Fourth Eurographics Workshop on Parallel Graphics and Visualization (2002)
D. Bartz, X. Pueyo, E. Reinhard (Editors)

An Out-of-core Method for Computing Connectivities of Large Unstructured Meshes

Shyh-Kuang Ueng† and K. Sikorski‡

Abstract

Adjacency graphs of meshes are important for visualizing or compressing unstructured scientific data. However, calculating adjacency graphs requires intensive memory space. For large data sets, the calculation becomes very inefficient on desk-top computers with limited main memory. In this article, an out-of-core *method is presented for finding connectivities of large unstructured FEA data sets. Our algorithm composes of three stages. At the first stage, FEA cells are read into main memory in blocks. For each cell block read, cell faces are generated and distributed into disjoint groups. These groups are small enough such that each group can reside in main memory without causing any page swapping. The resulted groups are stored in disk files. At the second stage, the face groups are fetched into main memory and processed there one after another. Adjacency graph edges are determined in each face group by sorting faces and examining consecutive faces. The edges contained in a group are kept in a disk file. At the third stage, edge files are merged into a single file by using external merge sort, and the connectivity information is computed.*

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Geometric Algorithms

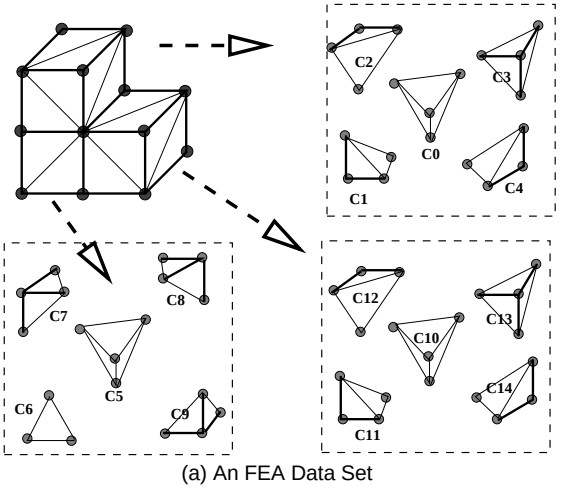
1. Introduction

The meshes produced in Finite Element Analysis (FEA) are usually unstructured. There is no effective method to determine neighboring cells on the fly. *Adjacency graphs* are used to represent the connectivities of FEA meshes. In an adjacency graph, each FEA cell is represented by a vertex. If two cells are neighbors, an edge is created between the corresponding vertices. Thus cell connectivities are stored in the adjacency graph. An FEA data set and its adjacency graph are shown in Figure 1. The adjacency graph composes of 15 vertices and 16 edges. Since the cells are tetrahedra, each vertex has at most 4 edges. Adjacency graphs are important data structures for visualizing and compressing unstructured scientific data. In *ray-casting* visualization methods [2, 3, 5], rays are cast into data sets and marched among cells to collect information. The connectivity information of cells is used to decide which cells should be penetrated by a ray. In *projection visualization methods* [7, 11, 13, 20], data cells are projected onto the image space to create volume rendering images, based on their *visibility order*. The connectivities of cells can be used to compute the visibility order of cells [21]. Streamline visualization is helpful in analyzing vector field data [1, 6, 9, 17, 18]. In order to construct a streamline, it is necessary to trace the streamline from one cell to another. The connectivity information enables us to decide which cell contains the streamline. In [4, 12, 16], three data compression methods were proposed to compress FEA meshes. These methods calculate cell connectivities and traverse adjacency graphs to convert FEA meshes into tetrahedral strips or other similar structures. Then the tetrahedral strips are encoded into bit-strings or byte-strings to reduce the size of data set.

