vol.46 (SIG12), pp.384–395 (2005).
21) Aida, K. and Osumi, T.: A case study in running a parallel branch and bound application on the Grid, *Proc. IEEE/IPSJ The 2005 Symposium on Applications & the Internet* (*SAINT2005*), pp.64–173 (2005).
22) AIST GRID CA. https://www.apgrid.org/ca/aist/production/index.html
23) NIST Net. http://snad.ncsl.nist.gov/nistnet/
24) van Nieuwpoort, R.V., Kelmann, T. and Bal,

H.E.: Efficient load balancing for wide-area divide-and-conquer applications, *Proc. eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming PPoPP'01*, pp.34–43 (2001).

(Received January 27, 2006)
(Accepted May 3, 2006)

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, a research scientist at the Department of Mathematical and Computing Sciences, Tokyo Institute of Technology in 1997, and an assistant professor at the Department of Computational Intelligence and Systems Science, Tokyo Institute of Technology in 1999, respectively. He is now an associate professor at the Department of Information Processing, Tokyo Institute of Technology from 2003. His research interests are parallel computing, Grid computing, scheduling and internet applications. He is a member of IEICE, IEEJ, ACM and IEEE-CS.

Yoshiaki Futakata received his B.E. and M.E. degrees from Tokyo Institute of Technology in 1999 and 2001, respectively. He joined IBM Japan in 2001. He is now a Ph.D. student at University of Virginia.

Tomotaka Osumi received his B.E. and M.E. degrees from Tokyo Institute of Technology in 2004 and 2006, respectively. He joined NTT Communications Corporation in 2006.