A Prefix Code Matching Parallel Load-Balancing Method for Solution-Adaptive Unstructured Finite Element Graphs on Distributed Memory Multicomputers[1]

Yeh-Ching Chung[2] and Ching-Jung Liao[3]

[2] Department of Information Engineering
Feng Chia University, Taichung, Taiwan 407, R.O.C
[3] Department of Accounting and Statistics
The Overseas Chinese College of Commerce, Taichung, Taiwan 407, R.O.C
Email: ychung, cjliao@cray.pdplab.iecs.fcu.edu.tw

Abstract

In this paper, we propose a prefix code matching parallel load-balancing method (PCMPLB) to efficiently deal with the load unbalancing problems of solution-adaptive finite element application programs on distributed memory multicomputers. The main idea of the PCMPLB method is first to construct a prefix code tree for processors. Based on the prefix code tree, a schedule for performing load transfer among processors can be determined by concurrently and recursively dividing the tree into two subtrees and finding a maximum matching for processors in the two subtrees until the leaves of the prefix code tree are reached. The experimental results show that the execution time of an application program under the PCMPLB method is less than that of the direct diffusion method and the multilevel diffusion method.

1. Introduction

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-completeness [7], many heuristic methods were proposed to find satisfactory sub-optimal solutions [2, 4, 6, 8, 12-14, 19-20]. If the number of nodes of a finite element graph will not be increased 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 will be increased discretely due to the refinement of some finite elements during the execution of a finite element application program. This will result in load unbalancing 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. Since node remapping or load-balancing algorithms were performed at run-time, their execution must be fast and efficient.

Many load-balancing methods have been proposed in the literature [5, 9-10, 15, 17-18, 21-22, 24]. In this paper, we propose a *prefix code matching parallel load-balancing* (PCMPLB) method to efficiently deal with the load unbalancing problems of solution-adaptive finite element application programs on distributed memory multicomputers with fully-connected interconnection networks such as multistage interconnection networks, crossbar networks, etc. The main idea of the PCMPLB method is first to construct a prefix code tree for processors according to the processor graph, where the leaves of the prefix code tree are processors. Based on the prefix code tree, a schedule for performing load transfer among processors can be determined by concurrently and recursively dividing the tree into two subtrees and finding a maximum matching for processors in the two subtrees until the leaves of the prefix code tree are reached.

To evaluate the performance of the PCMPLB method, we have implemented the PCMPLB method along with two load-balancing methods, the direct diffusion method [5, 21] and the multilevel diffusion method [9, 17-18], and five mapping methods, AE/MC [4], AE/ORB [4], JOSTLE-MS [20-21], ML*k*P [12], and PARTY [16] on an

[1] The work of this paper was partially supported by NSC of R.O.C. under contract NSC-87-2231-E-035-010.
[2] Correspondence addressee.

SP2 parallel machine. The experimental results show that (1) if a mapping method is used for the initial partitioning and this mapping method or a load-balancing method is used in each refinement, the execution time of an application program under a load-balancing method is always less than that of the mapping method. (2) The execution time of an application program under the PCMPLB method is less than that of the direct diffusion method and the multilevel diffusion method.

2. The Prefix Code Matching Parallel Load-Balancing Method

The PCMPLB method can be divided into the following four phases.

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

Phase 2: Construct a prefix code tree for processors in G.

Phase 3: Determine the load transfer sequence by using matching theorem.

Phase 4: Perform the load transfer.

In the following, we will describe them in details.

2.1. The Processor Graph

When nodes of a solution-adaptive finite element graph were 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. 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. We now give an example to explain it.

EXAMPLE 1: Figure 1 shows an example of a processor graph. Figure 1(a) shows an initial partition of a 100-node finite element graph on 10 processors by using the ML*k*P method. In Figure 1(a), all processors are assigned 10 finite element nodes. After the refinement, the number of nodes assigned to processors P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, and P_9 are 10, 11, 11, 12, 10, 19, 16, 13, 13, and 13, respectively, and is shown in Figure 1(b). The corresponding processor graph of Figure 1(b) is shown in Figure 1(c).