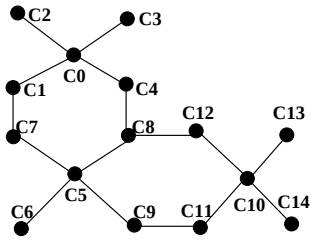
However, cell connectivities are not always available in FEA data. For example, the FEA data file format of the flow analysis software toolkit, *FAST*, does not include cell connectivities[19]. In [14], a linear time algorithm was designed to calculate the connectivities of FEA meshes. This algorithm is an incore method, in which all data structures are resident in main memory while the calculation is carried out. It is very efficient if the memory space is large enough to hold all data structures. However, if the data size becomes

† Department of Computer Science, National Taiwan Ocean University, No. 2, Pei-Ning Road, Keelung, Taiwan, skueng@cs.ntou.edu.tw

‡ School of Computing, University of Utah, Salt Lake City, Utah 84112, USA, sikorski@cs.utah.edu



(a) An FEA Data Set



(b) Adjacency Graph of The Data Set

Figure 1: *An FEA Data Set and its Adjacency Graph*

large and the required space exceeds the main memory capacity, *page-swapping* will occur to bring data back and forth between disk and main memory. Therefore the performance is badly damaged. Some testing results are exhibited in Figure 2. In the tests, a cube was split into different numbers of tetrahedral cells. The incore method was employed to compute the connectivities. The elapsed times of the computation were measured by using wall-clock time. The target machine is a SUN Sparc 5 workstation with 32 mega-byte main memory. As the figure depicts, the incore algorithm performs well when the data size is less than $200,000$ cells. However, the time complexity increases dramatically when the data size exceeds $250,000$ cells. It takes more than 30 minutes to process a data set of 1 million of cells. Thus, the incore method is not a practical algorithm for finding cell connectivities for large data sets on a computer with limited memory space.

In [15], a parallel algorithm was proposed to compute adjacency graphs for very large FEA data sets. In this method, the data structures were subdivided into groups. Each group was small enough to fit into the main memory of a single processor. Furthermore, a group contained all required information for finding a subgraph of the adjacency graph, and no data exchange was necessary during the computation. Therefore these groups would be efficiently processed in parallel.

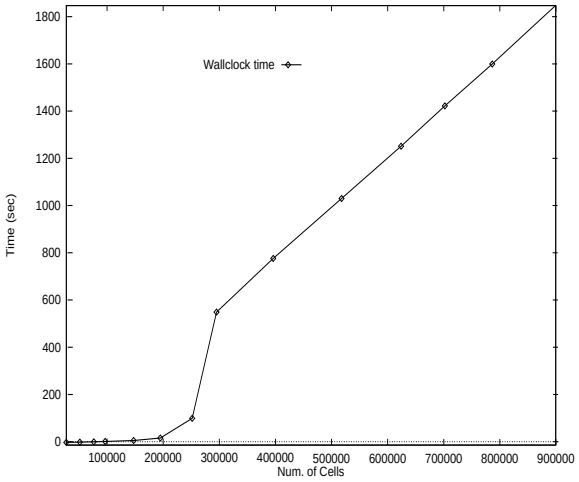


Figure 2: *Costs of Incore Method for Computing Adjacency Graphs*

in [8], an out-of-core method was presented to construct topological data structures for triangle meshes. In their method, vertex indices are absent in the input file. Hash tables are used to calculate connectivities for vertices and edges. In order to fit hash tables into main memory, hash tables are built and partitioned into blocks on the fly. A random access to a larger hash table is confined to a small partition of the hash table. Input data are then divided into groups and processed. At final stage, the results are merged and the target data structures are generated. In this article, an out-of-core method is presented for the calculation of connectivities for very large unstructured meshes. The out-of-core method composes of three stages, the data-division stage, the connectivity calculation stage, and the merging stage. At the first stage, the cell faces of data set are generated and divided into disjoint groups such that each group is small enough to fit into main memory. These groups are processed one after another to calculate adjacency graph edges at the second stage. The edges of each group are stored in a disk file. At the third stage, the graph edges of all groups are sorted and merged into a single file by using external merge-sort, and the adjacency graph is formed. This paper is organized as follows: In the next section, the essential data structures and strategies for calculating adjacency graphs are described. In section three, the three stages of the out-of-core method are presented. Analysis and testing results are given in section four. Conclusions are summarized in the last section.

