Proceedings of International Conference on Distributed Systems, Software Engineering and Database Systems

A Parallel Recursive H/V Partitioning Method for Mapping Unstructured Finite Element Graphs on Processor Meshes[1]

Ching-Jung Liao, Chin-Feng Lin, and Yeh-Ching Chung[2]

Department of Information Engineering
Feng Chia University, Taichung, Taiwan 407, ROC
Tel : 886-4-4517250 x2706 Fax : 886-4-4515517
Email : cjliao, cflin, ychung@pine.iecs.fcu.edu.tw

Abstract

To efficiently execute a finite element modeling program on a distributed memory multicomputer, we need to distribute the tasks of a finite element graph to the processors of a distributed memory multicomputer as evenly as possible and minimize the communication cost of processors. In this paper, we present a parallel recursive H/V (Horizontal/Vertical) partitioning method to efficiently map unstructured finite element graphs on processor meshes. Given an unstructured finite element graph and an $m \times n$ processor mesh, this method tries to balance the computation load and minimize the communication cost simultaneously by recursively applying the horizontal/vertical partitioning method to divide the unstructured finite element graph and the processor mesh into two subgraphs and two sub-processor meshes, respectively, until all sub-processor meshes contain one processor. To evaluate the performance of the parallel recursive H/V partitioning method, we have implemented the proposed method on simulated processor meshes along with the nearest-neighbor mapping method [17]. Five unstructured finite element graphs are used as test samples. The simulation results show that the proposed method outperforms the nearest-neighbor mapping method for all test samples.

***Keywords:** Unstructured finite element graphs, processor meshes, mapping, partitioning.*

1 Introduction

In parallel computing, it is important to map a parallel program on a parallel computer such that the total execution time of the parallel program is minimized. In general, a parallel program and a parallel computer can be represented by a task graph and a processor graph, respectively. For a task graph, nodes represent tasks of a parallel program and edges denote the data communication needed between tasks. The weights associated with the nodes and edges represent the computation and the communication cost, respectively. For a processor graph, nodes and edges denote processors and communication channels, respectively. By using the graph model, the mapping problem can be treated as the task allocation problem. In the task allocation problem, we try to distribute the tasks of a parallel program to the processors of a parallel computer as even as possible and minimize the communication cost among processors. Since the task allocation problem is known to be NP-completeness [9], many heuristic methods were proposed to find satisfactory sub-optimal solutions [7-8, 13-14, 16].

The finite element method is widely used for the structural modeling of physical systems [15]. In the finite element model, an object can be viewed as a finite element graph, which is a connected and undirected graph that consists of a number of finite elements. Each finite element is composed of a number of nodes. The number of nodes of a finite element is determined by an application. Due to the properties of computation-intensiveness and computation-locality, it is very attractive to implement the finite element method on distributed memory multicomputers [2-3, 11-12].

In the context of parallelizing a finite element modeling program that uses iterative techniques to solve system of equations [2-3], a parallel program may be viewed as a collection of tasks represented by nodes of a finite element graph. Each node represents a particular amount of computation and can be executed independently. In each iteration, a node needs to get data from other nodes in the same finite element before the next iteration can be performed. To efficiently execute a finite element modeling program on a distributed memory multicomputer, we need to distribute the tasks of a finite element graph to the processors of a distributed memory multicomputer as evenly as possible

[1] The work of this paper was partially supported by NCHC of R.O.C. under contract NCHC-86-08-021.
[2] Correspondence addressee.

(the load balancing criterion) and minimize the communication cost of processors (the minimum communication cost criterion).

Many finite element graph mapping methods have been addressed in the literature. In [5], *a pairwise interchange* method was proposed to map finite element graphs on a finite element machine. This approach assumes that the number of nodes of a finite element graph is fewer 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.

Berger and Bokhari [4] proposed a *binary decomposition* method to partition a non-uniform mesh into modules so that each module has the same amount of computation load. These modules were then mapped on meshes, trees, and hypercubes. This method does not try to minimize the communication cost.

In [6], a *two-way stripes partition mapping* and a *greedy assignment mapping* algorithms were addressed. The two-way stripes partition mapping tries 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 computation load of processors. The greedy assignment mapping tried to minimize the communication cost and balance the computation load simultaneously.

