A Parallel Run–Time Iterative Load Balancing Algorithm for Solution–Adaptive Finite Element Meshes on Hypercubes‡

Yeh–Ching Chung, Yaa–Jyun Yeh, and Chia–Cheng Liu

Department of Computer Science & Information Engineering
Feng Chia University
Taichung, Taiwan 407, ROC
e–mail : ychung, yeher, ccliu@pine.iecs.fcu.edu.tw

Abstract – To efficiently execute a finite element program on a hypercube, we need to map nodes of the corresponding finite element graph to processors of a hypercube such that each processor has approximately the same amount of computational load and the communication among processors is minimized. If the number of nodes of a finite element graph will not be increased during the execution of a program, the mapping only needs to be performed once. However, if a finite element graph is solution–adaptive, that is, the number of nodes will be increased discretely due to the refinement of some finite elements during the execution of a program, a run–time load balancing algorithm has to be performed many times in order to balance the computational load of processors while keeping the communication cost as low as possible. In this paper, we proposed a parallel iterative load balancing algorithm (ILB) to deal with the load imbalancing problem of a solution–adaptive finite element program. The proposed algorithm has three properties. First, the algorithm is simple and easy to implement. Second, the execution of the algorithm is fast. Third, it guarantees that the computational load will be balanced after the execution of the algorithm. We have implemented the proposed algorithm along with two parallel mapping algorithms, parallel orthogonal recursive bisection (ORB) [11] *and parallel recursive mincut bipartitioning (MC)* [2]*, on a 16–node NCUBE–2. Three criteria, the execution time of load balancing algorithms, the computation time of an application program under different load balancing algorithms, and the total execution of an application program (under several refinement phases) are used for performance evaluation. Simulation results show that (1) the execution time for ILB is very short compared to those of MC and ORB; (2) the mappings produced by ILB are much better than those of ORB and MC; and (3) the speedups produced by ILB are much better than those of ORB and MC.*

Key Words : Hypercube, load balancing, DIME, mapping, solution–adaptive finite element meshes.

1. Introduction

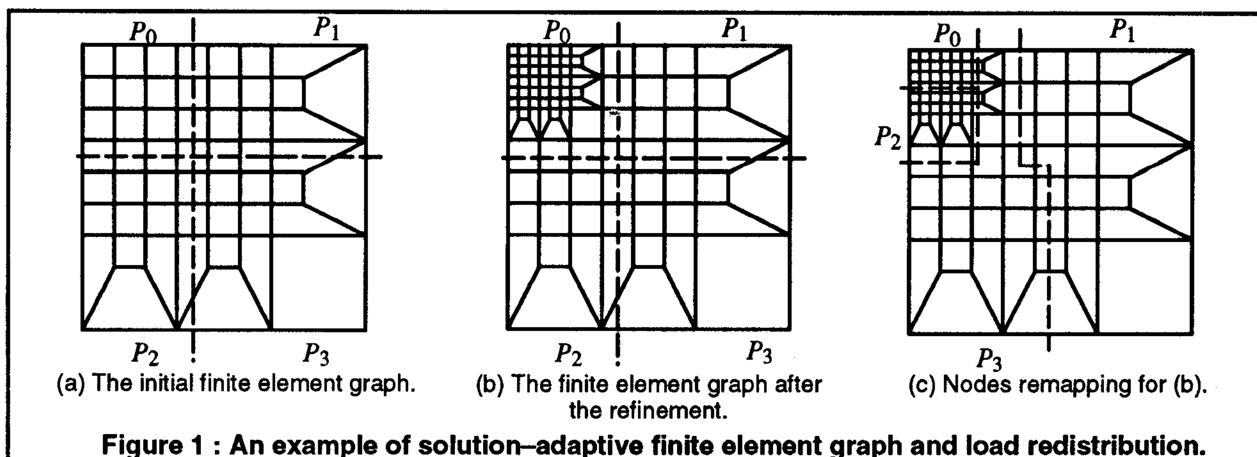
The finite element method is widely used for the structural modeling of physical systems [8]. In the finite element model, an object can be viewed as a finite element graph, which is a connected and undirected graph that consists of a number of finite elements. Each finite element is composed of a number of nodes. The number of nodes of a finite element is determined by an applications. Due to the properties of computation–intensiveness and computation–locality, it is very attractive to implement finite element method on distributed memory multiprocessors [1] [9] [11].