2. Background

2.1. The Incore Method

In this article, we assume that the input data contains tetrahedral cells only. If other types of cells are present, they are

divided into tetrahedra. A tetrahedral cell consists of four vertices. It can be defined by a quadruple of vertex indices, $< V_0, V_1, V_2, V_3 >$. Usually, these indices can be listed in any order. Therefore, this cell is not uniquely defined. To avoid ambiguity, these vertex indices are listed in ascending order. If they do not appear in ascending order in the input data set, they are sorted at a preprocessing stage. A tetrahedral cell has four faces. A face composes of three vertices. It can be identified by a triple of vertex indices $< V_0, V_1, V_2 >$. As in the cell definition, these indices are enumerated in ascending order such that each face has an unambiguous definition. The four faces of a cell are enumerated based on the lexicographical orders of their definitions. By adopting these rules, a cell as well as its four faces are uniquely defined and indexed.

Two cells are adjacent by sharing a face. Therefore, cell connectivities can be determined by verifying whether a common face of two arbitrary cells exists or not. This strategy was adopted by the incore algorithm of [14] to compute adjacency graphs. At first, the cell faces of the input data are enumerated and stored in face records. A face record includes five integers: The first three integers are the vertex indices of the face. The fourth integer is the index of the face in its master cell, and the fifth integer is the index of its master cell in the data set. For each cell, four face records are generated to represent its four faces. If a face is shared by two cells, then two face records are produced for this face. The vertex indices in the two face records are the same, though the face indices and cell indices are different. After being enumerated, the face records are kept in a list. This list is sorted by using bucket sort. The vertex indices are treated as the keys of the sorting. After the sorting, the two face records of a shared face will be adjacent in the list since they have the same vertex indices. Therefore, by examining every two consecutive face records, the edges of adjacency graph are determined. An example is illustrated in Figure 3 and Figure 4. In Figure 3, a data set of two cells is displayed in part (a). The cell faces are enumerated and drawn in part (b). The structure of face record is drawn in part (c). The face records of the cell are depicted in part (d). In Figure 4, part (a) shows the face records before sorting. The sorted list is contained in part (b). By examining the sorted list, we find that the two cells are adjacency by sharing face $< V_0, V_2, V_3 >$, which is the $0th$ face of cell $C1$ and the $2nd$ face of cell $C0$. Thus, $C0$ is the $0th$ neighbor of $C1$, and $C1$ is the $2nd$ neighbor of $C0$. A table, called the *adjacency table* is utilized to store the adjacency graph. Each table entry includes 4 integers which are the indices of the four neighboring cells. The adjacency table of the data set is shown in part (c). A -1 in a field of a table entry means that the neighboring cell does not exist and the corresponding face is an exterior face.

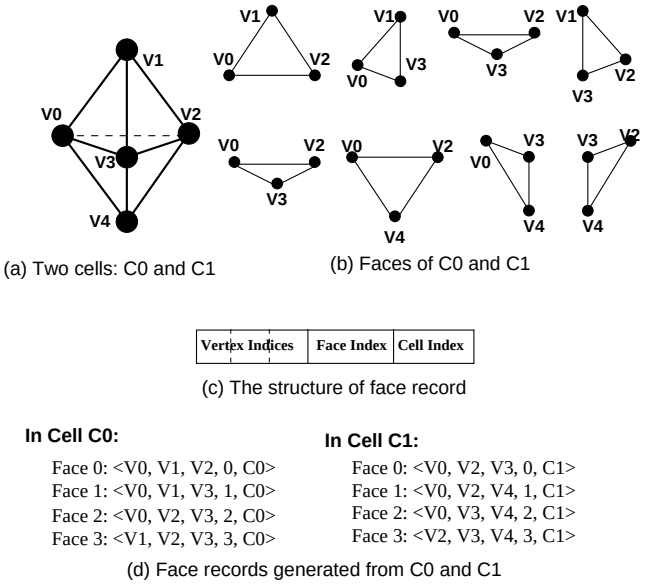
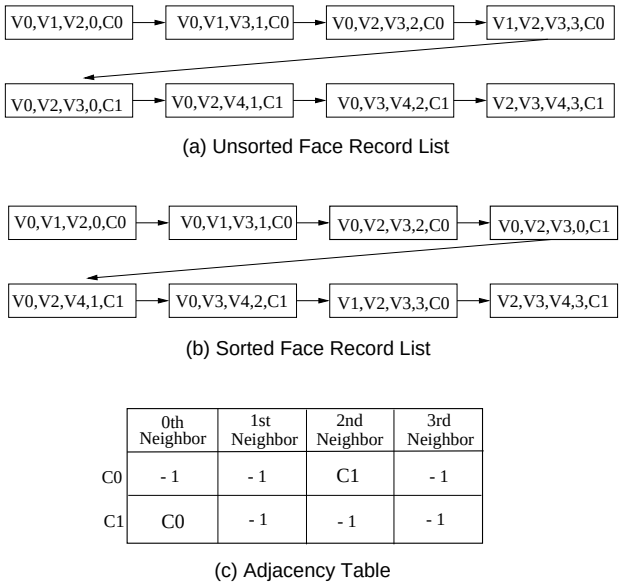