Grama and Kumar [10] presented scalability analysis of three finite element graph partitioning strategies, namely striped partitioning, binary decomposition, and scattered decomposition. The analysis was performed by using the *Isoefficiency* metric, which helps in predicting performance of these schemes on a range of processors and architectures.

In [17], a *nearest-neighbor mapping* approach was proposed to map planar finite element graphs on processor meshes. This approach uses the *stripes partition* (*stripes mapping*) strategy to minimize the communication cost of processors and then uses the *boundary refinement* heuristic to balance the computation load of processors. However, the boundary refinement heuristic does not guarantee the balancing of the computation load.

Williams [18] 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 executing of parallel simulated annealing is time consuming. But the mapping produced by simulated annealing saves of 21% in the execution time of a finite element mesh than the mapping produced by orthogonal recursive bisection.

In this paper, we propose a parallel recursive H/V (Horizontal/Vertical) partitioning method to efficiently map unstructured finite element graphs on processor meshes. Given an unstructured finite element graph and an $m \times n$ processor mesh, this method tries to balance the computation load and minimize the communication cost simultaneously by recursively applying the horizontal/vertical partitioning method to divide the unstructured finite element graph and the processor mesh into two subgraphs and two sub-processor meshes, respectively, until all sub-processor meshes contain one processor. To evaluate the performance of the parallel recursive H/V partitioning method, we have implemented the proposed method on simulated processor meshes along with the nearest-neighbor mapping method [17]. Five unstructured finite element graphs are used as test samples. The simulation results show that the proposed method outperforms the nearest-neighbor mapping method for all test samples.

The paper is organized as follows. The definitions and terminology used in this paper will be given in Section 2. In Section 3, we will present the cost model of mapping unstructured finite element graphs on processor meshes. The parallel recursive H/V partitioning method will be described in details in Section 4. In Section 5, the performance evaluation and simulation results will be presented. The conclusions and future work will be given in Section 6.

2 Preliminaries

2.1 Finite Element Graphs

The finite element method is widely used to solve partial differential equations by using either a direct or an iterative approach. In the finite element model, an object can be viewed as an finite element graph, which is a connected and undirected graph that consists of a number of finite elements. The shape and the number of nodes of a finite element are determined by applications. In this paper, we consider the mapping of two-dimensional finite element graphs on processor meshes.

Definition 1: In an finite element graph, two nodes *node*(x) and *node*(y) are *adjacent* if (*node*(x), *node*(y)) is an edge of the finite element graph.

Definition 2: In an finite element graph, two nodes *node*(x) and *node*(y) are *neighbors* if *node*(x) and *node*(y) are in the same finite element

In Figure 1, an example of a 21-node finite element graph consisting of 24 finite elements is shown.

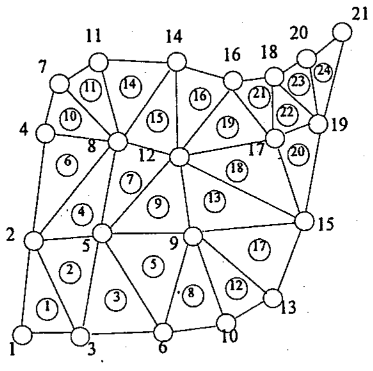


Figure 1. An example of a 21-node FEG with 24 finite elements (the circled and uncircled numbers denote the finite element numbers and node numbers, respectively).

2.2 Processor Meshes

In a processor mesh, processors are arranged into a two-dimensional lattice. Figure 2 illustrates a two-dimensional mesh. Processors in the boundary of a mesh is called *boundary* processors. For those processors other than boundary processors in a mesh are called *internal* processors.

Definition 3: Two processors are *adjacent* processors in a mesh if there is a link between them.

Definition 4: Two processors are *neighbor* processors in a mesh if they are adjacent processors or one processor is in the northeast, or the northwest, or the southeast, or the southwest of the other.

