Proceedings of IASTED International Conference APPLIED INFORMATICS:
International Symposium on Parallel and Distributed Computing and Networks,
pp. 364-369, ACTA press, February 18-21, 2002, Innsbruck, Austria.

ITERATIVE DATA PARTITIONING SCHEME OF PARALLEL PDE SOLVER FOR HETEROGENEOUS COMPUTING CLUSTER

SHUICHI ICHIKAWA and YOSHIKATSU FUJIMURA*
Department of Knowledge-based Information Engineering,
Toyohashi University of Technology
Hibarigaoka, Tempaku, Toyohashi, Aichi 441-8580, JAPAN
Email: ichikawa@tutkie.tut.ac.jp

ABSTRACT

This paper presents a static load balancing scheme for a parallel PDE solver targeting heterogeneous computing clusters. The proposed scheme adopts a mathematical programming approach and optimizes the execution time of the PDE solver, considering both computation and communication time. While traditional task graph scheduling algorithms only distribute loads to processors, the proposed scheme adopts a combined approach of iterative data partitioning and load distribution to make total execution time minimal. The approximation algorithm presented here shows good accuracy and is solvable in practical time.

KEYWORDS: Optimization, Load balancing, Scheduling, Cluster computing

1 INTRODUCTION

Many important scientific and technological applications are modeled and solved as partial differential equations (PDE). For the efficient processing of the PDE system, various parallel PDE solvers have been researched for years (e.g. //ELLPACK[1], PETSc[2], CTADEL[3], PDE^2[4]). However, only a limited degree of static load balancing has been implemented in the preceding parallel PDE systems.

Many preceding systems tried to equalize the computation time, while minimizing the communication time. Such a scheme is simple and easy to implement, but does not guarantee that the total execution time is minimal, particularly when the communication time is substantial. To optimize the execution time totally, it is essential to consider both calculation and communication as a whole. While there are some studies that focused on this scheme [5] [6], they only discussed heuristic approximation algorithms, because this kind of optimization is generally a hard computation problem and has been regarded as intractable. One of the contributions of the present research is that we model the problem as a combinatorial optimization problem and show that it is solvable in practical time. No parallel PDE systems have adopted this kind of approach.

The second contribution of this paper is that it discusses an optimization method suited for heterogeneous clusters. Preceding studies have scarcely considered heterogeneous computer clusters, which consists of *non*-uniform processors. Heterogeneous clusters are very widespread and an important target architecture, as they are found in every office and lab as various PCs connected in a LAN. On the other hand, they are an essentially difficult target for optimization, because non-uniformity of processors results in a vast number of free variables in optimization. It is another difficulty of heterogeneous clusters that relatively slow connections often make communication time dominant over calculation time. In some cases, you get worse performance by using more processors because of the increase of communication time. Let us call this paradox "*excessive use of processors*" in this paper.

One of the authors has been involved in developing a parallel PDE solver for NSL [7]. NSL is a numerical simulation system that automatically generates parallel programs for multicomputers from a high level description of PDE. NSL adopts an explicit finite difference method (FDM) based on a boundary fitted coordinate system and multi-block method.

Ichikawa et al. [8] showed that the static load balancing of this PDE solver could be modeled as a combinatorial optimization problem, considering both calculation time and communication time. In that paper [8], the target architecture was assumed to be distributed memory parallel processors or homogeneous clusters that consist of uniform processing elements (PEs). This optimization problem was shown to be solvable in practical time by an off-the-shelf PC for systems consisting of hundreds or more PEs.

Ichikawa and Yamashita [9] extended the scheme for heterogeneous clusters. Under the condition of a sufficient number of processors, it was shown that an adaptive recursive partitioning scheme gives a good load balance in practical time. It is also reported that the performance degradation caused by excessive use of processors can be avoided by selecting the most adequate subset of processors [10]. These results are encouraging, but some problems remain. When the number of processors is insufficient against the complexity of a given PDE domain, the preceding scheme [9][10] cannot give a good load balance. In such a case,

*Presently with Toyota System Research Inc.