Figure 3: *Face Records of Tetrahedral Cells*



	0th Neighbor	1st Neighbor	2nd Neighbor	3rd Neighbor
C0	- 1	- 1	C1	- 1
C1	C0	- 1	- 1	- 1

(c) Adjacency Table

Figure 4: *Calculating Adjacency Graph*

2.2. Memory Requirement for the Incore Method

For a data set with N cells, there are $4N$ face records generated. Since a face record contains three keys, it takes $O(12N)$ steps to sort the face records by using bucket sort. The incore method is a linear time algorithm for calculating adjacency information of FEA meshes. The memory requirement of the incore method grows when the size of data becomes larger. A face record contains five integers and a pointer. As described in above subsection, the five integers include three vertex indices, one cell index, and one face index. The pointer is served as a link to next face record. If

four bytes are used to represent an integer or a pointer, then a face record requires 24 bytes. For a data set with N cells, $4N$ face records are generated, the memory space for storing the face records is $96N$ bytes. The growth rate of memory requirement can not be ignored. Furthermore, if the data set contains M vertices, the bucket utilized in the bucket sorting needs M entries. Each entry composes of two pointers pointing to the head and the rear of a face record list. Thus the size of a bucket entry is eight bytes, and $8M$ bytes are needed for the bucket. In order to keep cell connectivities, an adjacency table of N entries is needed to store the indices of the neighboring cells for every cell. Each entry is comprised with four integers, and the size of this table is $16N$ bytes. Therefore, the total memory requirement is at least $112N + 8M$ bytes for processing this data set. If $N = 300,000$ and $M = 60,000$, the memory requirement is about 34 Mega-bytes. If this data set is processed by using a machine with only 32 mega-bytes of main memory, page swapping would definitely occur during the computation. Since page swapping is slow, the performance is declined. Figure 2 reveals the influence of memory requirement upon the performance of the incore method. The target machine has only 32 mega-byte memory. As the size of test data reaches $300,000$ cells, the memory space occupied by the data structures of the incore program and the operating system exceeds the capacity of the main memory. Some memory pages are swapped back and forth between disk and the main memory, and the performance drops dramatically.

3. The Out-of-core Method

In order to prevent page swapping, the out-of-core algorithm avoids bringing all data structures into the main memory. Instead, the face records are generated and divided into groups at the first stage. The maximum size of the groups is smaller than a pre-calculated limit such that any group can be processed inside the main memory without causing page swapping. Consequently, the groups are processed one after another to calculate the edges of the adjacency graph. The adjacency table is not explicitly built in the main memory. The graph edges of each group are stored in a disk file. At the final stage, the edges of all groups are sorted and merged into a file to form the adjacency table by using external merge sort.

3.1. Face Record Generation and Data Division

The cell definitions are fetched into the main memory from a disk file in blocks. For each cell read, its vertex indices are sorted and its face records are generated according to our indexing rules. The face records are distributed into K buffers based on a hashing function. When a buffer is full, its face records are written into a disk file. By using this method, the face records of the data set are created and divided into K groups.

