A Case Study in Running a Parallel Branch and Bound Application on the Grid

Kento Aida
Tokyo Institute of Technology/PRESTO, JST
aida@alab.ip.titech.ac.jp

Tomotaka Osumi
Tokyo Institute of Technology
osumi@alab.ip.titech.ac.jp

Abstract

This paper presents a case study to effectively run a parallel branch and bound application on the Grid. The application discussed in this paper is a fine-grain application and is parallelized with the hierarchical master-worker paradigm. This 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. This hierarchical manner reduces communication overhead by localizing frequent communication in tightly coupled computing resources, or a single PC cluster. The algorithm is implemented on a Grid testbed by using GridRPC middleware, Ninf-G and Ninf. In the implementation, communication among PC clusters is securely performed via Ninf-G, which uses Grid security service on the Globus Toolkit, and communication among computing nodes in each PC cluster is performed via Ninf, which enables fast invocation of remote computing routines. The experimental results showed that implementation of the application with the hierarchical master-worker paradigm using a combination of Ninf-G and Ninf effectively utilized computing resources on the Grid testbed in order to run the fine-grain application, where the average computation time of the single task was less than 1[sec].

1. Introduction

Grid computing is regarded as new computing technology that provides huge computational power with low costs by employing computing resources geographically distributed over the internet. 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. Performances of applications that consist of small tasks are 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][2][3][4]. For instance, the work presented in [1] shows experimental results for an application, which solves the quadratic assignment problem, on a Grid testbed; and the mean task grain size, or the mean execution time of the single task, in the application is 190 [sec]. The work in [3] also presents experimental results for an application, which solves the traveling salesman problem, on a Grid testbed; and the mean task grain sizes are 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, such as computational resources distributed on the Grid. Thus, implementation to effectively run these fine-grain applications on the Grid contributes for expanding applications of Grid computing.

This paper presents a case study to effectively run a parallel branch and bound application on the Grid. Branch and bound applications are widely used to solve optimization problems in many engineering fields, e.g. operations research, control theory,

multiprocessor scheduling [5][6][7][8]. However, many of these applications tend to be composed of a huge number of fine-grain tasks, i.e. they are fine-grain applications. The application presented in this paper is parallelized with the hierarchical master-worker paradigm [9] in order to efficiently compute fine-grain tasks on the Grid. This 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. This hierarchical manner avoids performance degradation, which is mainly caused by communication overhead between the master process and worker processes, by localizing frequent communication in tightly coupled computing resources, or a single PC cluster. The application is implemented on the Grid 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 proposed to the GGF for standardization [13]. In the implementation, communication among PC clusters is securely performed via Ninf-G, which uses the Grid security service 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 the fine-grain parallel branch and bound application with GridRPC on the Grid has not been sufficiently discussed. The contribution of this paper is to present implementation and detailed performance of the application on a Grid testbed constructed with standard Grid technology [13][14]. The experimental results showed that the implementation of the application with the hierarchical master-worker paradigm using combination of Ninf-G and Ninf effectively utilized computing resources on the Grid testbed in order to run the fine-grain application, where the average computation time of the single task was less than 1[sec].

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

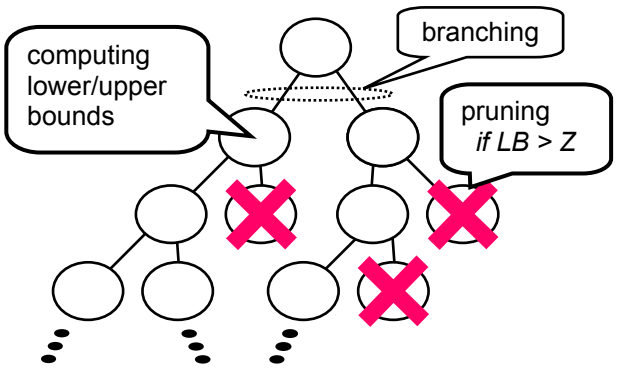

Figure 1. An example of the search tree

2. Target Application

This section summarizes an overview of the parallel branch and bound algorithm and parallelization of the application with the hierarchical master-worker paradigm.

