Fast Performance Prediction of Master-Slave Programs by Partial Task Execution

Yasuharu Mizutani Fumihiko Ino Kenichi Hagihara

Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, JAPAN

Abstract: In this paper, it is proposed to rapidly and accurately predict performance of master-slave (MS) parallel programs. To provide rapid prediction with high accuracy, our method reduces direct execution of the target MS program and estimates execution time of tasks of the program from only some directly executed tasks. In this estimation, we use a linear interpolation in order to reproduce the original order of task assignment, which affects the prediction accuracy of a performance saturation point. The experimental result shows that our proposed method predicts the performance of MS programs 1.7 times faster, at least, than the measured execution time which corresponds to the minimum time taken to predict the performance by prediction methods based on direct execution. Furthermore, our method predicts the performance with 7% error being as good as that of existing prediction method.

Key-Words: Master-slave, Performance prediction, Parallel computational model, Simulation

1 Introduction

With the rapid advances in cluster and grid technologies, parallel computing environments are increasing the heterogeneity of processors and interconnects. One adaptive programming paradigm for such heterogeneous environments is the master-slave (MS) paradigm, where computing nodes are classified into two groups, namely masters and slaves. In this paradigm, masters generate computational tasks and assign them to slaves, while slaves execute the assigned tasks. Although the MS paradigm provides us to develop high performance programs by allowing a dynamic load-balancing mechanism, the performance falls if the program is executed under inappropriate circumstances, for example, too many slaves for a master or too small granularity for a task. In order to prevent such undesired executions, performance prediction tools provide useful information. For example, these tools predict the performance on different numbers of slaves or granularities for tasks, so that optimal values for these execution parameters may easily be detected for every execution environment.

Existing performance prediction methods are based on a direct execution method [9, 10, 12], which accurately predicts the performance, or a symbolic estimation method [1, 14], which rapidly predicts. These methods can not achieve both the rapidity and the accuracy for MS programs. The direct execution method requires longer time than the execution time of the program due to executing the whole of the program for the prediction. This situation is against our purpose of using the prediction. After the performance prediction of a program by the method, we also obtain computational results of the program, because the method directly executes the whole of the program. This means that it is no longer necessary to execute the program with appropriate parameter values gotten by the prediction. On the other hand, the symbolic estimation method can not accurately predict the performance if workload of tasks are unpredictable, because the method predicts workload of parts of a program and derives the execution time of the program from the workload. Therefore, this method can not accurately predict the performance of MS programs, because we generally apply the MS paradigm to problems in which the distribution of workload is unknown.

In this paper, it is proposed to achieve both the accuracy and the rapidity of the prediction of MS programs performance, where we assume that the prediction is fast if the prediction time is shorter than the execution time of the program. Our method achieves the rapidity by reducing the directly executed tasks of MS programs. Instead of the reduction, the method estimates the execution time of all tasks of MS program from some directly executed tasks. On this estimation, we use a linear interpolation in order to keep a relationship between the order of tasks assignment and the size of tasks. This relationship is important to achieve good prediction accuracy.

The rest of the paper is organized as follows. Section 2 introduces some related work. Section 3 presents our

method and its design aspects. Section 4 presents experimental results on a cluster of PCs. Finally, Section 5 concludes the paper.

2 Related Work

There are some theoretical approaches for predicting MS programs performance [3, 4, 7]. Although these approaches are helpful in determining the number of slaves, they require strict assumptions like that all tasks assigned by the master must be of same-size [4] and workload must be a representative distribution such as an exponential distribution [7].

Other approaches are a direct execution and a symbolic estimation of parallel programs. MPI-SIM [12] is based on the direct execution method. MPI-SIM predicts the performance of MPI [13] programs by a discrete event simulation. In this simulation, MPI-SIM obtains events of the program behavior, for example, sending, receiving and computing, with directly executing the whole of the program. This contributes to reproducing the behavior of the program in detail. Therefore, MPI-SIM can accurately predict the performance. However, MPI-SIM requires longer time than the execution time of program due to executing the program for obtaining the events. The master slaves emulator [10], called MSE, is also based on the direct execution method. MSE targets to predict the performance of MS programs by emulating the programs and using a parallel computational model improved to represent the bottleneck at the master. Although MSE predicts accurately the performance and the behavior, MSE requires longer time than the execution time of the program due to its emulation approach. Thus, the direct execution method requires long time for the prediction.