2.2. The Construction of a Prefix Code Tree

Based on the processor graph, we can construct a prefix code tree. The construction of a prefix code tree T_{Prefix} is based on the Huffman's algorithm [11] and is given as follows:

Step 1: Let V be a set of P isolated vertices, where P is the number of processors in G. Each vertex P_i in V is the root of a complete binary tree (of height 0) with a weight $w_i = 1$.

Step 2: While $|V| > 1$, perform the following:

(a) Find a tree T in V with the smallest root weight w. If there are two or more candidates, choose the one whose leaf nodes have the smallest degree in G.

(b) For trees in V whose leaf nodes are adjacent to those in T, find a tree T' with the smallest root weight w'. If there are two or more candidates, choose the one whose leaf nodes have the smallest degree in G.

(c) Create a new (complete binary) tree T^* with root weight $w^* = w + w'$ and having T and T' as its left and right substrees, respectively.

(d) Place T^* in V and delete T and T'.

(e) Repeat (a) to (d) until $V' = 1$.

We now give an example to explain the above description.

EXAMPLE 2: An example of step by step construction of a prefix code tree from the processor graph shown in Figure 1(c) is given in Figure 2. The degrees of processors P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, and P_9 are 2, 4, 4, 5, 3, 3, 3, 4, 4, and 6, respectively. The initial configuration is shown in Figure 2(a). Initially, P_5 has the smallest degree. P_5 and P_6 are combined as a tree and we obtain a new configuration as shown in Figure 2(b). By applying Steps 2(a)-2(e), we can obtain Figures 2(c)-2(j). Once the construction of a prefix code tree is completed, each processor is assigned a prefix code word, that is, P_0 = 1000, P_1 = 1001, P_2 = 1010, P_3 = 111, P_4 = 1011, P_5 = 000, P_6 = 001, P_7 = 010, P_8 = 110, and P_9 = 011.

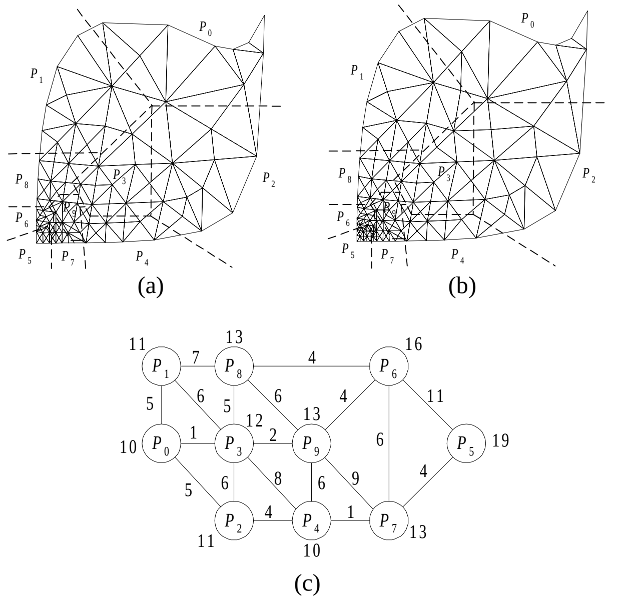

Figure 1: An example of a processor graph. (a) The initial partitioned finite element graph. (b) The finite element graph after a refinement. (c) The corresponding processor graph obtained from (b).

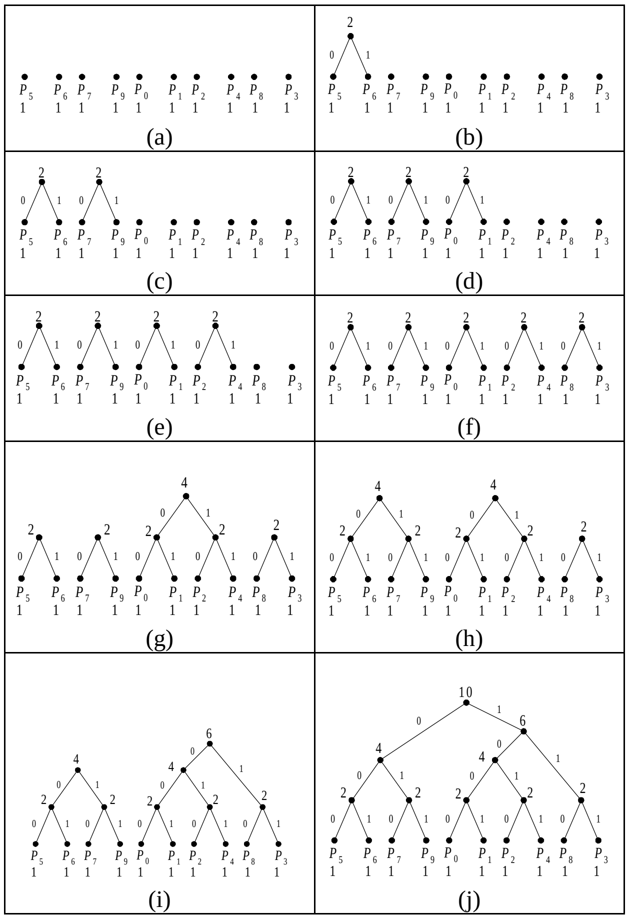
Figure 2: A step by step construction of a prefix code tree from Figure 1(c).

