Tree-Based Parallel Load-Balancing Methods for Solution-Adaptive Finite Element Graphs on Distributed Memory Multicomputers

Ching-Jung Liao, *Member*, *IEEE Computer Society*, and
Yeh-Ching Chung, *Member*, *IEEE Computer Society*

Abstract—To solve the load imbalance problem of a solution-adaptive finite element application program on a distributed memory multicomputer, nodes of a refined finite element graph can be remapped to processors or load of a refined finite element graph can be redistributed based on the current load of each processor. For the former case, remapping can be performed by some fast mapping algorithms. For the latter case, a load-balancing algorithm can be applied to balance the computational load of each processor. In this paper, three tree-based parallel load-balancing methods, the MCSTLB method, the BTLB method, and the CBTLB method, were proposed to deal with the load imbalance problems of solution-adaptive finite element application programs. To evaluate the performance of the proposed methods, we have implemented those methods along with three mapping methods, the AE/ORB method, the AE/MC method, and the ML*k*P method, on an SP2 parallel machine. Three criteria, the execution time of mapping/load-balancing methods, the execution time of a solution-adaptive finite element application program under different mapping/load-balancing methods, and the speedups achieved by mapping/load-balancing methods for a solution-adaptive finite element application program, are used for the performance evaluation. The experimental results show that 1) if the initial mapping is performed by a mapping method and the same mapping method and load-balancing methods were used in each refinement to balance the load of processors, the execution time of an application program under a load-balancing method is always shorter than that of the mapping method, and 2) the execution time of an application program under the CBTLB method is shorter than that of the BTLB method and the MCSTLB method.

Index Terms—Distributed memory multicomputers, partitioning, mapping, load balancing, solution-adaptive finite element graphs.

1 Introduction

The finite element method is widely used for the structural modeling of physical systems. In the finite element model, an object can be viewed as a finite element graph, which is a connected and undirected graph that consists of a number of finite elements. Each finite element is composed of a number of nodes. The number of nodes of a finite element is determined by an application. Due to the properties of computation-intensiveness and computation-locality, it is very attractive to implement the finite element method on distributed memory multicomputers [1], [11], [33], [36], [37]. In the context of parallelizing a finite element application program that uses iterative techniques to solve system of equations [2], a parallel program may be viewed as a collection of tasks represented by nodes of a finite element graph. Each node represents a particular amount of computation and can be executed independently. In each iteration, a node performs its computation and needs to get data from other nodes in the same finite element before the next iteration can be performed.

To efficiently execute a finite element application program on a distributed memory multicomputer, we need to map nodes of the corresponding finite element graph to processors of a distributed memory multicomputer such that each processor has approximately the same amount of computational load and the communication among processors is minimized. Since this mapping problem is known to be NP-complete [12], many heuristic methods were proposed to find satisfactory suboptimal solutions [3], [4], [8], [10], [13], [14], [17], [18], [22], [23], [24], [33], [36].

If the number of nodes of a finite element graph do not increase during the execution of a finite element application program, the mapping algorithm only needs to be performed once. For a solution-adaptive finite element application program, the number of nodes increases discretely due to the refinement of some finite elements during the execution. This may result in load imbalance of processors. A node remapping or a load-balancing algorithm has to be performed many times in order to balance the computational load of processors while keeping the communication cost among processors as low as possible. For the node remapping approach, some mapping algorithms can be used to partition a finite element graph from scratch. For the load balancing approach, some load-balancing algorithms can be used to perform the load balancing process according to the current load of processors. Since node remapping or load-balancing algorithms are performed at run-time, their execution must be fast and efficient.

• *The authors are with the Department of Information Engineering, Feng Chia University, Taichung, Taiwan 407, R.O.C.*
E-mail: {cjliao, ychung}@pine.iecs.fcu.edu.tw.

Manuscript received 16 July 1997.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 105388.

1045-9219/99/$10.00 © 1999 IEEE

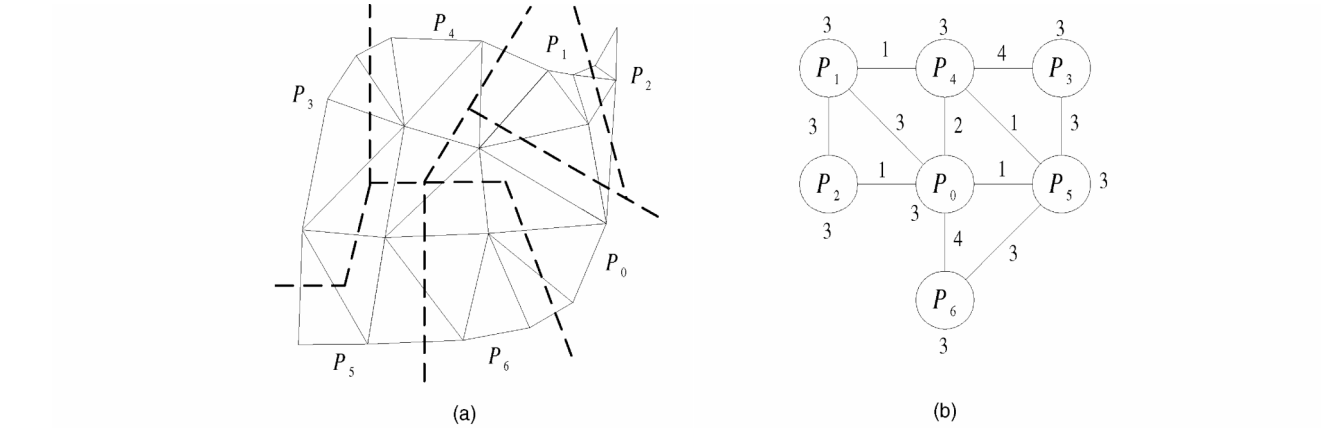

Fig. 1. (a) A partition of a finite element graph on seven processors. (b) The corresponding processor graph of Fig. 1a.

In this paper, we propose three tree-based parallel load-balancing methods to efficiently deal with the load imbalance problems of solution-adaptive finite element application programs on distributed memory multicomputers. They are the *maximum cost spanning tree load-balancing* (MCSTLB) method, the *binary tree load-balancing* (BTLB) method, and the *condensed binary tree load-balancing* (CBTLB) method. When nodes of a solution-adaptive finite element graph were evenly distributed to processors by some mapping algorithms, according to the communication property of the finite element graph, we can get a processor graph from the partition. For example, Fig. 1a shows a partition of a 21-node finite element graph on seven processors. The corresponding processor graph of Fig. 1a is shown in Fig. 1b. In a processor graph, nodes represent the processors and edges represent the communication needed among processors. The weights associated with nodes and edges denote the computation and the communication costs, respectively.

When a finite element graph is refined during run-time, it will result in load imbalance of processors. To balance the computational load of processors, the MCSTLB method first finds the maximum cost spanning tree from the processor graph. Based on the maximum cost spanning tree, the global load balancing information is calculated by the tree walking algorithm (TWA) [38]. According to the global load balancing information and the current load distribution, a load transfer algorithm is performed to balance the computational load of processors and minimize the communication cost among processors. For the BTLB method, a binary tree is obtained from the processor graph. The global load balancing information calculation and the load transfer method are the same as those of the MCSTLB method. In the CBTLB method, nodes of the processor graph are first grouped into metaprocessors. Each metaprocessor is a hypercube. We call the grouped processor graph as a *condensed processor graph*. Then the CBTLB method finds a binary tree from the condensed processor graph. Based on the binary tree, the global load balancing information is calculated by a similar TWA method. According to the global load balancing information and the current load distribution, a load transfer algorithm is performed to balance the computational load of metaprocessors and minimize the communication cost among metaprocessors. After the load transfer algorithm is performed, a *dimension exchange method* (DEM) [7], [39] is performed to balance the computational load of processors in a metaprocessor.