To perform the hashing, the three vertex indices of a face record are encoded as a string X of 12 bytes. (If the number of vertices is less than 2^{24}, which is about 16 millions, three bytes are enough to represent a vertex index. The length of the string, X, is 9.) Assume that the string $X = x_0x_1...x_{11}$; then the hash function is defined as:

$$h(X) = (\sum_{i=0}^{11} x_i d^i) \bmod K$$

where $d = 256$, mod is the modulus operator, and K is a pseudo-prime number, $2^n - 1$, . Since K is the number of groups, it must be carefully selected such that the face records are uniformly distributed and the sizes of groups are nearly equal. According to a study in [10], K should be a prime number to ensure that the hash function is random. In our application, a uniform hash function is feasible even though it is not random. Our tests indicated that a pseudo-prime number, K, is suitable for the data division. To avoid overflow, *Horner's Algorithm* is utilized to compute the hash function:

$s = x_{11} \bmod K;$
$for(i = 10; i >= 0; i--)$
    $s = (s * d + x_i) \bmod K;$
$return(s);$

By using the hash function, two face records with the same vertex indices are distributed into the same group. These two face records belong to two different cells, however, they represent a common face shared by the two cells. Therefore these face records yield an edge of the adjacency graph. Subsequently, the necessary information for computing an edge is completely contained in one group. No data dependency exists between two groups. Information exchange among groups is not required.

3.2. Calculating Adjacency Graph Edges

At the second stage, face record groups are processed one by one. For each group, the face records of the group are sorted by using bucket sort. The strings of vertex indices are treated as keys. A string is decomposed into 6 sub-keys such that each sub-key contains two bytes. Since the possible value of a sub-key is within 2^{16}, the size of the bucket is only $65,536$, and no longer grows with the number of vertices in the data set. The benefit is that the memory space for the bucket is reduced from $8M$, where M is the number of vertices, to only 512 K-bytes, though the number of sub-keys is increased and more iterations of bucket sorting are needed.

After sorting the face records, the adjacency graph edges are decided by sequentially examining the sorted face record list. If two consecutive face records have the same vertex indices, a common face is detected and an adjacency graph edge is found. Two *edge structures* are created to record the connectivity information for the two cells sharing the common face. An edge structure contains two integers and one byte. The two integers are the indices of the