2.3. Determine a Load Transfer Sequence by Using Matching Theorem

Based on the prefix code tree and the processor graph, we can obtain a communication pattern graph.

Definition 1: Given a processor graph $G = (V, E)$ and a prefix code tree T_{Prefix}, the communication pattern graph $G_c = (V_c, E_c)$ of G and T_{Prefix} is a subgraph of G. For every $(P_i, P_j) \in E_c$, P_i and P_j are in the left and the right subtrees of T_{Prefix}, respectively, and $P_i, P_j \in V_c$.

The communication pattern graph has several properties that can be used to determine the load transfer sequence.

Definition 2: A graph $G = (V, E)$ is called *bipartite* if $V = V_1 \cup V_2$ with $V_1 \cap V_2 = \phi$, and every edge of G is of the form $\{a, b\}$ with $a \in V_1$ and $b \in V_2$.

Theorem 1: A communication pattern graph G_c is a bipartite graph.

Proof: According to Definition 1, for every $(P_i, P_j) \in E_c$, P_i and P_j are in the left and right subtrees of T_{Prefix}, respectively. Therefore G_c is a bipartite graph. ■

Definition 3: A subset M of E is called a *matching* in G if its elements are edges and no two of them are adjacent in G; the two ends of an edge in M are said to be matched under M. M is a *maximum matching* if G has no matching M' with $|M'| > |M|$.

Theorem 2: Let $G = (V, E)$ be a bipartite graph with bipartition (V_1, V_2). Then G contains a matching that saturates every vertex in V_1 if and only if $|N(S)| \geq |S|$ for all $S \subseteq V_1$, where $N(S)$ is the set of all neighbors of vertices in S.

Proof: The proof can be found in [3]. ■

Corollary 1: Let $G_c = (V_c, E_c)$ be a communication pattern graph and V_L and V_R are the sets of processors in the left and the right subtrees of T_{Prefix}, respectively, where $V_L, V_R \subseteq V_c$. Then we can find a maximum matching M from G_c such that for every element $(P_i, P_j) \in M$, $P_i \in V_L$ and $P_j \in V_R$.

Proof: From Definition 3 and Hungarian method [3], we know that a maximum matching M from G_c can be found. ■

From the communication pattern graph, we can determine a load transfer sequence for processors in the left and the right subtrees of a prefix code tree by using the matching theorem to find a maximum matching among the edges of the communication pattern graph. Due to the construction process used in Phase 2, we can also obtain communication pattern graphs from the left and the right subtrees of a prefix code tree. A load transfer sequence can be determined by concurrently and recursively dividing a prefix code tree into two subtrees, constructing the corresponding communication pattern graph, finding a maximum matching for the communication pattern graph, and determining the number of finite element nodes need to be transferred among processors until a tree contains one vertex. Assume that there are P processors in a processor graph and N nodes in a refined finite element graph. We define N/P as the average load of a processor. The load of a processor is defined as the number of finite element nodes assigned to it. The load transfer sequence is determined as follows:

Step 1: Let S be a set that contains the prefix code tree obtained in Phase 2.

Step 2: While $|S| < P$, for each tree T_{Prefix} in S and the number of vertices in T_{Prefix} is greater than 1, perform the following:

(a) Let T_L and T_R be the left and the right subtrees of T_{Prefix}, respectively. P_L and P_R represent the number of processors (leaf nodes) in T_L and T_R, respectively. Find the communication pattern graph G_c from the processor graph G and the prefix code tree T_{Prefix}.