In a distributed memory multiprocessor, such as a hypercube, processors communicate to each other by message passing. To efficiently execute a finite element modeling program on a distributed memory multiprocessor, we need to map nodes of the corresponding finite element graph to processors of a distributed memory multiprocessor such that each processor has approximately the same amount of computational load and the communication among processors is minimized. If the number of nodes of a finite element graph will not be increased during the execution of a program, the mapping only needs to be performed once. However, if a finite element graph is solution–adaptive, that is, the number of nodes will be increased discretely due to the refinement of some finite elements during the execution of a program, a dynamic load balancing algorithm has to be performed many times in order to balance the computational load of processors while keeping the communication cost as low as possible. For example, in Figure 1, a finite element graph is refined once during run–time. Initially, each processor has 16 nodes. If no load balancing algorithm is performed, after the refinement, the number of nodes assigned to P_0, P_1, P_2, and P_3 are 64, 16, 16, and 16, respectively. However, if a load balancing algorithm is carried out for the refinement, the load may be evenly distributed as shown in Figure 1(c).

To solve the load imbalancing problem of a solution–adaptive finite element program on a distributed memory multiprocessor, nodes of a refined finite element graph can be remapped (nodes remapping approach) or load of a refined finite element graph can be redistributed based on the current load of processors (load redistribution approach). For the former case, nodes remapping can be performed by some fast mapping algorithms. In the load redistribution approach, after a finite element graph is refined, a load balancing heuristic is applied to balance the computational load of processors. For both approaches, a good nodes remapping or load redistribution algorithm

‡ The work of this paper is partially supported by NSC under contract NSC83–0408–E–035–005.

0-8186-6555-6/94 $04.00 © 1994 IEEE



(a) The initial finite element graph. (b) The finite element graph after the refinement. (c) Nodes remapping for (b).

Figure 1 : An example of solution–adaptive finite element graph and load redistribution.

should have two properties. First, its execution is fast. Second, it should produce a good mapping result.

In this paper, we present a parallel run–time iterative load balancing algorithm to deal with load imbalancing problems of executing solution–adaptive finite element programs on hypercubes. The algorithm has three properties. First, the algorithm is simple and easy to implement. Second, the execution of the algorithm is fast. Third, it guarantees that the computational load will be balanced after the execution of the algorithm. We have implemented the algorithm on a 16–node NCUBE–2 along with two parallel mapping algorithms, *orthogonal recursive bisection* [11] and *recursive min–cut bipartitioning* [2]. Three criteria, the execution time of load balancing algorithms, the computation time of an application program under different load balancing algorithms, and the total execution of an application program (under several refinement phases) are used for performance evaluation. Simulation results show that the proposed load balancing algorithm outperforms the other two and produces very good mapping results.

In Section 2, a brief survey of related work will be presented. A hypercube definition and the proposed load balancing algorithm will be described in Section 3. The comparisons of the proposed parallel run–time iterative load balancing algorithm, parallel orthogonal recursive bisection, and parallel recursive min–cut bipartitioning will be given in Section 4.

2. Related Work

Many finite element mapping algorithms have been addressed in the literature. In [1], a *binary decomposition* approach was used to partition a nonuniform mesh graph into modules such that each module has the same amount of computational load. These modules were then mapped on meshes, trees, and hypercubes. This method does not try to minimize the communication cost.

Sadayappan and Ercal [9] proposed *nearest–neighbor mapping* approach to map planar finite element graphs on processor meshes. This approach used the *stripes partition* (*stripes mapping*) strategy to minimize the communication cost of processors and then used the *boundary refinement* heuristic to balance the computational load of processors. However, the boundary refinement heuristic does not guarantee the computational load will be balanced.