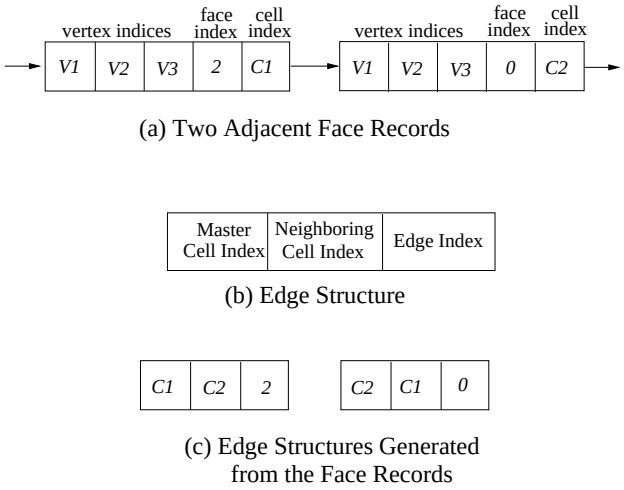


Figure 5: *Edge Record Generation*

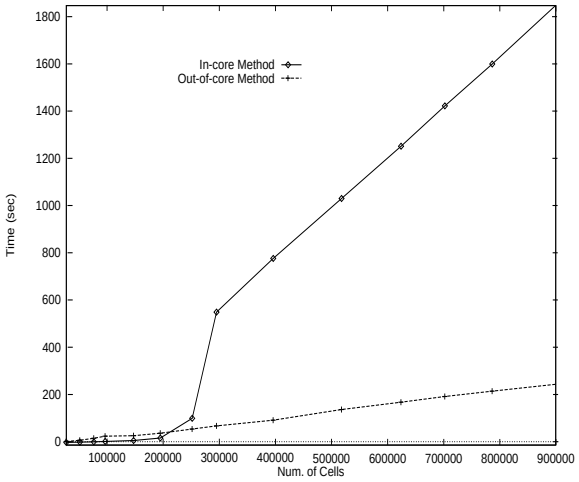


Figure 6: *Out-of-core Method vs. Incore Method*

cells. The first cell is called the *master cell* of the edge structure. The second integer is the index of the other cell. The two cells serve as the master cells of the two edge structures respectively. The byte represents the index of the edge in its master cell. An example is illustrated in Figure 5. In this example, two face records, $< V1, V2, V3, 2, C1 >$ and $< V1, V2, V3, 0, C2 >$, appear successively in the sorted face record list. Therefore, cells $C1$ and $C2$ are connected by an edge in the adjacency graph. The first face record is the 2nd face record of $C1$. Thus, $C2$ is designated as the 2nd neighbor of $C1$, and the edge is indexed as the 2nd edge of $C1$. On the other hand, $C1$ is the $0th$ neighbor of $C2$. Therefore, this edge is the $0th$ edge of $C2$. When all edge structures of a face record group are generated, they are sorted by using *quicksort*. The master cell index is treated as key in the sorting. The sorted edge structures are stored in a disk file.

3.3. Merging Adjacency Graph Edges

At the third stage, adjacency graph edges are merged to create the adjacency table. The merging is accomplished by using a two-way merge sort. The sorted edge structures in two disk files are merged into a larger sorted disk file. The merging is iteratively carried out until only one sorted file left. Totally, $\log_2 K$ passes of merging are performed, where K is the number of face record groups. After the merging, edge structures are fetched into the main memory in blocks. Since the edge structures are sorted on the master cell index, the edge structures belonging to a cell will appear successively in the sorted file. By scanning edge structures, the neighboring cells of each cell are determined and entered into a buffer. The structure of the buffer is the same as the adjacency table, though its size is much smaller so that it can fit into the main memory with other data structures. If a cell has less than 4 neighbors, the indices of missing neighbors are denoted by -1 in the buffer. When the buffer is full, its contents are written into a disk file and cleared. The processing repeats until all edge structures are examined. Then the remaining contents of the buffer is written into the disk file and the adjacency table is built and stored in the file.

4. Test Results and Analysis

The out-of-core algorithm was tested on two computers. The first computer, Machine A, is equipped with a Sun Sparc 5 CPU, 32 Mega-byte memory, and a SCSI interfaced disk. Its embedded operating system is Sun OS 4.1.4. The second machine, Machine B, has a Sun Ultra-Sparc II CPU, 64 Mega-byte memory, and a wide-SCSI interfaced disk. Its operating system is Sun OS 5.6. The CPU speed and disk I/O bandwidth of Machine B are nearly twice as fast as that of Machine A. The memory capacity of Machine B is also two times larger than that of Machine A. Modern personal computers usually have more memory and faster processors than our machines. However, the purpose of our tests is to demonstrate the performance of our out-of-core algorithm. Therefore, these machines are feasible platforms for the tests. The test data are generated by splitting a cube into different numbers of tetrahedral cells.

4.1. Out-of-core Method vs. Incore Method

In order to compare the out-of-core method and the incore method, both programs are carried out on Machine A. Wall clock time is used to measure the programs' costs. The results are displayed in Figure 6. The incore program is superior to the out-of-core program when the data size is less than $200,000$ cells. As the number of cells exceeds $250,000$, the execution time of the incore program increases dramatically.

Data Size	*Disk I/O Overhead (%)*
490,000	43.1
1,000,000	46.6
2,000,000	48.1
3,000,000	48.7
4,000,000	50.1

Table 1: *Disk I/O Overhead of Out-of-core Method*

The decline of efficiency in the incore program is caused by the page swapping taken by the OS to allocate memory space for data structures during the computation. The performance of the out-of-core method is always stable and efficient. As the data size reaches $900,000$, the out-of-core program is about 7.5 times faster than the incore method.

