CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 7(7), 615–631 (OCTOBER 1995)

A parallel dynamic load-balancing algorithm for solution-adaptive finite element meshes on 2D tori

YEH-CHING CHUNG, YAA-JYUN YEH AND J.-S LIU
Department of Information Engineering & Computer Science
Feng Chia University
Taichung, Taiwan 407, ROC

SUMMARY

To efficiently execute a finite element program on a 2D torus, we need to map nodes of the corresponding finite element graph to processors of a 2D torus such that each processor has approximately the same amount of computational load and the communication among processors is minimized. If nodes of a finite element graph do not increase 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, nodes of a finite element graph increase 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. In the paper we propose a parallel dynamic load-balancing algorithm (LB) to deal with the load-imbalancing problem of a solution-adaptive finite element program on a 2D torus. The algorithm uses an iterative approach to achieve load-balancing. We have implemented the proposed algorithm along with two parallel mapping algorithms, parallel orthogonal recursive bisection (ORB) and parallel recursive mincut bipartitioning (MC), on a simulated 2D torus. 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 time of an application program (under several refinement phases) are used for performance evaluation. Simulation results show that (1) the execution of LB is faster than those of MC and ORB; (2) the mappings of LB are better than those of ORB and MC; and (3) the speedups of LB are better than those of ORB and MC.

1. INTRODUCTION

The finite element method is widely used for the structural modeling of physical systems[1]. 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 applications. In Figure 1(a), a 40-node finite element graph with 25 4-node finite elements is shown. Due to the properties of computation-intensiveness and computation-locality, it is very attractive to implement the finite element method on distributed-memory multiprocessors[2–4]. In the context of parallelizing a finite element modeling program that uses iterative techniques to solve a system of equations[2,3], a parallel program may be viewed as a collection of tasks represented by the nodes of a finite element graph. Each node represents a particular amount of computation and can be executed independently. In each iteration, a node needs to get data from other nodes in the same finite element before the next iteration can be performed. The communication needed between nodes in the finite element graph of Figure 1(a) is shown in Figure 1(b).

CCC 1040–3108/95/070615–17

Received 29 September 1994
Revised 25 May 1995

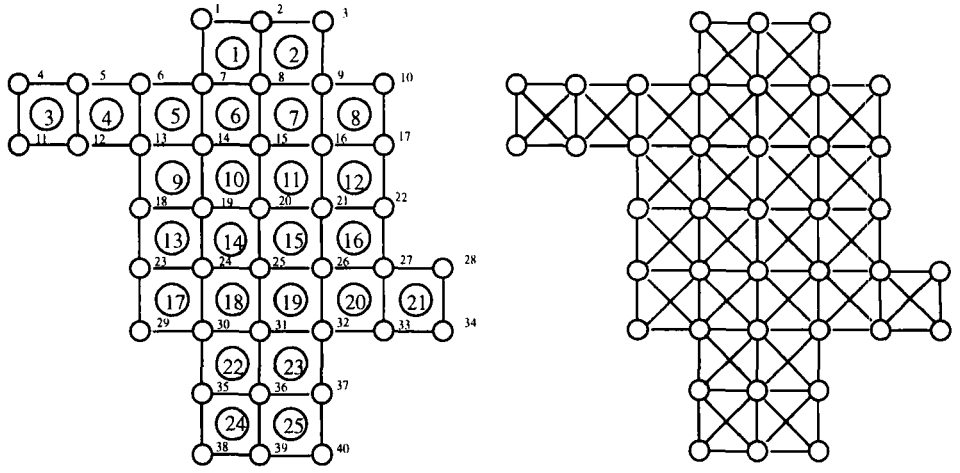
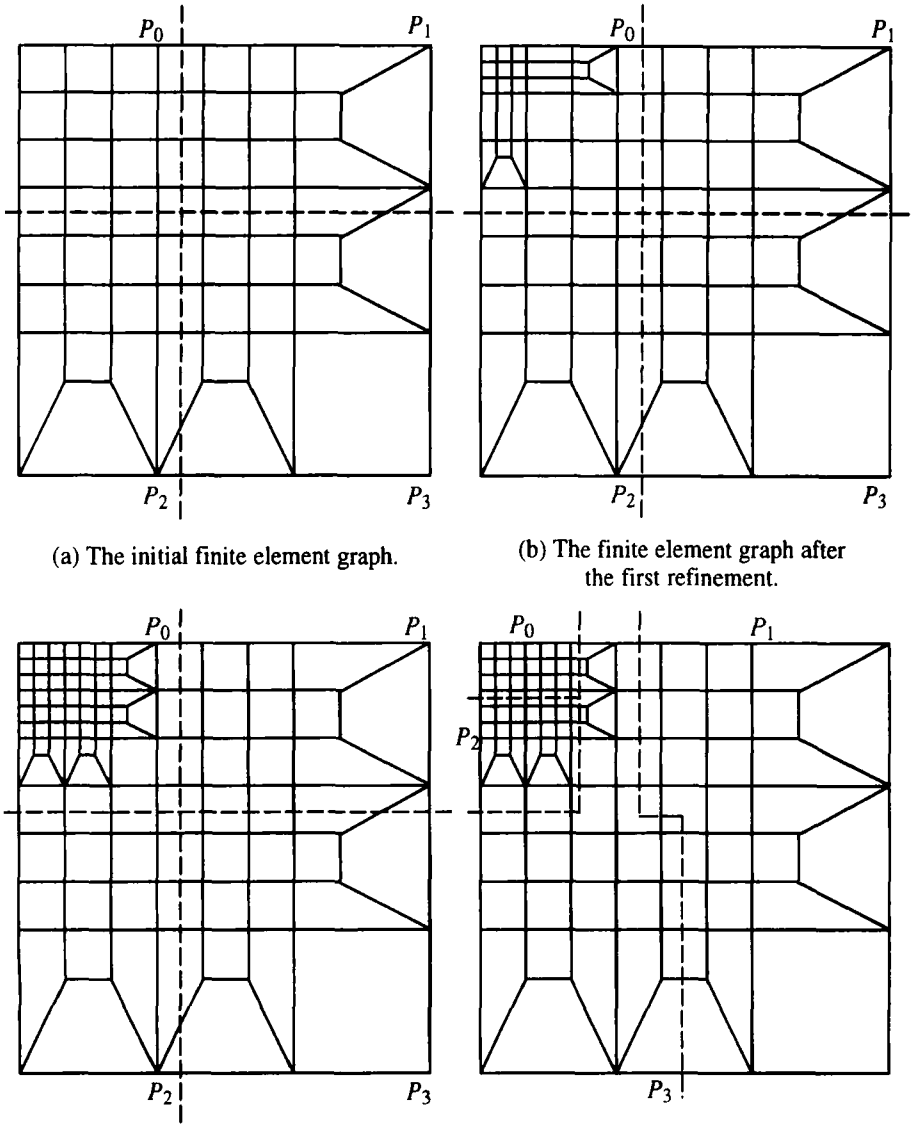