To represent the processors of a mesh, we use the row-major labeling method. Given an $m \times n$ processor mesh, when the row-major labeling method is applied, processor in the ith row and the jth column is labeled as P_k, where $i = 1$ to m, $j = 1$ to n, and $k = (i-1) \times n + (j-1)$. For example, the processor in the second row and the third column of the processor mesh shown in Figure 2 is labeled as P_7 when the row-major labeling method is applied. In Figure 2, the adjacent processors of P_6 are P_1, P_5, P_7, and P_{11}. The neighbor processors of P_7 are P_1, P_2, P_3, P_6, P_8, P_{11}, P_{12}, and P_{13}.

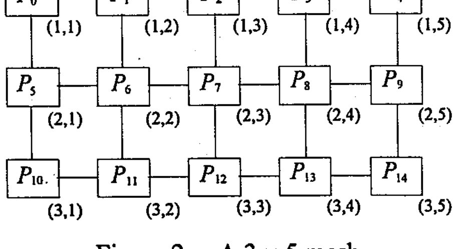


Figure 2. A 3 × 5 mesh.

3 The Cost Model of Mapping Unstructured Finite Element Graphs on Processor Meshes

To map an N-node unstructured finite element graph on an $M = m \times n$ processor mesh, we need to assign nodes of an unstructured finite element graph to processors of a processor mesh. There are M^N mapping ways. The execution time of an unstructured finite element graph on a processor mesh under a particular mapping MAP_i can be defined as follows:

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

where $T_{par}(MAP_i)$ is the execution time of a parallel finite element program on a processor under mapping MAP_i, $T_{comp}(MAP_i, P_j)$ is the computation cost of processor P_j under mapping MAP_i, and $T_{comm}(MAP_i, P_j)$ is the communication cost of processor P_j under mapping MAP_i, where $i = 1, \ldots, M^N$ and $j = 0, \ldots, M-1$.

The cost model used in Equation 1 is assuming a synchronous mode of communication in which each processor goes through a computation phase followed by a communication phase. Therefore, the computation cost of processor P_j under a mapping MAP_i can be defined as follows:

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

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

In our communication model, we assume that every processor can communicate with all other processors in one step. This assumption is valid for most of state-of-the-art parallel computers using the wormhole routing techniques. In general, it is possible to overlap communication with computation. In this case, $T_{comm}(MAP_i, P_j)$ may not always reflect the true communication cost since it would be partially overlapped with computation. However, $T_{comm}(MAP_i, P_j)$ should provide a good estimate for the communication cost. Since we use synchronous communication mode, $T_{comm}(MAP_i, P_j)$ can be defined as follows:

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

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

Let T_{seq} denote the execution time of an unstructured finite element graph on a processor mesh with one processor. The speedup of a mapping MAP_i is defined as

$$SpeedUp(MAP_i) = \frac{T_{seq}}{T_{par}(MAP_i)} \qquad (4)$$

The objective of mapping an unstructured finite element graph on a processor mesh is to minimize the execution time or maximize the speedup of a finite element modeling program, that is, $min\{T_{par}(MAP_i)\}$ or $max\{SpeedUp(MAP_i)\}$, where $i = 1, ..., M^N$. From Equation 1, we know that the processor with the maximal summation value of the computation cost and the communication cost determines the execution time of a finite element modeling program on a processor mesh under a particular mapping. There are three ways to minimize the execution time of a finite element modeling program on a processor mesh.

Method 1 : First minimize the communication cost, then balance the computation load.

Method 2 : First balance the computation load, then minimize the communication cost.

Method 3 : Minimize the communication cost and balance the computation load simultaneously.

4 The Parallel Recursive H/V Partitioning Method

Given an N-node unstructured finite element graph, G, and an $M = m \times n$ processor mesh, H, the parallel recursive H/V partitioning method tries to balance the computation load and minimize the communication cost simultaneously by recursive applying the horizontal/vertical partitioning method to divide the unstructured finite element graph and the processor mesh into two subgraphs and two sub-processor meshes, respectively, until all sub-processor meshes contain one processor.