Williams [11] proposed three parallel load balancing algorithms, *orthogonal recursive bisection*, *eigenvector recursive bisection*, and *a simple parallel simulated annealing*, for solution–adaptive finite element graph problems. The performance analysis shows that the time to execute orthogonal recursive bisection is the fastest and the execution of parallel simulated annealing is time consuming. But the mapping produced by simulated annealing saves of 21% in the execution time of the finite element mesh than the mapping produced by orthogonal recursive bisection.

For those work mentioned above, only the work of [11] deals with load imbalancing problem of solution–adaptive finite element programs. Others assume that the number of nodes of a finite element graph will not be changed during the execution of a program.

3. The Proposed Load Balancing Algorithm

In this section, we will first give a definition for hypercubes. Then, we will describe the proposed load balancing algorithm in details.

3.1. Hypercubes

Definition 1 : An n–dimensional hypercube Q_n, for $n > 1$, can be recursively defined in terms of the graph product $\times$ as follows [5]:

$$Q_n = K_2 \times Q_{n-1}, \quad (1)$$

where $K_2 = Q_1$ is the complete two–node graph.

Definition 2 : In an n–cube two processors, P_x and P_y, are *adjacent* if the address of P_x differs from that of P_y by one bit. Moreover, P_x is said to be the jth *adjacent processor* of P_y if P_x and P_y are adjacent processors and the address of P_x differs from that of P_y at the jth significant bit, where $j = 0, ..., n-1$. We use $P_y(j)$ to denote the jth adjacent processor of P_y.

3.2. A Parallel Run–Time Iterative Load Balancing Algorithm

Many dynamic load balancing algorithms have been addressed in the literature [6] [7] [10]. However, the prob-

Algorithm iterative_load_balancing_for_hypercubes()
1. **for** $j = 0$ to $(n-1)$ **do**
2 { Send $load(P_i)$ to $P_i(j)$ and receive $load(P_i(j))$ from $P_i(j)$.
3 **if** $(load(P_i) = load(P_i(j)))$ **then** $N \leftarrow 0$.
4. **else** /* need to load transfer */
5. { $avg = (load(P_i) + load(P_i(j))) \div 2$.
6. **if** $load(P_i) < avg$ **then** /* P_i must receive nodes from $P_i(j)$ */
7. { $N \leftarrow avg - load(P_i(j))$.
8. Receive N nodes from $P_i(j)$. }
9. $load(P_i) \leftarrow load(P_i) + |N|$. }
10. **else** /* P_i must send nodes to $P_i(j)$ */
11. { $N \leftarrow load(P_i(j)) - avg$.
12. Find the set of nodes M in P_i that are adjacent to nodes in $P_i(j)$. $M_1 \leftarrow \varnothing$.
13. **while** $(|M| + |M_1| < N)$ **do**
14. { $M \leftarrow M \cup M_1$.
15. Find the set M_1 of all nodes in P_i are adjacent to nodes in M. }
16. $M \leftarrow M \cup M_2$, where $M_2 \subseteq M_1$ and $|M| + |M_2| = N$.
17. send M to $P_i(j)$.
18. $load(P_i) \leftarrow load(P_i) - N$. }
19. }
20. }
end_of_iterative_load_balancing_for_hypercubes

Figure 2 : The proposed run–time iterative load balancing algorithm.

lem addressed in this paper is different from that in [6] [7] [10]. At run–time, the computational load increased in a solution–adaptive finite element program is discrete in nature while that in [6] [7] [10] is continuous. Therefore, those approaches proposed in [6] [7] [10] cannot efficiently handle the load imbalancing issue presented in this paper.