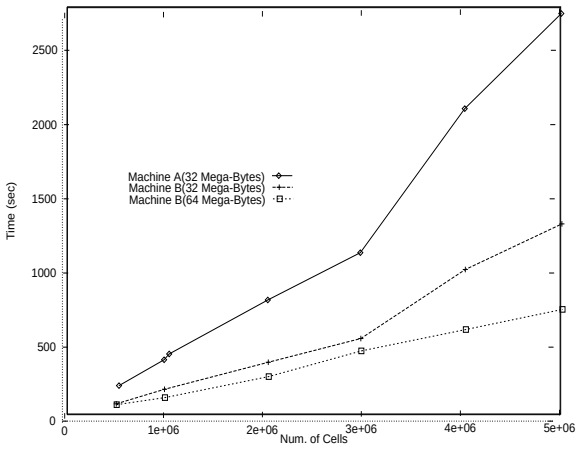


Figure 7: *Tests with Different CPU, Memory Space, and Disk I/O Bandwidth*

4.2. CPU Speed, Memory Capacity, and Disk I/O Bandwidth

The performance of the out-of-core method is dominated by three factors, the cpu speed, the memory capacity, and the disk I/O bandwidth. By using faster cpu, we can speed up calculations and reduce computational time. If the main memory capacity is enlarged, the maximum size of face groups increases. Then the number of groups is decreased, and less disk I/O is performed to read face groups. Even t he costs of merging edge structures are also declined. Disk I/O is slow, so the disk I/O bandwidth plays an important role in out-of-core methods. To understand the influence of the disk I/O bandwidth, the out-of-core program is profiled, and the costs of disk I/O are collected. The results are shown in Table 1. This table contains the percentages of disk I/O operation times in the total costs. Disk I/O cost constitutes more than 50% of the execution time, when the data size is 4 millions. Therefore, if the disk I/O bandwidth is increased, the performance can be improved significantly.

Three tests are executed to analyze the effects of these three factors. In the first test, the out-of-core program is run on Machine A. In the second test, the program is run on Machine B. However, the maximum memory capacity is limited to 32 Mega-bytes. The third test is performed by executing the program on Machine B, and raising the memory limit up to 64 Mega-bytes. The costs of the program are recorded and depicted in Figure 7. Since Machine A is twice as slow as Machine B in cpu speed and disk I/O bandwidth, Machine B outperforms Machine A even when both machines are equipped with the same amount of main memory. By increasing the main memory capacity to 64 mega-bytes, the performance of Machine B is improved. The improvement is less obvious when the data size is small. It is near 23% when the data size is about $500,000$. However, the effect of increasing memory space becomes significant when the data size exceeds 2 millions. The cost is reduced by about 40%.

5. Conclusions

An out-of-core algorithm is presented to compute adjacency graphs for large FEA data sets on computers with limited memory space. In our method, the essential data structures, face records, are divided into groups and stored in disk files. The groups are small enough such that each group can be processed inside the main memory without triggering any page swapping. The data division is accomplished by using a hash function. Then the groups are processed one after another, and the adjacency graph edges are calculated. Then the edges are merged into a single file in a merging stage, and the adjacency graph is calculated.

Test results show that the performance of the out-of-core method is stable and efficient. For example, the adjacency graph of a data set with 1 million cells can be computed in 6 minutes on a Sun Sparc 5 machine equipped with 32 mega-byte memory and a SCSI hard disk. The cost would exceed 30 minutes, if the incore algorithm was employed for the computation. Test results also demonstrate the effects of the cpu speed, the memory capacity, and the disk I/O bandwidth upon the out-of-core program's performance. The performance can be improved caused by using faster cpu, wider disk bandwidth, and larger memory space. For example, by running the same test on a Sun Ultra Sparc II machine with 64 mega-byte memory and wide SCSI hard disk, the cost of processing an 1-million cell data set is reduced to only 87 seconds which is at least 5 times faster.

