An Asynchronous Algorithm for Balancing Unpredictable Workload on Distributed-Memory Machines

Yongwha Chung, Jin-Won Park, and Suk-Han Yoon

CONTENTS

I. INTRODUCTION

II. IRREGULAR PROBLEMS HAVING UNPREDICTABLE WORKLOAD

III. A DYNAMIC LOAD BALANCING STRATEGY

IV. EXPERIMENTAL RESULTS

V. CONCLUDING REMARKS

REFERENCES

ABSTRACT

It is challenging to parallelize problems with irregular computation and communication. In this paper, we propose an asynchronous algorithm for balancing unpredictable workload on distributed-memory machines. By using an initial workload estimate, we first partition the computations such that the workload is distributed evenly across the processors. In addition, we perform task migrations dynamically for adapting to the evolving workload. To demonstrate the usefulness of our load balancing strategy, we conducted experiments on an IBM SP2 and a Cray T3D. Experimental results show that our task migration strategy can balance unpredictable workload with little overhead. Our code using C and MPI is portable onto other distributed-memory machines.

I. INTRODUCTION

Many problems with *regular* computation and communication have been parallelized on distributed-memory machines by using the explicit message passing paradigm or by using parallelizing compilers. However, it is challenging to parallelize problems with *irregular* computation and communication [1]-[3].

Such problems arise in intermediate and high-level computer vision, for instance. Parallel solutions to these problems are characterized by uneven distribution of symbolic features among the processors, unbalanced workload, and irregular inter-processor data dependency caused by the input image [4]. Furthermore, the nature of the irregularity which depends on the input image is not known at compile time. To achieve high performance, the irregularity should be considered carefully and dynamic techniques may be needed.

A key operation in intermediate and high-level vision is to search for symbolic features satisfying certain geometric constraints. For instance, in perceptual grouping, a set of symbolic features satisfying certain geometric constraints are grouped to form structural hypotheses [5]. In another example, image matching, correspondences between symbolic features extracted from two different images are determined based on geometrical relationships [6].

Parallelizing such feature search operations is challenging since the operation is highly data dependent (the size and shape of the search is different for each data element) [7]. Our motivation for this research is to develop a dynamic load balancing strategy for highly data dependent problems such as feature search on distributed-memory machines. We assume that the input data are replicated on each processor.

In general, we can balance the computational workload of an application either by using an algorithmic technique or by using a runtime library/system. In most algorithmic approaches [7]-[13], the computation performed by an application is structured as a sequence of iterations. After each iteration, an estimate of the remaining computation in each processor is collected and used to re-distribute the workload of the next iteration. In contrast, if a solution to an application needs to create tasks dynamically, as in computer chess for example, then general-purpose runtime libraries/systems [14], [15] can be used to balance unpredictable workload.

However, feature search operations such as those discussed above have different computational characteristics. The computational structure of the search operations is non-iterative and the workload is highly data dependent and not easily predictable during the computation. Hence the previous algorithmic techniques are not suitable for balancing such unpredictable workload. However, the total number of search operations is fixed by the input data so that dynamic task creation is not needed; hence,

approaches using a runtime library/system cause unnecessary overhead in task creation and context switching between tasks. To achieve desired speed-ups, we need to balance the unpredictable workload "on the fly" using an efficient algorithmic technique.

In this paper, we propose an asynchronous algorithm for balancing unpredictable workload. In our strategy, we first partition the workload using prior knowledge at the beginning of the computation. In addition, our strategy migrates tasks dynamically to adapt to evolving workload. We group tasks in a "supertask" which is used as a unit for task migration. After finishing execution of a supertask, a processor sends this information to its neighbors, and checks for similar information from them. Based on this communication, a supertask may be migrated from a busy processor to an idle neighbor.

To demonstrate the usefulness of our load balancing strategy, experiments were conducted on an IBM SP2 and a Cray T3D. The experimental results show that our task migration technique can balance unpredictable workloads with very little overhead and the speed-ups can be improved by a factor of two by using the proposed technique. Although we show the experimental results for the feature search operation, our idea can also be applied to many other irregular problems having unpredictable workload. We believe that it offers a general framework to parallelize many irregular problems having unpredictable workload.

The organization of the paper is as follows. An overview of the irregular problems having unpredictable workload is given in Section II. Section III discusses our load balancing strategy for solving the problem. Experimental results are shown in Section IV. Concluding remarks are made in Section V.

II. IRREGULAR PROBLEMS HAVING UNPREDICTABLE WORKLOAD

In the following, we illustrate the computational aspects of irregular problems arising in target tracking and computer vision applications. The main difficulty in parallelizing these irregular problems is that the workload is hard to predict.

1. Hypothesis Testing

Multi-hypothesis testing is a problem in which many possible answers (hypotheses) exist to some question, and each possibility must be considered to find the right answer. These hypothesis testing operations are used in areas such as object classification, discrimination, multi-object tracking, and damage assessment.