2.1. Branch and Bound Algorithm

The main idea of the 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/upper bounds of the objective function. These lower/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 on Figure 1. On 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 nodes of the root node. This partitioning process is called *branching*. After the branching, lower/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 [1]. By continuing in this way, a tree structure called *the 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

[1] This paper assumes an optimization problem that minimizes the objective function.

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 less than the certain interval.

2.2. Parallelization with Hierarchical Master-Worker Paradigm

The 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][18]. Also, the parallel algorithm with the hierarchical master-worker paradigm is proposed to improve performance on large-scale computing environment [9].

The hierarchical master-worker paradigm is one of solutions to avoid performance degradation in the 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. The 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. The 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 the master process and worker processes, which frequently communicate with each other, on tightly coupled computing resources. The second advantage is to avoid that a single heavily loaded master process becomes a performance bottleneck by distributing work among multiple master processes.

The parallel 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 algorithm 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 updates the best upper bound of the objective function by communicating with master processes. Updating of the best upper bound is crucial to improve the performance of the application, because it accelerates

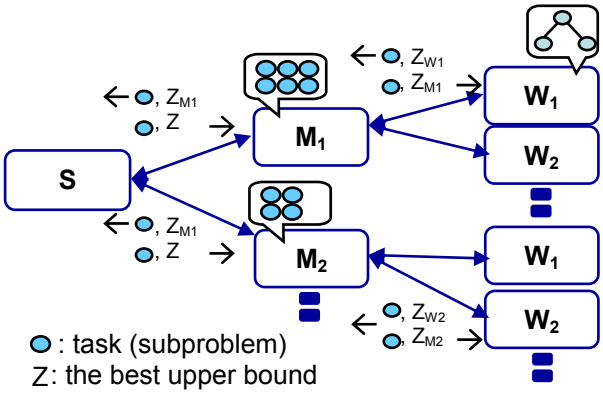

Figure 2. B&B algorithm with the hierarchical master-worker paradigm

pruning. Figure 2 shows an overview of the branch and bound (B&B) algorithm with the hierarchical master-worker paradigm. Symbols on the figure, Z_{Wi}, Z_{Mj} and Z, denote the current upper bound of the objective function stored on the worker process $\mathbf{W_i}$, the master process $\mathbf{M_j}$ and the supervisor process, respectively.

In each set of the 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 process. 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. Next, it computes the lower/upper bounds for each subproblem and performs pruning, 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 the upper bound computed on the worker process, the solution, and subproblems that have generated by branching and have not been pruned on the worker process.

The supervisor process periodically queries master processes about their statuses, which include the number of un-computed subproblems and the best 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. A strategy for the load balancing is

discussed in Section 4. When the supervisor process finds the new best upper bound on the master process M_i, where $Z_{Mi} < Z$, the supervisor process updates the best upper bound stored on the supervisor process (Z) and distributes Z to other master processes. Thus, the master process communicates both with its worker processes and with the supervisor processes. Finally, the supervisor process terminates computation if the termination condition is satisfied.

3. Implementation

The Grid testbed assumed 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 the application described in the previous section on the Grid testbed, mapping of processes on computing resources and communication methods among these processes are crucial. Particularly, implementation to reduce overhead is necessary to run the fine-grain application on the Grid testbed, because the performance of the 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. On the figure, multiple PC clusters, which are depicted by squares with dotted lines, are distributed on the internet. Symbols on the figure, **S**, **M** and **W** denote the supervisor process, the master process and the 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. It relays operations between the supervisor process and the master process. These relayed operations consist of queries about statuses of master processes, stealing/assigning subproblems from/to master processes and distributing the new best upper bound. As described in Section 2.2, the 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 dedicated high-speed network. This mapping is effective to reduce communication