On the other hand, the symbolic estimation method [1, 14] predicts the performance faster than the measured execution, because the symbolic estimation method executes no or few parts of the program unlike the direct execution method. This methods estimate workload of the program and derives the performance from the workload. For example, a method [1] measures the execution time of an iteration of loops and obtain the number of loop iterations, then derives the execution time of the whole of the loop. The symbolic execution method is effective for the programs in which the control flow is statically determined, because such programs allow us to easily estimate the workload from their source programs. However, this method can not accurately predict the performance of MS programs, because the workload of MS programs are generally unpredictable as described in Section 1.

Thus, the direct execution method and the symbolic estimation method can not achieve both the rapidity and the accuracy of the prediction of MS programs. In contrast to these methods, we target to realize a performance prediction method that achieves the both rapidity and accuracy.

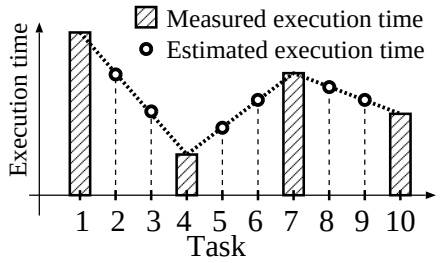


Fig. 1. Example of estimation of execution time by linear interpolation.

3 Performance Prediction by Partial Execution of Program

In this section, we describe our performance prediction method for MS programs. Our method is based on the following two steps. First, the method estimates the execution time of all tasks from some directly executed tasks by a linear interpolation. We refer this direct execution of some tasks as partial execution. Second, the method simulates the behavior of MS paradigm with the estimated tasks.

3.1 Estimating Execution Time of Tasks by Linear Interpolation

For the prediction of the MS programs performance, an obstacle of existing symbolic estimation methods is that the directly executed parts of program are nothing or too few, for example, *an* iteration of a loop, to estimate the workload, because MS programs contains dynamically determined factors such as the workload and the control flow. Therefore, in order to achieve an accurate prediction, it is necessary for a prediction method to use more information for the estimation by executing more parts of the programs. Besides, In order to achieve the rapidity of the prediction, it is necessary for the prediction method to reduce directly executed parts of the program.

Based on the above arguments, our method estimates execution times of all tasks by a linear interpolation from execution times of some directly executed tasks. Although it is difficult to estimate the accurate execution time of all tasks due to their unpredictable workload, we think that the linear interpolation gives us a good approximation of the execution time of tasks. This method realizes the increase of the information by measuring execution time of tasks, and the reduction of the direct execution by adjusting the number of measured tasks.

Fig.1 shows an example of estimating the time by the linear interpolation. In Fig.1, task 1, 4, 7 and 10

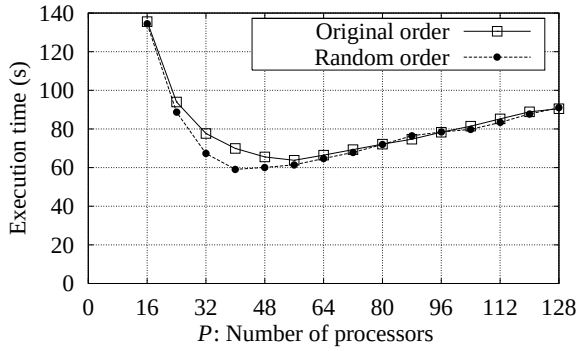


Fig. 2. Execution time of original and random order of assignment.

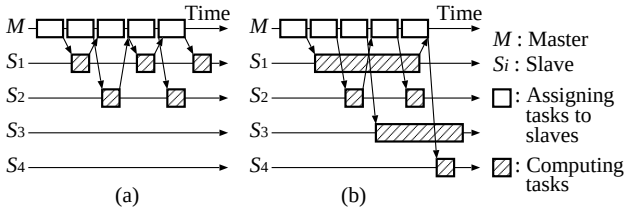


Fig. 3. Efficiency for using slaves.

are directly executed and measured their execution time. Then, the execution time of all remained tasks are estimated by the linear interpolation. Thus, our method avoids the direct execution of all tasks.