Figure 1. An example of a 40-node finite element graph and the communication needed between nodes: (a) a 40-node finite element graph with 25 finite elements (the circled and uncircled numbers denote the finite element numbers and node numbers, respectively); (b) the communication needed between nodes

To efficiently execute a finite element modeling program on a 2D torus, we need to map nodes of the corresponding finite element graph to processors of a 2D torus such that each processor has approximately the same amount of computational load and the communication among processors is minimized. If nodes of a finite element graph do not increase 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, nodes of a finite element graph increase 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 2, a finite element graph is refined twice during execution. Initially, each processor has 16 nodes. If no load-balancing algorithm is performed, after the first and the second refinement, the number of nodes assigned to P_0 are 36 and 64, respectively, and the number of nodes assigned to P_1, P_2 and P_3 are 16. However, if a load-balancing algorithm is carried out in each refinement, the load may be evenly distributed as shown in Figure 2(d).

In fact, the solution-adaptive finite element problem is a subset of a class of irregular loosely synchronous problems[5]. In [5], types of loosely synchronous problems are classified into *static single phase computations* such as explicit unstructured mesh fluids calculation[6,7], *multiple phase computations* such as unstructured multigrid[8] and particle-in-cell methods[9–10], *adaptive irregular computations* such as solution-adaptive finite element methods[4,11] and molecular dynamics calculations[12], *implicit multiphase loose synchronous computations* such as particle dynamics[13], and *static and dynamic structured problems*. Since data-dependency of algorithms for those problems is determined at run time, a good run-time mapping scheme is critical for the performance of those algorithms.



(a) The initial finite element graph.

(b) The finite element graph after the first refinement.

(c) The finite element graph after the second refinement.

(d) Nodes remapping for (c).

Figure 2. An example of solution-adaptive finite element graph and load redistribution

To solve the load-imbalancing problem of a solution-adaptive finite element program, 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 node remapping or load redistribution algorithm should have two properties. Firstly, its execution is fast. Secondly, it should produce a good mapping. Algorithms for node remapping and load-redistribution are called load-balancing algorithms throughout this paper.

In this paper we present a parallel dynamic load-balancing algorithm to deal with the load-imbalancing problem of a solution-adaptive finite element program. The algorithm uses iterative approach to achieve load-balancing. We have implemented the algorithm on a simulated 2D torus along with two parallel mapping algorithms, *orthogonal recursive bisection*[4] and *recursive min-cut bipartitioning*[14]. 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 time 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 is presented. The definition of a 2D torus and the proposed parallel dynamic load-balancing algorithm are described in Section 3. The comparisons of the proposed parallel dynamic load-balancing algorithm, parallel orthogonal recursive bisection and parallel recursive min-cut bipartitioning are given in Section 4.

2. RELATED WORK

Many finite element mapping algorithms have been addressed in the literature. In [5], a *binary decomposition* approach was used to partition a non-uniform 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[16] proposed a *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 balancing of computational load.

In [17], *a pairwise interchange algorithm* was proposed to map finite element graphs onto a finite element machine[18]. This approach assumes that the number of nodes of a finite element graph is less than or equal to the number of processors of a finite element machine. An initial mapping is generated by assigning node i of a finite element graph to processor i of the finite element machine. Then the pairwise interchange heuristic is applied to minimize the communication cost of processors.

Grama and Kumar[19] presented scalabilty analysis of three finite element graph partitioning strategies, namely *striped partitioning*, *binary decomposition* and *scattered decomposition*. The analysis was performed using the *Isoefficiency* metric, which helps in predicting the performance of these schemes on a range of processors and architectures.

In [20] and [21], a *two-way stripes partition mapping* and a *greedy assignment mapping* algorithm were proposed. The two-way stripes partition mapping tried to minimize the communication cost by assigning a node and its neighbor nodes of a finite element graph to the same processors or neighbor processors of a hypercube. Then a load transfer heuristic was performed to balance the computational load of processors. The greedy assignment

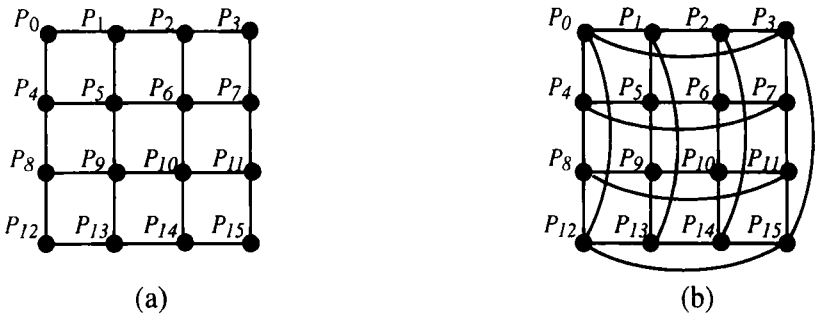


Figure 3. Two-dimensional meshes: (a) a mesh without wrap-around connections: (b) a mesh with wrap-around connections

mapping tried to minimize the communication cost and balance the computational load simultaneously.

Williams[4] proposed three parallel load-balancing algorithms, *orthogonal recursive bisection*, *eigenvector recursive bisection* and *a simple parallel simulated annealing*, to deal with the load-imbalancing problem of a solution-adaptive finite element program. 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 21% in the execution time of a finite element mesh over the mapping produced by orthogonal recursive bisection.