A parallel implementation of the hypothesis testing benchmark used in target tracking applications has been reported in [13]. In the benchmark, there are a series of radar frames which contains the x, y coordinate pairs of measurements detected by

```
For (each measurement frame)
    For (each existing hypothesis track)
        Predict track state and covariance to time of current frame
        Extend hypothesis corresponding to missed detection in current frame
        For (each detection point in the frame)
            Gate detection versus predicted track
            If (detection passes gate test)
                Create new hypothesis as extension of track by detection
    For (each detection point in the frame)
        Create new hypothesis for detection point as first track detection
Prune hypotheses
```

Fig. 1. Hypothesis testing operation [13].

radar. The series of frames are consecutive radar images of the same geographical area over a period of time. The purpose of the benchmark is to establish which measurements are for the same object in the different radar frames, thereby tracking the object. The main operation in the hypothesis testing benchmark is shown in Fig. 1.

The main difficulty in parallelizing hypothesis testing is that in order to determine the workload of a processor, we must counter not only the number of hypotheses it currently has, but also future offspring. Because it is impossible to predict what offspring hypothesis will be generated by a hypothesis, a typical load balancing algorithms cannot provide large speed-ups. Details of parallel hypothesis testing can be found in [13].

2. Feature Searching

Searching symbolic features satisfying certain geometric constraints is a primitive operation used in many intermediate and high-level vision algorithms. It is modeled as a search operation within a window in an image plane. We assume that the symbolic data is already stored in the image plane before performing the search operations. The symbolic features used in this paper are line segments extracted from raw images [5]. The line segments are represented symbolically by their end-point coordinates, lengths, orientations, and average contrast.

In Fig. 2, we show a typical search operation for each line segment. A search is performed within a region on both sides of the line segment to find other line segments. To make a general framework, we perform a dummy operation after finding each symbolic data. The dummy operation consists of a loop performing simple arithmetic operations. We choose the number of loop iterations such that the loop execution time is similar to the actual computation time of vision algorithms. By replacing the dummy operation, the search operation can refer to either a *grouping* process or an *image matching* process. In this paper, we use a set of line segments extracted from

```
For (each searching line segment)
   Construct search window (region on both sides of searching line segment) in an image plane
   For (each position in search window)
      Check line segment satisfying geometric constraint
      If (line segment is found)
         Perform dummy operation
```

Fig. 2. Feature searching operation.

an image as both the searching and the searched segments. Details of the grouping process and the image matching process can be found in [5], [6].

Let S denote the number of input line segments. Let $W(S)$ denote the total area of all the search windows generated by the S line segments. The serial time complexity of the search operation is $O(W(S))$.

III. A DYNAMIC LOAD BALANCING STRATEGY

For the purpose of explanation, only the feature search operation is discussed in the following sections. However, the same approach can be applied to other irregular problems having unpredictable workload.

The main difficulty in parallelizing the feature search operation is that the workload depends on the input data in a nontrivial way. In this section, we first describe a typical load balancing strategy that performs a task partitioning step at the beginning of the computation. Then, our task migration strategy is described.

Let P denote the number of processors. The following terminology is used in this paper:

- *Token*: a data structure containing information about an input line segment, such as end-point coordinates, length, orientation, and average contrast of the line segment. The total number of input tokens is S.
- *Task*: the computational work to be performed on a token. In feature search operations, the total number of tasks is also S.
- *Workload*: time to execute a task.

1. Task Partitioning

We assume that the input tokens are already replicated on each processor. The objective of load balancing is to assign the tasks to the processors such that total workload is distributed evenly across the processors.

A possible strategy for balancing total workload is to estimate the workload and to distribute the estimated workload across the processors at the beginning of the computation. We can estimate the workload of each task by computing the size of the search window. Then, we partition the list

of total tasks such that the workload assigned to each processor is nearly the same. This strategy is similar to other algorithmic approaches [7]-[13] in that all the processors participate in the initial load balancing step.

We denote a parallel search algorithm using this typical task partitioning strategy as the *Task Partitioning* algorithm. The load balancing step can be performed in $O(S)$ time. After that step, each processor performs the assigned tasks independently of the other processors. This algorithm can be performed in $O(W(S)/P)$ time, $1 \leq P \leq W(S)/S$. Although it is scalable, the actual execution time also depends on the shape of the search window and the number of tokens within the window. Therefore, we need to consider an additional step which can balance the unpredictable workload on the fly.

2. Task Migration

To balance the unpredictable workload after the initial task partitioning, we need to consider task migration from a heavily loaded processor to a lightly loaded processor. However, efficient task migration requires that several issues be addressed.