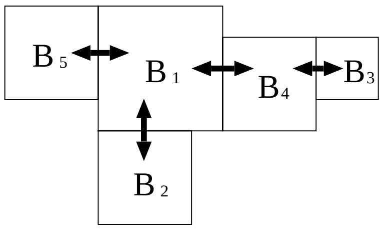


Figure 1. Computational Domain

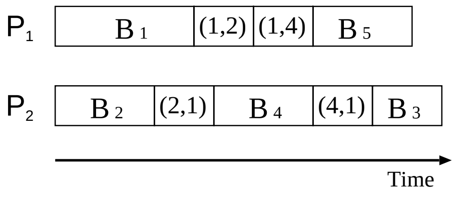


Figure 2. Load Distribution

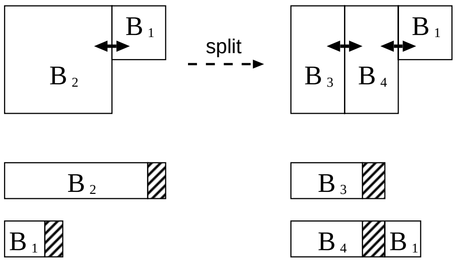


Figure 3. Improved Load Balance

load coalescing is required in addition to data partitioning.

This paper presents a combined approach of iterative partitioning and load coalescing. Section 2 outlines the model of the PDE solver and our new scheme. In section 3, some heuristics for load coalescing are examined. Section 4 describes the iterative partitioning scheme and the evaluation results.

2 OPTIMIZATION MODEL OF PDE SOLVER

In NSL [7], a physical domain is mapped to a computational domain, which consists of mutually connected rectangular blocks. Each block is an array of grid points, on which differential equations are calculated. Each calculation of the differential equations can be executed in parallel due to the nature of explicit FDM, followed by communications to exchange data across the connected borders. The parallel PDE solver in NSL invokes this set of operations iteratively. Figure 1 illustrates an example computational domain that consists of five blocks.

Let the number of blocks be m, and the number of processors be n. The relationship $m \ll n$ was assumed in preceding studies [8][9][10]. This assumption is valid in such cases that the problem is simply structured but contains so many grid points that many processors are required, which is often seen in real-world applications. In these cases, static load balance is realized by partitioning each block into pieces (*subblocks*), each of which is dispatched to a processor. Each processor handles only one subblock to suppress communication as much as possible.

However, there are still many applications that require $m \approx n$ or $m > n$. In such cases, each processor has to handle one or more subblocks, according to its performance. Let us examine Figure 1 as an example. The domain of Figure 1 includes five blocks ($m = 5$). Provided that we have two processors ($n = 2$), we can distribute these blocks (B_1, ..., B_5) between two processors (P_1 and P_2) to balance computational loads. Note that the best of 32 possible distributions (generally n^m ways) must be chosen to get the best load balance. Section 3 describes how to do this.

Figure 2 shows an example of load distribution. The communication from B_i to B_j is counted as (i, j) in the figure. Though four bidirectional communications are required in Figure 1, four communications $(1, 5), (5, 1), (3, 4)$, and $(4, 3)$ are not counted in Figure 2, because (B_1, B_5) and (B_3, B_4) are allocated on the same processor. As variables can be shared freely on the same processor, no additional communication cost is required.

This kind of optimization problem is very popular as a *task graph scheduling problem*, and has been thoroughly researched (see Kwok [11] for recent survey). However, the sole task graph scheduling cannot give a good load balance for our problem when $m \approx n$ holds or when there is much difference in block size. Figure 3 (left) illustrates an unsuitable case, in which $m = n = 2$ holds and block size differs much. Block B_2 is too big for one processor, consequently becoming the bottleneck. In such a case, splitting B_2 into two blocks (B_3, B_4) is a good idea to shorten total execution time (Figure 3 right).[1] Similar situations can occur even when all blocks are of the same size, because the performance of each processor varies in heterogeneous clusters. This is the reason why we need an approach that combines load balancing and data partitioning. The task scheduling approach is general but *not* all-powerful. We can derive better performance by exploiting problem-specific criteria.

The next section examines some algorithms with which to distribute blocks among processors for a given domain structure. Then, Section 4 describes the iterative partitioning scheme that is used in combination with load distribution algorithms.

[1] Though new communication sometimes emerges with such partitioning, the communication between B_1 and B_4 is suppressed in this case, because they are allocated on the same processor.