The proposed parallel run–time load balancing algorithm uses iterative approach to achieve load balancing. For an n–cube, initially, every processor P_i calculates the average value *avg* of its current computational load, denoted by $load(P_i)$, and the current computational load of $P_i(0)$, denoted by $load(P_i(0))$. If $load(P_i) < avg$, then $N = avg - load(P_i(0))$, and $load(P_i) = load(P_i) + |N|$. If $load(P_i) > avg$, then $N = load(P_i) - avg$ and $load(P_i) = load(P_i) - N$. Then every processor P_i executes physical load transfer according N (Note that, if N is negative, it denotes the number of nodes that processor P_i has to receive from $P_i(j)$; otherwise, it denotes the number of nodes that processor P_i has to transfer to $P_i(j)$). Assume that processor P_i needs to send N nodes to processor $P_i(j)$ and let M denote the set of nodes in P_i that are adjacent to those of $P_i(j)$. In order to keep the communication cost as low as possible, nodes in M are transferred first. If $|M|$ is less than $|N|$, then nodes adjacent to those in M are transferred. If the sum of $|M|$ and the number of nodes adjacent to those in M is less than $|N|$, then nodes adjacent to those nodes adjacent to those in M are transferred. This process is continued until the number of nodes transferred to $P_i(j)$ is equal to $|N|$. At the next iteration, every processor P_i calculates the average value of its current computational load and the current computational load of $P_i(1)$, determines the values of N and $load(P_i)$, and then executes the load transfer, and so on, until n iterations are executed, where n is the dimension of an n–cube. The algorithm is given in Figure 2.

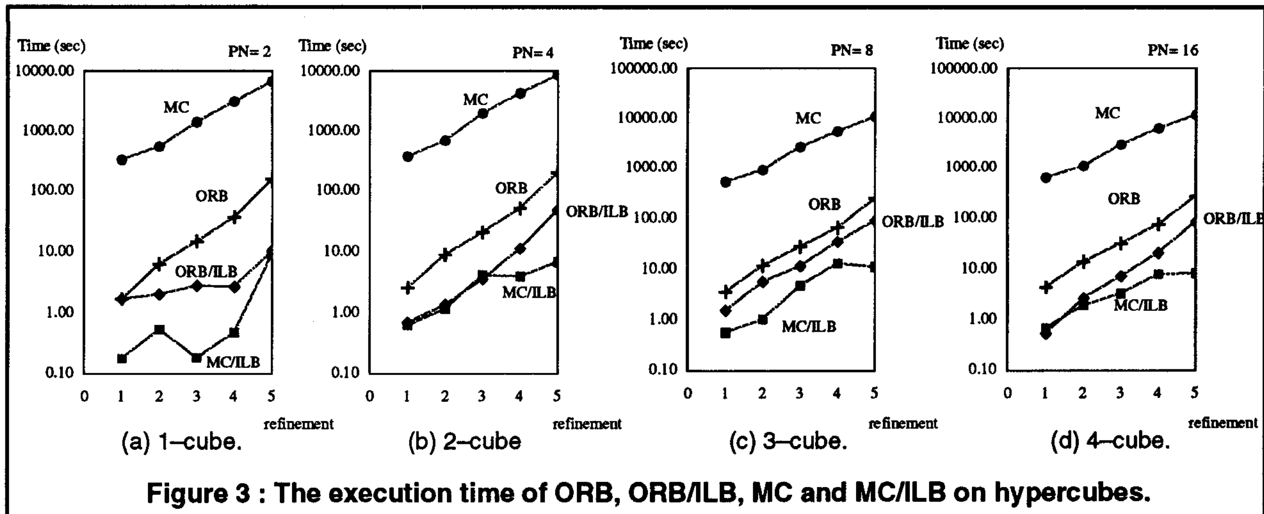
In algorithm *iterative_load_balancing_for_hypercubes,* lines 2 to 7 take constant time. Line 12 takes $|M|$ time, where $|M|$ is the cardinality of set M. Lines 13–15 take T time, where T is maximum number of nodes transferred between processors. Line 16 takes constant time. Let the time for a processor to send (receive) T nodes of data to (from) its adjacent processor in a hypercube take $t_s + T \times t_m$ time, where t_s is the startup time and the t_m is the data transmission time per data. Then line 8 and line 17 take $t_s + T \times t_m$ time. Line 18 takes constant time. Lines 1 to 20 form a loop. This loop is executed n times, where n is the dimension of a hypercube. The complexity of algorithm *iterative_load_balancing_for_hypercubes* is $O(n \times (t_s + T \times t_m))$.