Initially, all nodes of a finite element graph are in processor P_0. If $m \geq n$, then a horizontal partitioning method is applied. For the processor mesh, the horizontal partitioning method divides the processor mesh into two sub-processor meshes H_1 and H_2, where H_1 and H_2 have $\lfloor m/2 \rfloor \times n$ and $\lceil m/2 \rceil \times n$ processors, respectively. For the unstructured finite element graph, the horizontal partitioning method divides the unstructured finite element graph into two subgraphs, G_1 and G_2, where G_1 and G_2 have $\lfloor m/2 \rfloor / m \times N$ and $N - (\lfloor m/2 \rfloor / m \times N)$ nodes, respectively. To divide G into G_1 and G_2, nodes of G are first labeled by using the *horizontal stripe labeling method.* The horizontal stripe labeling method is performed as follows:

Step 1: Initially, let the value of *label* be 1.

Step 2: Let *WL* be the set of nodes in G without labels. For those nodes in *WL* with the minimal y coordinate, find node *node*(s) that has the minimal x coordinate and label *node*(s) as *label*.

Step 3: For those nodes that are neighbors of *node*(s), find the node *node*(t) that has the minimal y coordinate. If there are more than two such nodes, choose one of them randomly.

Step 4: If the x coordinate of *node*(t) is greater than that of *node*(s), then *node*(t) is labeled as *label*. Let *node*(s) be *node*(t) and continue Step 3. Otherwise, goes to Step 5.

Step 5: Increase the value of *label* by 1. For those nodes that are neighbors of nodes with label *label*−1, label them as *label*. Continue Step 5 until no more neighbor nodes can be found.

Step 6 : If there are nodes without labels, increase the value of *label* by 1. Continue Step 2.

After each node in G is labeled, nodes with label 1 are assigned to G_1. If the number of nodes assigned to G_1 is less than $\lfloor m/2 \rfloor / m \times N$, nodes with label 2 are assigned to G_1, and so on, until the number of nodes assigned to G_1 is equal to $\lfloor m/2 \rfloor / m \times N$. Nodes that are not assigned to G_1 are assigned to G_2. After G is divided into G_1 and G_2, G_1 and G_2 are assigned to H_1 and H_2, respectively, that is nodes of G_1 and G_2 are assigned to processors P_0 and P_k, respectively, where $k = \lfloor m/2 \rfloor \times n$.

If $m < n$, then a vertical partitioning method is applied. For the processor mesh, the vertical partitioning method divides the processor mesh into two sub-processor meshes H_1 and H_2, where H_1 and H_2 have $m \times \lfloor n/2 \rfloor$ and $m \times \lceil n/2 \rceil$ processors, respectively. For the unstructured finite element graph, the vertical partitioning method divides the unstructured finite element graph into two subgraphs, G_1 and G_2, where G_1 and G_2 have $\lfloor n/2 \rfloor / n \times N$ and $N - (\lfloor n/2 \rfloor / n \times N)$ nodes, respectively. To divide G into G_1 and G_2, nodes of G are first labeled by using the *vertical stripe labeling method.* The vertical stripe labeling method is similar to the horizontal stripe labeling method and is performed as follows:

Step 1: Initially, let the value of *label* be 1.

Step 2: Let *WL* be the set of nodes in G without labels. For those nodes in *WL* with the minimal x coordinate, find node *node*(s) that has the minimal y coordinate and label *node*(s) as *label*.

Step 3: For those nodes that are neighbors of *node*(s), find the node *node*(t) that has the minimal x coordinate. If there are more than two such nodes, choose one of them randomly.

Step 4: If the y coordinate of *node*(t) is greater than that of *node*(s), then *node*(t) is labeled as *label*. Let *node*(s) be *node*(t) and continue Step 3. Otherwise, goes to Step 5.

Step 5: Increase the value of *label* by 1. For those nodes that are neighbors of nodes with label *label*–1, label them as *label*. Continue Step 5 until no more neighbor nodes can be found.

Step 6 : If there are nodes without labels, increase the value of *label* by 1. Continue Step 2.

After each node in G is labeled, nodes with label 1 are assigned to G_1. If the number of nodes assigned to G_1 is less than $\lfloor n/2 \rfloor / n \times N$, nodes with label 2 are assigned to G_1, and so on, until the number of nodes assigned to G_1 is equal to $\lfloor n/2 \rfloor / n \times N$. Nodes that are not assigned to G_1 are assigned to G_2. After G is divided into G_1 and G_2, G_1 and G_2 are assigned to H_1 and H_2, respectively, that is nodes of G_1 and G_2 are assigned to processors P_0 and P_k, respectively, where $k = \lfloor n/2 \rfloor$.