3 LOAD DISTRIBUTION ALGORITHMS

3.1 FORMULATION

Now, we are ready to formulate the model as an optimization problem. Let T be the execution time of one iteration of PDE solver. Our purpose is to minimize T for a given domain structure by partitioning and distributing blocks among available processors.

$$T = \max_i T_i \quad (i = 1, ..., n), \tag{1}$$

$$T_i = \sum_{B_j \in G_i} (Ta_j + Tc_j). \tag{2}$$

T_i is the execution time of the i-th processor (P_i). T_i is given by the sum of the time for each block B_j in G_i, where G_i is the set of blocks dispatched to P_i. Ta_j and Tc_j are the calculation and communication times of B_j, respectively. Ta_j and Tc_j are estimated as the linear function of the number of grid points involved.

$$Ta_j = Cta_i\, Sa_j + Dta_i, \tag{3}$$

$$Tc_{j,l} = Ctc\, Sc_{j,l} + Dtc, \tag{4}$$

$$Tc_j = \sum_{B_l \notin G_i} Tc_{j,l}. \tag{5}$$

Here, Sa_j is the number of grid points of B_j. Cta_i and Dta_i are coefficients that are dependent on the performance of P_i. $Sc_{j,l}$ is the number of grid points involved in the communication from B_j to B_l. Ctc and Dtc are communication coefficients of the network.

Though it is very easy to calculate T for a given set of $G = \{G_i \,|\, i = 1, ..., n\}$, we have to check every possible G to find the optimal T, which takes $O(n^m)$ time. As is well known, this kind of enumeration approach is intractable. The branch-and-bound method [12] and heuristic approximation algorithms are hence adopted here. Find more details for these techniques in the preceding reports [8][9][10].

3.2 HEURISTIC ALGORITHMS

This section introduces five simple heuristic algorithms. As space is limited, only the core idea of each algorithm is described.

Approx1 dispatches blocks in a greedy manner, considering calculation time only. Approx1 first sorts blocks according to their size in decreasing order, then iteratively dispatches the largest block to the most lightly loaded processor at that time.

Approx2 first sorts the blocks in the same manner as Approx1. It then iteratively dispatches the largest block to the processor that will be most lightly loaded after the dispatch. Approx2 considers the communication time to the blocks that are already dispatched. The communication time to the blocks that are not allocated yet is ignored.

Approx3 first sorts the blocks in the same manner as Approx1. It then iteratively dispatches the largest block to the processor that minimizes the maximum tentative execution time of each processor. Approx3 considers the communication time in the same manner as Approx2.

Approx4 sorts the blocks considering both block size and possible communication time. It then dispatches blocks in the same manner as Approx3.

Approx5 first selects the processor P_i that is most lightly loaded at that point. Then, it calculates the priority factor Q_j for each block, and selects the block that has the maximal Q_j value for the next dispatch.

$$Q_j = Cta_i Sa_j + Dta_i + \sum_{B_l \in g_i} (Ctc\, Sc_{j,l} + Dtc) - \sum_{B_l \notin \bar{g} \cup g_i} (Ctc\, Sc_{j,l} + Dtc) \tag{6}$$

Here, g_i is the tentative set of blocks that is dispatched to P_i. The blocks not allocated yet are represented by $\bar{g}$. Q_j gets larger when the block is large or the communication is suppressed (if allocated on the same processor). Q_j gets smaller if the communication emerges by dispatching Q_j to P_i.

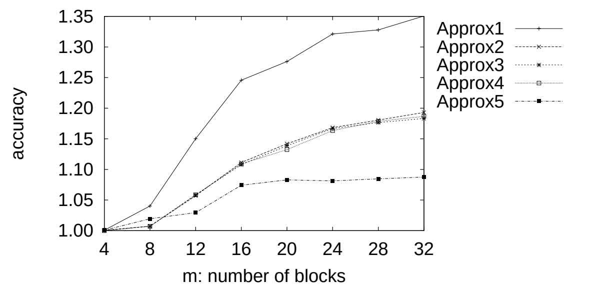
Table 1. Parameters

Parameter	Value
Cta_i	1.0
Dta_i	0.1
Ctc_i	20.0
Dtc_i	0.1

In addition to these approximation algorithms, *local search* technique [12] is examined. After dispatching all blocks by approximation algorithm, we can examine whether the possible swap of two blocks improves the total execution time. A local search iterates such swaps while improvement is derived.

Figures 4 and 5 illustrate various aspects of the proposed approximation algorithms. In these figures, the number of processors is four ($n = 4$), and all processors are equivalent (homogeneous environment is assumed). The computational domain consists of m blocks, and is randomly generated to have a tree topology. Each block is square, and contains 100 to 10000 grid points. In Figures 4 and 5, each point represents the average result of 20 trials. Other parameters are listed in Table 1.