The first issue is to decide whether we can use a barrier synchronization to perform task migration. In most load balancing algorithms, the computational structures of the applications consist of a sequence of iterations. Such workloads can be balanced by assigning equal number of data to each processor. After each iteration, a barrier synchronization is performed and data distribution for the next iteration is collected. Using this information, the workload of the next iteration can be redistributed across the processors.

However, in this approach, if a processor finishes earlier than others, then it remains idle until all the other processors complete their iteration. In addition to this idle time, the computational structure of the feature search operation is non-iterative and its workload is unpredictable: profile information for tasks that have already been executed does not help in predicting the execution times of the remaining tasks. To balance the workload of the non-iterative and unpredictable search operations, each processor performs task migration *asynchronously*. Also, to make our algorithm scalable and robust, we use a decentralized policy, rather than a centralized manager-based one.

The second issue is to determine whether a processor is in a suitable state to participate in a task migration. In general, task migration is performed between a heavily loaded processor and a lightly loaded processor. In solving the feature search operation, however, classification of a processor into a heavily/moderately/lightly-loaded category can change over time (due to the unpredictable nature of the evolving workload). Task migration based on such classification could cause unnecessary and repeated migrations. For example, if a

task migration is performed from a heavily loaded processor to a lightly loaded processor, then the transfer of the task may cause the lightly loaded processor to become a heavily loaded processor. This necessitates transfer of that task to another processor, and this *processor thrashing* may repeat indefinitely.

Instead of using a classification that varies with time, we choose a *busy-idle* classification of processors. In this strategy, a task migration is performed when a processor becomes idle and the processor thrashing problem can be avoided. Let *taker* denote an idle processor taking over a task initially assigned to some other processor. Let *giver* denote the busy processor transferring its task to the taker. Since a taker in our strategy is otherwise idle, any taken-over task is guaranteed to be executed by the taker.

The third issue is to determine the unit of a task migration. Since communicating a message is expensive on distributed-memory machines, we need to consider the overhead in transferring task(s). Also, variations in the number of tasks assigned to each processor make SPMD (Single Program Multiple Data) style of programming difficult.

To manage these problems, we define a flexible notion of a group of tasks assigned to a processor and call this a *supertask*. In our strategy, the unit of a task migration is a supertask rather than a task. A supertask is made non-preemptable since transferring a partially executed supertask is expensive. Tasks are grouped such that each processor holds the same number of supertasks; programming is simplified. The number of supertasks can be determined by a programmer considering the number of processors and the number of total tasks. For instance, we can group the tasks initially assigned to a processor into three supertasks as following (see Fig. 3):

- A *lower* supertask consists of tasks such that the sum of the search areas of the tasks is nearly equal to 0.2 $W(S)/P$.
- A *middle* supertask consists of tasks such that the sum of the search areas of the tasks is nearly equal to 0.6 $W(S)/P$.
- An *upper* supertask consists of tasks such that the sum of the search areas of the tasks is nearly equal to 0.2 $W(S)/P$.

The fourth issue is to decide the possible partners for a task migration. For simplicity, we restrict possible partners of a processor to its neighbors. However, our strategy can be extended easily to non-neighbor partners. As in the Task Partitioning algorithm, the list of total tasks is partitioned into P blocks and each block is assigned to a processor. Therefore, for each processor, we denote a *lower* and an *upper* neighbor as the processor having the preceding and the subsequent block of tasks, respectively. We consider the list of total tasks as a circular list. P_0 becomes an upper neighbor for P_{P-1} and P_{P-1} becomes a lower neighbor for P_0 (see Fig. 3).

The fifth issue is how to recognize the states of the neighbors. A possible solution

