 CMES, vol.45, no.1, pp.31-55, 2009

Fast Searching Algorithm for Candidate Satellite-node Set in NLMG

Yufeng Nie[1], Ying Liu[2], Yuantong Gu[3] and Xiangkuo Fan[1]

Abstract: The Node-based Local Mesh Generation (NLMG) algorithm, which is free of mesh inconsistency, is one of core algorithms in the Node-based Local Finite Element Method (NLFEM) to achieve the seamless link between mesh generation and stiffness matrix calculation, and the seamless link helps to improve the parallel efficiency of FEM. Furthermore, the key to ensure the efficiency and reliability of NLMG is to determine the candidate satellite-node set of a central node quickly and accurately. This paper develops a Fast Local Search Method based on Uniform Bucket (FLSMUB) and a Fast Local Search Method based on Multilayer Bucket (FLSMMB), and applies them successfully to the decisive problems, i.e. presenting the candidate satellite-node set of any central node in NLMG algorithm. Using FLSMUB or FLSMMB, the NLMG algorithm becomes a practical tool to reduce the parallel computation cost of FEM. Parallel numerical experiments validate that either FLSMUB or FLSMMB is fast, reliable and efficient for their suitable problems and that they are especially effective for computing the large-scale parallel problems.

Keywords: local mesh generation, uniform bucket, multilayer bucket, linear quadtrees, neighbor searching, satellite node

1 Introduction

The increasing solving scale of the discrete system by finite element methods and the rapid development of the software and hardware for parallel computers have made it to be valuable to pay much attention to the parallel techniques of finite element methods [Shephard et al. (1997); Bielak, Ghattas and Kim (2005); Ha,

[1] School of Natural and Applied Sciences, Northwestern Polytechnical University, Xi'an 710072, China, yfnie@nwpu.edu.cn

[2] School of Mechatronics, Northwestern Polytechnical University, Xi'an 710072, China, qqingbbao333@163.com

[3] School of Engineering System, Queensland University of Technology, Brisbane, Australia, yuantong.gu@qut.edu.au

Seo and Sheen (2006); Rao et al. (2004); Hassan et al. (2004); Hajjar and Abel (1989); Santiago and Law (1996)]. But the parallel skills in many literatures and monograph just focus on each function block such as mesh generation, global stiffness matrix forming, linear equation system solving, etc. [Saad (1996); Cartwright, Oliveira and Stewart (2001); Du and Wang (2006); Bern, Eppstein and Teng (1999)]. This kind of research model ignores the influence on parallel efficiency coming from the seam link characteristic between the different function block. Comparing with the potential parallel algorithm for the meshless method [Atluri (2004); Sladek and Sladek (2006); Moulinec et al. (2008); Griebel and Schweitzer (2000); Danielson and Adley (2000); Danielson et al. (2000)], this is a great shortcoming. In order to improve the parallel efficiency of the finite element software system, the parallel mechanism of Node-based Local Finite Element Method (NLFEM) has been proposed recently by Nie, Chang and Fan (2007). This new parallel mechanism implements naturally the seamless link between pre-processing and main processing, and breaks through the serial characteristics between them. One of core techniques of NLFEM is the Node-based Local Mesh Generation (NLMG) Algorithm in which the mesh generation process is based on nodes independently and concurrently[Nie, Fan and Yuan (2006), Chang and Nie (2005), Yagawa (2004), Maus (1984), Nie and Chang (2006)]. The NLMG algorithm, which is free of mesh inconsistency, improves the efficiency of the parallel computation of finite element method (FEM) by achieving the seamless link between pre-processing and main processing.

It lays the foundation for the NLMG algorithm that every row in the global stiffness matrix is determined by the element patches (collar domain) which are generated by the corresponding central node. Selecting quickly the candidate satellite-node set of central node is the key of NLMG algorithm, which has definitive influence on computational efficiency and reliability of NLMG algorithm. The papers by Chang et al. (2005) and Nie et al. (2006) employed the optimal exploring circle method to find the candidate satellite-node set of central node, which guarantees no loss of the candidate satellite-nodes of central node, and resulting in the mesh consistency [Yagawa (2004)]. However, the work appointed the same initial exploring circle radius for every central node in practical computation to guarantee including its all satellite-nodes for each central node, then all of the nodes in the inclusive exploring circle were found by Local Search Method as the candidate node set of searching satellite-nodes. This idea of giving unified radius resulted in the huge computational cost. When nodes aren't distributed uniformly, the problem will become serious. In addition, it is also very hard to determine this unified radius in practical computation.

It is necessary to design a fast local search method for searching the candidate

satellite-node set for each circle node in the domain. If nodal distribution is relatively uniform, the Fast Local Search Method based on Uniform Bucket (FLSMUB) which introduces the data structure of uniform bucket for finding the candidate satellite-node set of central node is very fast and has very good performance [Yagawa (2004); Fan, Nie and Chang (2008)]; If nodal distribution is nonuniform, the data structure of uniform bucket isn't suitable. It is necessary to design a more adaptive and effective algorithm. Therefore a Fast Local Search Method based on Multilayer Bucket (FLSMMB) is proposed in this paper.

Local Search Method [Nie and Chang (2006)] made use of the data structure of uniform bucket, but the major shortcoming is that the initial searching circle radius must be assigned uniformly and it increases significantly computational cost. This paper presents the FLSMUB which gives a rational method of assigning initial exploring circle radius, and improves remarkably computational efficiency. Yagawa (2004) used multilayer bucket in NLMG algorithm , but they only combined multilayer bucket with gift-wrapping method [Su, Robert and Scot (1995)] in NLMG algorithm. We know that the gift-wrapping method needs a known Delaunay edge resulting in increase of computational complexity. In order to avoid this problem, the FLSMUB is modified. Using the data structure of linear quadtree in NLMG algorithm fits naturally the variety of node density and doesn't need the exploring circle in searching process, so the algorithm is simpler and more adaptive. When nodal distribution is uniform, its efficiency is worse than that of NLMG algorithm using uniform bucket's efficiency [Fan, Nie and Chang (2008)]. However, it isn't distinct.