Figure 4 presents the accuracy of approximation algorithms (**Approx1, ..., Approx5**) and their derivatives with



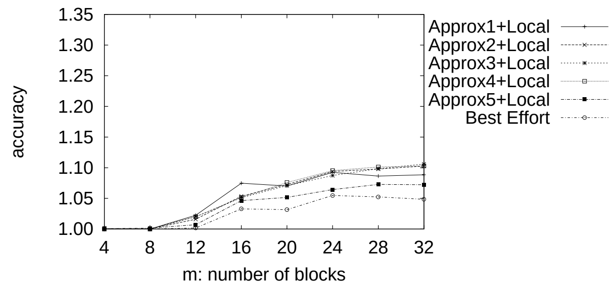
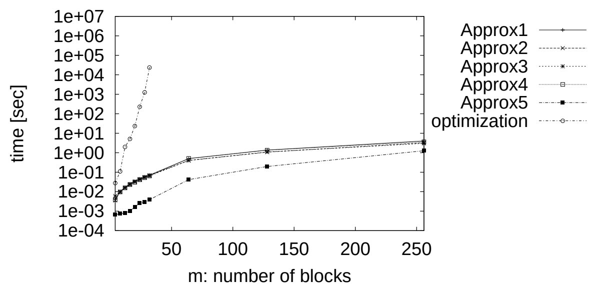


Figure 4. Accuracy of Approximation Algorithms



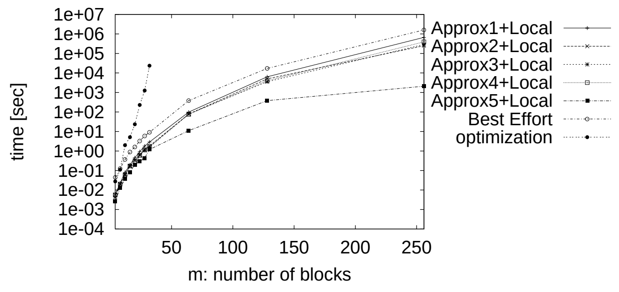


Figure 5. Elapsed Time for Approximation

local search (**+Local**). The best results selected from these algorithms is plotted as **Best Effort** in Figure 4. Accuracy is defined by the ratio of approximated T to the optimal T derived by combinatorial optimization (**optimization**). As is easily seen, Approx5 shows good accuracy. Approx5+Local is slightly better than Approx5 in accuracy, but takes a far longer time to finish (more than 100 times), as seen in Figure 5. All others (Approx1, ..., Approx4) show worse accuracy than Approx5. With local search, their accuracy improves to almost equal Approx5, but far more time is required to finish.

From these observations, we choose Approx5 for our research. The simulations for heterogeneous environment (omitted here for lack of space) show similar trends, except that all algorithms require several times more time to finish.

3.3 COMPARISON WITH TASK GRAPH SCHEDULING ALGORITHM

The algorithms described in Section 3 simply dispatch blocks to processors without any partitioning.[2] That is, general task graph scheduling algorithms can achieve an equivalent result in place of our Approx5 algorithm. We therefore examined CP/DT/MISF algorithm [13] as a possible competitor. CP/DT/MISF is a heuristic task graph scheduling algorithm that is derived from CP/MISF (Critical Path / Most Immediate Successors First), taking data transfer cost (DT) into consideration.

[2]Partitioning is described in Section 4.

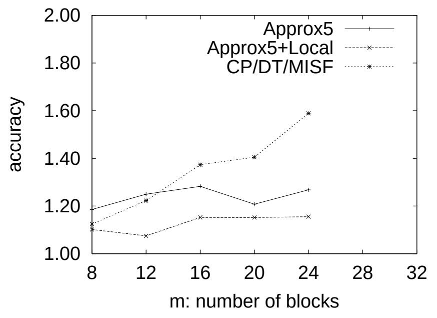
Table 2. Parameters

Parameter	Value	Parameter	Value
Cta_0, Cta_1	0.25	Dta_i	0.1
Cta_2, Cta_3	0.33	Ctc	100.
Cta_4, Cta_5	0.5	Dtc	10000.
Cta_6, Cta_7	1.0		

Figure 6 shows the simulation results of CP/DT/MISF, Approx5, and Approx5+Local for heterogeneous clusters. In this simulation, the computational domain is constructed by m square blocks connected in a random tree topology. The edge length b of each block is also randomly generated as $10 \leq b \leq 800$. The target cluster consists of 8 heterogeneous processors ($n = 8$). The performance ratios of the PEs are set to $4 : 4 : 3 : 3 : 2 : 2 : 1 : 1$. Each point in the figure represents the average value of 20 trials. Other parameters are listed in Table 2.