To evaluate the performance of the proposed methods, we have implemented these methods along with three mapping methods, the AE/ORB method [6], the AE/MC method [6], and the ML*k*P method [22], on an SP2 parallel machine. The finite element graph *Truss* is used as the test sample. The experimental results show that 1) if the initial mapping is performed by a mapping method and the same mapping method and load-balancing methods were used in each refinement to balance the computational load of processors, the execution time of an application program under a load-balancing method is always shorter than that of the mapping method, and 2) the execution time of an application program under the CBTLB method is shorter than that of the BTLB method and the MCSTLB method.

The paper is organized as follows: The relative work will be given in Section 2. In Section 3, the proposed tree-based parallel load-balancing methods will be described in details. In Section 4, we will present the cost model of mapping/load-balancing methods for finite element graphs on distributed memory multicomputers. The performance evaluation and experimental results will also be presented in this section.

2 Related Work

Many methods have been proposed to deal with the load imbalance problems of solution-adaptive finite element application programs on distributed memory multicomputers in the literature. They can be classified into two categories, the remapping methods and the load redistribution methods. The remapping methods, in general, can be divided into five classes, the *orthogonal section* approach [6], [21], [31], [34], the *min-cut* approach [6], [8], [10], [13], [24], the *spectral* approach [3], [4], [17], [33], the *multilevel* approach [3], [4], [18], [22], [23], and miscellaneous approaches [14], [26], [28], [36]. These methods were implemented in several graph partition libraries, such as Chaco [16], DIME [37], JOSTLE [34], METIS [23], ParMetis [31], PARTY [29], Scotch [27], and TOP/DOMDEC [9], etc., to solve graph partition problems.

For the load redistribution methods, many load-balancing algorithms have been proposed in the literature. In [35], a recent comparison study of dynamic load balancing strategies on highly parallel computers is given. The dimension exchange method (DEM) is applied to application programs without geometric structure [7]. It is conceptually designed for a hypercube system but may be applied to other topologies, such as *k*-ary *n*-cubes [39]. Ou and Ranka [26] proposed a linear programming-based method to solve the incremental graph partition problem. Wu [38] proposed the tree walking, the cube walking, and the mesh walking run-time scheduling algorithms to balance the load of processors on tree-based, cube-based, and mesh-based paradigms, respectively. Diffusion based load-balancing methods were proposed in [7], [19], [20], [31], [32], [34].

3 The Parallel Load Balancing Methods

3.1 The Maximum Cost Spanning Tree Load-Balancing (MCSTLB) Method

The main idea of the MCSTLB method is to find a maximum cost spanning tree from the processor graph that is obtained from the initial partitioned finite element graph. Based on the maximum cost spanning tree, it tries to balance the load of processors. The MCSTLB method can be divided into the following four phases:

Phase 1: Obtain a processor graph G from the initial partition.

Phase 2: Use a similar *Kruskal's* [25] algorithm to find a maximum cost spanning tree $T = (V, E)$ from G, where V and E denote the processors and edges of T, respectively. There are many ways to determine the shape of T. In this method, the shape of T is constructed as follows:

1) The processor with the largest degree in V is selected as the root of T.
2) For each nonterminal processor v in T, if $\{u_1, \ldots, u_m\}$ are the m children of v and $|u_1| \leq |u_2| \leq \ldots \leq |u_m|$, then u_1 will be the leftmost child of v, u_2 will be the second leftmost child of v, and so on, where $|u_i|$ is the degree of u_i and $i = 1, \ldots, m$.

If the depth of T is greater than $\log M$, where M is the number of processors, we will try to adjust the depth of T. The adjusted method is first to find the longest path (from a terminal processor to another terminal processor) of T. After the longest path is determined, the middle processor of the path is selected as the root of the tree and reconstruct the tree according to the above construction process. If the depth of the reconstructed tree is less than that of T, the reconstructed tree is the desired tree. Otherwise, T is the desired tree. The purpose of the adjustment is trying to reduce the load balancing steps among processors.

Phase 3: Calculate the global load balancing information and schedule the load transfer sequence of processors by using the TWA [38]. Assume that there are M processors in a tree and N nodes in a refined finite element graph. We define N/M as the average weight of a processor. In the TWA method, the *quota* and the *load* of each processor in a tree are calculated, where the quota is the sum of the average weights of a processor and its children processors and the load is the sum of the weights of a processor and its children processors. The difference of the quota and the load of a processor is the number of nodes that a processor should send to or receive from its parent. If the difference is negative, a processor should send nodes to its parent. Otherwise, a processor should receive nodes from its parent. According to the global load balancing information, a schedule can be determined.

Phase 4: Perform load transfer (send/receive) based on the global load balancing information, the schedule, and T. The main purposes of load transfer are balancing the computational load of processors and minimizing the communication cost among processors. Assume that processor P_i needs to send m nodes to processor P_j and let N denote the set of nodes in P_i that are adjacent to those of P_j. In order to keep the communication cost as low as possible, in the load transfer, nodes in N are transferred first. If $|N|$ is less than m, then nodes adjacent to those in N are transferred. This process is continued until the number of nodes transferred to P_j is equal to m. We now give an example to explain the above description.

EXAMPLE 1. An example of the behavior of the MCSTLB method is shown in Fig. 2. Fig. 2a shows an initial partition of a 61-node finite element graph on seven processors by using the AE/ORB method. In Fig. 2a, the number of nodes assigned to processors P_0, P_1, P_2, P_3, P_4, P_5, and P_6 are 8, 9, 9, 8, 9, 9, and 9, respectively. After a refinement, the number of nodes assigned to processors P_0, P_1, P_2, P_3, P_4, P_5, and P_6 are 26, 10, 16, 13, 12, 13, and 10, respectively, as shown in Fig. 2b. To apply the MCSTLB method to balance the load of processors shown in Fig. 2b, the corresponding processor graph is obtained from Fig. 2b and is shown in Fig. 2c. From Fig. 2c, a maximum cost spanning tree can be obtained and is shown in Fig. 2d. The global load balancing information calculated by the TWA is shown in Fig. 2e. From the global load balancing information of Fig. 2e, we can determine the following load transfer sequence.

Step 1: $P_0 \rightarrow P_2$, $P_3 \rightarrow P_4$, $P_5 \rightarrow P_6$;

Step 2: $P_2 \rightarrow P_1$;

Step 3: $P_2 \rightarrow P_3$;

Step 4: $P_2 \rightarrow P_5$.

Fig. 2f shows the load balancing result of Fig. 2b after the load transfer is performed.