The structure of this paper is as follows. Section 2 introduces NLMG. The data structure of uniform bucket and the linear quadtree structure of multilayer bucket are given in section 3; Section 4 gives the details of the searching algorithm of candidate satellite-node set in NLMG algorithm including the FLSMUB and the FLSMMB; The capability and adaptability of the new developed algorithm are validated in section 6 through variable numerical examples under parallel environment of distributed memory MIMD system; At final, the conclusion is given in section 7.

2 NLMG algorithm

The traditional mesh generation algorithms such as the famous Bowyer-Watson incremental algorithm by Bowyer (1981) and Watson (1981), Guiba-Stolf divide-and-conquer method by Guiba and Stolf (1985), and Fortune sweepline algorithm by Fortune (1987) have characteristics as follows. In the implementation procedure of these algorithms, the topological relations of nodes aren't decided until global mesh generation is finished, and it is to say that the mesh generations have global

feature. In these algorithms , the element set relating to some nodes isn't decided until global mesh generation finished, so the stiffness matrix isn't computed and assembled until pre-processing completes. This global feature makes them impossible to complete seamless connection between pre-processing and main processing. However, the NLFEM algorithm requires a mesh generation algorithm with local feature. That is the topological relations of nodes are decided in the local region, and mesh can be generated on nodes independently and concurrently. Therefore the Node-based Local Mesh Generation algorithm (NLMG) has been developed.

NLMG algorithm is the core technique of NLFEM, and the new finite element parallel mechanism developed based on it can achieve naturally seamless link between pre-processing and main processing. An effective NLMG algorithm must guarantee that the local mesh elements generated in local region of each circle node are just a part of optimal mesh in global region. NLMG method can avoid mesh inconsistency, guarantee mesh quality, and, therefore, form the theoretical foundation for the reliability of NLFEM method. Nie et al. (2006) and Chang et al. (2005) proposed an initial NLGM algorithm which satisfies the above-mentioned requirement. However, it used the data structure of the uniform bucket in searching the candidate satellite-node set locally. This method is too coarse, requires a unified exploring circle radius, and poor efficiency.

In fact, the final aim of NLMG is to find the satellite-node set of each central node in order to complete its local partition. Therefore, the key of NLMG algorithm is how to find the candidate satellite-node set quickly and exactly, then to get the satellite-node set for computation. If the uniform bucket is used, because this kind of buckets have no response to the variety of node density, the exploring circle is needed[Chang and Nie (2005); Nie and Chang (2006)]. The data structure of uniform bucket is used to confirm and refine the exploring circle radius for each central node and, then, to get the candidate satellite-node set. If the multilayer bucket is used, because it can represent the variety of the node density, there are relatively more buckets in the region with high nodal density, and less buckets in the region with low nodal density. Therefore, the algorithm is simple and efficient without the notion of exploring circle as we will see in the following sections.

3 The data structures of uniform bucket and multilayer bucket

3.1 Uniform bucket

The data structure of uniform bucket is relatively simple, and its algorithm is fast and compact. For the node set in a computational region, the data structure of uniform bucket is built in this paper as follows. Let M be the cardinality of the node set. The information of nodes is organized and includes node serial number,

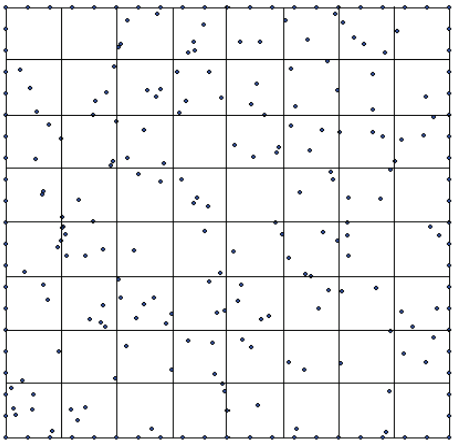

Figure 1: Linear quadtrees code norm

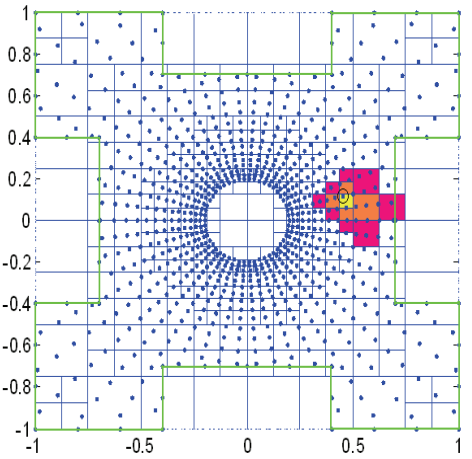

Figure 2: Example of linear quadtree coding

coordinates, etc. The node serial numbers are: 0, 1, ..., $M-1$.

Algorithm 1: The algorithm of building the data structure of uniform bucket in computational region

Step 1 Choose a box which is able to cover the computational region and is as possible as small. Let el be the length of its edge, and coordinates of its lower sinister corner are (Cx, Cy).