(b) Find a maximum matching $M = \{(P_i, Q_i) | P_i$ and Q_i are processors in T_L and T_R, respectively, and P_i and Q_i are adjacent in $G\}$ from G_c.

(c) Calculate $quota(T_L)$, $quota(T_R)$, $load(T_L)$ and *load*(T_R), where *quota*(T_L) and *quota*(T_R) denote the sum of the average load of processors in T_L and T_R, respectively; and *load*(T_L) and *load*(T_R) represent the sum of the load of processors in T_L and T_R, respectively.

(d) If *load*(T_R) > *quota*(T_R), processors in T_R need to send m = *load*(T_R) − *quota*(T_R) finite element nodes to processors in T_L. If *load*(T_R) < *quota*(T_R), processors in T_L need to send m = *load*(T_L) − *quota*(T_L) finite element nodes to processors in T_R. If *load*(T_R) = *quota*(T_R), go to step 2(g).

(e) For each element (P_i, Q_i) in M, determine the number of finite element nodes that P_i (Q_i) needs to send to Q_i (P_i) based on $|M|$, the load of P_i (Q_i), and the value of m. Assume that $M = \{(P_1, Q_1), (P_2, Q_2), \ldots , (P_k, Q_k)\}$ and *load*(T_R) > *quota*(T_R). The number of finite element nodes that Q_i needs to send to P_i is $w_i = m \times load(Q_i) / \sum_{j=1}^{k} load(Q_j)$, where *load*($Q_i$) is the number of finite element nodes assigned to processor Q_i. If *load*(Q_i) < w_i, an exception handling procedure is carried out by moving finite element nodes from processors in T_R to Q_i to ensure that *load*(Q_i) ≥ w_i.

(f) Place T_L and T_R in S and delete T_{Prefix} from S.

(g) Repeat (a) to (f) until $|S| = P$.

We now give an example to explain the above description.

EXAMPLE 3: Figure 3 shows the communication pattern graphs and their corresponding maximum matching for the examples shown in Figures 1 and 2 step by step when performing the procedure described in this subsection. Figure 1(a) shows the communication pattern graph for the prefix code tree with root at level 1. In Figure 3(a), an arrow is an element of a matching. The number associated with an arrow denotes the number of finite element nodes that a processor needs to send to the other processor. Figure 3(b) to Figure 3(d) show the communication pattern graphs for the prefix code trees with roots at levels 2, 3, and 4, respectively. When the matching of each communication pattern graph is found, the load transfer sequence can be determined as follows.

Step 1: $P_6 \xrightarrow{4} P_8$, $P_7 \xrightarrow{3} P_4$, $P_9 \xrightarrow{3} P_3$;

Step 2: $P_5 \xrightarrow{3} P_7$, $P_6 \xrightarrow{2} P_9$, $P_3 \xrightarrow{3} P_0$, $P_8 \xrightarrow{4} P_1$;

Step 3: $P_5 \xrightarrow{3} P_6$, $P_0 \xrightarrow{2} P_2$, $P_8 \xrightarrow{1} P_3$;

Step 4: $P_1 \xrightarrow{2} P_0$.

2.4. Perform the Load Transfer

After the determination of the load transfer sequence, the physical load transfer can be carried out among the processors according to the load transfer sequence in parallel. The goals of the physical load transfer are to balance the load of processors and to minimize the communication cost among processors. By following the load transfer sequence, the goal of load balancing can be achieved easily. Assume that processor P_i needs to send m finite element nodes to processor Q_i. To minimize the communication cost between processors P_i and Q_i, P_i sends finite element nodes that are adjacent to those in Q_i (we called these nodes as boundary nodes) to Q_i. If the number of boundary nodes is greater than m, nodes with smaller degrees will be sent from P_i and Q_i. If the number of boundary nodes is less than m, the boundary nodes and nodes that are adjacent to the boundary nodes will be sent from P_i and Q_i.