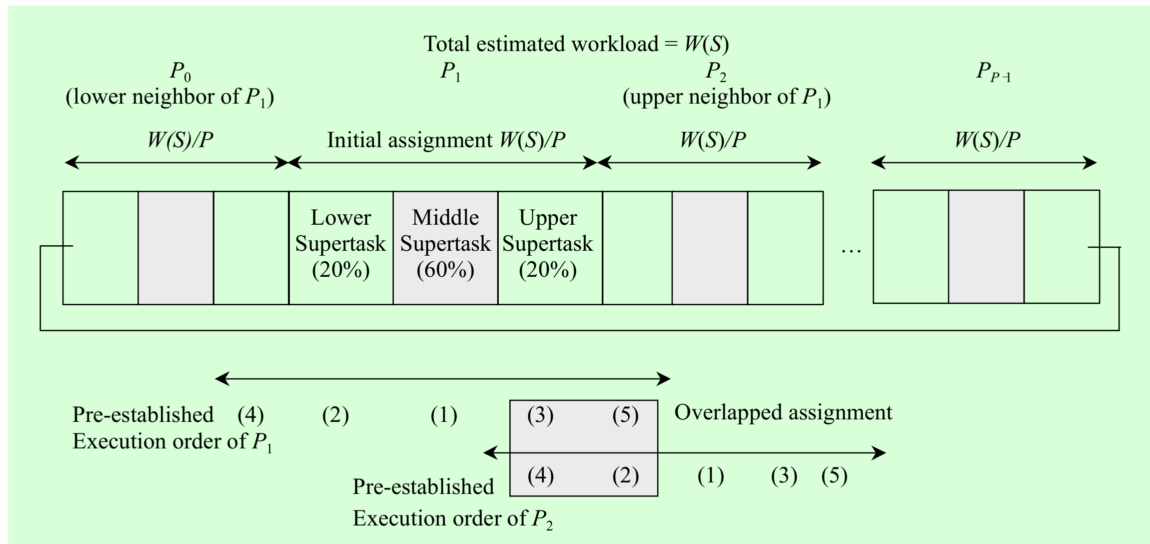

Fig. 3. An example illustrating the notation.

is to let an idle processor poll its neighbors. However, in a message-passing programming style, collecting status information of neighbors requires two-way communication. Furthermore, processors should check for the non-deterministic arrival of the polling messages continuously for a quick response. This continuous message checking causes additional overhead. To reduce these overheads, we let each processor disseminate its state to neighbors after completing a supertask. Since each processor keeps the complete list of tasks, it sends an index of a completed supertask to its neighbors. We call this message a *progress message*. By checking these messages, each processor can recognize the current state of its neighbors.

Finally, some processor has to decide which supertask is to be migrated. In general, once the pair of processors for each task migration is decided, the giver sends the task(s) to be migrated to the taker. This policy causes overhead in transferring the supertask. To reduce this overhead, we can also exploit the facts that each processor keeps the complete list of tasks and additional tasks are not created dynamically during the computation.

Our mapping between supertasks and processors is not one-to-one; some supertasks, called *overlapped supertasks*, are also assigned to a neighbor processor. For instance, each lower supertask is assigned to both a processor and its lower neighbor, and an upper supertask is assigned to a processor and its upper neighbor. We also assign an execution priority (lower number indicates higher priority in Fig. 3) to each supertask. The middle supertask has the

Procedure: A Task Migration Algorithm for Irregular Problems

Step 1: Post non-blocking receives.
Step 2: Partition the list of tasks.
Step 3: Send "done" message and perform assigned tasks.
Step 4: If initially assigned tasks are completed, go to Step 6.
Step 5: Check termination condition.
If FALSE (no "takeover" message), go to Step 3.
Step 6: Check termination condition.
If FALSE (no "done" message), go to Step 7.
Step 7: Send "takeover" message and perform taken-over tasks
Which were initially assigned to neighbors in Step 2. Go to Step 6.

Fig. 4. An outline of the Task Migration algorithm.

highest priority. Usually, a processor having a higher priority executes an overlapped supertask, but a taker (having a lower priority) can execute it depending on its workload. Note that a task migration is performed whenever a taker executes an overlapped supertask.

The main advantage of our method is that both a giver and a taker can execute any overlapped supertask uniformly regardless of the initially assigned priority. Before executing any overlapped supertask, each processor only needs to check for the progress messages to determine whether it has already been executed by its neighbors. We use MPI non-blocking commands (MPI_Isend, MPI_Irecv, MPI_Test) to communicate asynchronously. Another feature of our strategy is to preserve the initial ordering of the task indices across the processors such as $start_task_id(P_i) = [end_task_id(P_{i-1})+1] \ mod \ S$. This feature is important when neighboring tasks need to be executed by a processor.

We show a parallel search algorithm using such task migration in Fig. 4 and denote it as the *Task Migration* algorithm. Its state diagram is shown in Fig. 5. Let S and S' denote the total number of tasks and the total number of supertasks initially assigned to a processor, respectively. We assume $S' \ll S$.

In Step 1, we post non-blocking receives for possible "done" and "takeover" messages. This can be performed in $O(S')$ time. In Step 2, we partition the list of total tasks such that each processor holds the same number of supertasks. This can be performed in $O(S)$ time. According to the pre-established execution order of the assigned supertasks, each processor performs tasks in a supertask and sends a "done" message to its neighbors (Step 3). For each supertask initially assigned to a processor, each processor checks whether a "takeover" message for that supertask was received from a neighbor (Step 5). If so, the processor terminates. When a processor completes the initially assigned supertasks, it checks whether a "done" message was received from neighbors (Step 6). If so, the