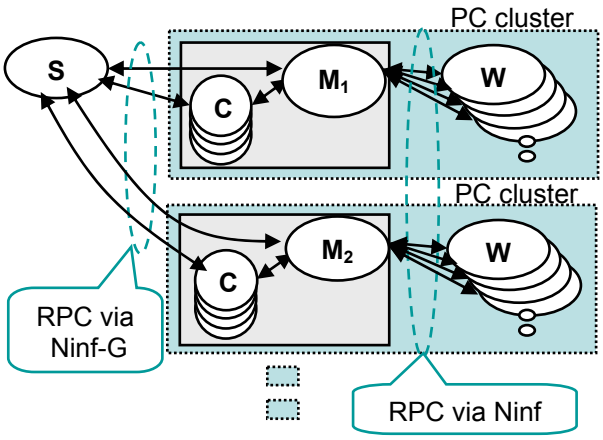

Figure 3. Process mapping

overhead in the application, because the amount of data transferred between the supervisor process and master processes is much smaller than that between the master process and worker processes. The discussion for the amount of the transferred data is presented in Section 4. 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 the 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 the master process is performed by Grid RPC middleware Ninf-G [11], which uses the Grid security service on the Globus Toolkit [14]. Also, communication between the 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 get 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 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 at the execution. Next, it initiates the master process on the designate node for each PC cluster using Ninf-G. An example of program codes 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 a function handle to invoke a 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.

The master process initiates worker processes on computing nodes in the same PC cluster and dispatches subproblems to idle worker processes using Ninf. An 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 a 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 a client program until one of invoked executables 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 the 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 the remote computing routine are same for every invocation. The application presented in this paper avoids the redundant communication by re-using constant input data transferred at the first invocation. When the master process dispatches the first subproblem on the worker process, the master process transfers all input data to the worker process. At this time, the worker process saves 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 saved constant data.

Load balancing and updating 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** on Figure 3, obtains the number of un-computed subproblems and the 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 distributing the updated best upper bound, are performed in the same way.

Table 1. The Grid testbed

	specification of a single node	Grid software	RTT [ms]
Client PC	PIII 1.0GHz, 256MB mem. 100BASE-T NIC	GTK 2.2 Ninf-G 1.1.1	
Blade	PIII 1.4GHz x2 512MB mem. 100BASE-T NIC	GTK 2.2 Ninf-G 1.1.1	0.04
Presto III	Athlon 1.6GHz x2, 768MB mem. 100BASE-T NIC	GTK 2.4 Ninf-G 1.1.1	1
Mp	Athlon 1.6GHz x2 512MB mem. 100BASE-T NIC	GTK 2.4 Ninf-G 1.1.1	20
Sdpa	Athlon 2GHz x2, 1024MB mem. 1000BASE-T NIC	GTK 2.4 Ninf-G 1.1.1	14

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. Table 1 shows resources on the testbed. Four PC clusters, **Blade**, **PrestoIII**, **Mp** and **Sdpa**, are installed in different four sites. The client PC and **Blade** are installed in the same site. Distances from the site for the client PC and sites for other PC clusters, **PrestoIII**, **Mp** and **Sdpa**, are 30[km], 500[km] and 50[km], respectively. 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 the master process and worker processes runs on each PC cluster. Certificates for users/hosts on the testbed are issued from the AIST GTRC CA[17].

The benchmark problem solved by the application in this experiment is the Bilinear Matrix Inequality Eigenvalue Problem (BMI-EP). The objective of the problem is to find an optimal solution that minimizes the greatest eigenvalue of the following bilinear matrix function with given constant matrices (F_{ij}).

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

$$F_{ij} = F_{ij}^T,\ i = 0,\ldots,n_x,\ j = 0,\ldots,n_y$$
$$x = (x_1,\ldots,x_{n_x})^T,\ y = (y_1,\ldots,y_{n_y})^T$$

The BMI-EP is recognized as a general framework for analysis and synthesis of control systems in variety of industrial applications, such as position control of a helicopter and control of robot arms. Thus, speedup of the computation is expected in the control theory 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 problem that has never been solved [7]. The problem size of the BMI-EP is defined by the size of optimal solution, n_x and n_y, and the size of F_{ij}, m^2.