Step 2 Divide the box into uniform square grids, number and record the coordinates of each grid which are called bucket also in this paper. Let's assume that there are n square grids, so the average node number of each bucket is M/n, and the length of the edge of all of these buckets is $el/\sqrt{n}$. Then we number buckets from the bucket in the lower sinister corner (number 0) to the upper dexter corner (number $n-1$) in the order of from left to right and from bottom to top, and in the same time record the location of each bucket, that is the coordinates of its the lower sinister corner. For every bucket,let integer k ($0 \le k \le n-1$) be its number. Its location (Ckx, Cky) is:

$$\left(Cx + \frac{el}{\sqrt{n}} \times \left(k - k \times \left\lfloor \frac{k}{\sqrt{n}} \right\rfloor\right), Cy + \frac{el}{\sqrt{n}} \times \left\lfloor \frac{k}{\sqrt{n}} \right\rfloor\right)$$

Step 3 Build the node information for each node, that is all nodes in computational region are distributed into buckets, and the data structure is organized well. For

each node i, let its coordinates be (Cix, Ciy), and the number of its bucket is:

$$\left\lfloor \frac{Ciy - Cy}{el/\sqrt{n}} \right\rfloor \times \sqrt{n} + \left\lfloor \frac{Cix - Cx}{el/\sqrt{n}} \right\rfloor$$

Fig. 1 plots a example for this algorithm.

3.2 Multilayer bucket

Constructing the data structure of multilayer bucket is complex compared with that of uniform bucket. Quadtree, specially linear quadtree, is an important method to represent binary image, and is widely used in research areas of computer graphics, image processing, computer vision, robot and so on. The linear quadtree structure of multilayer bucket is constructed in this paper by recursion, and succeeds in deciding the candidate satellite-node set in NLMG algorithm with it.

3.2.1 Code of linear quadtree

Because linear quadtree is partitioned by recursion, each box which does not satisfy the requirement is divided into four sub-boxes. The codes of the four sub-buckets are 0, 1, 2, 3 respectively, as shown in Fig. 3.

The code of linear quadtree has the following characteristics:

(1) Directionality. According to the size of the code in Fig. 3, the code increases from west to east and from north to south.

(2) Hierarchy. After the mth partition, the quaternary code of the sub-bucket of the mth level is:

$$q_1 q_2 \cdots q_m \quad q_i \in \{0,1,2,3\}, i = 1,2,\cdots,m.$$

(3) Compressibility. If the code is stored in quaternary form, it will need m digits for a mth level sub-bucket. To save memory space, the quaternary code is transformed into decimal code.

(4) Convertibility. When the quaternary code of a sub-bucket is known, its decimal code and quaternary code can be converted reciprocally, i.e. the transformation is both surjection and injection. That is one quaternary code is corresponding with one decimal code and one lever number of primary quaternary code.

There are two kinds of neighbor for sub-bucket: one kind is sharing adjacent edge, called edge neighbor; the other kind is sharing adjacent angle, called angle neighbor. As Fig. 4 shown, sub-bucket 21 is a edge neighbor of sub-bucket 30 with similar size, and sub-bucket 30 is a edge neighbor of sub-bucket 33 with different size; sub-bucket 30 is a angle neighbor of sub-bucket 33 with similar size, and

sub-bucket 30 is a angle neighbor of sub-bucket 231 with different size. The neighbor searching technique of linear quadtrees is a important role in finding candidate satellite-node set of central node with multilayer bucket.

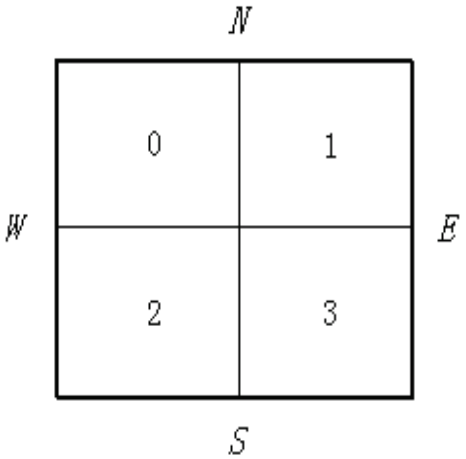

Figure 3: Linear quadtrees code norm

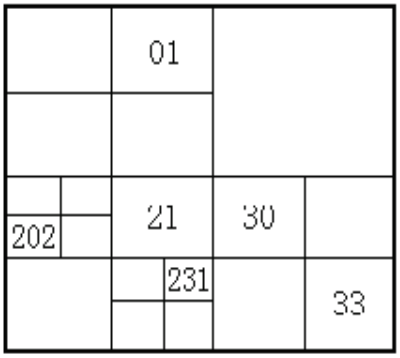

Figure 4: Example of linear quadtree coding

3.2.2 Building the linear quadtree structure of multilayer bucket

The linear quadtree structure is used in finding the candidate satellite-node set in NLMG algorithm. Let's assume that the number of nodes is M, and the node serial numbers are: 0, 1, ..., M−1. The information of nodes is organized which includes node serial number, coordinates, etc. The ending condition of recursion of bucket partition is that the number of nodes in the bucket is no more than a positive integer p.

For the convenience of searching, let the global variable *bknum* record the serial number of the bucket which satisfies the requirement of partition according to the order(starts with 0). The global array $Tcodim[M][1]$ stores each node's bucket number, and its row number is equal to node number. The global array $informbk[M][5+p]$ stores the bucket information, and its row number is equal to bucket number. Their implementation steps is following:

Algorithm 2: The algorithm of building the linear quadtree structure of multilayer bucket in the computational region

Step 1 Choose the smallest square region which can covers the computational region, and call it as initial bucket. Let *ed* be the length of its edge, and (Dx, Dy) be

the coordinates of the lower sinister corner. Then the bucket's central coordinates are $(Dx+ed/2, Dy+ed/2)$. The list $nodelist$ stores all node serial numbers, and the lists $getcode0$, $getcode1$, $getcode2$ and $getcode3$ record the buckets' quaternary code after partition.