After a horizontal/vertical partitioning method is applied once, processors P_0 and P_k continue applying the horizontal/vertical partitioning method on nodes assigned to them recursively until all sub-processor meshes contain one processor. The algorithm is given as follows.

```
algorithm_parallel_H/V_partitioning($P_i$, $G$, $H$, $m$, $n$, $n'$, $N$)
/*  Initially, $P_i = P_0$ and $n' = n$.  */
 1. if (($m = 1$) && ($n = 1$)) then exit.
 2. For each processor, if ($my\_rank = P_i$) {
 3.     if ($m \ge n$) then {
 4.         Divide $H$ into two sub-processor meshes $H_1$
            and $H_2$, where $H_1$ and $H_2$ have $\lfloor m/2 \rfloor \times n$ and
            $\lceil m/2 \rceil \times n$ processors, respectively.
 5.         Divide $G$ into two subgraphs $G_1$ and $G_2$ by
            using the horizontal partitioning method,
            where $G_1$ and $G_2$ have $\lfloor m/2 \rfloor / m \times N$ and $N -$
            $(\lfloor m/2 \rfloor / m \times N)$ nodes, respectively.
 6.         $k = i + \lfloor m/2 \rfloor \times n'$.    $N_1 = \lfloor m/2 \rfloor / m \times N$.
            $N_2 = N - (\lfloor m/2 \rfloor / m \times N)$.
 7.         doall {
 8.                 parallel_H/V_partitioning($P_i$, $G_1$, $H_1$,
                                       $\lfloor m/2 \rfloor$, $n$, $n'$, $N_1$).
 9.                 parallel_H/V_partitioning($P_k$, $G_2$, $H_2$,
                                       $\lceil m/2 \rceil$, $n$, $n'$, $N_2$). }
10.     } /* end of then part */
11.     else {
12.         Divide $H$ into two sub-processor meshes $H_1$
            and $H_2$, where $H_1$ and $H_2$ have $m \times \lfloor n/2 \rfloor$ and $m$
            $\times \lceil n/2 \rceil$ processors, respectively.
13.         Divide $G$ into two subgraphs $G_1$ and $G_2$ by
            using the vertical partitioning method, where
            $G_1$ and $G_2$ have $\lfloor n/2 \rfloor / n \times N$ and $N - (\lfloor n/2 \rfloor /$
            $n \times N)$ nodes, respectively.
14.         $k = i + \lfloor n/2 \rfloor$.    $N_1 = \lfloor n/2 \rfloor / n \times N$.
            $N_2 = N - (\lfloor n/2 \rfloor / n \times N)$.
15.         doall {
16.                 parallel_H/V_partitioning($P_i$, $G_1$, $H_1$,
                                       $m$, $\lfloor n/2 \rfloor$, $n'$, $N_1$).
17.                 parallel_H/V_partitioning($P_k$, $G_2$, $H_2$,
                                       $m$, $\lceil n/2 \rceil$, $n'$, $N_2$). }
18.     } /* end of else part */
19. }
end_of_parallel_H/V_partitioning
```

In algorithm *parallel_H/V_partitioning*, initially we assume that an N-node unstructured finite element graph G and an $M = m \times n$ processor mesh H are given. Lines 7 to 9 and 15 to 17 form two **doall** clauses. Statements in **doall** clauses are executed in parallel. Lines 1, 2, 3, 4, 6, 11, 12, and 14 take constant time. Lines 5 and 13 take $O(N)$ time. Lines 8, 9, 16, and 17 are recursive calls. The depth of the recursive calls is $O(\log m + \log n)$. Lines 8 and 16 take $O((\log m + \log n) \times N)$ time. The recursive calls of lines 9 and 17 involve data communication between processors P_i and P_k. The total time to perform data communication is $O((\log m + \log n) \times T_{setup} + N \times T_c)$, where T_{setup} and T_c are the setup time and data transmission time per byte of the I/O channel, respectively. Lines 9 and 17 take $O((\log m + \log n) \times N + (\log m + \log n) \times T_{setup} + N \times T_c)$ time. The time complexity of algorithm *parallel_H/V_partitioning* is $O((\log m + \log n) \times N + (\log m + \log n) \times T_{setup} + N \times T_c)$.