3.2 The Binary Tree Load Balancing (BTLB) Method

The BTLB method is similar to the MCSTLB method. The only difference between these two methods is that the MCSTLB method is based on a maximum cost spanning tree to balance the computational load of processors while

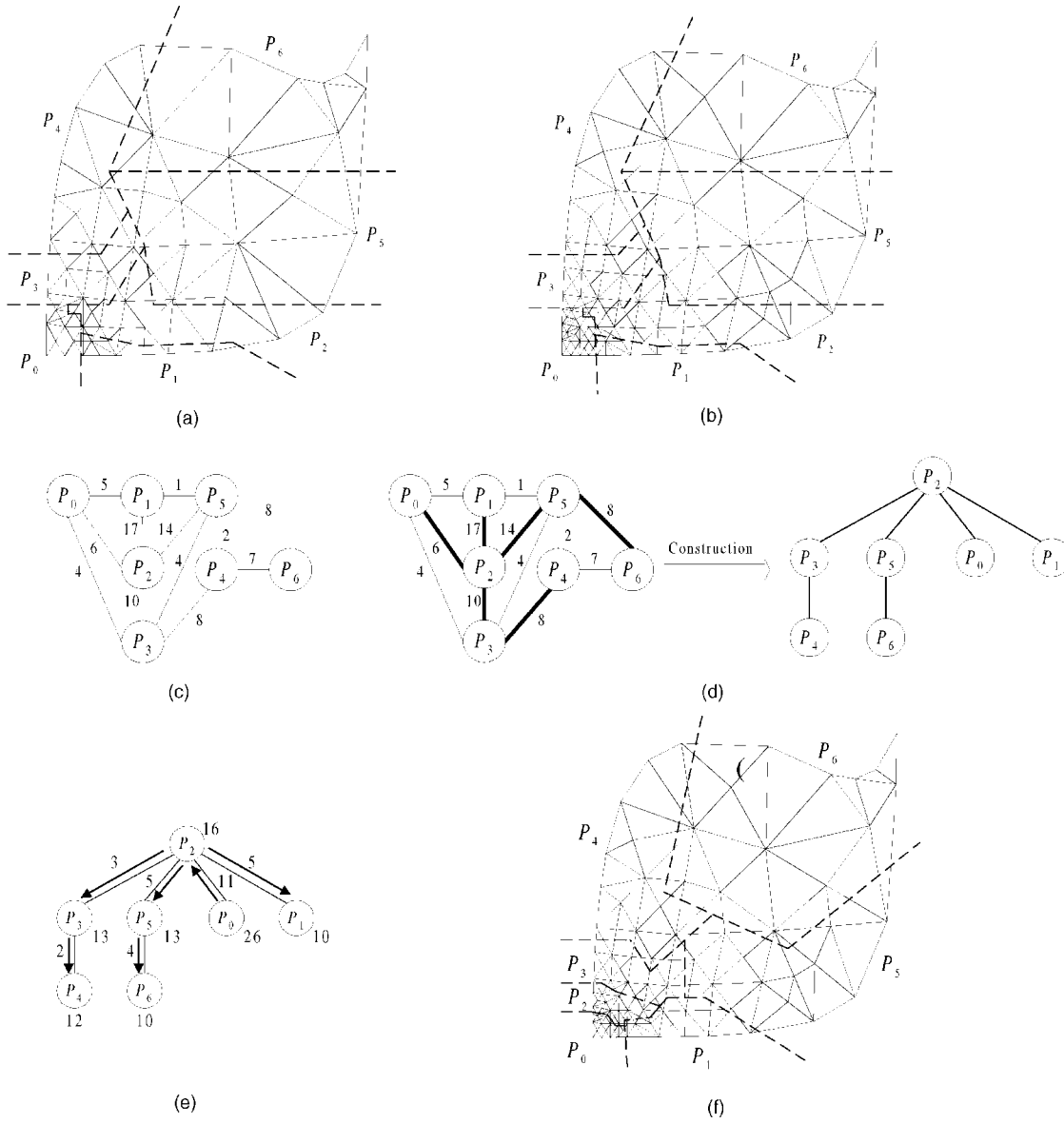

Fig. 2. An example of the behavior of the MCSTLB method. (a) The initial partitioned finite element graph. (b) The finite element graph after a refinement. (c) The corresponding processor graph obtained from (b). (d) The maximum cost spanning tree obtained from (c). (e) The global load balancing information calculated by TWA. (f) The load balancing result of (b) after performing the MCSTLB method.

the BTLB method is based on a binary tree. The BTLB method can be divided into the following four phases:

Phase 1: Obtain a processor graph G from the initial partition.

Phase 2: Use a similar *Kruskal's* algorithm to find a binary tree $T = (V, E)$ from G, where V and E denote the processors and edges of T, respectively. The method to determine the shape of a binary tree is the same as that of the MCSTLB method.

Phase 3: Calculate the global load balancing information and schedule the load transfer sequence of processors by using the TWA.

Phase 4: Perform load transfer (send / receive) based on the global load balancing information, the schedule, and T. The load transfer method is the same as that of the MCSTLB method. We now give an example to explain the behavior of the BTLB method.

EXAMPLE 2. An example of the behavior of the BTLB method is shown in Fig. 3. To apply the BTLB method to balance the load of processors shown in Fig. 2b, the corresponding processor graph is obtained from Fig. 2b and is shown in Fig. 3a. From Fig. 3a, a binary tree can be constructed and is shown in Fig. 3b. The global load balancing information calculated by the TWA is shown in Fig. 3c. From the global load balancing information of Fig. 3c, we can determine the following load transfer sequence.

Step 1: $P_0 \rightarrow P_2$, $P_5 \rightarrow P_4$;

Step 2: $P_2 \rightarrow P_1$, $P_5 \rightarrow P_6$;

Step 3: $P_2 \rightarrow P_3$;

Step 4: $P_3 \rightarrow P_5$.

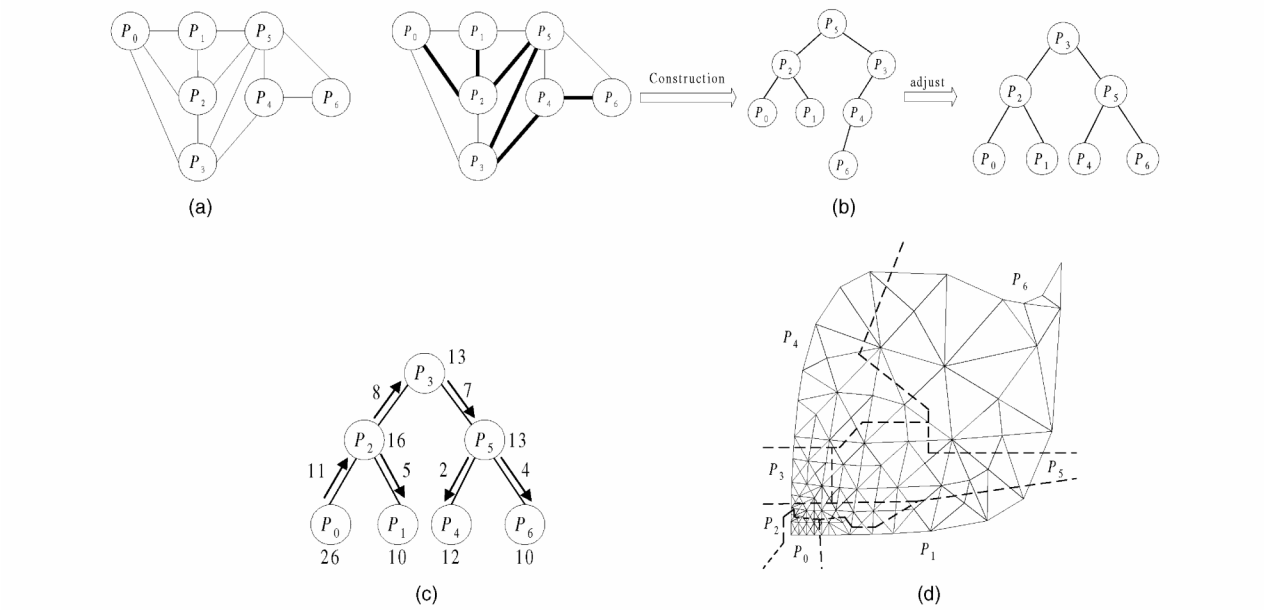

Fig. 3. An example of the behavior of the BTLB method. (a) The corresponding processor graph obtained from Fig. 2b (b) A binary tree constructed from the processor graph. (c) The global load balancing information calculated by TWA. (d) The load balancing result of Fig. 2b after performing the BTLB method.