4. Simulation and Experimental Results

We have implemented algorithm *iterative_load_balancing_for_hypercubes* (ILB) on a 16–node NCUBE–2 along with two parallel mapping algorithms, orthogonal recursive bisection (ORB) [11] and recursive min–cut bipartitioning (MC) [2]. All programs are written in EXPRESS C. Three criteria, the execution time of load balancing algorithms, the computation time of an application program under different load balancing algorithms, and the total execution of an application program (under several refinement phases) are used for performance evaluation.

In dealing with finite element meshes, the distributed irregular mesh environment (DIME) [12] is used to generate test samples. To create test samples, an initial finite element mesh, which has 311 nodes, is created by DIME. Then, the initial finite element mesh is refined 5 times. The refined process is carried out by DIME. In each refinement, the corresponding mesh structure is saved to a data file. Those data files will be used as test samples. The number of nodes for test samples are shown in Table 1.

To emulate the execution of a solution–adaptive finite element program on an NCUBE–2, we first read the



(a) 1–cube. (b) 2–cube (c) 3–cube. (d) 4–cube.

Figure 3 : The execution time of ORB, ORB/ILB, MC and MC/ILB on hypercubes.

Table 1 : The number of nodes of test samples.

Sample No.	Number of Nodes
Sample 1 (Initial mesh)	311
Sample 2 (The 1st refinement)	870
Sample 3 (The 2nd refinement)	1824
Sample 4 (The 3rd refinement)	2928
Sample 5 (The 4th refinement)	4671
Sample 6 (The 5th refinement)	9347

mesh structure of the initial finite element mesh (Sample 1). Then, algorithm ORB or MC is applied to map nodes of the initial finite element mesh to processors. After the mapping, the computation for each processor is carried out. In our example, the computation is to solve Laplaces's equation (Laplace solver). Since it is difficult to predict the number of iterations for convergence, we assume that the maximum iterations executed by our Laplace solver is 10000. When the computation is converged, the mesh structure of the first refined finite element mesh (Sample 2) is read. To balance the computational load, ORB or MC or ILB is applied. After a load balancing algorithm in performed, the computation for each processor is carried out. The refinement, load balancing, and computation processes are performed in turn until the execution of a solution–adaptive finite element program in completed.

To evaluate the performance of ORB, MC, and ILB, five cases are considered:

Case 1 : The test samples are executed in sequential.

Case 2 : Nodes in the initial finite element mesh is mapped to processor by ORB. In each refinement, ORB is applied to balance the computational load of processors.

Case 3 : Nodes in the initial finite element mesh is mapped to processor by ORB. In each refinement, ILB is applied to balance the computational load of processors. We use ORB/ILB to represent the proposed load balancing algorithm used in this case.

Case 4 : Nodes in the initial finite element mesh is mapped to processor by MC. In each refinement, MC is applied to balance the computational load of processors.

Case 5 : Nodes in the initial finite element mesh is mapped to processor by MC. In each refinement, ILB is applied to balance the computational load of processors. We use MC/ILB to represent the proposed load balancing algorithm used in this case.

4.1. Comparisons of The Execution Time of ORB, MC, and ILB

The execution time of ORB, ORB/ILB, MC, and MC/ILB for test samples on n–cube are shown in Figure 3, where n = 1, 2, 3, and 4. From Figure 3, we can see that the execution time of MC ranges from hundreds of seconds to a few hours, the execution time of ORB ranges from a few seconds to hundreds of seconds, the execution time of ORB/ILB ranges from a few seconds to tens of seconds, and the execution time of MC/ILB is about a few seconds. Obviously, the execution time of ILB is much less than those of ORB and MC. It is also interesting that the execution time of ORB/ILB is much higher than that of MC/ILB when the number of processors and the number of nodes increased. For example, if the number of processors is 16, the execution time of ORB and MC/ILB for the fifth refinement (sample 5) are 82.70 and 8.14 seconds, respectively, i.e., the execution time of ORB/ILB is 10 times of the execution time of MC/ILB. The possible reason is that the partitions produced by MC is better (nodes in each partition are tied together) than that of ORB, especially for unstructured meshes. When ILB is applied to balance the computational load in each refinement, the time of migration nodes from overloaded processors to underloaded processors for the case where MC is used to map the initial finite element mesh on a hypercube is less than that of the case where ORB is used to map the initial finite element mesh on a hypercube.