5 Performance Evaluation and Simulation Results

Since we do not have a processor mesh, we simulate a processor mesh on a SP2 parallel machine. In a SP2 parallel machine, the interconnection network used is a crossbar network. Therefore, it is quite easy to embed a processor mesh on a SP2 machine using the row-major labeling method. We have implemented the parallel recursive H/V partitioning algorithm (H/V) on simulated processor meshes along with the nearest-neighbor mapping (NNM) method proposed in [17]. The proposed algorithm is written in C and MPI communication primitives and the nearest-neighbor mapping algorithm is written in C.

In dealing with the unstructured finite element graphs, the distributed irregular mesh environment (DIME) [19] is used. DIME is a programming environment for doing distributed calculations with unstructured triangular meshes. The mesh covers a two-dimensional manifold, whose boundaries may be defined by straight lines, arcs of circles, or Bezier cubic sections. In also provides functions for creating, manipulating, and refining unstructured triangular meshes. Since the number of nodes in an unstructured triangular mesh cannot over 10,000 in DIME, in this paper, we only use DIME to generate the initial test samples. From the initial test samples, we use our refining algorithms and data structures to generate the desired test samples. The five test samples used for the performance evaluation are *simple*, *letter*, *hook*, *font*, and *truss*. The shapes of the five test samples are shown in Figure 3. The number of nodes and elements for each test sample are shown in Table 1. For the presentation purpose, the number of nodes and the number of finite elements shown in Figure 3 are less than those shown in Table 1.

To compare the performance of the proposed method with the nearest-neighbor mapping method, nodes of the test samples are first distributed to processors of processor meshes using both methods. Then, 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 [1]. 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 are 1000.

Table 2 shows the execution time of the test samples on processor meshes under both mapping methods. According to Equation 4, we can calculate the speedups of the test samples under both mapping methods. The speedups show that the proposed method outperforms the nearest-neighbor mapping method for all the cases. The main reason is that the nearest-neighbor mapping method needs to maintain the *neighbor mapping* property. According to [17], a mapping is called a neighbor mapping if any two neighbor nodes in a finite element graph are assigned to the same processor or two neighbor processors of a processor mesh under the mapping. Since the nearest-neighbor mapping needs to maintain the neighbor mapping property, the nodes may not be equally assigned to each processor. This will result in a load imbalancing situation, that is, increase the overall computation time. In our method, nodes are guaranteed to evenly distributed to each processor. Therefore, the speedups produced by our method are 10% ~ 25% better than those of the nearest-neighbor mapping method for most of the cases.

6 Conclusions and Future Work

To efficiently execute a finite element modeling program on a distributed memory multicomputer, we need to distribute the tasks of a finite element graph to the processors of a distributed memory multicomputer as evenly as possible and minimize the communication cost of processors. In this paper, we have presented a parallel recursive H/V (Horizontal/Vertical) partitioning method to efficiently map unstructured finite element graphs on processor meshes. Given an unstructured finite element graph and an $m \times n$ processor mesh, this method tries to balance the computation load and minimize the communication cost simultaneously by recursively applying the horizontal/vertical partitioning method to divide the unstructured finite element graph and the processor mesh into two subgraphs and two sub-processor meshes, respectively, until all sub-processor meshes contain one processor. The cost model of mapping finite element graphs on processors meshes were also described. To evaluate the performance of the parallel recursive H/V partitioning method, we have implemented the proposed method on simulated processor meshes along with the nearest-neighbor mapping method [17]. Five unstructured finite element graphs were used as test samples. The simulation results show that the proposed method outperforms the nearest-neighbor mapping method for all test samples.

In this paper, we only consider the mapping of two-dimensional finite element graphs on processor meshes. In the future, we will explore the methods of mapping three-dimensional finite element graphs on distributed memory multicomputers.