Step 2 Divide the bucket region into four sub-buckets, i.e. bucket 0, bucket 1, bucket 2, bucket 3. The lists $nodelist0$, $nodelist1$, $nodelist2$ and $nodelist3$ store the serial numbers of the nodes of bucket 0, 1, 2 and 3 respectively. The $nodenum0$, $nodenum1$, $nodenum2$ and $nodenum3$ variables denote the numbers of the nodes in bucket 0, 1, 2 and 3 respectively. It is valuable to notice that one node belongs to one and only one bucket, especially for the nodes which locate just on the segments of the dividing line of the sub-buckets, additional regulations are needed to satisfy the above-mentation rule.

Step 3 For each new bucket(bucket 0, 1, 2 and 3), take bucket 0 for example, its location of the lower sinister corner is $(Dx, Dy+ed/2)$, and its edge length is $ed/2$. Its current number 0 is putted in the $getcode0$ list. If the number of nodes $nodenum0$ included in bucket 0 is bigger than p, the bucket must be partitioned further, turn back to Step 2; otherwise the bucket meets the partition requirement, the partition stops, and go to the next step.

Step 4 For the buckets meeting the partition requirement,their quaternary codes are transformed into decimal numbers. Row $bknum$ of $informbk[M][5+p]$ records all the information corresponding to a bucket: the row number $bknum$ denotes the serial number of the bucket, the bucket decimal code, the bit size of quaternary code, the bucket's location(coordinates of the lower sinister corner), the node list including in bucket. For each node belonging to the bucket, let its code be $i(0 \leq i < M)$, then the i row of array $Tcodim[M][1]$ stores the bucket number.

Step 5 $bknum \Leftarrow bknum+1$.

According to above steps, the data structure which is suitable for the quadtree structure of the candidate satellite-node set in NLMG algorithm can be built quickly and effectively. Finally the value of $bknum$ is the biggest bucket number.

The number of nodes in the multiply connected region in Fig. 2 is 880, where there are 107 boundary nodes, and the nodal distribution is nonuniform. Let $p=5$ be the ending condition. The initial bucket location is $(-1,-1)$, and its size is 2. According to the recursive partition method as above, the value of $bknum$ is 363, this denotes the biggest bucket number is 363, and the number of buckets is 364. The global array $informbk[M][5+p]$ records the buckets' information as Table 1 shows. The local array $Tcodim[M][1]$ records the bucket number of each node as Table 2 shows.

Table 1: Bucket information recorded in global array, i.e. informbk[M][5+p]

Number	Decimal value	Level	Location	size	nodes included
0	0	4	(-1.000000,0.875000)	0.125000	14 877
1	1	4	(-0.875000, 0.875000)	0.125000	15 867
2	2	4	(-1.000000,0.750000)	0.125000	13 868
3	3	4	(-0.875000, 0.750000)	0.125000	851
4	1	3	(-0.750000, 0.750000)	0.250000	16 827 828 849 850
⋮	⋮	⋮	⋮	⋮	⋮
272	47	3	(-0.250000,-1.000000)	0.250000	void
273	192	4	(0.000000,-0.125000)	0.125000	void
274	772	5	(0.125000,-0.062500)	0.062500	void
275	3092	6	(0.187500,-0.031250)	0.031250	56 105 155
⋮	⋮	⋮	⋮	⋮	⋮
363	255	4	(0.875000,-1.000000)	0.125000	42 879

Table 2: Information that each node belongs to a bucket is recorded in global array Tcodim[M][1]

Code number	0	1	2	3	...	381	382	383	394	...	876	877	878	879
Bucket number	260	258	253	187	...	67	196	198	204	...	103	0	260	363

4 Fast local searching method of the candidate satellite-node set in NLMG algorithm

The main purpose of NLMG algorithm without mesh inconsistency is to find the satellite-node set of each central node accurately and quickly. In this sections the fast searching methods based on the two data structures as previous developed, which are named as the Fast Local Search Method based on Uniform Bucket (FLSMUB) and the Fast Local Search Method based on Multilayer Bucket (FLSMMB), are given. They are the major work of this paper.

4.1 Fast Local Search Method based on Uniform Bucket (FLSMUB)

When the node distribution is relatively uniform, using the uniform bucket structure is the best choice. FLSMUB is used to refine the initial circle radius and to search for the candidate satellite-node set by uniform bucket. It is more efficient

and practical compared with the unified circle radius method [3,6] to determine the candidate satellite-node set.

After constructing the data structure of uniform bucket, each present computational node A is called central node. The flow chart of the FLSMUB is plotted in Fig. 5, and its detailed steps are given as follows.

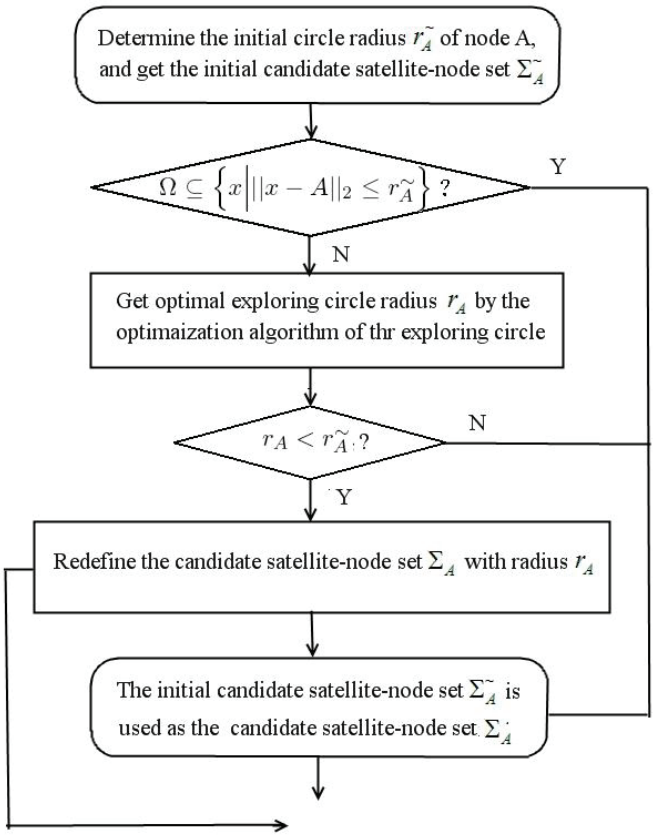

Figure 5: The flow chart of the FLSMUB

Algorithm 3: The algorithm of FLSMUB

Step 1 Determine the initial circle radius $r_A^{\sim}$ of node A, and get the initial candidate satellite-node set $\sum_A^{\sim}$.