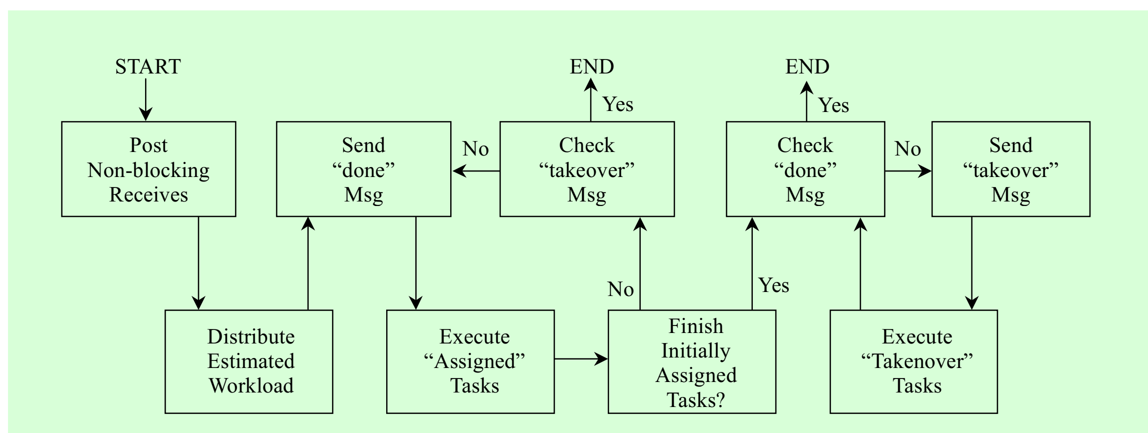

Fig. 5. State diagram of the Task Migration algorithm.

processor terminates. Otherwise, it sends a "takeover" message to a neighbor and executes the taken-over supertask as a taker (Step 7).

Note that, due to the non-zero communication time for the "done" message, there is a possibility for two processors to execute the same supertask. To minimize such duplication, the taker checks for the "done" message after completing each task and quits if such a message has been received. Since any taker would stay idle, total execution time does not increase in spite of some duplicated computation in the taker. The check operations and the non-blocking send operations in Step 3-7 can be performed in $O(S')$ time. The algorithm executes in $O(W(S)/P)$ time, $1 \leq P \leq W(S)/S$. Note that the worst case complexity of the computation time of the Task Migration algorithm is the same as that of the Task Partitioning algorithm. Furthermore, additional communication times caused by constant numbers of the progress messages can be included in the proposed algorithm. In practice, however, the proposed algorithm can speed-up the execution time significantly. Compared with the Task Partitioning algorithm, both the performance improvement and the overhead of the proposed algorithm caused by migrating tasks from a busy processor to an idle neighbor are discussed in the next section.

IV. EXPERIMENTAL RESULTS

The Task Partitioning algorithm and the Task Migration algorithm were implemented on an IBM SP2 [16] and a Cray T3D [17]. The code was written using C and MPI message passing library. In Fig. 6, we show the input line segments extracted from two 1024×1024 Modelboard images. For each line segment, a search was performed within the region on both sides of

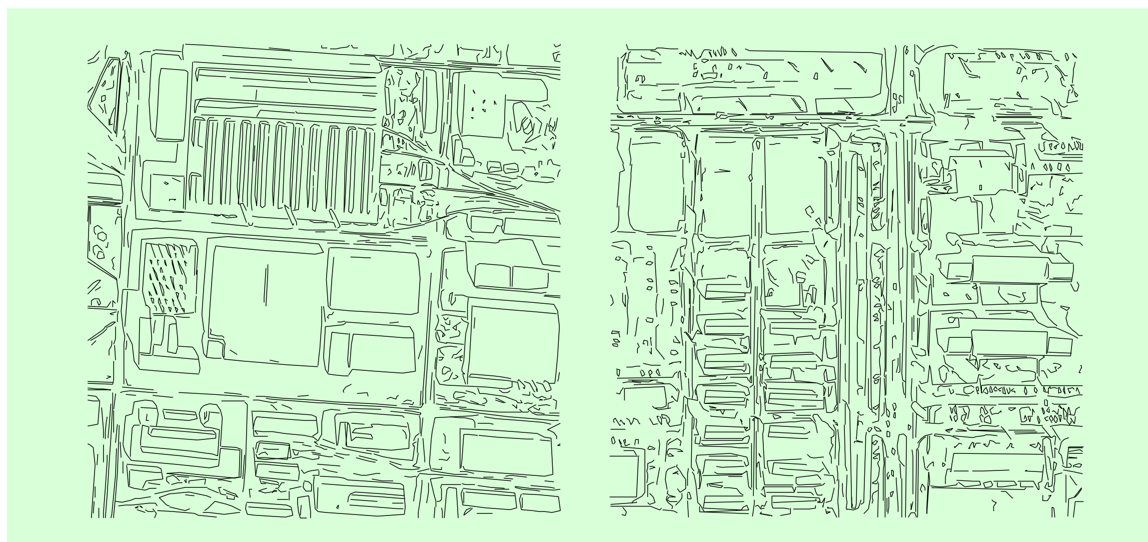

Fig. 6. Extracted line segments from 1024×1024 Modelboard1(left) and Modelboard2(right) images. The numbers of input line segments from Modelboard1 and Modelboard2 were 3519 and 7825, respectively.

it within a 4-pixels width to find other line segments, using the algorithm in Fig. 2. As we discussed in Section II, we did not target any specific application. Thus, a dummy loop was used to simulate the computation time t for comparing the load balancing algorithms. Note that, t denote the amount of computation needed by a processor after finding each line segment. Since the number of line segments found within a search window is unknown in advance, we can generate unpredictable workload.