References :

1. I.G. Angus, G.C.Fox, J.S. Kim, and D.W. Walker, *Solving Problems on Concurrent Processors*, Vol. 1, N. J.: Prentice-Hall, 1990.
2. C. Aykanat, F. Ozguner, S. Martin, and S.M. Doraivelu, "Parallelization of a Finite Element Application Program on a Hypercube Multiprocessor," *Hypercube Multiprocessor*, pp. 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 Transactions on Computers*, Vol. 37, No. 12, pp. 1554-1568, 1988
4. M.J. Berger and S.H. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors," *IEEE Transactions on Computers*, Vol. C-36, No. 5, pp. 570-580, 1987.
5. S.H. Bokhari, "On the mapping problem," *IEEE Trans. on Computers*, Vol. C-30, pp. 207-214, 1981.
6. Y.C. Chung and S. Ranka, "Mapping Finite Element Graphs on Hypercubes," *The Journal of Supercomputing*, Vol. 6, No.3, pp. 257-282, 1992.

7. F. Ercal, J. Ramanujam, and P. Sadayappan, "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning," *Journal of Parallel and Distributed Computing*, 10, pp. 35-44, 1990.
8. F. Ercal, J. Ramanujam, and P. Sadayappan, "Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube," *Parallel Computing*, 13, pp. 1-16, 1990.
9. M.R. Garey and D.S. Johnson, Computers and Intractability, A Guide to Theory of NP-Completeness. San Francisco, CA: Freeman, 1979.
10. 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.
11. S. Hammond and R. Schreiber, "Mapping Unstructured Grid Problems to the Connection Machine," Technical Report 90.22, RIACS, October 1990.
12. H. Jordan, "A special purpose architecture for finite element analysis," *Proceedings of International Conference on Parallel Processing*, pp. 263-266, 1978.
13. B.W. Kernigham and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," Bell Syst. Tech. J., Vol. 49, No. 2, pp. 292-370, February 1970.
14. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, Vol. 220, pp. 671-680, May 1983.
15. L. Lapidus and C.F. Pinder, *Numerical Solution of Partial Differential Equations in Science and Engineering*, New York: Wiley, 1983.
16. H. Muhlenbein, M. Gorges-Schleuter, and O. Kramer, "New Solutions to the Mapping Problem of Parallel Systems: The Evolution Approach," *Parallel Computing*, 4, pp. 269-279, 1987.
17. P. Sadayappan and F. Ercal, "Nearest-Neighbor Mapping of Finite Element Graphs on Processor meshes," *IEEE Transactions on Computers*, Vol. C-36 No. 12, pp. 1408-1424, 1987.
18. R.D. Williams, "Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations," *Concurrency : Practice and Experience*, Vol. 3(5), pp. 457-481, October 1991.
19. R.D. Williams, "DIME: A User's Manual," Caltech Concurrent Computation Report C3P 861, Feb. 1990.

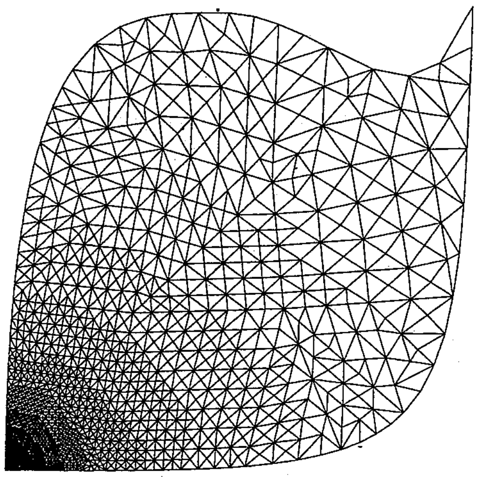
Table 1. The number of nodes and elements of the test samples.

Samples	#node	#element
Simple	50019	99003
Letter	84895	167625
Hook	64212	126569
Font	76336	151368
Truss	46751	91968

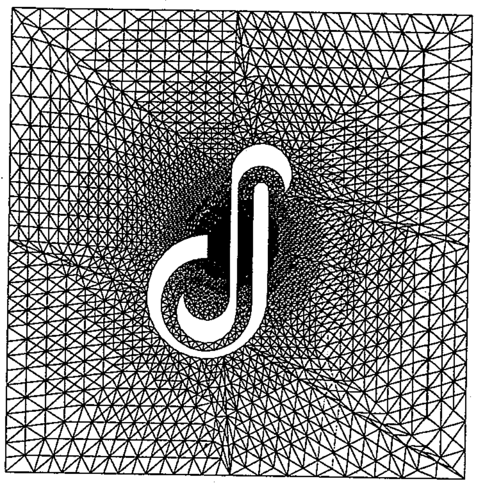
Table 2. The execution time of the test samples on processor meshes under both mapping methods.