Sizes of transferred data in the application to solve the BMI-EP are calculated by its problem size. The master process transfers the following sizes of data to dispatch the first subproblem to each worker process, or each computing node in the PC cluster:

$$D_{in1} = 4(m^2 + m)(n_x + n_y + n_x n_y + 1) + 28n_x + 28n_y + 28n_x n_y + 136$$
$$D_{out1} = N(24n_x + 24n_y + 16) + 8n_x + 8n_y + 16.$$

Here, D_{in1} and D_{out1} denote sizes [Bytes] of data transferred from the master process to the worker process and those transferred in the reverse way, respectively; N means the number of subproblems returned to the master process.

The supervisor process transfers initial input data once to each master process when the application starts. The size of the transferred data is almost same as D_{in1}. After the transfer of the initial data, the amount of transferred data between the supervisor process and master processes is small. For instance, when the supervisor process queries about the status of the master process, sizes of transferred data are:

$$D_{in2} = 8$$
$$D_{out2} = 8n_x + 8n_y + 32.$$

Here, D_{in2} and D_{out2} denote sizes [Bytes] of data transferred from the supervisor process to the master process and those transferred in the reverse way, respectively. Also, when the supervisor process steals un-computed subproblems from the master process, the following sizes of data are transferred:

$$D_{in2} = 12$$
$$D_{out2} = N(24n_x + 24n_y + 16) + 4.$$

For the BMI-EP with the size of $n_x = 6$, $n_y = 6$, $m = 24$, the master process transfers 120[KB] of data to each worker process when it dispatches the first subproblem. It is obvious that the amount of data transferred between the supervisor process and master processes is much smaller than that between the master process and worker processes.

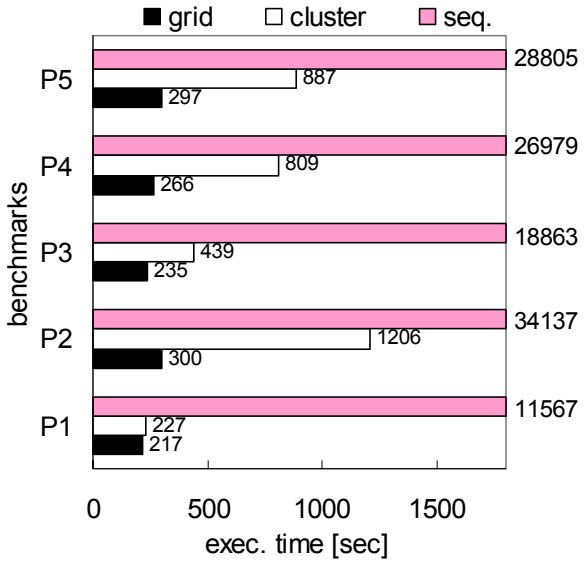

Figure 4. Execution time on the Grid testbed

4.1. Results on Grid Testbed

Figure 4 shows 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, 348 CPUs over four sites, 73 CPUs (one for the master process and 72 CPUs for worker processes) on **Blade**, 97 CPUs on **PrestoIII**, 81 CPUs on **Sdpa** and 97 CPUs on **Mp**, are employed to solve problems. On 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. Values on the right hand side of bar diagrams indicate the digitized execution time [sec].

The results show that 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 5 minutes on the Grid testbed, while it requires nine hours and half on the single CPU. Also, the execution for **P2** on the Grid testbed is four times faster than that on the single PC cluster.

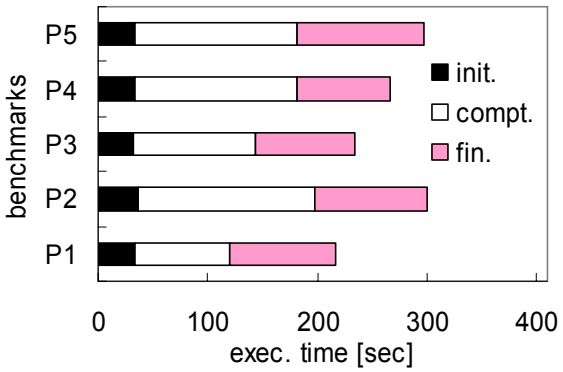

Figure 5. The breakdown of execution time

Figure 5 shows the breakdown of execution time for the benchmark problems on the Grid testbed. On the figure, **init**, **compt** and **fin** mean overhead to initialize Ninf-G processes, computation time to solve problems, and overhead to finalize Ninf-G processes, respectively. The results on the Figure 5 indicate that overhead to finalize Ninf-G processes significantly affects the overall performance. It might be one of reasons why the performance for **P1** on the Grid testbed is not well improved compared with that on the single PC cluster. However, in the implementation, computed results are obtained before the finalization phase of the application. Thus, from the user's point of view, the user can obtain an optimal solution within shorter time than the execution time on Figure 4, e.g. execution time without the finalization phase is 120[sec] for **P1**; that is, the performance on the Grid testbed is well improved.