In order to keep good prediction accuracy on our method, it is important to keep both (A) the total execution time of all tasks and (B) the order of assignment of tasks near to those of the original tasks. (A) is almost trivially important, especially for computation-intense applications. (B) is important for the MS programs as follows. Fig.2 shows an execution time of an MS program with original and randomized order of tasks assignment. As shown Fig.2, the execution time with randomized order is 15% smaller than that with original order around $P = 40$. This difference is fatal for detecting the appropriate number of slaves. Thus, the order of task assignments affects the performance.

The reason for this difference is an efficiency of using slaves as follows. Fig.3 shows a behavior of MS paradigm, where the master assigns only small tasks to slaves in Fig.3(a) and the master assigns both small and large tasks to slaves in Fig.3(b). As shown Fig.3(a), two slaves, S_3 and S_4, are idle due to too small tasks. For example, the master receives a result from S_1 before assigning a task to S_3, so that the master assigns the task to S_1. On the other hand, in Fig.3(b), the master assigns

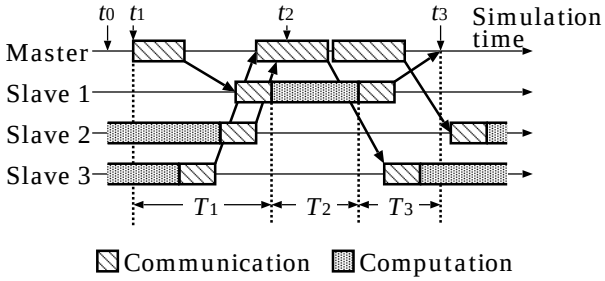


Fig. 4. Example of simulation.

small task and large task by turns to slaves, so that the MS paradigm uses all slaves. Thus, the order of task assignments affects the efficiency of using slaves. Therefore, reproducing the original order of tasks assignment is important for precise prediction. The linear interpolation provide the both (A) and (B) by adjusting the number of directly executed tasks.

3.2 Simulating Behavior of MS Paradigm by Parallel Computational Model

A performance bottleneck of MS programs is concentration of messages at the master. This concentration causes a delay of tasks assignments to slaves. In order to accurately predict the performance, a prediction method must predict a performance degradation caused by this bottleneck. Therefore, in order to reproduce the bottleneck, our prediction method simulates the behavior of MS paradigm by a simulator.

The simulator repeats the following three behaviors of MS paradigm until the master receives results of all tasks: (1) the master assigns tasks to each slave, (2) the slave returns the result to the master time after computing the task, and (3) the master receives the result from a slave and assigns a new task to the slave if tasks remain. In order to simulate the bottleneck at the master, the master delays a task assignment if a result arrives at the master when the master is assigning a task to other slave. The simulator manages an execution time by using a simulation time. At each behavior, the simulator sets the simulation time forward based on the estimated execution time of tasks and the communication time described at the later. In order to manage the unpredictable arrival of messages at the master from slaves, the simulator has a queue which has a list of the arrival time. When the master assigns a task to a slave in the simulation, the simulator calculates the time at which the result of the task arrives to the master.

Fig.4 shows an example of the simulation. In this example, the queue has two arrival times, t_1 and t_2, at t_0. First, the simulator dequeues the oldest arrival time from the queue and sets the virtual time t_1. This means that the master receives a result from a slave with taking the

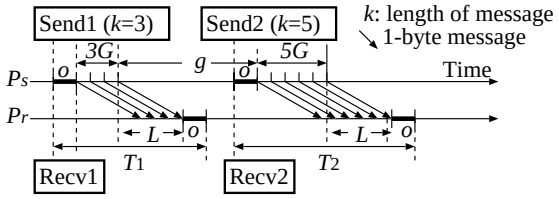


Fig. 5. Communication under the LogGP model.

receiving overhead $t_1 - t_0$. Then, the simulator calculate three time T_1, T_2 and T_3 which correspond to the time required assigning a task, computing the task, and returning the results, respectively.