Fig. 3d shows the load balancing result of Fig. 2b after the load transfer is performed.

3.3 The Condensed Binary Tree Load Balancing (CBTLB) Method

The main idea of the CBTLB method is to group processors of the processor graph into metaprocessors. Each metaprocessor is a hypercube. We call the grouped processor graph as a condensed processor graph. From the condensed processor graph, the CBTLB method constructs a binary tree. Based on the binary tree, the global load balancing information is calculated by a similar TWA method. According to the global load balancing information and the current load distribution, a load transfer algorithm is performed to balance the computational load of metaprocessors and minimize the communication cost among metaprocessors. After the load transfer in performed, a dimension exchange method (DEM) is performed to balance the computational load of processors in a metaprocessor. The CBTLB method can be divided into the following five phases:

Phase 1: Obtain a processor graph G from the initial partition.

Phase 2: Group processors of G into metaprocessors to obtain a condensed processor graph G_c incrementally. Each metaprocessor of G_c is a hypercube. The metaprocessors in G_c are constructed as follows: First, a processor P_i with the smallest degree in G and a processor P_j that is a neighbor processor of P_i and has the smallest degree among those neighbor processors of P_i are grouped into a metaprocessor. Then, the same construction is applied to other ungrouped processors until there are no processors can be grouped into a hypercube. Repeat the grouping process to each metaprocessor until there are no metaprocessors can be grouped into a higher order hypercube.

Phase 3: Find a binary tree $T = (V, E)$ from G_c, where V and E denote the metaprocessors and edges of T, respectively. The method of constructing a binary tree is the same as that of the BTLB method.

Phase 4: Based on T, calculate the global load balancing information and schedule the load transfer sequence by using a similar TWA method for metaprocessors. Assume that there are M processors in a tree and N nodes in a refined finite element graph. We define N/M as the average weight of a processor. To obtain the global load balancing information, the quota and the load of each processor in a tree are calculated. The quota is defined as the sum of the average weights of processors in a metaprocessor C_i and processors in children processors of C_i. The load is defined as the sum of the weights of processors in a metaprocessor C_i and processors in children metaprocessors of C_i. The difference of the quota and the load of a metaprocessor is the number of nodes that a metaprocessor should send to or receive from its parent metaprocessor. If the difference is negative, a metaprocessor should send nodes to its parent metaprocessor. Otherwise, a metaprocessor should receive nodes from its parent metaprocessor. After calculating the global load balancing information, the schedule is determined as follows. Assume that m is the number of nodes that a metaprocessor C_i needs to send to another metaprocessor C_j. We have the following two cases:

Case 1: If the weight of C_i is less than m, the schedule of these two metaprocessors is postponed until the weight of C_i is greater than or equal to m.

Case 2: If the weight of C_i is greater than or equal to m, a schedule can be made between processors of C_i and C_j. Assume that ADJ denotes the set of

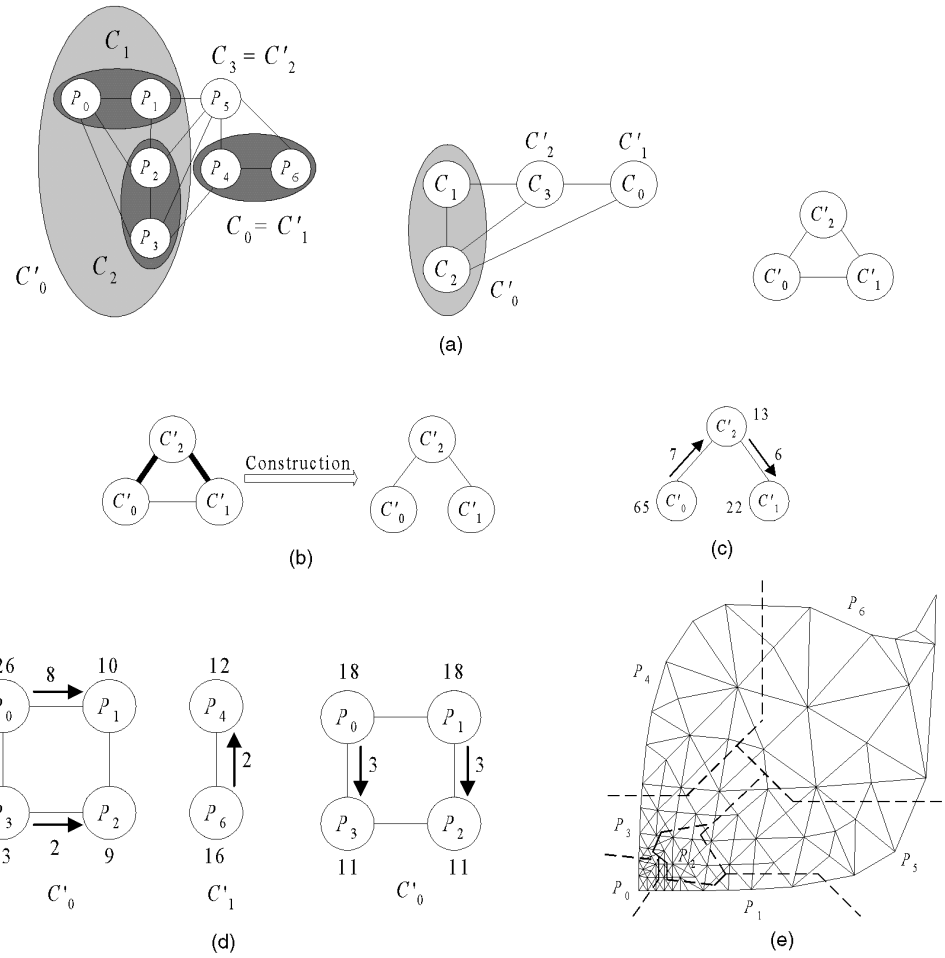

Fig. 4. An example of the behavior of the CBTLB method. (a) The process of constructing a condensed processor graph from Fig. 2b. (b) A binary tree constructed from the condensed processor graph. (c) The global load balancing information of the condensed binary tree. (d) The load transfer process in each metaprocessor by using the DEM method. (e) The load balancing result of Fig. 2b after performing the CBTLB method.

processors in C_i that are adjacent to those in C_j. If the sum of the weights of processors in *ADJ* is less than *m*, a schedule is made to transfer nodes of processors in C_i to processors in *ADJ* such that the weights of processors in *ADJ* is greater than or equal to *m*. If the sum of the weights of processors in *ADJ* is greater than or equal to *m*, a schedule is made to send *m* nodes from processors in *ADJ* to those in C_j.