As seen in Fig. 6, the accuracy of CP/DT/MISF is inferior to Approx5 for a heterogeneous environment, though the elapsed time of CP/DT/MISF is almost the same as Approx5. Approx5+Local shows far better accuracy than either CP/DT/MISF or Approx5, but also requires more time than either of them.

We also examined the cases of homogeneous clusters, and found that CP/DT/MISF can not outperform Approx5, though it shows almost equal accuracy and elapsed time as



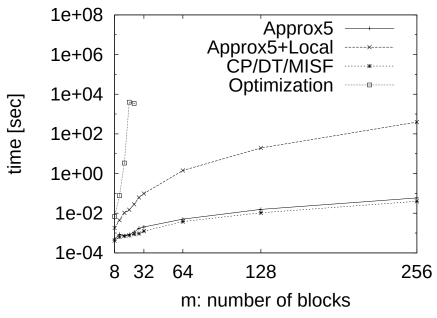
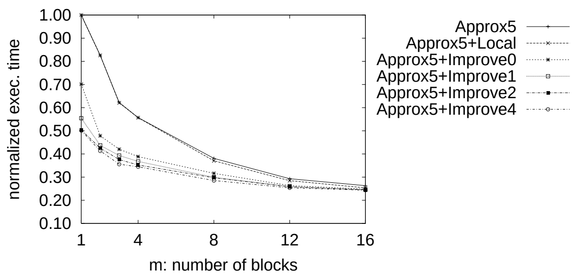


Figure 6. Approx5 vs. CP/DT/MISF



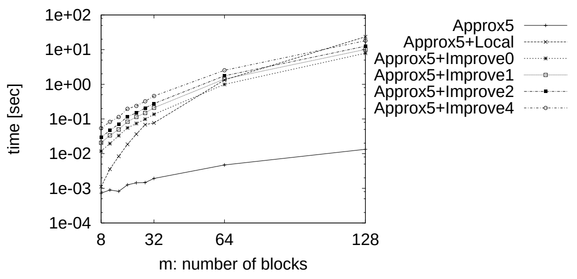


Figure 7. Iterative Improvement

Approx5.

Consequently, we found no merit in adopting CP/DT/MISF in our research. CP/DT/MISF is targeted for general task graphs, while Approx5 and Approx5+Local are more problem-specific and better for our purpose.

4 ITERATIVE PARTITIONING

As shown in Figure 3, load balance can be improved by partitioning *big* blocks into smaller ones. If one partition is not enough, we can apply this procedure iteratively. Note that this kind of partitioning can create new communication, which tends to increase execution time T after all; we therefore have to stop iterative partitioning at the optimal (or sub-optimal) point.

We have to decide the following three matters to realize the iterative partitioning scheme.

- How to split the selected block?
- Which block should be selected for partitioning?
- When should we stop iteration?

Let us examine the first question. There are many ways to split a block into two; we must decide the direction, equally or inequally, etc. The advantages and disadvantages of various partitioning schemes are already discussed in preceding papers [9] [10]. Here, for simplicity, we divide a block equally into two rectangular sub-blocks, in the manner that makes the communication minimal. The choice of a better partitioning scheme is left for future study.

For the second question, the simplest answer is to choose the biggest block. However, this is neither thorough nor good enough (just think you have two or more blocks of the same size). A more elegant way is to examine each block as a candidates for partitioning and take one that makes T minimal after partitioning. We continue this procedure iteratively, while we improve T. Let us call this method **Improve0**.

Improve0 still does not work well enough, because the figure of objective function T is jagged and has many local minima. A usual countermeasure for local minima is to continue iterative improvement even if T increases a couple of times. Let **Improve**N be the method that allows the increase of T at most N times. If we can find a better tentative solution in N partitioning steps, we update the tentative record to the new solution, and continue iteration. If we cannot update the current record in N steps, we terminate the iteration and return the tentative record as the solution.

Figure 7 presents evaluation results of iterative improvement. For the load distribution algorithm, Approx5 is adopted. Simulation conditions are almost the same as in Section 3.3, except that the edge length b of each block is randomly generated as $10 \leq b \leq 3200$. The value 3200 was selected to make the deviation of block sizes sufficiently big. Other parameters are listed in Table 2 of Section 3.3.