To show the effect of the number of supertasks in the Task Migration algorithm, our experiments were conducted with three supertasks and nine supertasks per processor, separately. In the experiment with three supertasks, tasks assigned to a processor were grouped into one lower supertask $(0.2\,W(S)/P)$, one middle supertask $(0.6\,W(S)/P)$, and one upper supertask $(0.2\,W(S)/P)$. In the experiment with nine supertasks, tasks were grouped into four lower supertasks $(0.1\,W(S)/P)$, one middle supertask $(0.2\,W(S)/P)$, and four upper supertasks $(0.1\,W(S)/P)$. The "takeover" and "done" messages were implemented as null messages with different "tag" fields.

In Fig. 7 and 8, we show the speed-ups for a fixed problem as a function of the number of processors used. Note that the speed-up used in this paper is defined by the ratio of the execution time of the best sequential implementation and that of the parallel implementation, given a fixed input. The speed-up of the Task Partitioning algorithm is sensitive to the input image and the number of processors used. The Task Migration algorithm provides superior performance.

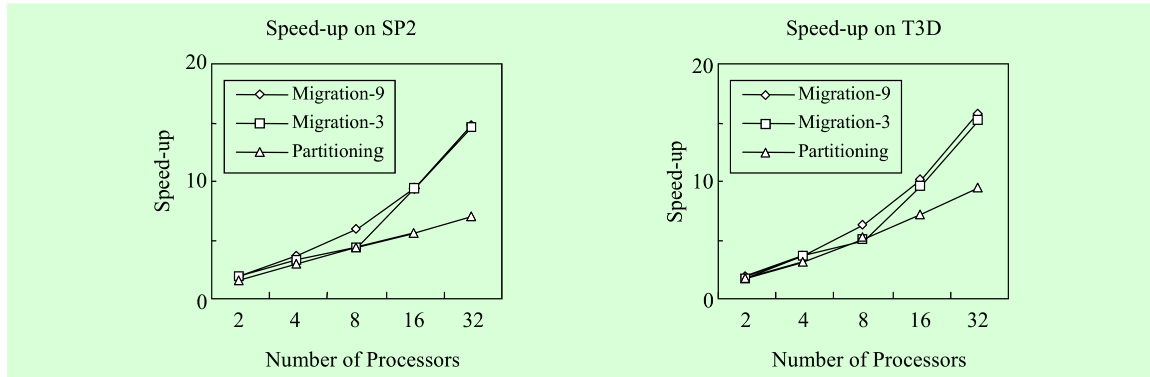

Fig. 7. Speed-ups for an 1024×1024 Modelboard1 image.

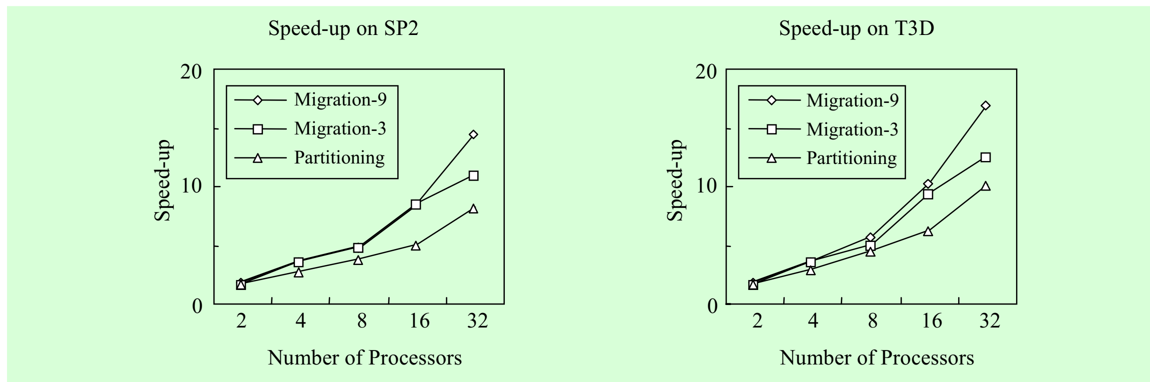

Fig. 8. Speed-ups for an 1024×1024 Modelboard2 image.