Phase 5: Perform load transfer (send/receive) among metaprocessors based on the global load balancing information, the schedule, and *T*. The load transfer method is similar to that of the BTLB method. After performing load transfer process among metaprocessors, a dimension exchange method (DEM) is performed to balance the computational load of processors in metaprocessors. We now give an example to explain the above description.

EXAMPLE 3. An example of the behavior of the CBTLB method is shown in Fig. 4. Fig. 4a shows the process of constructing a condensed processor graph from Fig. 2b. In Fig. 4b, a binary tree is constructed from the condensed processor graph. The global load balancing information of the condensed binary tree is shown in Fig. 4c. From the global load balancing information of Fig. 4c, we can determine the following load transfer sequence.

Step 1: $P_2 \rightarrow P_5$;

Step 2: $P_5 \rightarrow P_6$.

Fig. 4d shows the load transfer process in each metaprocessor using the DEM method. The load transfer sequence is give as follows:

Step 1: $P_0 \rightarrow P_1$, $P_3 \rightarrow P_2$, $P_6 \rightarrow P_4$;

Step 2: $P_0 \rightarrow P_3$, $P_1 \rightarrow P_2$.

Fig. 4e shows the load balancing result of Fig. 2b after performing the CBTLB method.

4 Performance Evaluation and Experimental Results

To evaluate the performance of the proposed methods, we have implemented the MCSTLB method, the BTLB method, and the CBTLB method along with three mapping methods, the AE/ORB method [6], the AE/MC method [6], and the ML*k*P method [22], on an SP2 parallel machine. All of the algorithms were written in C with MPI

TABLE 1
THE NUMBER OF NODES AND ELEMENTS OF THE TEST SAMPLE TRUSS

	Sample *Truss*	
Refinement	No. of Nodes	No. of Elements
Initial (0)	18407	35817
1	23570	46028
2	29202	57181
3	36622	71895
4	46817	92101
5	57081	112494

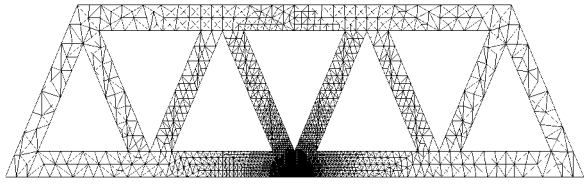

Fig. 5. The test sample *Truss* (7,325 nodes, 14,024 elements).

communication primitives. Three criteria, the execution time of mapping/load-balancing methods, the computation time of an application program under different mapping/load-balancing methods, and the speedups achieved by the mapping/load-balancing methods for an application program, are used for the performance evaluation.

In dealing with the unstructured finite element graphs, the distributed irregular mesh environment (DIME) [37] is used. DIME is a programming environment for doing distributed calculations with unstructured triangular meshes. The mesh covers a two-dimensional manifold, whose boundaries may be defined by straight lines, arcs of circles, or Bezier cubic sections. It also provides functions for creating, manipulating, and refining unstructured triangular meshes. Since the number of nodes in an unstructured triangular mesh cannot exceed 10,000 in DIME, in this paper, we only use DIME to generate the initial test sample. From the initial test graph, we use our refining algorithms and data structures to generate the desired test graphs. The initial test graph used for the performance evaluation is shown in Fig. 5. The number of nodes and elements for the test graph after each refinement are shown in Table 1. For presentation purpose, the number of nodes and the number of finite elements shown in Fig. 5 are less than those shown in Table 1.

To emulate the execution of a solution-adaptive finite element application program on an SP2 parallel machine, we have the following steps: First, read the initial finite element graph. Then, the initial partitioning method, the AE/ORB method, the AE/MC method, or the ML*k*P method, is applied to map nodes of the initial finite element graph to processors. After the mapping, the computation of each processor is carried out. In our example, the computation is to solve Laplaces's equation (Laplace solver). The algorithm of solving Laplaces's equation is similar to that of [1]. Since it is difficult to predict the number of iterations for the convergence of a Laplace solver, we assume that the maximum number of iterations executed by the Laplace solver is 1,000. When the computation is converged, the first refined finite element graph is read. To balance the computational load of processors, the AE/ORB method, the AE/MC method, the ML*k*P method, the MCSTLB method, the BTLB method, or the CBTLB method is applied. After a mapping/load-balancing method is performed, the computation for each processor is carried out. The procedures of the mesh refinement, the load balancing, and the computation processes are performed in turn until the execution of a solution-adaptive finite element application program is completed.

By combining the initial mapping methods and methods for load balancing, there are 20 methods used for the performance evaluation. We defined

M_ϕ = {AE/ORB, AE/MC, ML*k*P, AE/ORB/MCSTLB,
AE/MC/MCSTLB, ML*k*P/MCSTLB,
AE/ORB/BTLB, AE/MC/BTLB, ML*k*P/BTLB,
AE/ORB/CBTLB, AE/MC/CBTLB,
ML*k*P/CBTLB}.

In M_ϕ, AE/ORB means that the AE/ORB method is used to perform the initial mapping and the AE/ORB method is used to balance the computational load of processors in each refinement. AE/ORB/MCSTLB means that the AE/ORB method is used to perform the initial mapping and the MCSTLB method is used to balance the computational load of processors in each refinement.

4.1 The Cost Model for Mapping Solution-Adaptive FEGs on Distributed Memory Multicomputers

To map an N-node finite element graph on a P-processor distributed memory multicomputer, we need to assign nodes of the graph to processors of the multicomputer. There are P^N mappings. The execution time of a finite element graph on a distributed memory multicomputer under a particular mapping/load-balancing method L_i can be defined as follows:

$$T_{par}(L_i) = max\{T_{comp}(L_i, P_j) + T_{comm}(L_i, P_j)\}, \quad (1)$$

where $T_{par}(L_i)$ is the execution time of a finite element application program on a distributed memory multicomputer under L_i, $T_{comp}(L_i, P_j)$ is the computation cost of processor P_j under L_i, and $T_{comm}(L_i, P_j)$ is the communication cost of processor P_j under L_i, where $i = 1, ..., P^N$ and $j = 0, ..., P–1$.

The cost model used in (1) is assuming a synchronous communication mode in which each processor goes through a computation phase followed by a communication phase. Therefore, the computation cost of processor P_j under a mapping/load-balancing method L_i can be defined as follows:

$$T_{comp}(L_i, P_j) = S \times load_i(P_j) \times T_{task}, \quad (2)$$

where S is the number of iterations performed by a finite element method, $load_i(P_j)$ is the number of nodes of a finite element graph assigned to processor P_j, and T_{task} is the time for a processor to execute a task.

In our communication model, we assume that every processor can communicate with all other processors in one step. In general, it is possible to overlap the communication with the computation. In this case, $T_{comm}(L_i, P_j)$ may not always reflect the true communication cost since

it could be partially overlapped with that of the computation. However, $T_{comm}(L_i, P_j)$ can provide a good estimate for the communication cost. Since we use a synchronous communication mode, $T_{comm}(L_i, P_j)$ can be defined as follows:

$$T_{comm}(L_i, P_j) = S \times (\delta \times T_{setup} + \phi \times T_c), \quad (3)$$

where S is the number of iterations performed by a finite element method, δ is the number of processors that processor P_j has to send data to in each iteration, T_{setup} is the setup time of the I/O channel, ϕ is the total number of bytes that processor P_j has to send out in each iteration, and T_c is the data transmission time of the I/O channel per byte.