In the upper graph of Figure 7, the optimized parallel execution time T is shown, normalized by the value that is derived from a single fastest processor (scalar execution time). When $m \leq n = 8$, Approx5 does not work well because big blocks tend to be bottlenecks. Approx5+Local scarcely improves things because swap of blocks does not resolve bottlenecks. Approx5+Improve0 works well by the effect of iterative partitioning, but is not good enough because Improve0 is easily trapped at local minima. Improve1, Improve2, and Improve4 show better performance than Improve0, but do not differ so much from each other. The apparent lower bound of the normalized execution time is 0.2, because the performance ratio of the fastest processor to all processors is given by 4 / (4+4+3+3+2+2+1+1). From this point of view, Improve1 or Improve2 is regarded as good enough. Approx5+Improve$N$ shows a remarkable achievement, compared with the preceding method [9][10] that could not provide a good load balance in the area of $m \geq 4$ ($m \approx n$ or $m \geq n$),

The lower graph of Figure 7 shows the elapsed time of each approximation algorithm on a Pentium-II 400 MHz processor. Approx5 is extremely fast, but all other algorithms also finish in practical time.

5 CONCLUSION

This paper presented a combined approach of data partitioning and load distribution for a parallel PDE solver. Our simulation results show that Approx5+Improve1 (or Approx5+Improve2) is a practical choice for our purpose.

Another important item, which was not mentioned in this paper, is an algorithm to avoid the excessive use of processors. This problem comes down to finding the most adequate subset of processors, as examined in the preceding papers. It is difficult to examine every possible subset, but $O(n)$ greedy approximation algorithm worked well in past studies [8][9][10]. We expect this kind of algorithm would work well also for the present case. This work is still in progress, and we intend to confirm the findings in the near future.

ACKNOWLEDGMENTS

This work was partially supported by a Grant-in-Aid from the Ministry of Education, Culture, Sports, Science and Technology of Japan.

References

[1] E. N. Houstis, et al. PELLPACK: A Problem-Solving Environment for PDE-Based Applications on Multicomputer Platforms. *ACM Trans. Math. Softw.*, Vol. 24, No. 1, pp. 30–73, March 1998.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pp. 163–202. Birkhauser Press, 1997.

[3] R. von Engeln, L. Wolters, and G. Cats. CTADEL: A Generator of Multi-Platform High Performance Codes for PDE-based Scientific Applications. In *Proc. ICS '96*, pp. 86–93. ACM, 1996.

[4] M. Prieto, I. Martin, and F. Tirado. An Environment to Develop Parallel Code for Solving Partial Differential Equations Based-Problems. *J. of Systems Architecture*, Vol. 45, pp. 543–554, 1999.

[5] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works!*, chapter 11. Morgan Kaufmann, 1994.

[6] N. Chrisochoides, N. Mansour, and G. Fox. Comparison of Optimization Heuristics for the Data Distribution Problem. *J. Concurrency Practice and Experience*, Vol. 9, No. 5, pp. 319–344, 1997.

[7] T. Kawai, S. Ichikawa, and T. Shimada. NSL: High-Level Language for Parallel Numerical Simulation. In *Proc. IASTED Int'l Conf. Modeling and Simulation (MS '99)*, pp. 208–213. Acta Press, 1999.

[8] S. Ichikawa, T. Kawai, and T. Shimada. Mathematical Programming Approach for Static Load Balancing of Parallel PDE Solver. In *Proc. 16th IASTED Int'l Conf. Applied Informatics (AI '98)*, pp. 112–114. Acta Press, 1998.

[9] S. Ichikawa and S. Yamashita. Static Load Balancing of Parallel PDE Solver for Distributed Computing Environment. In *Proc. ISCA 13th Int'l Conf. Parallel and Distributed Computing Systems (PDCS-2000)*, pp. 399–405. ISCA, 2000.

[10] S. Ichikawa and S. Yamashita. Static Load-balancing for Distributed Processing of Numerical Simulations. *IPSJ Journal*, Vol. 42, No. 3, pp. 552–564, 2001. (in Japanese).

[11] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, Vol. 31, No. 4, pp. 406–471, 1999.

[12] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Dover Publications, 1998.

[13] H. Kasahara and S. Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Trans. Comput.*, Vol. c-33, No. 11, pp. 1023–1029, 1984.