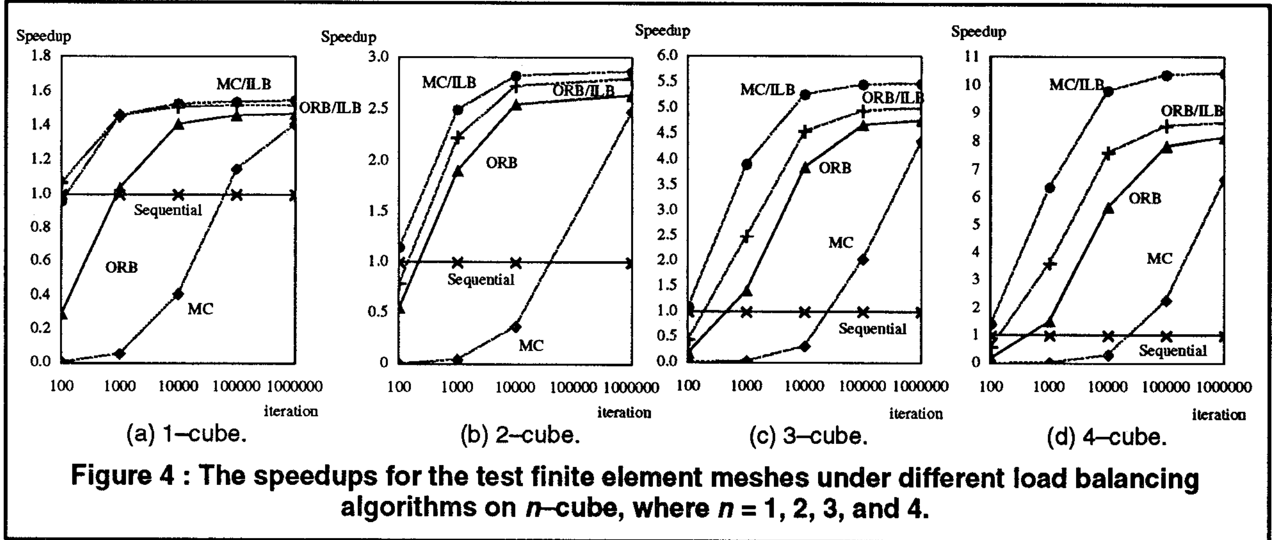
4.2. Comparisons of The Execution Time of Test Samples under Different Load Balancing Algorithms

In Table 2, we show the time for the Laplace solver to execute one iteration (computation + communication) for test samples under different load balancing algorithms

Table 2 : The time for the Laplace solver to execute one iteration (computation + communication) for the test samples under different mapping schemes on hypercubes.

Time unit : 1×10^{-3} second

Cube	algorithm \ sample	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Total
0–cube	sequential	10.8	30.5	64.3	100.0	165.6	335.2	706.4
1–cube	ORB	8.8	27.2	47.4	72.2	105.7	219.4	480.7
	ORB/ILB	8.8	22.7	43.8	66.3	104.2	218.6	464.4
	MC	12.6	27.8	48.9	77.6	113.2	218.5	498.6
	MC/ILB	12.6	25.0	46.4	69.0	104.6	199.4	457.0
2–cube	ORB	5.8	15.5	26.7	39.1	61.9	118.5	267.5
	ORB/ILB	5.8	11.6	22.0	34.3	61.5	117.0	252.2
	MC	9.1	20.1	29.9	41.9	55.4	111.6	268.0
	MC/ILB	9.1	17.9	28.3	37.2	51.2	102.0	245.7
3–cube	ORB	5.5	9.9	14.3	20.0	35.1	63.1	147.9
	ORB/ILB	5.5	8.5	12.8	17.9	34.2	62.4	141.2
	MC	7.0	11.7	15.9	21.9	28.6	55.9	141.0
	MC/ILB	7.0	10.6	15.1	19.5	26.5	50.2	128.9
4–cube	ORB	6.4	9.0	9.1	12.1	17.5	32.1	86.2
	ORB/ILB	6.4	7.3	8.5	9.9	17.8	31.5	81.4
	MC	6.9	8.8	10.5	13.3	15.5	28.2	83.2
	MC/ILB	6.9	8.3	10.0	11.8	14.4	25.3	67.7



(a) 1–cube. (b) 2–cube. (c) 3–cube. (d) 4–cube.