Of the papers mentioned above, only the work of [4] deals with the load-imbalancing problem of a solution-adaptive finite element program. Others assume that nodes of a finite element graph do not change during the execution of a program.

3. THE PROPOSED LOAD-BALANCING ALGORITHM

In this section we first give the definition of a 2D torus. Then we describe the proposed parallel dynamic load-balancing algorithm in detail.

3.1. 2D tori

A *2D torus network* is a variant of the *mesh network*, where nodes are arranged into a two-dimensional lattice. Figure 3(a) illustrates a two-dimensional mesh. A 2D torus allows wrap-around connections between processors on the edge of the mesh. These connections may connect processors in the same row and column (Figure 3(b)). We use $P(a, b)$ to denote the processor in row a and column b of an $m \times n$ torus, where $0 \leq a < m$ and $0 \leq b < n$. In an $m \times n$ torus, each processor has *north*, *east*, *west* and *south* neighbors. The north, east, west, and south neighbors of a processor $P(a, b)$ are $P((a\text{-}1) \bmod m, b)$, $P(a, (b\text{+}1) \bmod n)$, $P(a, (b\text{-}1) \bmod n)$ and $P((a\text{+}1) \bmod m, b)$, respectively, where $0 \leq a < m$ and $0 \leq b < n$. Also, we use P_x to denote the processor $P(x/m, x \bmod n)$ in an $m \times n$ torus. For example, in Figure 3(b), processor P_0 (or $P(0,0)$) has a north neighbor P_{12} (or $P(3, 0)$), an east neighbor P_1 (or $P(0, 1)$), a west neighbor P_3 (or $P(0, 3)$) and a south neighbor P_4 (or $P(1, 0)$).

3.2. A parallel dynamic load-balancing algorithm

Many dynamic load-balancing algorithms have been addressed in the literature[22–25]. However, the problem addressed in this paper is different from that in [22–25]. At run time, the computational load increased in a solution-adaptive finite element program is discrete in nature while that in [22–25] is continuous. Therefore, those approaches proposed in [22–25] cannot efficiently handle the load-imbalancing issue presented in this paper.

In this paper we propose a parallel dynamic load-balancing algorithm to deal with the load-imbalancing problem of a solution-adaptive finite element program. The algorithm uses an iterative approach to achieve load-balancing. For an $m \times n$ torus, if $m > n$ it will first balance the computational load of processors at the same column. Then, it will balance the computational load of processors at the same row. If $m \leq n$, it will balance the computational load of processors at the same row followed by balancing the computational load of processors at the same column. The process of balancing the computational load of processors at the same row is performed as follows. Initially, every processor $P(a, b)$ with even (odd) column co-ordinate will balance its current computational load and the current computational load of its *east* (*west*) neighbor processor, where $0 \leq a < m$ and $0 \leq b < n$. Then, processor $P(a, b)$ will balance its current computational load and the current computational load of its *west* (*east*) processor. The balancing process of processor $P(a, b)$ with its east and west neighbor processors is performed in turn until the computational load of processors at the same row is balanced. The process of balancing the computational load of two adjacent processors $P(a, b)$ and $P(a, c)$ consists of two phases:

Phase 1: Determine the number of nodes needed to be sent from one processor to another.

Phase 2: Perform load transfer while keeping the communication cost of finite element nodes of these two processors as low as possible.

Let the current computational load of processors $P(a, b)$ and $P(a, c)$ be denoted by $load(P(a, b))$ and $load(P(a, c))$, respectively. In phase 1, if $load(P(a, b)) > load(P(a, c))$, then processor $P(a, b)$ needs to send $N = \lceil (load(P(a,b)) - load(P(a,c))) / 2 \rceil$ nodes to processor $P(a, c)$. If $load(P(a, b)) < load(P(a, c))$, then processor $P(a, c)$ needs to send $N = \lceil (load(P(a, c)) - load(P(a, b))) / 2 \rceil$ nodes to processor $P(a, b)$. In phase 2, load transfer is performed. Assume that processor $P(a, b)$ needs to send N nodes to processor $P(a, c)$. In order to minimize the communication cost of finite element nodes of processors $P(a, b)$ and $P(a, c)$, we have the following four cases:

Case 1: Send nodes in $P(a, b)$ that are only adjacent to nodes of $P(a, c)$ to $P(a, c)$.

Case 2: Send nodes in $P(a, b)$ that are not adjacent to nodes of other processors to $P(a, c)$.

Case 3: Send nodes in $P(a, b)$ that are adjacent to nodes of $P(a, c)$ to $P(a, c)$.

Case 4: Send a node in $P(a, b)$ to $P(a, c)$.