When estimating the communication time, it is necessary to rapidly estimate the time in order to realize the rapid prediction. For this reason, we use a parallel computational model. The parallel computational model abstracts communication of messages with some parameters, so that we can rapidly estimate the communication time by using the parameters. In this work, we used extended version of the LogGP model [2, 10]. LogGP abstracts the communication by using the following five parameters:

- L: the latency, incurred in sending a message from its source processor to its target processor.
- o: the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message. In the improved version of LogGP, o is a linear function of P in order to represent the overhead for retrieving arrival messages [10].
- g: the gap between messages, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor.
- G: the gap per byte for long messages, defined as the time per byte for a long message.
- P: the number of processors.

Fig.5 shows an example of two messages, Send1 with $k = 3$ and Send2 with $k = 5$, under LogGP, where k is the length of message in bytes. As shown in Fig.5, the communication time for k-bytes message, T_1 for Send1 and T_2 for Send2, is $2o + L + kG$. Thus, the parallel computational model enables us to rapidly estimate the communication time by representing the time as an expression.

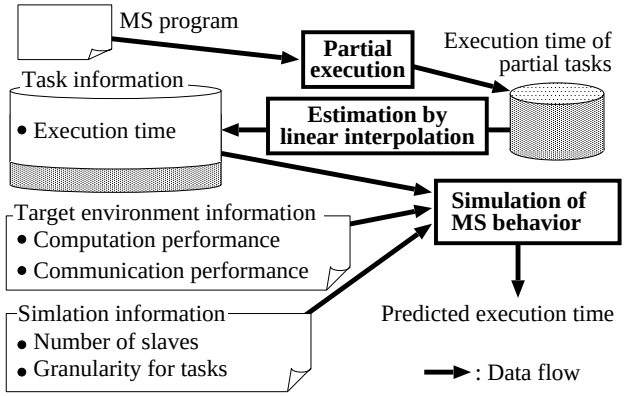


Fig. 6. Flow of performance prediction.

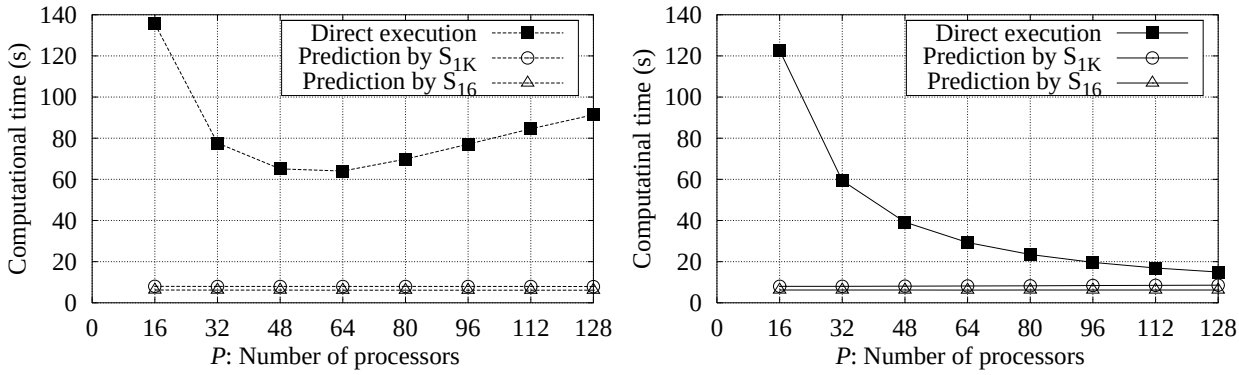
3.3 Flow of Performance Prediction

Now, we describe a flow of our performance prediction method with Fig.6. First, a user modifies the MS program to partially execute tasks and to measure their execution time. From the measured execution time, we estimate the execution time of all remain tasks by the linear interpolation. Next, we simulate the behavior of MS paradigm by the simulator. The simulator has three input data, namely a task information, a target environment information, and a simulation information. The task information represents a set of execution times of tasks. The target environment information includes the computation and communication performance of the target environment of performance prediction. We represent the computation performance as a ratio of computational speed of target environment to that of host used on partial execution. The communication performance consists of parameters values of LogGP. The simulation information includes the execution parameters of MS paradigm such as the number of slaves. Finally, we obtain the predicted execution time of the MS program on the target environment and the execution parameters.