Fig. 9 shows histograms of the distribution of the execution times of 32 processors to illustrate the detailed effect of proposed load balancing strategy; more compact distributions indicate a better balanced load on the processors. The performance of the Task Partitioning algorithm is seriously affected by a few overloaded processors. This unbalanced workload was distributed more evenly in the Task Migration algorithm by migrating tasks over time. For instance, 106 and 113 (4083 and 4127) tasks were migrated on a 32-processor T3D for Modelboard1 (Modelboard2) with three and nine supertasks, respectively. Although we show the histograms obtained from a T3D, very similar shapes of the histograms were obtained from an SP2.

To evaluate the overhead of the proposed load balancing algorithm caused by task migrations, the following notation is used. Let T_{avg} denote $(\sum_{0 \le i < P} T_i)/P$, where T_i represents the execution time of processor P_i to perform the assigned search opera-

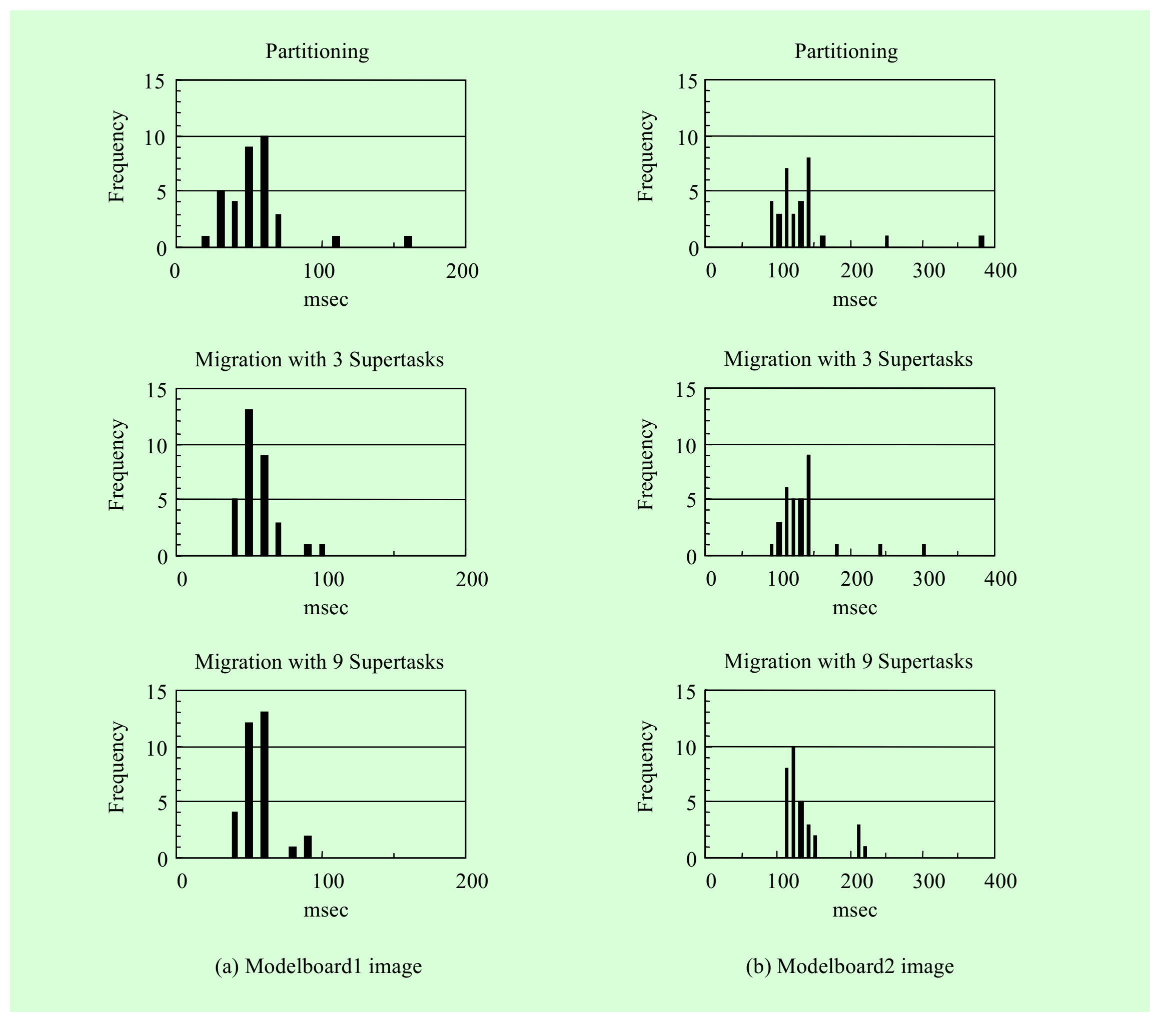

Fig. 9. Distribution of execution time on a 32-processor T3D.

tions. Then, the overhead of the Task Migration algorithm to maintain supertasks can be evaluated by comparing T_{avg} of the Task Migration algorithm with that of the Task Partitioning algorithm. T_{avg} of the Task Migration algorithm was larger than that of the Task Partitioning algorithm by less than 10%. Especially, the communication overhead caused by constant numbers of progress messages (implemented as null messages with meaningful tag fields) was less than 1 msec.