To send nodes from processor $P(a, b)$ to processor $P(a, c)$, nodes in case 1 are considered. If nodes in case 1 do not exist, then nodes in case 2 are considered, and so on. Let M denote the set of nodes in $P(a, b)$ that are selected from one of the cases stated above. If $|M|$ is less than N, then nodes adjacent to those of M are selected. If the sum of $|M|$ and the number of nodes adjacent to those of M is less than N, then nodes adjacent to those nodes adjacent to nodes of M are selected. This process is continued until the number of nodes selected is equal to N. Then those nodes are transferred from processor $P(a, b)$ to processor $P(a, c)$.

```
Algorithm dynamic_load_balancing_for_2D_tori()
 /* For an m × n torus, perform the load transfer for every processor P(a,b) */
 if (m ≤ n) then { balance_row_column(m, n, 1); balance_row_column(m, n, 2); }
            else { balance_row_column(m, n, 2); balance_row_column(m, n, 1); }
end_of_dynamic_load_balancing_for_2D_tori
```

```
Algorithm balance_row_column(m, n, row_col)
 1.  i ← 0.
 2.  For every processor P_x = P(a,b) do {
 3.    if (row_col = 1) then
 4.      if (b and i are both even or odd)
 5.        then P_y = P(a, (b+1) mod n) else P_y = P(a, (b–1) mod n)
 6.    else if (b and i are both even or odd)
 7.           then P_y = P((a+1) mod m, b) else P_y = P((a–1) mod m, b)
 8.    Send load(P_x) to P_y and receive load(P_y) from P_y.
 9.    if (load(P_x) < load(P_y)) then { /* P_x needs to receive nodes from P_y */
10.      N ← ⌈(load(P_y) – load(P_x) ÷ 2⌉; load(P_x) ← load(P_x) + N.
11.      Receive N nodes from P_y. }
12.   if (load(P_x) > load(P_y)) then { /* P_x needs to send nodes to P_y */
13.      N ← ⌈(load(P_x) – load(P_y) ÷ 2⌉; load(P_x) ← load(P_x) – N.
14.      M ← ∅. M_1 ← ∅. done = false.
15.      K = the set of nodes assigned to P_y.
16.       do {
17.          M ← M ∪ M_1.
18.          M_1 = the set of nodes of P_x that are only adjacent to nodes of K ∪ M.
19.          if (M_1 <> ∅) then goto L1.
20.          M_1 = the set of a node of P_x that are not adjacent to nodes of other processors.
21.          if (M_1 <> ∅) then goto L1.
22.          M_1 = the set of nodes of P_x that are adjacent to nodes of K ∪ M.
23.          if (M_1 <> ∅) then goto L1.
24.          M_1 = the set of a node of P_x.
25.  L1:     if (|M| + |M_1| < N) then M ← M ∪ M_1 else done = true.
26.        } until (done = true)
27.      M ← M ∪ M_2, where M_2 ⊆ M_1 and |M| + |M_2| = N.
28.      Send the set M to P_y. }
29.    i ← i + 1.
30.  } until (load is balanced)
end_of_balance_row_column
```

Figure 4. The proposed parallel dynamic load-balancing algorithm

The process of balancing the computational load of processors at the same column is similar to that of balancing the computational load of processors at the same row. The proposed algorithm is given in Figure 4.

In algorithm *balance_row_column*, lines 1–10, 12–17, 19, 21, 23–27 and 29–30 take constant time. Lines 18, 20 and 22 take L time, where L is the maximum number of nodes

assigned to processors. Let the time for a processor to send (receive) T nodes of data to (from) its adjacent processor on a 2D torus 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 lines 11 and 28 take $t_s + T \times t_m$ time. Lines 2-23 and 16–26 form loops. The loops are executed $O(m)$ or $O(n)$ and $O(1)$ time, respectively, where m and n are the length of the row and column of the torus. The complexity of algorithm *balance_row_column* is $O((m+n) \times (t_s + T \times t_m))$. The complexity of algorithm *dynamic_load_balancing_for_2D_tori* is $O((m+n) \times (t_s + T \times t_m))$.

We now give an example to show how algorithm *dynamic_load_balancing_for_2D_tori* works. Assume that, initially, we are given a 64-node finite element mesh and nodes of the finite element mesh are evenly distributed to a 1×4 torus as shown in Figure 5(a), that is, each processor is assigned 16 nodes. During the execution, the finite element mesh is refined once. After the refinement, P_0, P_1, P_2 and P_3 have 32, 20, 16 and 16 nodes, respectively, which is shown in Figure 5(b). When algorithm *dynamic_load_balancing_for_2D_tori* is applied, at the first iteration, the number of nodes N which need to be sent or received is calculated for every processor. After the calculation of N, a load transfer heuristic is performed to balance the computational load of processors as shown in Figure 5(c). The calculation of N and the physical load transfer execution for the next iteration are shown in Figure 5(d). After the execution of algorithm *dynamic_load_balancing_for_2D_tori*, nodes are evenly distributed to each processor (as shown in Figure 5(d)).

4. SIMULATION AND EXPERIMENTAL RESULTS

Since we do not have a 2D torus machine and a 2D torus can be embedded in a hypercube, algorithms for 2D torus are implemented on a 16-node NCUBE-2. To embed a $2^x \times 2^y$ torus on an $(x+y)$-cube, the binary reflected Gray code (BRGC) coding scheme is used. The binary reflected Gray code is defined as follows:

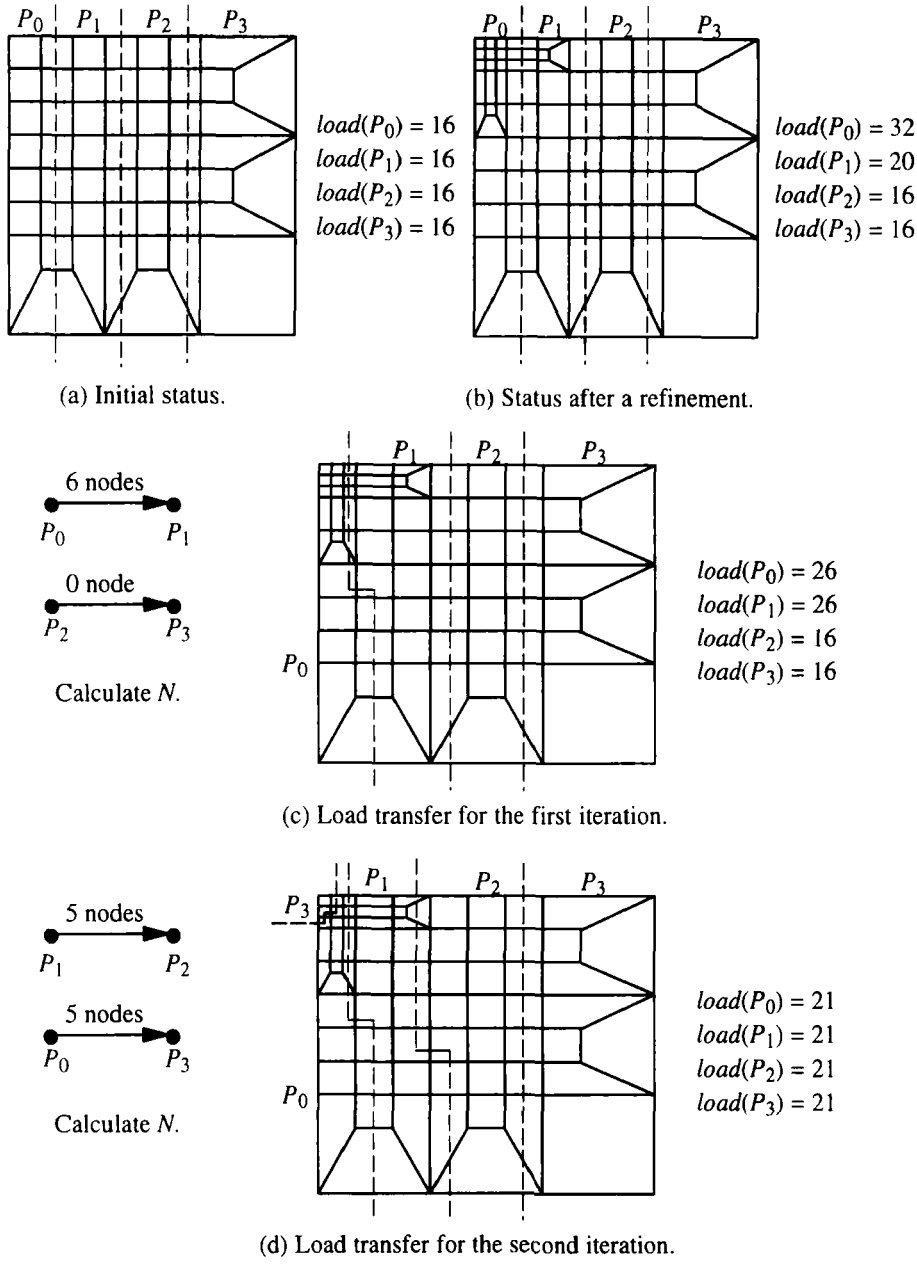
$$N_k = \begin{cases} (0,1) & \text{if } k=1 \\ 0N_{k-1} + 1N_{k-1}^* & \text{if } k>1 \end{cases} \quad (1)$$