4 Experimentation

In order to validate our method on its prediction accuracy and computational time required to predict the performance, we applied it to a MS program: a parallel Mandelbrot set explorer for fractal visualization. This MS program has 1,048,576 tasks in this experiment.

We used a 64-node cluster as a target environment of performance prediction. Each node in the cluster connects to Myrinet [5] and Fast Ethernet switches, yielding full-duplex bandwidth of 2 Gb/s and 100 Mb/s, respectively. Each node has two Pentium III 1GHz processors, so that the cluster has 128 CPUs. We used a node to execute the simulator. We implemented the MS program by using MPICH [8] on Ethernet and MPICH-SCore [11] on Myrinet.



(a) Ethernet (b) Myrinet

Fig. 7. Direct execution time and computational time.

In this experiment, we use two task sets, referred as S_{1K} and S_{16}, in order to validate the effect of the number of tasks for the partial execution. S_{1K} and S_{16} consist of 1,048,576 tasks, and execution times of these tasks are estimated from 1,024 and 16 directly executed tasks, respectively. Furthermore, in order to validate the improvement of the prediction accuracy by the linear interpolation, we use a task set, referred as S_R, which consists of 1,048,576 tasks randomly generated under the distribution of the size of original tasks.

4.1 Computational Time

In this section, we validate the rapidity of our prediction method. We compare the computational time with the direct execution time, because the direct execution time corresponds to the minimum time taken to predict the performance by prediction methods based on direct execution. The computational time consists of the following four times: (1) time taken to partially execute tasks, (2) time taken to interpolate, (3) time taken to write the task information to a file and read the file when starting the simulation, and (4) time taken to simulate the behavior of MS paradigm.

Fig.7 shows the computational time and the direct execution time. As shown in Fig.7, the computational times are shorter than the direct execution time. In this experiment, the minimum ratio of the direct execution time to the computational time is 8.0 and 1.7 on Ethernet at $P = 64$ and Myrinet at $P = 128$, respectively. This means that our method predicts the performance 1.7 times faster, at least, than the measured execution time in this experiment. Therefore, our method achieves the rapidity of the prediction.

Table 1 shows the detail of the computational time. This result shows that time for the linear interpolation is trivial for the computational time. Although the time for the file I/O is large, we think that the time can be reduced by improving the implementation of our prediction method to pass the task information to the simulator directly. The time for the partial execution depends on the number of directly executed tasks. Although we expect that the prediction accuracy rises by increasing the number, the time also become large. Therefore, we must carefully determine the number in order to keep the computational time shorter than the direct execution time. We think that the number must be less than N_a/P where N_a is the total number of tasks and P is the maximum number of processors, because it is expected that a processor computes an average of N_a/P tasks in parallel computation and the partial execution is processed on a processor. In this experiment, the number must be less than 8,192, because of $N_a =$ 1,048,576 and $P = 128$.

4.2 Prediction Accuracy

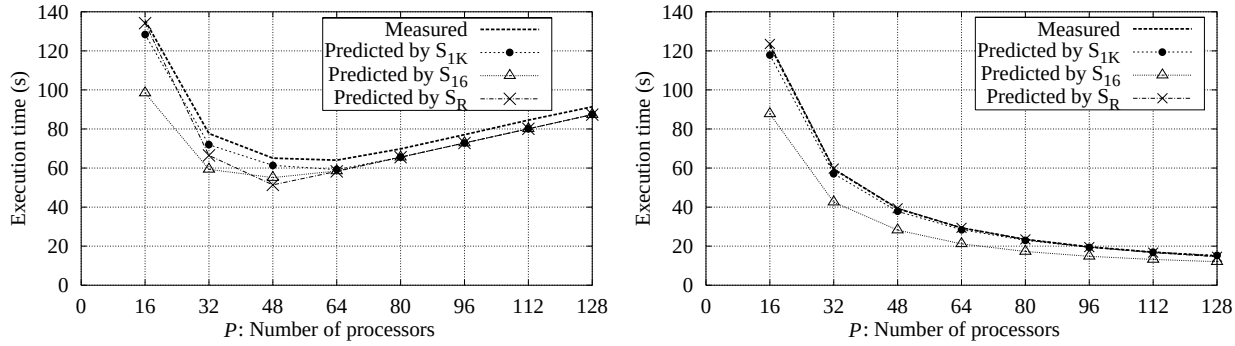
We let E_x be an execution time predicted by using task set S_x. Fig.8 shows a comparison of measured execution time with predicted execution time. The maximum error ratio of measured execution time to E_{1K} is 7.0% and 4.0% on Ethernet at $P = 64$ and Myrinet at $P = 32$, respectively. These results show that our prediction method achieve a good accuracy.