V. CONCLUDING REMARKS

We have presented an asynchronous algorithm for balancing unpredictable work-

load, and have shown the results of implementation on an IBM SP2 and a Cray T3D.

We first partition the list of total tasks such that the estimated workload is distributed evenly across the processors. In addition, each processor sends progress messages to neighbors to let them recognize its workload asynchronously. By maintaining the supertasks and deciding a suitable processor for them based on the progress messages, we can balance the unpredictable workload on the fly.

The experimental results are very encouraging. Our task migration strategy adapts itself to the evolving workload with very little overhead. Our code written in C and MPI permits portability to various distributed-memory machines. We believe that the task migration strategy offers a general framework to parallelize many irregular problems having unpredictable workload.

REFERENCES

[1] S. Ranka, ed., *Proc. of Workshop on Solving Irregular Problems on Distributed-Memory Machines, IPPS'96*, 1996.

[2] A. Ferreira and J. Rolim, ed., *Proc. of Workshop on Parallel Algorithms for Irregularly Structured Problems, Irregular'96,* 1996.

[3] A. Ferreira and J. Rolim, ed., *Proc. of Workshop on Solving Irregularly Structured Problems in Parallel, Irregular'97*, 1997.

[4] C. Wang, P. Bhat, and V. Prasanna, "High Performance Computing for Vision," *Proceedings of IEEE,* Vol. 84, No. 7, 1996, pp. 931–946.

[5] A. Huertas, C. Lin, and R. Nevatia, "Detection of Buildings from Monocular Views of Aerial Scenes Using Perceptual Grouping and Shadows," *Proc. of Image Understanding Workshop*, 1993, pp. 253–260.

[6] G. Medioni and R. Nevatia, "Matching Images Using Linear Features," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. 6, No. 6, 1984, pp. 675–685.

[7] V. Prasanna and C. Wang, "Scalable Parallel Implementations of Perceptual Grouping on Connection Machine CM-5," *Proc. of International Conference on Pattern Recognition*, 1994, pp. 229–233.

[8] D. Bader and J. JaJa, "Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection," *Proc. of International Parallel Processing Symposium*, 1996, pp. 292–301.

[9] A. Choudhary, B. Narahari, and R. Krishnamurti, "An Efficient Heuristic Scheme for Dynamic Remapping of Parallel Computations," *Parallel Computing*, Vol. 19, 1993, pp. 621–632.

[10] N. Chrisochoides, N. Mansour, and G. Fox, "Performance Evaluation of Load Balancing Algorithms for Parallel Single-Phase Iterative PDE Solvers," *Proc. of Scalable High-Performance Computing Conference*, 1994, pp. 764–772.

[11] I. Foster and B. Toonen, "Load Balancing Algorithms for the Parallel Community Climate Model," *Proc. of Scalable High-Performance Computing Conference*, 1994.

[12] D. Gerogiannis and S. Orphanoudakis, "Load Balancing Requirements in Parallel Implementations of Image Feature Extraction Tasks," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 9, 1993, pp. 994–1013.

[13] B. VanVoorst, R. Jha, L. Pires, and M. Muhammad, "Implementation and Results of Hypothesis Testing from the C3I Parallel Benchmark Suite," *Proc. of Parallel Processing Symposium*, 1997, pp. 192–196.

[14] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Proc. of Symposium on Principles and Practice of Parallel Programming*, 1995.

[15] C. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, and K. Yelick, "Runtime Support for Portable Distributed Data Structures," *Proc. of Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, 1995, pp. 111−120.

[16] T. Agerwala, J. Martin, J. Mirza, D. Sadler, D. Dias, and M. Snir, "SP2 System Architecture," *IBM Systems Journal,* Vol. 34, No. 2, 1995, pp. 152−184.

[17] Cray Research, Inc., *Cray T3D System Architecture Overview,* 1993.

Yongwha Chung received his B.S. and M.S. degrees from Hanyang University, Korea in 1984 and 1986, respectively. He received his Ph.D. degree from the University of Southern California, USA in 1997. He joined ETRI in 1986 and he is the senior member of technical staff in Performance Evaluation Team. His research interests include computer architecture, parallel algorithm, distributed processing, and multimedia computing.

Jin-Won Park received his B.S. degree from Seoul National University, Korea in 1975. He received his M.S. and Ph.D. degrees from the Ohio State University, USA in 1982 and 1987, respectively. He joined ETRI in 1988 and he is currently working as the head of Performance Evaluation Team. His research interests include computer architecture, parallel algorithm, distributed processing, and multimedia computing.

Suk-Han Yoon
See this issue, p. 325.