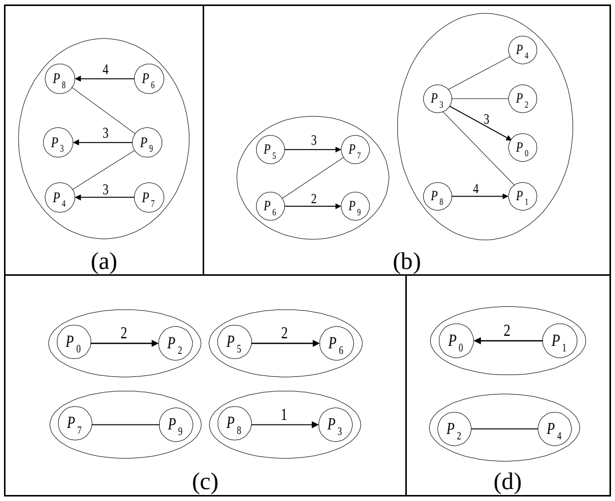

Figure 3: The matching of the communication pattern graph. (a) The first matching. (b) The second matching. (c) The third matching. (d) The fourth matching.

3. Performance Evaluation and Experimental Results

To evaluate the performance of the proposed method, we have implemented the PCMPLB method along with two load-balancing methods, the direct diffusion method (DD) and the multilevel diffusion method (MD), and five mapping methods, the AE/MC method, the AE/ORB method, the JOSTLE-MS method, the ML*k*P method, and the PARTY library method, on an SP2 parallel machine. All algorithms were written in C with MPI 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) [23] is used. 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 Figure 4. The number of nodes and elements for the test graph after each refinement are shown in Table 1. For the presentation purpose, the number of nodes and the number of finite elements shown in Figure 4 are less than those shown in Table 2.

Table 1. The number of nodes and elements of the test graph *truss*.

Refine / *Truss*	Initial (0)	1	2	3	4	5
node #	18407	23570	29202	36622	46817	57081
Element #	35817	46028	57181	71895	92101	112494

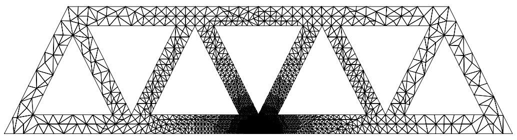

Figure 4. The test sample *truss* (7325 nodes, 14024 elements).

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 use the AE/MC method or the AE/ORB method or the JOSTLE-MS method or the ML*k*P method or the PARTY library method 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 equation (Laplace solver). The algorithm of solving Laplaces 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 our Laplace solver is 1000. When the computation is converged, the first refined finite element graph is read. To balance the computational load of processors, the AE/MC method or the AE/ORB method or the JOSTLE-MS method or the ML*k*P method or the PARTY library method or the direct diffusion method or the multilevel diffusion method or the PCMPLB method is applied. After a mapping/load-balancing method is performed, the computation for each processor is carried out. The procedures of mesh refinement, load balancing, and 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 twenty methods used for the performance evaluation. For examples, the AE/ORB method uses AE/ORB to perform the initial mapping and AE/ORB to balance the computational load of processors in each refinement. The AE/ORB/PCMPLB method use AE/ORB to perform the initial mapping and PCMPLB to balance the computational load of processors in each refinement.

3.1. Comparisons of the Execution Time of Mapping/Load-Balancing Methods

The execution time of different mapping/load-balancing methods for the test unstructured finite element graph *truss* on an SP2 parallel machine with 10, 30, and 50 processors are shown in Table 2. In Table 2, we list the initial mapping time and the refinement time for mapping/load-balancing methods. The initial mapping time is the execution time of mapping methods to map finite element nodes of the initial test sample to processors. The refinement time is the sum of the execution time of mapping/load-balancing methods to balance the load of processors after each refinement. Since we deal with the load balancing issue in this paper, we will focus on the refinement time comparison of mapping/load-balancing methods. From Table 2, we can see that, in general, the refinement time of load-balancing methods is shorter than that of the mapping methods. The reasons are (1) the mapping methods has higher time complexity than those of the load-balancing methods; and (2) the mapping methods need to perform gather-scatter operations that are time consuming in each refinement.