Fig.8(a) shows the advantage of our linear interpolation estimation, because E_{1K} succeeds to predict the P which enables us to get the best performance of the MS program while E_R fails to predict the P. As described in Section 3.1, when the assignment order of tasks are randomized, the performance become better because of an efficient usage of slaves. This prevents an accurate prediction of a saturation point of the MS program performance. Thus, it is important to reproduce the order for achieving an accurate prediction. Opposite to Fig.8(a), E_{1K} and E_R have almost no difference in Fig.8(b). On the Myrinet, the performance does not saturate at $P = 128$ because of a high performance network, so that slaves are efficiently used.

Difference between the measured execution time and

Table 1. Breakdown of computational time.

	P	Partial execution (s)	Interpolation (s)	File I/O (s)	Simulation (s)		Total (s)	
					Ethernet	Myrinet	Ethernet	Myrinet
S_{1K}	64	1.7	0.3	5.4	0.6	0.8	8.0	8.2
	128				0.6	1.2	8.0	8.6
S_{16}	64	0.02	0.2	5.4	0.6	0.6	6.2	6.2
	128				0.6	0.6	6.2	6.2



(a) Ethernet

(b) Myrinet

Fig. 8. Comparison of prediction accuracy on Mandelbrot set explorer.

E_{16} is large in both Ethernet and Myrinet. This difference is caused by a difference of the total execution time of all tasks. An error ratio of the measured execution time to E_{16} is 28% and 27% in Ethernet and Myrinet, respectively. On the other hand, an error ratio of the total execution time of directly executed tasks to the total execution time of S_{16} is 29%. Therefore, in order to achieve an accurate prediction, we must use a task set whose total execution time is near to the total execution time of directly executed tasks. For keeping the total execution time of tasks near to that of directly executed tasks, we consider a policy for selecting a number of directly executed tasks in Section 4.3.

4.3 Deciding the Number of Directly Executed Tasks

As described in Section 4.2, the number of directly executed tasks strongly affects the prediction accuracy. However, we think that it is difficult to analytically estimate the appropriate number of directly executed tasks, since we have no knowledge about the distribution of execution times of tasks before executing the program. Therefore, we estimate the number by a statistical method, called the bootstrap method [6].

First, we guess an average execution time of all tasks by using the bootstrap. Since the total number of tasks is fixed, we can estimate the total execution time of all tasks from the average time. We expect that the range of the error of the average time corresponds to that of the total execution time. Now, we describe the detail of the bootstrap as follows.

1. Randomly sample n tasks and measure an average execution time of them.
2. Repeat the step 1 N times, that is, get N average execution times.
3. Sort the N average execution times and remove α% number of elements from both side of sorted times. Then, the range which consists of remained times is called $100 - 2\alpha$% confidence interval.

We consider that a median of the remained times as an estimated average execution time of tasks. Furthermore, we consider that an interval derived from the error ratio of the confidence interval to the average execution time corresponds to an interval of the prediction error (referred as $\mathcal{I}_e$).

By the central limit theorem, as n increases, the confidence interval becomes narrow, that is, $\mathcal{I}_e$ becomes narrow. Therefore, we can decide an appropriate number of directly executed tasks by increasing n step by step and verifying the width of $\mathcal{I}_e$. Table 2 shows a result of applying the bootstrap to tasks used in this experiment, where N is 40 and α is 2.5%. R represents an error ratio of the total execution time of measured tasks to the total execution time of estimated tasks. As shown in Table 2, $\mathcal{I}_e$ becomes narrow as n increases. For all n, $\mathcal{I}_e$ contains or is close to the R. Thus, $\mathcal{I}_e$ well corresponds to R. Therefore, if we want the prediction error to be less than x%, we should select such n that derives $\mathcal{I}_e$ contained by