where + and * represent *sequence concatenation* and *sequence reversal operations*, respectively. For example, $N_1 = (0,1)$, $N_1^* = (0,1)^* = (1,0)$; $N_2 = 0N_1 + 1N_1^* = 0(0,1) + 1(1,0) = (00,01) + (11,10) = (00, 01, 11, 10)$; $N_3 = (000, 001, 011, 010, 110, 111, 101, 100)$, $N_3(0) = 000$, and $N_3(3) = 010$. Note that $N_k(r)$ denotes the $(r+1)$th element of N_k, where $r = 0,..., 2^k - 1$. To embed a $2^x \times 2^y$ torus in an $(x+y)$-cube, we assign processor $P(i, j)$ of a torus to the processor of an $(x+y)$-cube according to the following equation:

$$P(i,j) = N_x(i) \wedge N_y(j) \quad (2)$$

where $0 \le i \le 2^x - 1, 0 \le j \le 2^y - 1$, and $\wedge$ is the *binary string concatenation operation*. An example of embedding a 2×4 torus in a 3-cube by using the embedding method mentioned above is shown in Figure 6. In Figure 6(b), the addresses of $P(0,2)$ and $P(1,0)$ are $N_1(0) \wedge N_2(2) = 011$ and $N_1(1) \wedge N_2(0) = 100$, respectively.

We have implemented algorithm *dynamic_load_balancing_for_2d_tori* (LB) on a simulated 2D torus along with two parallel mapping algorithms, orthogonal recursive bisection (ORB)[4] and recursive min-cut bipartitioning (MC)[14]. 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



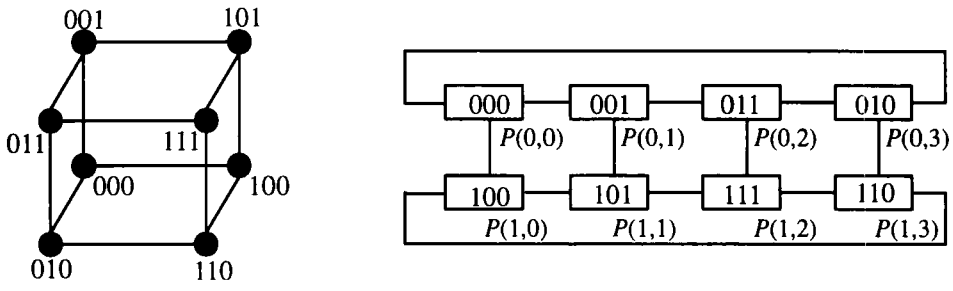
(a) Initial status.

(b) Status after a refinement.

(c) Load transfer for the first iteration.

(d) Load transfer for the second iteration.

Figure 5. The behavior of the parallel dynamic load-balancing algorithm on a 2D torus



(a) A 3–cube. (b) The emulation of a 2×4 2D–torus.

Figure 6. An example of embedding a 2×4 mesh in a 3-cube

execution time 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)[26] is used to generate test samples. 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. Although DIME is a programming environment, in this paper we only use DIME to generate desired finite element meshes.

To create test samples, an initial finite element mesh, which has 310 nodes, is created by DIME. Then, the initial finite element mesh is refined five 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 a simulated 2D torus we first read the 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). The algorithm of solving Laplaces's equation is similar to that of [27]. Since it is difficult to predict the number of iterations for the convergence of a Laplace solver, we assume that the maximum iterations executed by our Laplace solver is 10,000. 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 LB is applied. After a load-balancing algorithm is 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 is completed.

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

Case 1: The test samples are executed sequentially.

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

Table 1. The number of nodes of test samples

Sample no.	Number of nodes
Sample 1 (the initial mesh)	311
Sample 2 (the first refinement)	870
Sample 3 (the second refinement)	1824
Sample 4 (the third refinement)	2928
Sample 5 (the fourth refinement)	4671
Sample 6 (the fifth refinement)	9347

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

Case 4: Nodes in the initial finite element mesh are mapped to processors 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 processors by MC. In each refinement, LB is applied to balance the computational load of processors. We use MC/LB to represent the proposed load-balancing algorithm used in this case.