For the same initial mapping method, the refinement time of the PCMPLB method, in general, is shorter than that of the direct diffusion and the multilevel diffusion methods. The reasons are as follows:

(1) The PCMPLB method has less time complexity than those of the direct diffusion and the multilevel diffusion methods.
(2) The physical load transfer is performed in parallel in the PCMPLB method.
(3) The number of data movement steps among processors in the PCMPLB method is less than those of the direct diffusion and the multilevel diffusion methods.

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

The time of a Laplace solver to execute one iteration (computation + communication) for the test sample under different mapping/load-balancing methods on an SP2 parallel machine with 10, 30, and 50 processors are shown in Figure 5, Figure 6, and Figure 7, 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 Figure 5 to Figure 7, we can see that if the initial mapping is performed by a mapping method (for example AE/ORB) and the same mapping method or a load-balancing method (DD, MD, PCMPLB) is performed for each refinement, the execution time of a Laplace solver under the proposed load-balancing method is shorter than that of other methods. The reasons are as follows:

(1) The PCMPLB method uses the maximum matching to determine the load transfer sequence. Data migration can be done between adjacent processors. This local data migration ability can greatly reduce the amount of global data migration and therefore reduce the communication cost of a Laplace Solver.

(2) In the physical load transfer, the PCMPLB method tries to transfer boundary nodes between processors. This will also reduce the communication overheads of a Laplace Solver.

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

The speedups and the maximum speedups under the mapping/load-balancing methods on an SP2 parallel machine with 10, 30, and 50 processors for the test sample are shown in Table 3 and Table 4, respectively. The maximum speedup is defined as the ratio of the execution time of a sequential Laplace solver to the execution time of a parallel Laplace solver. From Table 3, we can see that if the initial mapping is performed by a mapping method (for example AE/ORB) and the same mapping method or a load-balancing method (DD, MD, PCMPLB) is performed for each refinement, the proposed load-balancing method has the best speedup among mapping/load-balancing methods.

From Table 4, we can see that if the initial mapping is performed by a mapping method (for example AE/ORB) and the same mapping method or a load-balancing method (DD, MD, PCMPLB) is performed for each refinement, the proposed load-balancing method has the best maximum speedup among mapping/load-balancing methods. For the mapping methods, AE/MC has the best maximum speedups for test samples. For the load-balancing methods, AE/MC/PCMPLB has the best maximum speedups for test samples. From Table 4, we can see that a better initial mapping method is used, a better maximum speedup can be expected when the PCMPLB method is used in each refinement.

4. Conclusions

In this paper, we have proposed a prefix code matching parallel load-balancing method, the PCMPLB method, to deal with the load unbalancing problems of solution-adaptive finite element application programs. To evaluate the performance of the proposed method, we have implemented this method along with two load-balancing methods, the direct diffusion method and the multilevel diffusion method, and five mapping methods, AE/MC, AE/ORB, JOSTLE-MS, ML*k*P, and PARTY, on an SP2 parallel machine. The unstructured 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 under 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 a mapping method is used for the initial partitioning and this mapping method or a load-balancing method is used in each refinement, the execution time of an application program under a load-balancing method is always shorter than that of the mapping method. (2) The execution time of an application program under the PCMPLB method is less than that of the direct diffusion method and the multilevel diffusion method.

References

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

[2] 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.

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

[4] Y.C. Chung and C.J. Liao, "A Processor Oriented Partitioning Method for Mapping Unstructured Finite Element Graphs on SP2 Parallel Machines," Technical Report, Institute of Information Engineering, Feng Chia University, Taichung, Taiwan, Sep. 1996.

[5] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 7, No. 2, pp. 279-301, Oct. 1989.

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

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

[8] B. Hendrickson and R. Leland, "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations," *SIAM Journal on Scientific Computing*,

Vol. 16, No.2, pp. 452-469, 1995.

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

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

[11] D.A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE 40*, pp. 1098-1101, 1952.

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

[13] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," Technical Report 95-035, Department of Computer Science, University of Minnesota, Minneapolis, 1995.

[14] B.W. Kernigham and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. J.*, Vol. 49, No. 2, pp. 292-370, Feb. 1970.

[15] 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.

[16] R. Preis and R. Diekmann, "The PARTY Partitioning – Library User Guide – Version 1.1," HENIZ NIXDORF INSTITUTE Universität Paderborn, Germany, Sep. 1996.