Table 2. Result of estimation by bootstrap for 40 iterations ($N = 40$).

n: # of sampled tasks	$\mathcal{I}_e$: Interval of prediction error		R: Error ratio of total execution time (%)
	Min. (%)	Max. (%)	
4	-70	81	-72
16	-41	46	-29
64	-19	17	-17
256	-10	11	-11
1K	-4.9	5.8	-4.2
4K	-2.9	2.6	-2.5
16K	-1.4	1.1	-1.7
64K	-0.7	0.4	-0.2

the interval between $-x$% and x%. For example, if we want an error ratio of total execution time of tasks to be less than 10%, we should select the number of directly executed tasks larger than 1K from Table 2 because both the minimum and the maximum value of $\mathcal{I}_e$ of $n = 1$K is less than the 10%.

5 Conclusion

We have presented a performance prediction method for rapidly and accurately predicting the performance of MS programs. In order to achieve the rapidity of the prediction, our method reduces the direct execution of the programs by estimating the execution time of all tasks from some directly executed task. In the estimation, we also have presented the importance of representing the order of tasks assignment for the estimation for accurate prediction of the performance saturation point. We validated the prediction accuracy and the computational time of our method. The results shows that the method predicts the execution time with 7% and 1.7 times faster.

Future work includes the integration of the statistical method, mentioned in Section 4.3, to determine the appropriate number of directly executed tasks.

Acknowledgments

This work was partly supported by Grant-in-Aid for Scientific Research (B)(16300006), NEC System Platforms Research Laboratories, and the 21st Century COE Program of Osaka University. We are also grateful to the anonymous reviewers for their valuable comments.

References:

[1] V. S. Adve, R. Bagrodia, E. Deelman, and R. Sakellariou. Compiler-optimized simulation of large-scale applications on high performance architectures. *J. Parallel and Distributed Computing*, vol. 62, no. 3, Mar. 2002, pp. 393–426.

[2] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *J. Parallel and Distributed Computing*, vol. 44, no. 1, July 1997, pp. 71–79.

[3] C. Anglano. Predicting parallel applications performance on non-dedicated cluster platforms. In *Proc. 12th ACM Int'l Conf. Supercomputing (ICS'98)*, July 1998, pp. 172–179.

[4] O. Beaumont, A. Legrand, and Y. Robert. The master-slave paradigm with heterogeneous processors. In *Proc. 3rd IEEE Int'l Conf. Cluster Computing (CLUSTER'01)*, Oct. 2001, pp. 419–426.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local-area network. *IEEE Micro*, vol. 15, no. 1, Feb. 1995, pp. 29–36.

[6] B. Efron. Bootstrap methods: Another look at the Jackknife. *Annals of Statistics*, vol. 7, no. 1, Jan. 1979, pp. 1–26.

[7] A. G. Greenberg and P. E. Wright. Design and analysis of master/slave multiprocessors. *IEEE Trans. Computers*, vol. 40, no. 8, Aug. 1991, pp. 963–976.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, vol. 22, no. 6, 1996, pp. 789–828.

[9] D. F. Kvasnicka, H. Hlavacs, and C. W. Ueberhuber. Simulating parallel program performance with CLUE. In *Proc. 2001 Int'l Symp. Performance Evaluation of Computer and Telecommuication Systems (SPECTS'01)*, July 2001, pp. 140–149.

[10] Y. Mizutani, F. Ino, and K. Hagihara. Evaluation of performance prediction method for master/slave parallel programs. *IEICE Trans. Information and Systems*, vol. E87-D, no. 4, Apr. 2004, pp. 967–975.

[11] F. O'Carroll, H. Tezuka, A. Hori, and Y. Ishikawa. The design and implementation of zero copy MPI using commodity hardware with a high performance network. In *Proc. 12th ACM Int'l Conf. Supercomputing (ICS'98)*, July 1998, pp. 243–250.

[12] S. Prakash, E. Deelman, and R. Bagrodia. Asynchronous parallel simulation of parallel programs. *IEEE Trans. Software Engineering*, vol. 26, no. 5, May 2000, pp. 385–400.

[13] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[14] A. J. C. van Gemund. Symbolic performance modeling of parallel systems. *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 2, Feb. 2003, pp. 154–165.