Let T_{seq} denote the execution time of a finite element graph on a distributed memory multicomputer with one processor. The speedup resulted from a mapping/load-balancing method L_i for an application program is defined as

$$Speedup(L_i) = \frac{T_{seq}}{T_{par}(L_i)}, \quad (4)$$

Let $T_i(L)$ denote the time for the Laplace solver to execute one iteration for the ith refinement of the test finite element graph under a mapping/load-balancing method, where $i = 0, 1, \ldots, 5$ and $L \in M_\phi$. For the presentation purpose, we assume that the initial finite element graph as the 0th refined finite element graph. $T_i(L)$ is defined as follows:

$$T_i(L) = T_{comp}(L, P_j) + T_{comm}(L, P_j), \quad (5)$$

The total execution time of test finite element graphs on a distributed memory multicomputer is defined as follows:

$$T_{total}(L) = T_{exec}(L) + \sum_{i=0}^{5} T_i(L) \times S_i, \quad (6)$$

where $T_{total}(L)$ is the total execution time of the test samples under a mapping/load-balancing method L on a distributed memory multicomputer, $L \in M_\phi$, $T_{exec}(L)$ is the total execution time of a mapping/load-balancing method L for test samples, and S_i is the number of iterations executed by the Laplace solver for the ith refinement. From (6), we can derive the speedup achieved by a mapping/load-balancing method as follows:

$$Speedup(L) = \frac{\sum_{i=0}^{5} Seq_i \times S_i}{T_{exec}(L) + \sum_{i=0}^{5} T_i(L) \times S_i}, \quad (7)$$

where Speedup(L) is the speedup achieved by a mapping/load-balancing L for test samples, $L \in M_\phi$, and Seq_i is the time for the Laplace solver to execute one iteration for the ith refinement of test graphs in sequential.

The maximum speedup achieved by a mapping/load-balancing L can be derived by setting the value of S_i to ∞. In this case, $T_{exec}(L)$ is negligible. We have the following equation:

$$Speedup_{\max}(L) = \frac{\sum_{i=0}^{5} Seq_i}{\sum_{i=0}^{5} T_i(L)}. \quad (8)$$

TABLE 2
THE EXECUTION TIME OF MAPPING/LOAD-BALANCING METHODS FOR THE TEST SAMPLE ON DIFFERENT NUMBERS OF PROCESSORS

Methods	Number of processors			
	10	30	50	70
AE/MC	3166.39	1552.69	697.4	582.9
AE/MC/MCSTLB	50.659	28.061	60.212	59.065
AE/MC/BTLB	47.74	24.329	58.821	58.93
AE/MC/CBTLB	47.359	23.82	58.622	58.394
AE/ORB	8.126	7.35	7.68	7.793
AE/ORB/MCSTLB	4.832	5.132	3.473	3.365
AE/ORB/BTLB	1.583	1.684	2.554	3.337
AE/ORB/CBTLB	1.185	1.309	2.057	2.496
MLkP	4.832	5.868	6.679	6.969
MLkP/ MCSTLB	4.025	5.305	4.076	3.803
MLkP/ BTLB	1.396	1.522	2.576	2.955
MLkP/ CBTLB	1.008	1.11	1.91	2.017

Time unit: seconds

where $Speedup_{\max}(L)$ is the maximum speedup achieved by mapping/load-balancing L and $L \in M_\phi$.

4.2 Comparisons of the Execution Time of Mapping/Load-Balancing Methods

The execution time of different mapping/load-balancing methods for the test sample *Truss* on SP2 with 10, 30, 50, and 70 processors are shown in Table 2. From Table 2, we can observe that the execution time of the proposed load-balancing methods is less than that of the mapping methods. The main reason is that the load-balancing methods use the current load distribution of processors to do the local load transfer task, while the mapping methods need to repartition the finite element graph and redistribute nodes to processors. The overheads of the load-balancing methods are less than those of the mapping methods. For the proposed methods, the execution time of the CBTLB method is less than that of the MCSTLB method and the BTLB method. This is because the CBTLB method can reduce the size of a tree with a large ratio so that the overheads to do the load transfer among metaprocessors are less than those of the MCSTLB method and the BTLB method. Therefore, it can reduce the load transfer time efficiently. The disadvantage of the MCSTLB method is when all of the processors except the root want to send nodes to their parents, the bottleneck will be occurred in the root. The BTLB method can avoid this situation since the degree of the root in a binary tree is two.

4.3 Comparisons of the Execution Time of the Test Sample under Different Mapping/Load-Balancing Methods

The times of a Laplace solver to execute one iteration (computation + communication) for the test sample under different mapping/load-balancing methods on a SP2 parallel machine with 10, 30, 50, and 70 processors are shown in Fig. 6, Fig. 7, Fig. 8, and Fig. 9, respectively. Since we assume a synchronous mode of communication in our model, the total time for a Laplace solver to complete its job is the sum of the computation time and the communication time. From Fig. 6 to Fig. 9, we can see that the execution time of a Laplace solver under the proposed load-balancing methods

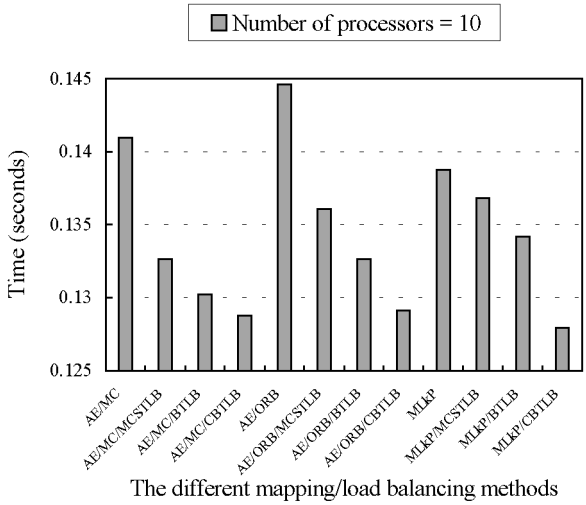

Fig. 6. The time for a Laplace solver to execute one iteration (computation + communication) for the test sample under different mapping/load-balancing methods on 10 processors.

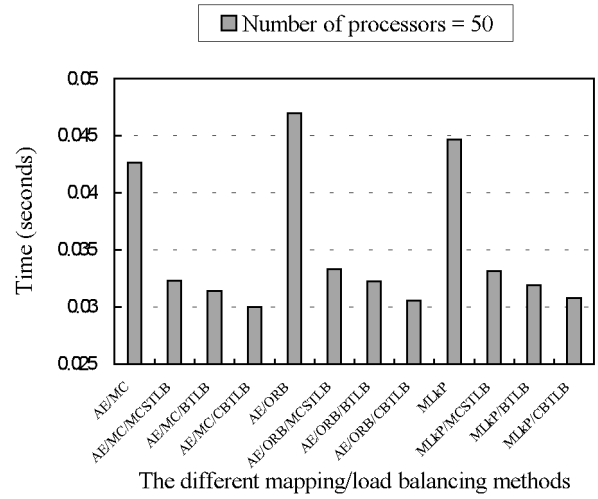

Fig. 8. The time for a Laplace solver to execute one iteration (computation + communication) for the test sample under different mapping/load-balancing methods on 50 processors.