Step 2 Check whether the initial circle radius contains the whole region Ω or not. If $\Omega \subseteq \left\{x \middle| ||x-A||_2 \le r_A^{\sim}\right\}$, the candidate satellite-node set $\sum_A$ is the initial candidate satellite-node set $\sum_A^{\sim}$, and return.

Step 3 Get optimal exploring circle radius r_A by the optimization algorithm of the initial exploring circle radius.

Step 4 If $r_A < r_A^{\sim}$, the initial candidate satellite-node set $\sum_A^{\sim}$ is used as the candidate satellite-node set $\sum_A$ of the central node A, and return.

Step 5 Redefine the candidate satellite-node set $\sum_A$ using the exploring circle radius r_A and the data structure of uniform bucket, and return.

The theory by Nie and Chang (2006) ensures that the candidate satellite-node set which is got by the above algorithm includes all the satellite nodes of the central node. Only when the refined r_A is greater than the initial circle radius, we redefine its candidate satellite-node set by uniform bucket; otherwise, we still use the unified circle radius. In addition, the refined r_A is usually smaller than the initial circle radius. So this fast algorithm is more efficient, and the results of numerical examples in next section confirm this conclusion clearly.

As shown in Fig. 5, there are two steps using the uniform bucket technique. One is the uppermost block diagram corresponding to Step 1 to define the initial exploring circle radius and the candidate satellite-node set of the central node A, the other is the last second block diagram corresponding to Step 5 using the refined radius r_A to redefine the candidate satellite-node set of the central node A, which will be used as the final candidate satellite-node set $\sum_A$ of the central node A. The algorithm details of Step 1 and Step 5 are given as follows:

Algorithm 4: (Step 1 of algorithm 3) The algorithm of the initial exploring circle radius and the candidate satellite-node set of the central node A

Step 1 List *MList* records the order number of the outer layer bucket, and its initial value is the order number of the bucket in which the central node A lies(i.e. the central bucket); variable *MR* is a flag which states whether the candidate satellite-node set $\sum_A^{\sim}$ meets the partition requirement or not [3]; the initial values $MR = false$, $\sum_A^{\sim} =$, the bucket layer $Lp = 1$.

Step 2 Put all the nodes in the bucket indicated by list *MList* into the candidate satellite-node set $\sum_A^{\sim}$.

Step 3 Check whether $\sum_A^{\sim}$ meets the partition requirement or not. Set $MR = true$ and turn to Step 5 if meets, or $MR = false$.

Step 4 Compute the outer adjacent buckets of he buckets indicated by $MList$. And use the order numbers of these adjacent buckets to cover the original value of list $Mlist$. Set the bucket's level $Lp = Lp + 1$. Turn back to Step 2.

Step 5 Remove the central node A from $\sum_A^{\sim}$. Compute the initial searching circle radius $r_A^{\sim} = (Lp+1) \times d$, and d is the edge length of each uniform bucket. Return.

Algorithm 5: (Step 5 of algorithm 3) The algorithm of using the refined radius r_A

to redefine the candidate satellite-node set of the central node A

Step 1 If $\Omega \subseteq \left\{ x \middle| ||x-A||_2 \leq r_A^{\sim} \right\}$, All the nodes in Ω is used as the candidate satellite-node set $\sum_A$ ($\sum_A = \sum$), and return.

Step 2 According to the exploring circle radius r, define the exploring layer of bucket Lp, the bucket layer counts from the central bucket(there is the central node in it). Let d be the edge length of bucket, then $Lp = \left\lfloor \frac{r}{d} \right\rfloor + 2$

Step 3 Search for all the buckets whose layer number does not exceed Lp counting from the central bucket. Record these buckets as set B_A^r. Its region covers fully the searching circle of node A.

Step 4 Compute $\sum_A = \left\{ S \middle| ||S-A||_2 \middle|, \forall S \in U, U \in B_A^r \right\} \backslash \{A\}$, and be used as the candidate satellite-node set of node A. That is, for every node included in each bucket U of B_A^r, it is a candidate node if the distance between the node and node A is not bigger than the exploring radius r.

4.2 The Fast Local Search Method based on Multilayer Bucket (FLSMMB)

4.2.1 The procedure of FLSMMB

The data structure of multilayer bucket is constructed in section 3.2. Let's induce a parameter *bklayer* which is used to express the bucket layer number starting with the central bucket in the computation. Because the size of multilayer bucket varies with node density, there are relatively many buckets in the region of relatively high node density, and there are relatively few buckets in the region of relatively low node density. This fact makes it possible to control the searching region for satellite-nodes of a central node by using the value of variable *bklayer* defined property instead of the exploring circle used in the uniform bucket case. Thus FLSMMB algorithm can be simple and efficient. Its core part is searching the neighbor of quadtree. The detailed steps of FLSMMB is as follows.

Now, let's mark the present computing node as node B. Let $i(0 \leq i < M)$ be its code, and the bucket is situated in be called as the central bucket.

Algorithm 6: The algorithm of FLSMMB

Step 1 Record the list of the outer layer buckets in list *exbkList*, the initial value is the bucket number of the central bucket which stored in $Tcodim[i][0]$. Set the initial candidate satellite-node set of control node B $\sum_B^{\sim} = \phi$, and the initial bucket layer $Ly = 1$.