Figure 4 : The speedups for the test finite element meshes under different load balancing algorithms on *n*–cube, where *n* = 1, 2, 3, and 4.

on n–cube, where n = 1, 2, 3, and 4. Let $T_i(S)$ denote the time for the Laplace solver to execute one iteration for Sample i under load balancing algorithm S, where i = 1, 2, ..., 6 and $S \in$ {ORB, ORB/ILB, MC, MC/ILB}. From Table 2, we can see that for Samples 1–3, in general, T_i(ORB/ILB) < T_i(ORB) < T_i(MC/ILB) < T_i(MC), where i = 1, 2, and 3. For Sample 4, T_4(ORB/ILB) < T_4(MC/ILB) < T_4(ORB) < T_4(MC). For Sample 5–6, T_i(MC/ILB) < T_i(MC) < T_i(ORB/ILB) < T_i(ORB), where i = 5 and 6. If we assume that the Laplace solver executes the same number of iterations for each test samples, then $\sum_{i=1}^{6} T_i(MC/ILB) < \sum_{i=1}^{6} T_i(ORB/ILB) < \sum_{i=1}^{6} T_i(ORB) < \sum_{i=1}^{6} T_i(MC)$ for 1–cube and 2–cube; and $\sum_{i=1}^{6} T_i(MC/ILB) < \sum_{i=1}^{6} T_i(ORB/ILB) < \sum_{i=1}^{6} T_i(MC) < \sum_{i=1}^{6} T_i(ORB)$ for 3–cube and 4–cube. From the above observations, ILB produces much better mappings than those of MC and ORB.

4.3. Comparisons of The Total Execution Time for Test Samples

The total execution time of test samples on a hypercube is defined as follows:

$$T_{total}(S) = T_{exec}(S) + \sum_{i=1}^{6} T_i(S) \times iteration_i \quad (2)$$

where $T_{total}(S)$ is the total execution time of test samples under load balancing algorithm S on a hypercube, $S \in$ {ORB, ORB/ILB, MC, MC/ILB}, $T_{exec}(S)$ is the total execution time of load balancing algorithm S for test samples, and $iteration_i$ is the number of iterations executed by the Laplace solver for Sample i. From Equation 2, we can derive the speedup achieved by a load balancing algorithm as follows:

$$Speedup(S) = (\sum_{i=1}^{6} Seq_i \times iteration_i) + (T_{exec}(S) + \sum_{i=1}^{6} T_i(S) \times iteration_i) \quad (3)$$

where $Speedup(S)$ is the speedup achieved by a load balancing algorithm S, $S \in$ {ORB, ORB/ILB, MC, MC/ILB}; and Seq_i is the time for the Laplace solver to execute one iteration for Sample i in sequential. The maximum speedup achieved by a load balancing algorithm S is derived by setting the value of $iteration_i$ to ∞. Then, we have the following equation:

$$Speedup_{max}(S) = \sum_{i=1}^{6} Seq_i + \sum_{i=1}^{6} T_i(S) \quad (4)$$

where $Speedup_{max}(S)$ is the maximum speedup achieved by a load balancing algorithm S and $S \in$ {ORB, ORB/ILB, MC, MC/ILB}.