[17] K. Schloegel, G. Karypis, and V. Kumar, Parallel Multilevel Diffusion Algorithms for Repartitioning of Adaptive Meshes, Technical Report #97-014, University of Minnesota, Department of Computer Science and Army HPC Center, 1997.

[18] K. Schloegel, G. Karypis, and V. Kumar, Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes, Technical Report #97-013, University of Minnesota, Department of Computer Science, Jun. 1997.

[19] H.D. Simon, "Partitioning of Unstructured Problems for Parallel Processing," *Computing Systems in Engineering*, Vol. 2, No. 2/3, pp. 135-148, 1991.

[20] C.H. Walshaw, M. Cross, and M.G. Everett, "A Localized Algorithm for Optimizing Unstructured Mesh Partitions," *The International Journal of Supercomputer Applications*, Vol. 9, No. 4, pp. 280-295, Winter 1995.

[21] C. Walshaw, M. Cross, and M. G. Everett, Dynamic Load-Balancing for Parallel Adaptive Unstructured Meshes, In M. Heath et al. Editor, *Parallel Processing for Scientific Computing*, SIAM, Philadelphia, 1997.

[22] C. Walshaw, M. Cross, and M. G. Everett, "Dynamic Mesh Partitioning: A Unified Optimisation and Load-Balancing Algorithm," Technical Report 95/IM/06, University of Greenwich, London, London SE18 6PF, UK, Dec. 1995.

[23] R.D. Williams, *DIME: Distributed Irregular Mesh Environment*, California Institute of Technology, 1990.

[24] 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.

Table 2: The execution time of different mapping/load-balancing methods for the test sample on different numbers of processors.

# of processors / Method	10		30		50	
	initial mapping	refinement	initial mapping	refinement	Initial Mapping	refinement
AE/MC	5.054	37.563	7.964	67.061	10.256	129.929
AE/MC/DD	5.035	1.571	7.671	1.383	10.041	1.585
AE/MC/MD	5.035	7.231	7.671	4.043	10.041	4.245
AE/MC/PCMPLB	5.035	0.444	7.671	0.652	10.041	0.458
AE/ORB	0.633	7.493	0.637	6.713	0.742	6.938
AE/ORB/DD	0.614	1.607	0.614	2.086	0.586	2.763
AE/ORB/MD	0.614	4.586	0.614	5.028	0.586	6.013
AE/ORB/PCMPLB	0.614	0.474	0.614	0.769	0.586	1.475
JOSTLE-MS	1.055	3.459	1.02	4.426	2.26	5.763
JOSTLE-MS/DD	1.036	0.741	0.997	1.968	0.704	2.954
JOSTLE-MS/MD	1.036	3.45	0.997	4.838	0.704	6.173
JOSTLE-MS/PCMPLB	1.036	0.483	0.997	1.57	0.704	0.922
MLkP	0.567	4.96	0.589	5.279	0.771	5.908
MLkP/DD	0.548	1.289	0.566	1.872	0.621	2.295
MLkP/MD	0.548	4.142	0.566	4.867	0.621	5.612
MLkP/PCMPLB	0.548	1.083	0.566	0.684	0.621	1.233
PARTY	1.969	18.195	1.809	19.6	1.752	19.262
PARTY/DD	1.937	1.347	1.786	2.009	1.577	2.578
PARTY/MD	1.937	4.255	1.786	5.157	1.577	6.278
PARTY/PCMPLB	1.937	1.58	1.786	1.09	1.577	0.941

Time unit: seconds

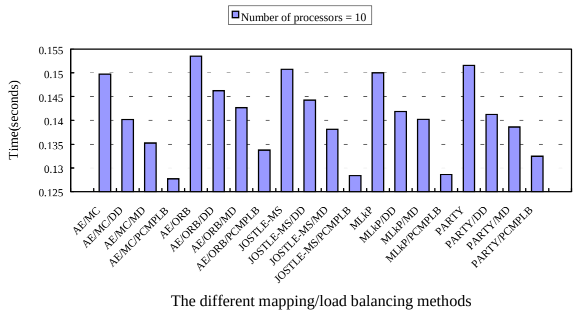

Figure 7: The time for Laplace solver to execute one iteration (computation + communication) for the test sample under different mapping/load-balancing methods on 10 processors.