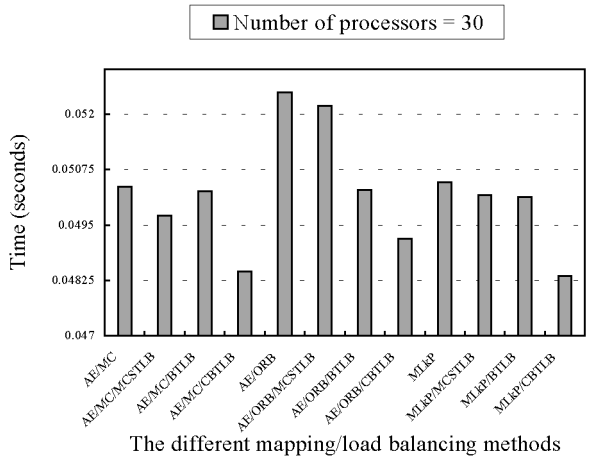

Fig. 7. The time for a Laplace solver to execute one iteration (computation + communication) for the test sample under different mapping/load-balancing methods on 30 processors.

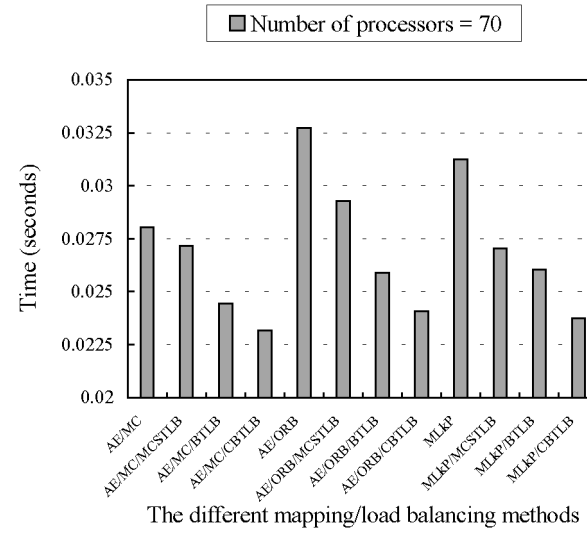

Fig. 9. The time for a Laplace solver to execute one iteration (computation + communication) for the test sample under different mapping/load-balancing methods on 70 processors.

(for example AE/ORB/BTLB) is less than that of their counterparts (AE/ORB). For the proposed methods, the execution time of a Laplace solver under the CBTLB method is less than that of the MCSTLB and the BTLB methods for all cases (Assume that the same initial mapping method is used).

4.4 Comparisons of the Speedups under the Mapping/Load-Balancing Methods for the Test Sample

The speedups and the maximum speedups achieved by the mapping/load-balancing methods with 10, 30, 50, and 70 processors for the test sample are shown in Table 3 and Table 4, respectively. In Table 3, the ML*k*P/CBTLB method, in general, has the best performance among the mapping/load-balancing methods for the test sample. The speedups produced by the AE/MC method and its counterparts (AE/MC/MCSTLB, AE/MC/BTLB, and AE/MC/CBTLB) are much smaller than other mapping/load-balancing methods. The main reason is that the execution of the AE/MC method is time consuming. However, if the number of iterations executed by a Laplace solver is set to ∞, the AE/MC method (its counterparts), in general, produces better speedups than those of AE/ORB (its counterparts) and the ML*k*P (its counterparts) methods for the test sample. We can see this situation from Table 4. Therefore, a fast and efficient mapping/load-balancing method is of great important to deal with the load imbalance problems of solution-adaptive finite element application programs on distributed memory multicomputers.

5 Conclusions

In this paper, we have proposed three tree-based parallel load-balancing methods, the MCSTLB method, the BTLB method, and the CBTLB method, to deal with the load

TABLE 3
THE SPEEDUPS ACHIEVED BY THE MAPPING/LOAD-BALANCING METHODS FOR THE TEST SAMPLE ON DIFFERENT NUMBERS OF PROCESSORS

	Number of processors			
Methods	10	30	50	70
AE/MC	0.30	0.62	1.35	1.63
AE/MC/MCSTLB	5.44	12.81	10.77	11.55
AE/MC/BTLB	5.60	13.36	11.05	11.95
AE/MC/CBTLB	5.66	13.79	11.25	12.22
AE/ORB	6.55	16.65	18.24	24.60
AE/ORB/MCSTLB	7.07	17.39	27.12	30.54
AE/ORB/BTLB	7.42	19.17	28.62	34.07
AE/ORB/CBTLB	7.65	19.73	30.55	37.50
ML*k*P	6.91	17.69	19.40	26.08
ML*k*P/ MCSTLB	7.07	17.96	26.78	32.29
ML*k*P/ BTLB	7.35	19.29	28.92	34.35
ML*k*P/ CBTLB	7.73	20.15	30.50	38.69

TABLE 4
THE MAXIMUM SPEEDUPS ACHIEVED BY THE MAPPING/LOAD-BALANCING METHODS FOR THE TEST SAMPLE ON DIFFERENT NUMBERS OF PROCESSORS

	Number of processors			
Methods	10	30	50	70
AE/MC	7.07	19.79	23.38	35.54
AE/MC/MCSTLB	7.51	20.04	30.85	36.66
AE/MC/BTLB	7.65	19.83	31.74	40.78
AE/MC/CBTLB	7.74	20.57	33.23	43.01
AE/ORB	6.92	18.99	21.22	30.46
AE/ORB/MCSTLB	7.32	19.09	29.95	34.05
AE/ORB/BTLB	7.51	19.82	30.88	38.46
AE/ORB/CBTLB	7.71	20.26	32.61	41.38
ML*k*P	7.18	19.75	22.30	31.90
ML*k*P/ MCSTLB	7.28	19.86	30.07	36.83
ML*k*P/ BTLB	7.43	19.88	31.26	38.24
ML*k*P/ CBTLB	7.79	20.61	32.40	41.97

imbalance problems of solution-adaptive finite element application programs on distributed memory multicomputers. To evaluate the performance of the proposed methods, we have implemented those methods along with three mapping methods, the AE/ORB method, the AE/MC method, and the ML*k*P method, on an SP2 parallel machine. The finite element graph *Truss* is used as test sample. Three criteria, the execution time of mapping/load-balancing methods, the execution time of a solution-adaptive finite element application program under different mapping/load-balancing methods, and the speedups achieved by mapping/load-balancing methods for a solution-adaptive finite element application program, are used for the performance evaluation. The experimental results show that 1) if the initial mapping is performed by a mapping method and the same mapping method and load-balancing methods were used in each refinement to balance the computational load of processors, the execution time of an application program under a load-balancing method is better than that of the mapping method, and 2) The execution time of an application program under the CBTLB method is better than that of the BTLB method and the MCSTLB method.

ACKNOWLEDGMENTS

The work of this paper was partially supported by National Center for High-Performance Computing of the Republic of China under contract NCHC-86-08-021.

REFERENCES

[1] I.G. Angus, G.C. Fox, J.S. Kim, and D.W. Walker, *Solving Problems on Concurrent Processors*, vol. 2. Englewood Cliffs, N.J.: Prentice Hall, 1990.