The speedups for test samples under different load balancing algorithm are shown in Figure 4. Since it is difficult to predict the number of iterations executed by the Laplace solver for test samples, in Figure 4, we assume that the Laplace solver executes the same number of iterations for each test sample. From Figure 4, we can see that, in general, *Speedup*(MC/ILB) > *Speedup*(ORB/ILB) > *Speedup*(ORB) > *Speedup*(MC). We also obverse that if the number of iterations executed by the Laplace solver is less than 10000, *Speedup*(MC) is less than 1. This implies that if the convergence rate of a Laplace solver is fast, MC is not a good load balancing algorithm for a solution-adaptive finite element program. The maximum speedups achieved by load balancing algorithms for test samples on hypercubes are shown in Table 3. From Table 3, we observe that, in general, $Speedup_{max}$(MC/ILB) > $Speedup_{max}$(ORB/ILB) > $Speedup_{max}$(MC) > $Speedup_{max}$(ORB).

Table 3 : The maximum speedups achieved by load balancing algorithms for test samples on n-cube, where n = 1, 2, 3, and 4.

Algorithm \ n-cube	1-cube	2-cube	3-cube	4-cube
ORB	1.47	2.64	4.78	8.19
ORB/ILB	1.52	2.8	5.00	8.68
MC	1.42	2.64	5.01	8.49
MC/ILB	1.55	2.88	5.48	10.43

5. Conclusions

In this paper, a parallel run-time iterative load balancing algorithm (ILB) is proposed to deal with the load imbalancing problem of a solution-adaptive finite element program. The proposed algorithm has three properties. First, the algorithm is simple and easy to implement. Second, the execution of the algorithm is fast. Third, it guarantees that the computational load will be balanced after the execution of the algorithm. We have implemented the proposed algorithm along with two parallel mapping algorithms, parallel orthogonal recursive bisection (ORB) [11] and parallel recursive mincut bipartitioning (MC) [2], on a 16-node NCUBE-2. Three criteria, the execution time of load balancing algorithms, the computation time of an application program under different load balancing algorithms, and the total execution of an application program (under several refinement phases) are used for performance evaluation. Simulation results show that (1) the execution time for ILB is very short compared to those of MC and ORB; (2) the mappings produced by ILB are much better than those of ORB and MC; and (3) the speedups produced by ILB are much better than those of ORB and MC.

References:

[1] M.J. Berger and S.H. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors," *IEEE Trans. on Computers*, Vol. C-36, No. 5, pp. 570-580, 1987.

[2] F. Ercal, J. Ramanujam, and P. Sadayappan, "Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube," *Parallel Computing*, 13, pp. 1-16, 1990.

[3] S. Hammond and R. Schreiber, "Mapping Unstructured Grid Problems to the Connection Machine," Technical Report 90.22, RIACS, October 1990.

[4] A.Y. Grama and V. Kumar, "Scalability Analysis of Partitioning Strategy for Finite Element Graphs: A Summary of Results," *Proceedings of Supercomputing'92*, pp. 83-92, 1992.

[5] F. Harary, *Graph Theory*, Addison Wesley, Mass., 1969

[6] D.Y. Hinz, "A Run-Time Load Balancing Strategy for Highly Parallel Systems," *Proceedings of Distributed Memory Multiprocessor Conference*, pp. 951-961, 1990

[7] D. King and E.J. Wegman, "Hypercube Dynamic Load Balancing," *Proceedings of Distributed Memory Multiprocessor Conference*, pp. 962-965, 1990

[8] L. Lapidus and G.F. Pinder, *Numerical Solution of Partial Differential Equations in Science and Engineering*. New York: Wiley, 1983

[9] P. Sadayappan and F. Ercal, "Nearest-Neighbor Mapping of Finite Element Graphs on Processor meshes," *IEEE Trans. on Computers*, Vol. C-36 No. 12, pp. 1408-1424, 1987.

[10] V.K. Saletore, "A Distributed and Adaptive Dynamic load balancing algorithm for Parallel Processing of Medium-Grain Tasks," *Proceedings of Distributed Memory Multiprocessor Conference*, pp. 994-999, 1990

[11] R.D. Williams, "Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations," *Councurrency : Practice and Experience*, Vol. 3(5), pp. 457-481, October 1991.

[12] R.D. Williams, "DIME: A User's Manual," Caltech Concurrent Computation Report C3P 861, Feb. 1990.