4.1. Comparisons of the execution times of ORB, MC and LB

The execution time of ORB, ORB/LB, MC and MC/LB for test samples on 1×16, 2×8 and 4×4 tori are shown in Figure 7. From Figure 7 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/LB ranges from a few seconds to tens of seconds, and the execution time of MC/LB is a few seconds. Obviously, the execution time of LB is less than those of ORB and MC. We also observe that the execution times of ORB/LB and MC/LB are high when the torus is not symmetrical. For example, on a 1×16 torus, the execution times of ORB/LB and MC/LB for the fifth refinement (sample 5) are 95.98 s and 22.18 s, respectively. For a 4×4 torus, the execution times of ORB/LB and MC/LB for the fifth refinement (sample 5) are 50.85 s and 11.87 s, respectively. The execution time of the 1×16 torus is almost twice that of the 4×4 torus. This is because for a 4×4 torus the number of steps to reach load-balancing is less than that of a 1×16 torus.

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 on 1×16, 2×8 and 4×4 tori. Let $T_i(S)$ denote the time for the Laplace solver to execute one iteration for sample i under load-balancing algorithm S, where $i = 1, 2, \ldots, 6$ and $S \in \{\text{ORB, ORB/LB, MC, MC/LB}\}$. From Table 2, if we assume that the Laplace solver executes the same number of iterations for each test samples, then $\sum_{i=1}^{6} T_i(MC/LB) < \sum_{i=1}^{6} T_i(ORB/LB) < \sum_{i=1}^{6} T_i(ORB) < \sum_{i=1}^{6} T_i(MC)$ for a 2×8 torus, and $\sum_{i=1}^{6} T_i(MC/LB) < \sum_{i=1}^{6} T_i(ORB/LB) < \sum_{i=1}^{6} T_i(MC) < \sum_{i=1}^{6} T_i(ORB)$ for 1×16 and 4×4 tori. From the above observations,

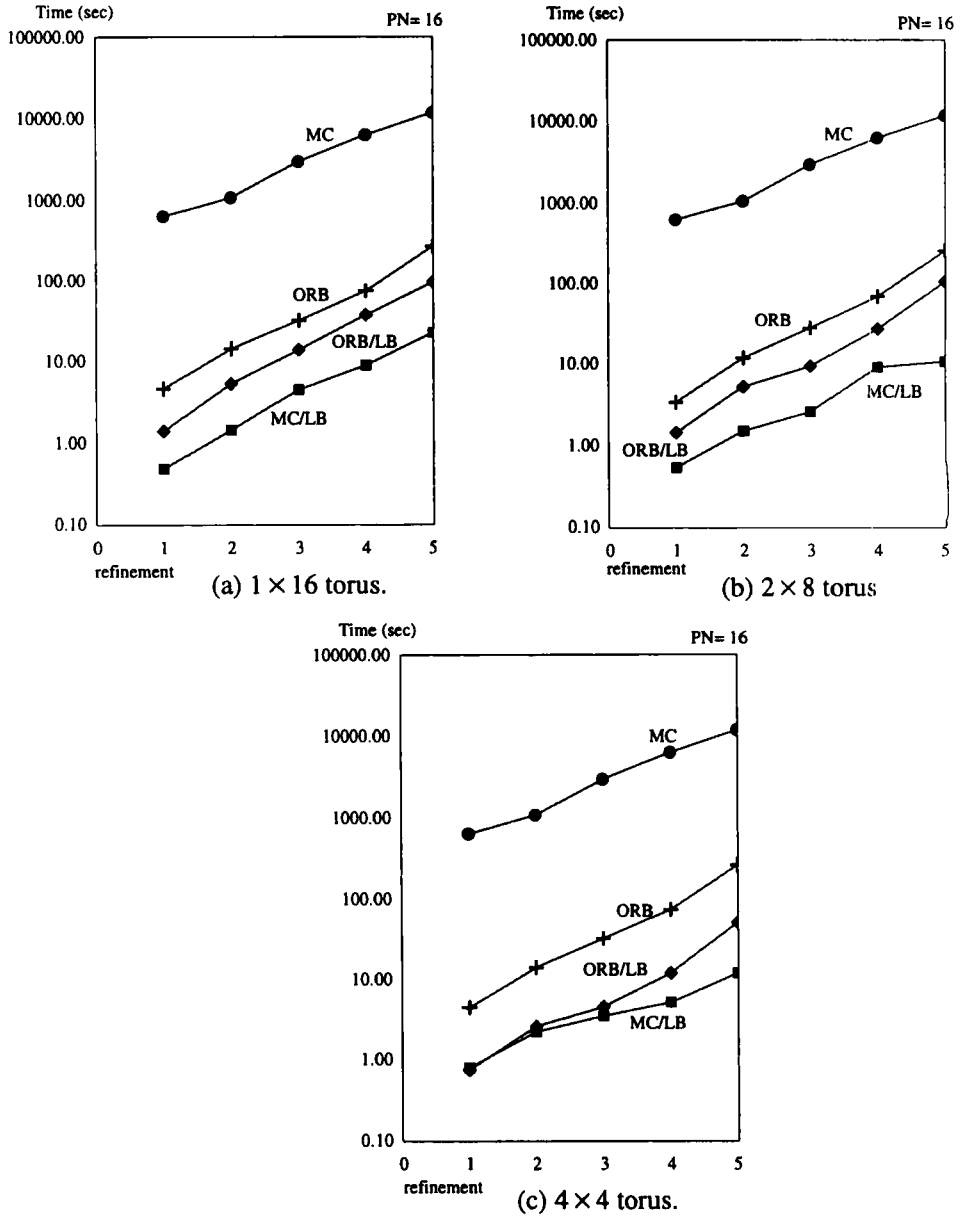


Figure 7. The execution time of ORB, ORB/LB, MC and MC/LB on 2D tori

Table 2. The time for the Laplace solver to execute one iteration (computation + communication) for the test samples under different load-balancing algorithms on 2D tori

Torus	Sample/ Algorithm	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Total
1×1	Sequential	10.8	30.5	64.3	100.0	165.6	335.2	706.4
	ORB	15.3	18.9	22.7	26.9	33.3	51.2	168.3
	ORB/LB	15.2	17.4	21.4	25.4	32.2	51.9	163 6
1×16	MC	14.5	18.6	24.9	25.5	31.9	49	164.4
	MC/LB	14.5	17.2	20.5	24.1	29.2	43.6	149.1
	ORB	10.3	12.8	15.5	18.4	22.7	34.6	114.3
	ORB/LB	10.3	11.8	14.2	17.2	20.3	32.5	106.3
2×8	MC	9	11.7	15.7	21	24.9	33.6	115.9
	MC/LB	9.1	11	12.7	14.8	18	26.7	92.3
	ORB	8.5	10.6	12.6	14.9	18	26.1	90.7
	ORB/LB	8.6	9.6	11.2	12.9	15.6	24.7	82.6
4×4	MC	7	9.3	12.5	16.9	18.1	26	89.8
	MC/LB	7.3	8.6	10.2	11.9	14.5	22.2	74.7

Time unit: 1×10^{-3}s.