[2] C. Aykanat, F. Ozgüner, S. Martin, and S.M. Doraivelu, "Parallelization of a Finite Element Application Program on a Hypercube Multiprocessor," *Hypercube Multiprocessor*, pp. 662-673, 1987.

[3] S.T. Barnard and H.D. Simon, "Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems," *Concurrency: Practice and Experience*, vol. 6, no. 2, pp. 101-117, Apr. 1994.

[4] S.T. Barnard and H.D. Simon, "A Parallel Implementation of Multilevel Recursive Spectral Bisection for Application to Adaptive Unstructured Meshes," *Proc. Seventh SIAM Conf. Parallel Processing for Scientific Computing*, pp. 627-632, San Francisco, Feb. 1995.

[5] J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*. New York: Elsevier North Holland, 1976.

[6] Y.C. Chung and C.J. Liao, "A Processor Oriented Partitioning Method for Mapping Unstructured Finite Element Graphs on SP2 Parallel Machines," technical report, Dept. of Information Eng., Feng Chia Univ., June 1996.

[7] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279-301, Oct. 1989.

[8] F. Ercal, J. Ramanujam, and P. Sadayappan, "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning," *J. Parallel and Distributed Computing*, vol. 10, pp. 35-44, 1990.

[9] C. Farhat and H.D. Simon, "TOP/DOMDEC—A Software Tool for Mesh Partitioning and Parallel Processing," Technical Report RNR-93-011, NASA Ames Research Center, 1993.

[10] C.M. Fiduccia and R.M. Mattheyes, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. 19th IEEE Design Automation Conf.*, pp. 175-181, 1982.

[11] C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salman, and D.W. Walker, *Solving Problems on Concurrent Processors*, vol. 1. Englewood Cliffs, N.J.: Prentice Hall, 1988.

[12] M.R. Garey and D.S. Johnson, *Computers and Intractability, A Guide to Theory of NP-Completeness*. San Francisco: Freeman, 1979.

[13] J.R. Gilbert and E. Zmijewski, "A Parallel Graph Partitioning Algorithm for a Message-Passing Multiprocessor," *Int'l J. Parallel Programming*, vol. 16, no. 6, pp. 427-449, 1987.

[14] J.R. Gilbert, G.L. Miller, and S.H. Teng, "Geometric Mesh Partitioning: Implementation and Experiments," *Proc. Ninth Int'l Parallel Processing Symp.*, Santa Barbara, Calif., pp. 418-427, Apr. 1995.

[15] A. Heirich and S. Taylor, "A Parabolic Load Balancing Method," *Proc. ICPP '95*, pp. 192-202, 1995.

[16] B. Hendrickson and R. Leland, "The Chaco User's Guide: Version 2.0," Technical Report SAND94-2692, Sandia Nat'l Laboratories, Albuquerque, N.M., Oct. 1994.

[17] B. Hendrickson and R. Leland, "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations," *SIAM J. Scientific Computing*, vol. 16, no. 2, pp. 452-469, 1995.

[18] B. Hendrickson and R. Leland, "An Multilevel Algorithm for Partitioning Graphs," *Proc. Supercomputing '95*, Dec. 1995.

[19] G. Horton, "A Multi-Level Diffusion Method for Dynamic Load Balancing," *Parallel Computing*, vol. 19, pp. 209-218, 1993.

[20] Y.F. Hu and R.J. Blake, "An Optimal Dynamic Load Balancing Algorithm," Technical Report DL-P-95-011, Daresbury Laboratory, Warrington, U.K., 1995.

[21] M.K. Jones and P.E. Plassmann, "Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes," *Proc. IEEE Scalable High Performance Computing Conf.*, pp. 478-485, 1994.

[22] G. Karypis and V. Kumar, "Multilevel *k*-way Partitioning Scheme for Irregular Graphs," Technical Report 95-064, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, 1995.

[23] G. Karypis and V. Kumar, *MeTiS—Unstructured Graph Partitioning and Spares Matrix Ordering System*. Univ. of Minnesota, 1995.
[24] B.W. Kernigham and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Systems Technology J.*, vol. 49, no. 2, pp. 292-370, Feb. 1970.
[25] J.B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salseman Problem," *Proc. AMS*, vol. 7, pp. 48-50, 1956.
[26] C.W. Ou and S. Ranka, "Parallel Incremental Graph Partitioning," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 8, pp. 884-896, Aug. 1997.
[27] F. Pellegrini and J. Roman, "Scotch: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs," *Proc. HPCN '96*, pp. 493-498, Apr. 1996.
[28] J.R. Pilkington and S.B. Baden, "Dynamic Partitioning of Non-Uniform Structured Workloads with Spacefilling Curves," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 3, pp. 288-300, Mar. 1996.
[29] R. Preis and R. Diekmann, "The PARTY Partitioning—Library User Guide—Version 1.1," Heniz Nixdorf Inst., Universität Paderborn, Germany, Sept. 1996.
[30] S. Ranka, Y. Won, and S. Sahni, "Programming a Hypercube Multicomputer," *IEEE Software*, vol. 5, no. 5, pp. 69-77, Sept. 1988.
[31] K. Schloegel, G. Karypis, and V. Kumar, "Parallel Multilevel Diffusion Algorithms for Repartitioning of Adaptive Meshes," Technical Report #97-014, Univ. of Minnesota, Dept. of Computer Science and Army HPC Center, 1997.
[32] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes," Technical Report #97-013, Dept. of Computer Science, Univ. of Minnesota, June 1997.
[33] H.D. Simon, "Partitioning of Unstructured Problems for Parallel Processing," *Computing Systems in Eng.*, vol. 2, nos. 2/3, pp. 135-148, 1991.
[34] C. Walshaw, *The Jostle User Manual: Version 2.0*. London: Univ. of Greenwich, July 1997.
[35] M. Willebeek-LeMair, and A.P. Reeves, "Strategies for Dynamic Load Balancing on Highly Parallel Computers," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 9, pp. 979-993, Sept. 1993.
[36] R.D. Williams, "Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations," *Councurrency : Practice and Experience*, vol. 3, no. 5, pp. 457-481, Oct. 1991.
[37] R.D. Williams, *DIME: Distributed Irregular Mesh Environment*. California Inst. of Technology, 1990.
[38] M.Y. Wu, "On Runtime Parallel Scheduling for Processor Load Balancing," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 2, pp. 173-186, Feb. 1997.
[39] C.Z. Xu, and F.C.M. Lau, "The Generalized Dimension Exchange Method for Load Balancing in *k*-ary *n*-cubes and Variants," *J. Parallel and Distributed Computing*, vol. 24, no. 1, pp. 72-85, 1995.

Ching-Jung Liao received his BS and MS degrees in computer science and biomedical engineering from Chung Yuan Christian University in 1988 and 1990, respectively. He was an instructor in the Department of Information Management at Ling Tung College from 1992 to 1994. He is currently a PhD candidate in the Department of Information Engineering at Feng Chia University. His research interests include parallel processing for scientific computation, parallel programming tools, partitioning, and load balancing. He is a member of the IEEE Computer Society.

Yeh-Ching Chung received a BS degree in computer science from Chung Yuan Christian University in 1983, and MS and PhD degrees in computer and information science from Syracuse University in 1988 and 1992, respectively. Since 1992, he has been an associate professor in the Department of Information Engineering at Feng Chia University. His research interests include parallel compilers, parallel programming tools, mapping, scheduling, and load balancing. He is a member of the IEEE Computer Society.