The benchmark problem solved in this experiment is a fine-grain problem. The average execution time of the single task, or computation dispatched by the master process to the worker process, is less than 1[sec]. Thus, it is obvious that the implementation with the conventional master-worker paradigm on the Grid shows unacceptable performance because of overhead to dispatch fine-grain tasks via the internet[2]. However, the results show that the implementation with the hierarchical master-worker paradigm using a combination of Ninf-G and Ninf effectively utilizes computing resources on the Grid testbed in order to run the fine-grain application.

[2] The performance degradation of the application implemented with the conventional master-worker paradigm on WAN is discussed in [9].

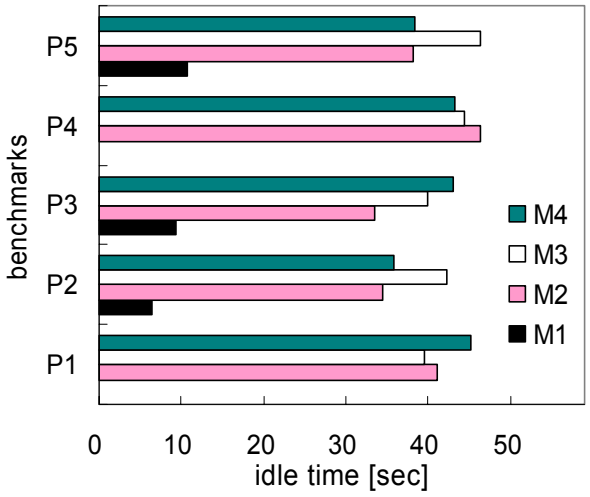

Figure 6. Idle time on master processes

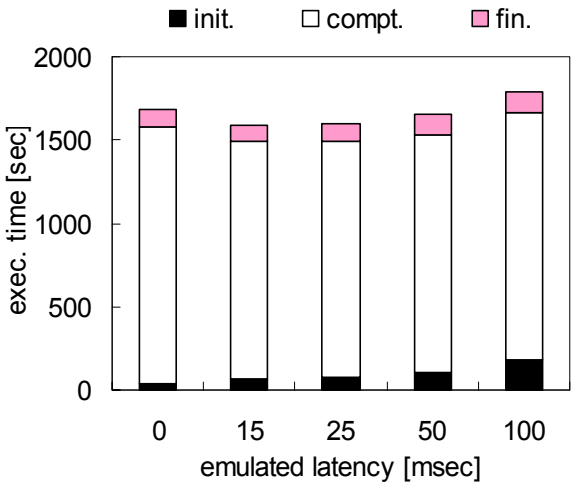

Figure 7. Execution time on the emulated Grid testbed

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 [3], the supervisor process steals/assigns un-computed subproblems from/to master processes so that the number of un-computed subproblems on master processes, $N_{task(i)}$, becomes as follows:

$$N_{task(i)} = N_{task} \left(T_{task(i)} N_{workers(i)} / \sum_{j} (T_{task(j)} N_{workers(j)}) \right) ,$$

where N_{task}, $T_{task(i)}$ and $N_{workers(i)}$ mean the number of un-computed tasks, the average task execution time measured on the PC cluster *i* during the execution and the number of worker processes running on the PC cluster *i*, respectively.