Samples (nodes)	Algorithms	Processor meshes							
		1 × 1	2 × 3	7 × 2	3 × 5	9 × 2	4 × 5	5 × 6	4 × 8
Simple (50019)	H/V	66.193	13.714	6.086	5.73	4.985	4.541	3.186	2.94
	NNM	66.193	15.638	7.029	6.881	5.816	5.332	3.29	3.13
Letter (84895)	H/V	100.701	18.943	8.597	8.121	7.868	6.394	4.953	4.544
	NNM	100.701	20.836	10.449	9.841	9.177	7.709	5.078	5.017
Hook (64212)	H/V	74.784	13.920	6.142	5.699	4.759	4.383	3.068	3.003
	NNM	74.784	15.636	6.815	6.337	5.349	4.848	4.003	3.596
Font (76336)	H/V	94.198	17.924	8.581	8.346	7.313	6.844	3.849	3.639
	NNM	94.198	18.868	9.363	9.928	8.721	7.025	4.402	4.119
Truss (46751)	H/V	54.607	9.910	4.008	4.039	3.367	3.036	2.485	2.299
	NNM	54.607	11.580	5.032	4.772	4.055	3.645	3.034	2.685

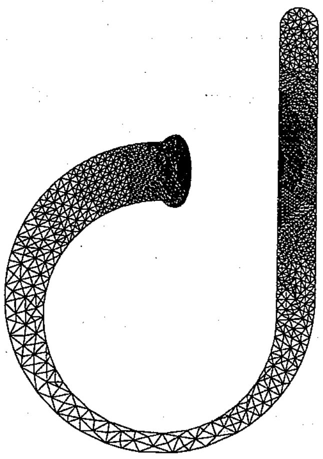
Time unit : second



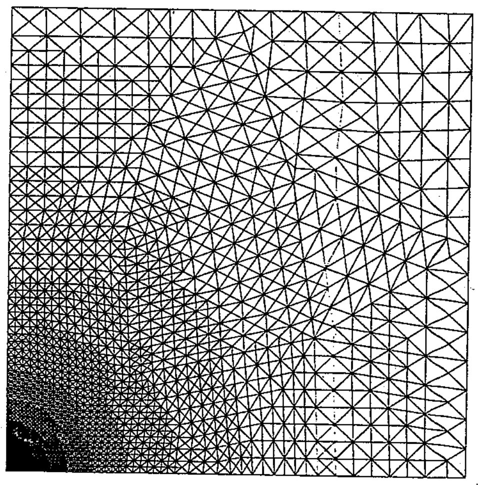
(a) *Simple* (2850 nodes, 5493 elements)



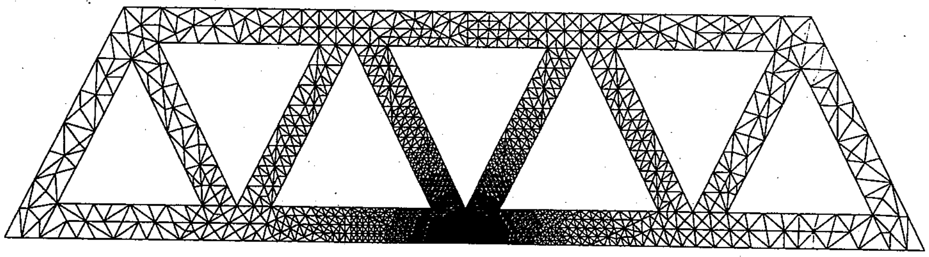
(b) *Letter* (6075 nodes, 11597 elements)



(c) *Hook* (1849 nodes, 3411 elements)



(d) *Font* (5648 nodes, 10992 elements)



(e) *Truss* (7325 nodes, 14024 elements)

Figure 3. The test samples.