LB produces better mappings than those of MC and ORB. One possible reason is that LB uses the locality characteristic of a finite element mesh to do load transfer (see algorithm *dynamic_load_balancing_for_2D_tori*) which results in a better mapping.

4.3. Comparisons of the total execution time for test samples

The total execution time of test samples on a 2D torus is defined as follows:

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

where $S \in \{\text{ORB, ORB/LB, MC, MC/LB}\}$, $T_{total}(S)$ is the total execution time of test samples under load-balancing algorithm S, $T_{exec}(S)$ 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 (3), we can derive the speedup of a load-balancing algorithm as follows:

$$Speedup(S)=(\sum_{i=1}^{6} Seq_i\times iteration_i)\div(T_{exec}(S)+\sum_{i=1}^{6} T_i(S)\times iteration_i) \tag{4}$$

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

$$Speedup_{max}(S)=\sum_{i=1}^{6} Seq_i\div\sum_{i=1}^{6} T_i(S) \tag{5}$$

Table 3. The maximum speedups achieved by load-balancing algorithms for test samples on 2D tori

n-cube Algorithm	1×16	2×8	4×4
ORB	4.22	6.21	7.83
ORB/LB	4.34	6.68	8.59
MC	4.32	6.12	7.90
MC/LB	4.76	7.69	9.50

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

The speedups for test samples under different load-balancing algorithms are shown in Figure 8. Since it is difficult to predict the number of iterations executed by the Laplace solver for test samples, in Figure 8 we assume that the Laplace solver executes the same number of iterations for each test sample. From Figure 8 we can see that, in general, *Speedup*(MC/LB) > *Speedup*(ORB/LB) > *Speedup*(ORB) > *Speedup*(MC). We also observe that, if the number of iterations executed by the Laplace solver is less than 10,000, *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 of load-balancing algorithms for test samples on 2D tori are shown in Table 3. From Table 3, we observe that, in general, $Speedup_{max}$(MC/LB) > $Speedup_{max}$(ORB/LB) > $Speedup_{max}$(MC) > $Speedup_{max}$(ORB). From Figure 8 and Table 3 we can see that the speedups of LB are better than those of MC and ORB.

5. CONCLUSIONS

In this paper, a parallel dynamic load-balancing algorithm (LB) is proposed to deal with the load-imbalancing problem of a solution-adaptive finite element program on a 2D torus. We have implemented the proposed algorithm along with two parallel mapping algorithms, parallel orthogonal recursive bisection (ORB) and parallel recursive min-cut bipartitioning (MC), on a simulated 2D torus. 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 of LB is faster than those of MC and ORB; (2) the mappings of LB are better than those of ORB and MC; and (3) the speedups of LB are better than those of ORB and MC.