Figure 6 shows idle time on master processes, or PC clusters, during execution of the application. On the figure, **M1**, **M2**, **M3** and **M4** denote idle time on master processes on **Blade**, **PrestoIII**, **Sdpa** and **Mp**, respectively. The results on the figure show that idle time of approximately 30-45[sec] is observed on PC clusters except **Blade**. The detailed analysis of the results shows that most of the idle time is observed during the initial phase of the execution, where there are not enough subproblems to utilize multiple PC clusters. Note that tasks, or subproblems, are generated by branching during the execution, and there are not enough tasks to make all PC clusters busy during the initialization phase. Thus, idle time on master processes is not much observed after the initialization phase, and it means that the load balancing strategy performs well in this experiment.

[3] An idle PC cluster means that with no un-computed subproblems in the queue of the master process.

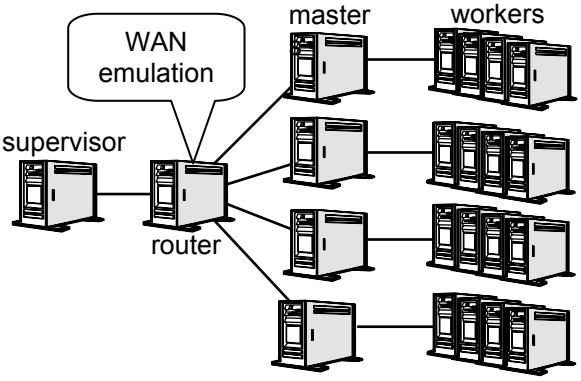

Figure 8. The emulated Grid testbed

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 7 shows execution time for **P2** on the emulated Grid

testbed illustrated on Figure 8, 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 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 2.4GHz, 512MB mem.) for the master process and four computing nodes (PIII 1.4GHz x2, 512MB mem.) for worker processes. Communication between the supervisor process and master processes is performed via the PC router (P4 2.4GHz, 512MB mem.), which emulates communication latency on wide area network by the software, NIST Net [19].

The results on Figure 7 show that performance degradation of the application is small even under high communication latency. The performance degradation is mainly caused by increase of overhead to initialize Ninf-G processes. The results mean that the application implemented with the hierarchical master-worker paradigm using GridRPC is robust even on Grid environment with high communication latency.

4.4. User Interface

A user of the application can operate through the web interface as illustrated on Figure 9 and can observe interim results of the computation. The upper window on the interface depicts the best upper/lower 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.

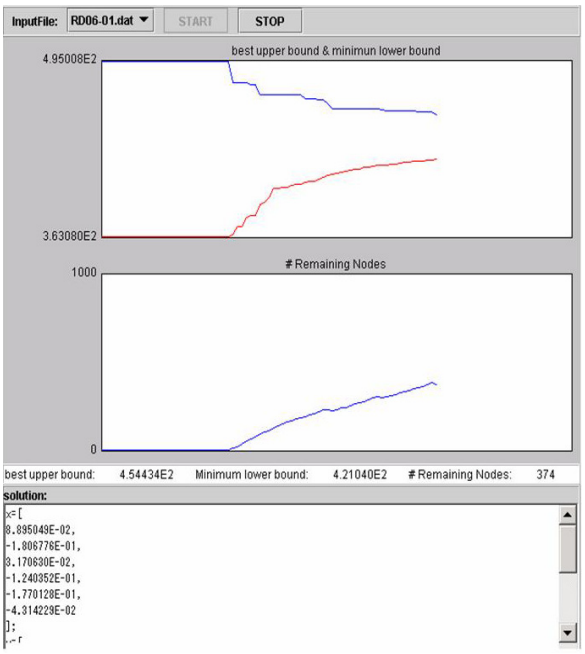

Figure 9. The user interface

5. Related Work

Fine-grain applications on distributed systems have been discussed in literatures [15][16]. The work presented in [15] discusses performance of applications on multiple PC clusters connected via slow network. The experimental results show an impact on performance by 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. The experiment for fine-grain divide-and-conquer applications on the Grid is reported in [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 parallel branch and bound algorithm with the hierarchical master-worker paradigm is proposed in [5], and the preliminary evaluation is presented. However, detailed performance of the fine-grain parallel branch and bound application on the Grid constructed with standard Grid technology, which this paper presents, has not been reported.

The work presented in [20] discusses load balancing strategies on distributed systems, where applications are parallelized in a hierarchical manner. The work reports evaluation 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** [20] in the view that load balancing is performed in a hierarchical way via designated nodes on PC clusters.