Step 2 Put all of the nodes of each bucket in *exbkList* into the candidate satellite-node set $\sum_B^{\sim}$.

Step 3 If $Ly == bklayer$, go to Step 5.

Step 4 Find the outer bucket of *ekbkList*, which involve the neighbor searching of a linear quadtree and will be introduced emphatically in section 4.2.2. Then using the data of these new outer bucket to replace the original data in *ekbkList*. Let the bucket layer $Ly = Ly + 1$. Turn back to step 2 .

Step 5 Remove the central node from $\sum_B^{\sim}$. Then the obtained $\sum_B^{\sim}$ is candidate satellite-node set of node B $\sum_B$, and return.

Now,let's set *exbkList.begin* and *exbkList.end* be the first and the last element of *exbkList* respectively. The step 2 in algorithm 6 can be done as follows.

Algorithm 7: The steps of putting the nodes of each bucket in *exbkList* into the candidate satellite-node set $\sum_B^{\sim}$

Step 1 $listIter \Leftarrow exbkList.begin$.

Step 2 Because the row number of array $imformbk[M][5+p]$ denotes the bucket number, all the nodes in the $listIter$ bucket can be found to be the last p elements of the array $informbk[listIter][5+p]$. And they put them into the candidate satellite-node set $\sum_B^{\sim}$.

Step 3 $listIter \Leftarrow listIter + 1$.

Step 4 If $listIter == exbkList.end$, $\sum_B^{\sim}$ is found. Or go back to Step 2.

Suppose that *bkvisited*, which is used to record the bucket visited, is a find of variable of map$< int, bool >$ type in Visual C++. Because the initial value of the outermost layer bucket list *exbkList* is the bucket $Tcodim[i][0]$ of the central node B, the initial value of variable *bkvisited* is $bkvisited[Tcodim\ [i][0]] = true$. The procedure of Step 4 is following.

Algorithm 8: The steps of computing the outer bucket of *ekbkList* and replacing the original buckets in *ekbkList*

Step 1 $listIter \Leftarrow exbkList.begin$.

Step 2 Compute the adjacent buckets of bucket $listIter$(only the edge adjacent buckets), store them in $neigbklist$.

Step 3 $iter \Leftarrow neigbklist.begin$.

Step 4 Check whether the bucket A $iter$ is visited or not. If $bkvisited[iter] == false$, the bucket $iter$ isn't visited, then put it into the list $tempbklist$ and be recorded by not being visited, i.e. $bkvisited[iter] = true$.

Step 5 If $iter! = neigbklist.end$, $iter \Leftarrow iter + 1$, and go back to Step 4 . Or the bucket list $tempbklist$ includes all of the buckets of the outer layer.

Step 6 $exbkList \Leftarrow tempbklist$, and renew $Ly = Ly + 1$.

4.2.2 Searching edge neighbor in the data structure of linear quadtree

For the linear quadtree structure of the multilayer bucket, computing the buckets of the outer layer, which involves searching neighbor of the linear quadtree, is needed among the steps of FLSMMB algorithm. The algorithm of searching neighbor is useful in binary graphic analysis, 3-D entity analysis, boundary definition, and connectivity judgement and so on [Liu and Yan (1997); Liu, Qiu and Yang (2004); Xiao, Gong and Xie (1998)]. There are two kinds of neighbor for buckets. They are edge neighbor and angle neighbor. To define the candidate satellite-node set in NLMG algorithm, only the algorithm of searching edge neighbor is needed because the angle neighbor can be regarded as the edge neighbor of next layer. Moreover, the methods of searching edge neighbor designed here include searching the same size edge neighbor method and searching the different size edge neighbor method. Searching the same size edge neighbor method is simple, and only scanning the last several bits is needed. Searching the different size edge neighbor method needs the combination between the same size edge neighbor computed and the hierarchy of the quaternary code.

Let some bucket's quaternary code be

$$C = q_1 q_2 \cdots q_n, \quad q_i \in \{0,1,2,3\},\ i = 1,2,\cdots,n.$$

Let $\triangle_C$ be the corresponding decimal code of the bucket. Then $\triangle_C = \Sigma_{l=1}^{n} q_l \times 4^{n-l}$.

1) The method for searching the same size edge neighbor

For searching the same size edge neighbor, firstly, we compute the possible edge neighbor, then check whether the possible edge neighbor is in the bucket structure constructed or not.

If $q_n = 0$, according to the standard code system, the decimal code of the possible east neighbor is $\triangle_C + 1$, and the decimal code of the possible south neighbor is $\triangle_C + 2$. Computing the possible west neighbor and the possible north neighbor is following: for the possible west neighbor, scan orderly from right to left until some bit of code $q_i (i \in \{1,2,\cdots,n-1\})$ is equal to 1 or 3 for the first time. The bucket of code $q_1 q_2 \cdots q_i$ is on the left side of its father bucket of code $q_1 q_2 \cdots q_{i-1}$. Firstly, subtract 1 from q_i, add 1 to each $q_{i+1}, q_{i+2}, \cdots, q_n$, and the values of $q_1, q_2, \cdots, q_{i-1}$ are kept invariable. So the new code $q_1' q_2' \cdots q_n'$ is the code of the computed neighbor. Let Δ_W be its decimal code. Then the difference between the codes of the two buckets Δ_{C-W} is:

$$\Delta_{C-W} = 4^{n-i} - \sum_{l=0}^{n-i-1} 4^l.$$

And the decimal code of the possible west neighbor is:

$$\Delta_W = \Delta_C - \Delta_{C-W} = \Delta_C - \left(4^{n-i} - \sum_{l=0}^{n-i-1} 4^l\right).$$