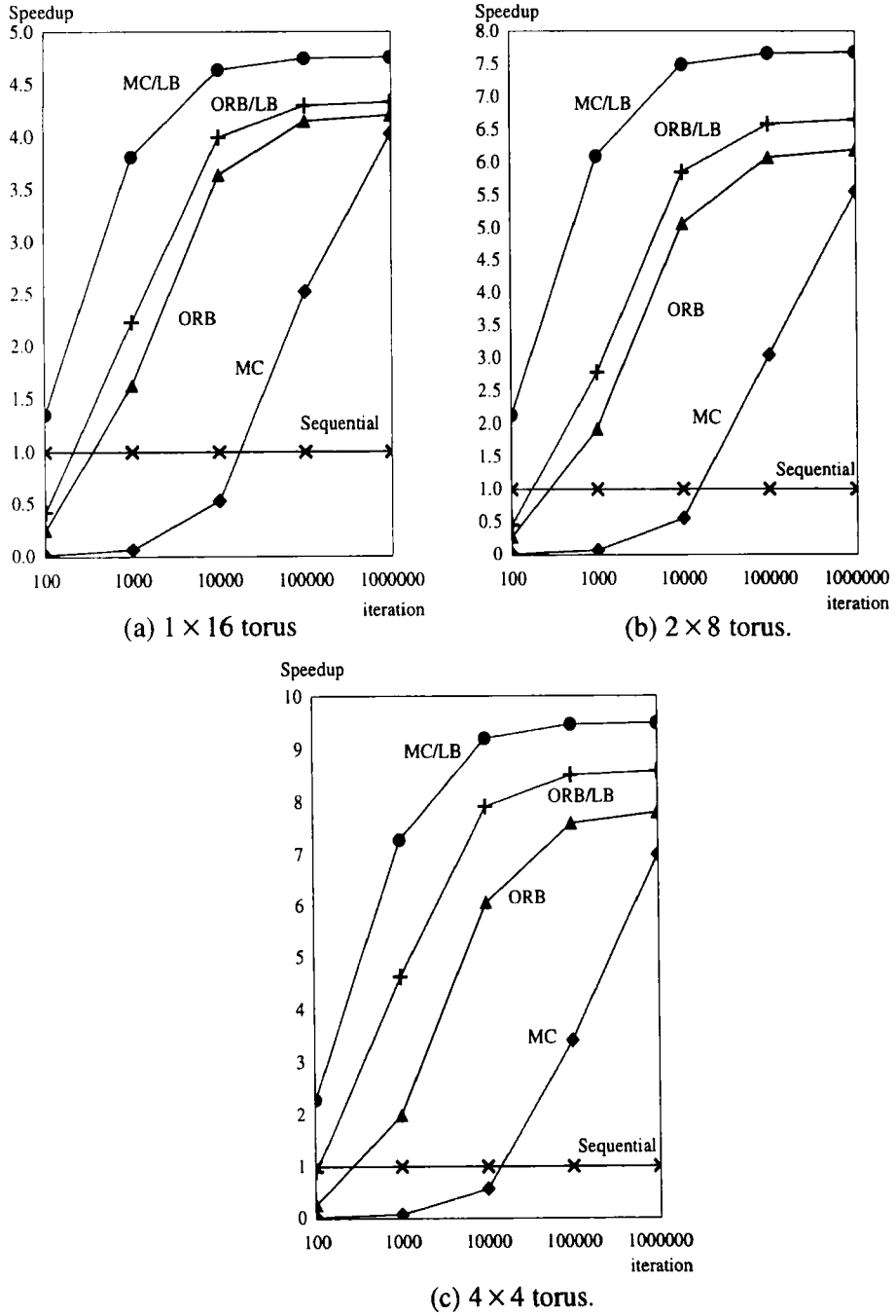


Figure 8. The speedups for the test finite element meshes under different load-balancing algorithms on 2D tori

REFERENCES

1. L. Lapidus and G.F. Pinder, *Numerical Solution of Partial Differential Equations in Science and Engineering*, Wiley, 1983.
2. C. Aykanat, F. Ozguner, S. Martin and S.M. Doraivelu, 'Parallelization of a finite element application program on a hypercube multiprocessor,' *Hypercube Multiprocessor*, 662–673 (1987).
3. C. Aykanat, F. Ozguner, F. Ercal and P. Sadayaooan, 'Iterative algorithms for solution of large sparse systems of linear equations on hypercubes,' *IEEE Trans.*, **C-37**, (12), 1554–1568 (1988).
4. R.D. Williams, 'Performance of dynamic load balancing algorithms for unstructured mesh calculations,' *Councurrency : Pract. Exp.*, **3**, (5), 457–481 (1991).
5. G.C. Fox *et al.*, 'A classification of irregular loosely synchronous problems and their support in scalable parallel software systems,' NPAC-SCCS Technical Report, Syracuse University, April 1992.
6. S. Hammond and R. Schreiber, 'Mapping unstructured grid problems to the connection machine,' Technical Report 90.22, RIACS, October 1990.
7. D.L. Whitaker, D.C. Slack and R.W. Walters, 'Solution algorithms for the two-dimensional Euler equations on unstructured meshes,' *Proceedings of AIAA 28th Aerospace Science Meeting*, Reno Nevada, January 1990.
8. D.J. Mavriplis, 'Three dimensional unstructured multigrid for the Euler equations,' *Proceedings of AIAA 10th Computational Fluid Dynamics Conference*, June 1991.
9. P.C. Liewer, B.A. Zimmerman, V.K. Decyk, J.M. Dawson and G.C. Fox, 'A general concurrent algorithm for plasma particle-in-cell simulations,' *Technical Report C3P-758*, California Institute of Technology, March 1989.
10. P.C. Liewer and V.K. Decyk, 'A general concurrent algorithm for plasma particle-in-cell simulation codes', *J. Comput. Phys.*, **85**, (2), 302–322 (1989).
11. M.J. Berger and A. Jameson, 'Automatic adaptive grid refinement for the euler equations,' *AAIA J.*, **23**, 561–568 (1985).
12. B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan and M. Karplus, 'Charmm : a program for macromolecular energy, minimization, and dynamics calculations,' *J. Comput. Chem.*, **4**, 187 (1983).
13. J. Carrier, L. Greegard and V. Rokhlin, 'A fast adaptive multipole algorithm for particle simulations,' *SIAM J. Sci. Stat. Comput.*, **9**, 669–686 (1988).
14. F. Ercal, J. Ramanujam and P. Sadayappan, 'Cluster partitioning approaches to mapping parallel programs onto a hypercube,' *Parallel Comput.*, **13**, 1–16 (1990).
15. M. J. Berger and A. Jameson, 'A partitioning strategy for nonuniform problems on multiprocessors,' *IEEE Trans.*, **C-36**, (5), 570–580 (1987).
16. P. Sadayappan and F. Ercal, 'Nearest-neighbor mapping of finite element graphs on processor meshes,' *IEEE Trans.*, **C-36**, (12), 1408–1424 (1987).
17. S.H. Bokhari, 'On the mapping problem,' *IEEE Trans.*, **C-30**, 207–214 (1981).
18. H. Jordan, 'A special purpose architecture for finite element analysis,' *Proceedings of International Conference on Parallel Processing*, 1978, pp. 263–266.
19. A.Y. Grama and V. Kumar, 'Scalability analysis of partitioning strategy for finite element graphs: a summary of results,' *Proceedings of Supercomputing '92*, 1992, pp. 83–92.
20. Y.C. Chung and S. Ranka, 'Mapping finite element graphs on hypercubes,' *J. Supercomput.*, **6**, (3), 257–282 (1992).
21. Y.C. Chung and S. Ranka, 'Mapping finite element graphs onto hypercubes,' *Proceedings of Frontier of Massively Parallel Computations*, 1990. pp. 135–144.
22. D.Y. Hinz, 'A run-time load balancing strategy for highly parallel systems,' *Proceedings of Distributed Memory Multiprocessor Conference*, 1990, pp. 951-961.
23. D. King and E.J. Wegman, 'Hypercube dynamic load balancing,' *Proceedings of Distributed Memory Multiprocessor Conference*, 1990, pp. 962–965.
24. V.K. Saletore, 'A distributed and adaptive dynamic load-balancing algorithm for parallel processing of medium-grain tasks,' *Proceedings of Distributed Memory Multiprocessor Conference*, 1990, pp. 994-999.
25. J. Xu and K. Hwang, 'Heuristic methods for dynamic load balancing in a message-passing supercomputer,' *Proceedings of Supercimputing '90*, 1990, pp. 888–897.

26. R.D. Williams, 'DIME: a user's manual,' Caltech Concurrent Computation Report C3P 861, February 1990.
27. I.G. Angus, G.C.Fox, J.S. Kim and D.W. Walker, *Solving Problems on Concurrent Processors*, Vol. 1, Prentice-Hall, 1990.