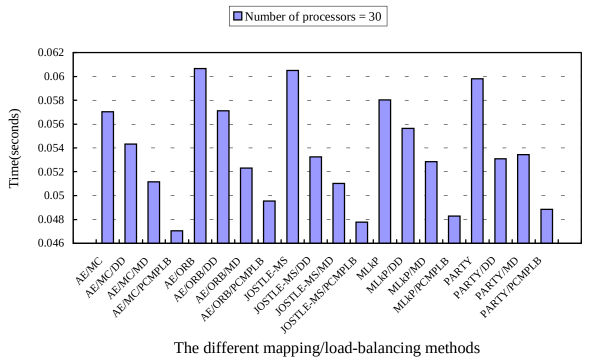

Figure 6: The time for Laplace solver to execute one iteration (computation + communication) for the test sample under different mapping/load-balancing methods on 30 processors.

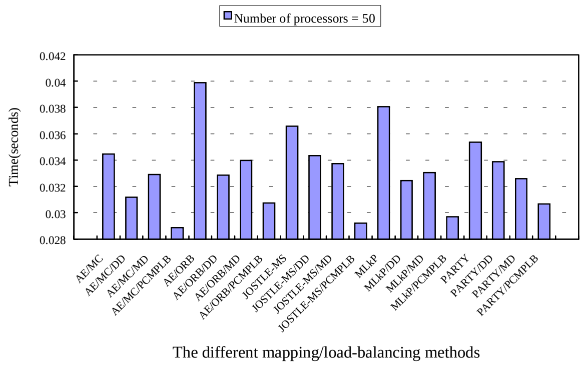

Figure 7: The time for Laplace solver to execute one iteration (computation + communication) for the test sample under different mapping/load-balancing methods on 50 processors.

Table 3: The speedups under the mapping/load-balancing methods for the test sample on an SP2 parallel machine.

# of processors / Method	10	30	50
AE/MC	5.18	7.54	5.71
AE/MC/DD	6.79	15.73	23.28
AE/MC/MD	6.9	15.85	21.78
AE/MC/PCMPLB	7.48	18.13	25.32
AE/ORB	6.16	14.65	20.95
AE/ORB/DD	6.71	16.66	27.52
AE/ORB/MD	6.74	17.2	24.57
AE/ORB/PCMPLB	7.39	19.57	30.38
JOSTLE-MS	6.42	15.11	22.35
JOSTLE-MS/DD	6.82	17.73	26.22
JOSTLE-MS/MD	6.99	17.53	25.65
JOSTLE-MS/PCMPLB	7.67	19.8	32.31
ML*k*P	6.41	15.59	22.27
ML*k*P/DD	6.93	17.16	28.19
ML*k*P/MD	6.87	17.1	25.85
ML*k*P/PCMPLB	7.65	20.11	31.59
PARTY	5.8	12.27	17.68
PARTY/DD	6.9	17.52	26.21
PARTY/MD	6.88	16.5	25.13
PARTY/PCMPLB	7.33	19.27	30.04

Table 4: The maximum speedups under the mapping/load-balancing methods for the test sample on an SP2 parallel machine.

# of processors / Method	10	30	50
AE/MC	6.66	17.47	28.92
AE/MC/DD	7.11	18.35	31.96
AE/MC/MD	7.37	19.48	31.67
AE/MC/PCMPLB	7.8	21.19	34.53
AE/ORB	6.49	16.43	24.98
AE/ORB/DD	6.81	17.45	30.32
AE/ORB/MD	6.98	19.05	29.35
AE/ORB/PCMPLB	7.45	20.11	32.41
JOSTLE-MS	6.61	16.47	27.25
JOSTLE-MS/DD	6.91	18.72	29.01
JOSTLE-MS/MD	7.21	19.53	31.17
JOSTLE-MS/PCMPLB	7.77	20.86	34.1
ML*k*P	6.64	17.17	26.18
ML*k*P/DD	7.02	17.91	30.72
ML*k*P/MD	7.1	18.85	30.83
ML*k*P/PCMPLB	7.75	20.64	33.56
PARTY	6.57	16.66	28.19
PARTY/DD	7.06	18.77	29.43
PARTY/MD	7.19	18.65	31.34
PARTY/PCMPLB	7.52	20.4	32.5