References

1. CRAWFIS, R., AND MAX, N. Direct Volume Visualization of Three-Dimensional Vector Fields. In *Proceedings of the 1992 Workshop on Volume Visualization* (1992), ACM SIGRAPH, pp. 55–60.
2. GARRITY, M. P. Raytracing Irregular Volume Data. *Computer Graphics 24*, 5 (November 1990), 35–40.
3. GIERTSEN, C. Volume Visualization of Sparse Irregular Meshes. *IEEE Computer Graphics & Applications 12*, 2 (March 1992), 40–48.
4. GUMHOLD, S., GUTHE, S., AND STRASSER, W. Tetrahedral Mesh Compression with the Cut-Border Machine. In *Proceedings of the IEEE Visualization '99* (1999), pp. 51–58.
5. KOYAMADA, K., AND NISHIO, T. Volume Visualization of 3D Finite Element Method Results. *IBM J. Res. Develop. 35* (March 1991), 12–25.
6. LOHNER, R. A Vectorized Particle Tracer for Unstructured Grids. *Journal of Computational Physics 91* (1990), 22–31.
7. MAX, N., HANRAHAN, P., AND CRAWFIS, R. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *Computer Graphics 24*, 5 (November 1990), 27–33.
8. MCMAINS, S., HELLERSTEIN, J., AND SEQUIN, C. Out-of-Core Build of a Topological Data Structure from Polygon Soup. In *Proceedings of 6th ACM Synposium on Solid Modeling and Applications* (June 2001), pp. 171–182.
9. MURMAN, E., AND POWELL, K. Trajectory Integration in Vortical Flows. *AIAA 27*, 7 (1988), 982–984.
10. SEDGEWICK, R. *Algorithms in C, Parts 1-4, Third Edition*. Addison-Wesley, 1999.
11. SHIRLEY, P., AND TUCHMAN, A. A Polygon Approximation to Direct Scalar Volume Rendering. *Computer Graphics 24*, 5 (November 1990), 63–70.
12. SZYMCZAK, A., AND ROSSIGNAC, J. Grow & Fold: Compression of Tetrahedral Meshes. In *Technical Report SM99-021* (Georgia Institute of Technology, February 1999).
13. UENG, S.-K., AND SIKORSKI, K. Parallel Visualization of 3D FEA Data. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing* (San Francisco, CA, February 1995), pp. 808–813.
14. UENG, S.-K., AND SIKORSKI, K. A Note on a Linear Time Algorithm for Constructing Adjacency Graphs of 3D FEA Data. *The Visual Computer 12*, 9 (1996), 445–450.
15. UENG, S.-K., AND SIKORSKI, K. A Parallel Algorithm for Constructing Adjacency Graphs of FEA Data. In *Proceedings of PDPTA'2001 Conference* (Las Vegas, NV, June 2001).
16. UENG, S.-K., AND SIKORSKI, K. A Data Compression Method for Tetrahedral Meshes. In *Proceedings of Visualization and Data Analysis 2002* (San Jose, CA, January 2002), SPIE and IS&T.
17. UENG, S.-K., SIKORSKI, K., AND MA, K.-L. Efficient Construction of Streamlines, Streamribbons, and Streamtubes on Unstructured Grids. *IEEE Transactions on Visualization and Computer Graphics 2*, 2 (1996), 100–108.
18. UENG, S.-K., SIKORSKI, K., AND MA, K.-L. Out-Of-Core Streamline Visualization on Large Unstructured Meshes. *IEEE Transactions on Visualization and Computer Graphics 3*, 4 (1997), 100–108.
19. WALATKA, P., CLUCAS, J., KEVIN, M., AND PLESSEL, T. *FAST User Guide*. NASA Ames Research Center, 1993.
20. WILLIAMS, P. L. Interactive Splatting of Nonrectilinear Volumes. In *Proceedings of Visualization '92* (October 1992), A. E. Kaufman and G. M. Nielson, Eds., IEEE Computer Society Press, pp. 37–44.
21. WILLIAMS, P. L. Visibility Ordering Meshed Polyhedra. *ACM Trans. on Graphics 11*, 2 (1992), 103–126.