If the $q_i (i \in \{1,2,\cdots,n-1\})$ to be 1 or 3 can't be found, the possible west neighbor does not exist. Similarly, for the possible north neighbor, scan from the last bit of the code q_n, orderly from right to left, until some bit of code $q_j (j \in \{1,2,\cdots,n-1\})$ is equal to 2 or 3 for the first time. The bucket of code $q_1 q_2 \cdots q_j$ is on the south side of its father bucket of code $q_1 q_2 \cdots q_{j-1}$. Firstly, subtract 2 from q_j, add 2 to each $q_{j+1}, q_{j+2}, \cdots, q_n$, and the values of $q_1, q_2, \cdots, q_{j-1}$ are kept invariable. So the new code is the code of the computed north neighbor. Then the difference between the new codes of this bucket and the bucket $C = q_1 q_2 \cdots q_n$ is:

$$\Delta_{C-N} = 2 \times 4^{n-j} - \sum_{l=0}^{n-j-1} 2 \times 4^l.$$

And the decimal code of the possible north neighbor is:

$$\Delta_N = \Delta_C - \Delta_{C-N} = \Delta_C - \left(2 \times 4^{n-j} - \sum_{l=0}^{n-j-1} 2 \times 4^l\right).$$

If $q_i (i \in \{1,2,\cdots,n-1\})$ to be 2 or 3 can't be found, the possible north neighbor does not exist. For the other three cases, i.e. $q_n = 1, 2$ or 3, if the corresponding possible neighbor of bucket C exist and $q_i (i \in \{1,2,\cdots,n-1\})$ satisfying a special location condition similarly as in the case $q_n = 0$, the computing methods of the neighbors are shown as follows.

If $q_n = 1$, the decimal code of the possible west neighbor is $\triangle_C - 1$, the possible south neighbor is $\triangle_C + 2$, the possible east neighbor is $\Delta_C + 4^{n-i} - \sum_{l=0}^{n-i-1} 4^l$, and the possible north neighbor is $\Delta_C - (2 \times 4^{n-i} - \sum_{l=0}^{n-i-1} 2 \times 4^l)$.

If $q_n = 2$, the decimal code of the possible north neighbor is $\triangle_C - 2$, the possible east neighbor is $\triangle_C + 1$, the possible south neighbor is $\Delta_C - (2 \times 4^{n-i} - \sum_{l=0}^{n-i-1} 2 \times 4^l)$, and the possible west neighbor is $\Delta_C - (4^{n-i} - \sum_{l=0}^{n-i-1} 4^l)$.

If $q_n = 3$, the decimal code of the possible west neighbor is $\triangle_C - 1$, the possible north neighbor is $\triangle_C - 2$, the possible south neighbor is $\Delta_C + (2 \times 4^{n-i} - \sum_{l=0}^{n-i-1} 2 \times 4^l)$, and the possible east neighbor is $\Delta_C + (4^{n-i} - \sum_{l=0}^{n-i-1} 4^l)$.

As above, the method of searching the same size edge neighbor is simpler, and only the last several bits of the quaternary code of the buckets are needed to scan. And the neighbor computing needs only the one-step arithmetic operation in half of the situations. This benefit is owing to using the convertibility and hierarchy of the linear quaternary code.

2) The method of searching different size edge neighbor

The method of searching the different size edge neighbor, which is based on the method of searching the same size edge neighbor, is simpler. Suppose the east, south, west and north same size edge neighbors of the bucket C have been computed, and their decimal codes are noted as $\triangle_E$, $\triangle_S$, $\triangle_W$ and $\triangle_N$ respectively. Let bucket $D = e_1 e_2 \cdots e_m (e_i \in \{0,1,2,3\}, i = 1,2,\cdots,m.\ m \neq n)$. Now we check whether the bucket D is the different size edge adjacent bucket of the bucket C or not. Its detailed steps is as follows:

If $m > n$, i.e. the layer number of bucket D is bigger than that of bucket C, bucket D is smaller than bucket C. Intercepting the front n bits of that of bucket D's quaternary code, we can get one of its father buckets, denoted as $D_n = e_1 e_2 \cdots e_n (e_i \in \{0,1,2,3\}, i = 1,2,\cdots,n)$. Its decimal code is $\Delta_{D_n} = \sum_{l=1}^{n} e_l \times 4^{n-l}$. The necessary condition of adjacency bucket D and bucket C is that the father bucket D_n of bucket D is one of the east, west, south or north same size edge adjacent buckets of bucket C.

If Δ_{D_n} is equal to Δ_E, bucket D_n is the east same size edge neighbor of bucket C. For the last $m-n$ bits of the quaternary code of bucket D, if having e_i =0 or 2, for any $e_i (i = n+1, n+2, \cdots, m)$, bucket D is the east different size edge adjacent bucket of bucket C.

If Δ_{D_n} is equal to Δ_S, bucket D_n is the south same size edge neighbor of bucket C. For the last $m-n$ bits of the quaternary code of bucket D, if having e_i =0 or 1, for any $e_i (i = n+1, n+2, \cdots, m)$, bucket D is the south different size edge adjacent bucket of bucket C.

If Δ_{D_n} is equal to Δ_W, bucket D_n is the west same size edge neighbor of bucket C. For the last $m-n$ bits of the quaternary code of bucket D, if having e_i =1 or 3, for any $e_i (i = n+1, n+2, \cdots, m)$, bucket D is the west different size edge adjacent bucket of bucket C.

If Δ_{D_n} is equal to Δ_N, bucket D_n is the north same size edge neighbor of bucket C. For the last $m-n$ bits of the quaternary code of bucket D, if having $e_i = 2$ or 3, for any $e_i (i = n+1, n+2, \cdots, m)$, bucket D is the north different size edge adjacent bucket of bucket C.