6. Conclusions

This paper presented a case study to effectively run a parallel branch and bound application on the Grid. The application discussed in this paper is a fine-grain application, and is parallelized with the hierarchical master-worker paradigm, where communication overhead on WAN is effectively reduced by localizing frequent communication in tightly coupled computing

resources, or a PC cluster. 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 implementation of the application with the hierarchical master-worker paradigm using a combination of Ninf-G and Ninf effectively utilized computing resources on the Grid testbed in order to run the fine-grain application, where the average computation time of the single task was less than 1[sec].

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.

Acknowledgements: 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] J. Goux, S. Kulkarni, J. Linderoth, and M. Yoder, An enabling framework for master-worker applications on the computational grid, In *Proc. the 9th IEEE Symposium on High Performance Distributed Computing (HPDC9)*, 2000.

[2] E. Heymann, M. A. Senar, E. Luque, and M. Livny, Adaptive scheduling for master-worker applications on the computational grid, *Proc. of the 1st IEEE/ACM International Workshop on Grid Computing (Grid2000)*, 2000.

[3] M. O. Neary and P. Cappello, Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing, *Proc. of the 2002 joint ACM-ISCOPE conference on Java Grande*, 2002.

[4] H. Takemiya, K. Shudo, Y. Tanaka and S. Sekiguchi. Development of Grid Applications on Standard Grid Middleware, *Porc. of the GGF8 Workshop on Grid Applications and Programming Tools*, 2003.

[5] R. Horst, P. M. Pardalos and N. V. Thoai, *Introduction to Global Optimization*, Kluwer Academic Publishers, 1995.

[6] K. C. Goh, M. G. Safonov and G. P. Papavassilopoulos, A Global Optimization Approach for the BMI Problem, *Proc. of the 3rd Conference on Decision and Control*, pp.2009-2014, 1994.

[7] M. Fukuda and M. Kojima, Branch-and-Cut Algorithms for the Bilinear Matrix Inequality Eigenvalue Problem, *Computational Optimization and Applications*, 19(1):79-105, 2001.

[8] H. Kasahara and S. Narita, Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing, *IEEE Trans. on Computers,* C-33(11), pp.1023-1029, 1984.

[9] K. Aida, W. Natsume and Y. Futakata, Distributed Computing with Hierarchical Master-worker Paradigm for Parallel Branch and Bound Algorithm, *Proc. of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003.

[10] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C Lee and H. Casanova, Overview of GridRPC: A Remote Procedure Call API for Grid Computing, *Grid Computing – Grid 2002, LNCS2536*, pp.274-278, 2002.

[11] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura and S. Matsuoka, Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing, *J. of Grid Computing*, 1(1):41-51, 2003.

[12] S. Matsuoka, H. Nakada, M. Sato and S. Sekiguchi, Design Issues of Network Enabled Server Systems for the Grid, *Grid Computing – Grid 2000, LNCS1971*, pp.4-17, 2000.

[13] Global Grid Forum, *http://www.ggf.org/*

[14] I. Foster and C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, *Int. J. of Supercomputing Applications*, 11(2):115-128, 1997.

[15] Plaat, H. E. Bal and R. F. Hofman, Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects, *Porc. of High Performance Computer Architecture (HPCA-5)*, pp. 244-253, 1999.

[16] R. van Nieuwpoort, J. Massen, T. Kielmann and H. E. Bal, Satin: Simple and Efficient Java-based Grid Programming, Proc. *Workshop on Adaptive Grid Middleware (AGridM 2003)*, 2003.

[17] ApGrid, *http://www.apgrod.org/*

[18] Y. Tanaka, M. Sato, M. Hirano, H. Nakada, and S. Sekiguchi, Performance evaluation of a firewall compliant globus-based wide-area cluster system, *Proc. of 9th IEEE Symposium on High-Performance Distributed Computing*, 2000.

[19] NIST Net, *http://snad.ncsl.nist.gov/nistnet/*

[20] R. van Nieuwpoort, T. Kielmann and H. E. Bal, Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications, *Porc. the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, 2001.