If $m < n$, i.e. the layer number of bucket D is smaller than that of bucket C, bucket D is bigger than bucket C. So if bucket D is a edge adjacent bucket of bucket C, bucket D must be the father bucket of the bucket which is the same size edge adjacent bucket of bucket C. That is to say there must exist some same size edge neighbor of bucket C which is included in bucket D. So for the east, west, south and north same size edge adjacent buckets of bucket C, check the front m bits of their quaternary codes. If any one of them, the front m bits of its quaternary codes

is the same as the quaternary code of bucket D, bucket D is this neighbor's father, and bucket D is the different size edge adjacent bucket of bucket C.

3) The algorithm of searching edge neighbor

Now, we are ready to give the algorithm frame of searching edge neighbor by using the skill developed previously. Suppose that the serial number of bucket C in the data structure is *bkindex*, its decimal code is $informbk[bkindex][0]$, and its layer(the number of bits of its quaternary code) is $informbk[bkindex][1]$. Variable $Iter$ denotes the traversal bucket at present, its initial value $Iter = 0$. List *neigbklist* stores its adjacent buckets. *Bknum* is the biggest serial number of the buckets.

Algorithm 9: The algorithm of computing all edge neighbors of bucket C

Step 1 Compute the east, west, south and north same size edge adjacent buckets of bucket C by the method of searching same size edge neighbor as above developed.

Step 2 If $Iter! = bkindex$ and $Informbk[Iter][1] == informbk[bkindex][1]$, the present bucket of the serial number $Iter$ indicated has the same size as bucket C. Check whether one of the east, west, south or north same size edge adjacent buckets of bucket C is bucket $Iter$ or not. If there exists, the bucket $Iter$ is the same size edge adjacent bucket of bucket C, and put its serial number into list $neigbklist$.

Step 3 If $Iter! = bkindex$ and $Informbk[Iter][1]! = informbk[bkindex][1]$, the present bucket of the serial number $Iter$ has the different size with bucket C. Check whether bucket Iter is the adjacent bucket of bucket C or not, by the method of searching different size edge neighbor as above developed. If it is, put its serial number into list $neigbklist$.

Step 4 If $Iter < Bknum$, $Iter \Leftarrow Iter + 1$, turn back to step 2 .Or the list *neigbklist* at present includes all adjacent buckets of bucket C.

5 Numerical examples

In this section, the following possion equation is used to demonstrate the newly developed algorithm above.

$$\begin{cases} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x), & (x,y) \in \Omega \\ u|_{\partial\Omega} = 0, & \Omega \text{ is a 2-d arbitrary domain} \end{cases}$$

The NLMG algorithm is applied because it is easy to perform the parallel calculation. The parallel models, which can be used, include the static allocation node model, the manager/worker-based dynamic allocation node model, and adaption model posed by Nie, Chang and Fan (2007). In different practical computational situations, a appropriate parallel model is chosen.

The parallel computing environment is the Hp rx2600 mustering system in the High-Performance Computing and Development Center of Northwestern Polytechnical University, which is composed of 40 computing nodes and 2 managing nodes (Here, each node is a HP rx2600 framework server), and which makes the high speed computing network via Myrinet network. The parallel programming mode is the MPI and we use C++ for the programme realization.

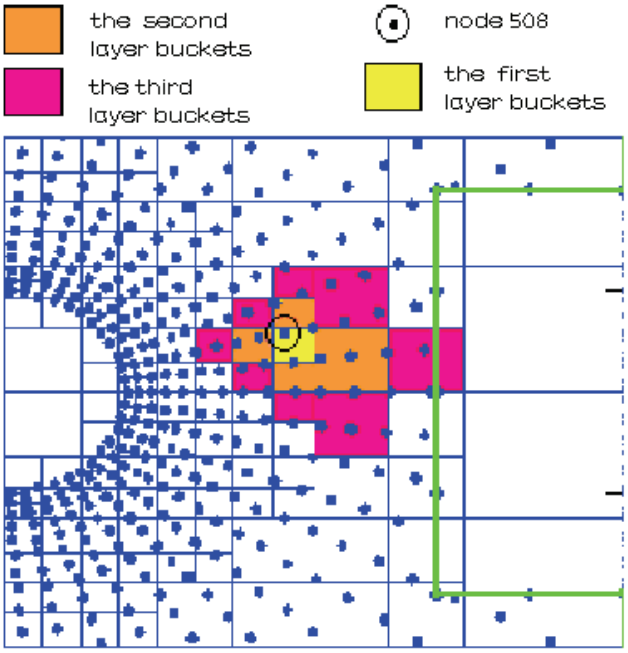

Figure 6: The local area and candidate satellite-node set of the node 508

Example 1: Determining the candidate satellite-node set in FLSMMB

Fig. 2 shows this example's computational domain, node distribution and the data structure of multilayer bucket. The number of nodes is 880, in which there are 107 boundary nodes. The division standard takes $p = 5$. Taking bucket $Tcodim[i][0]$ of the central node B as example(the serial number of node B is 508, its bucket number is $i = 169$), the candidate satellite-node set of node B is determined by the Fast Local Searching Algorithm based on Multilayer Bucket. Let the searching layer of the bucket *bklayer* be 3. The results are given as follows.

1) The serial number of the first layer bucket of node 508 is 169, and it can be obtained by the quadtree structure of multilayer buckets that bucket 169 includes

node 508.

2) The edge adjacent buckets of bucket 169 determined by algorithm 9: 163, 168, 171, 179.

3) The serial number of the second layer buckets: 163, 168, 171, 179.

The adjacent buckets of bucket 163: 161, 162, 169, 177; Store node 509 into list *Candlist*.

The adjacent buckets of bucket 168: 162, 165, 169, 170; Store nodes 408 and 458 into list *Candlist*.

The adjacent buckets of bucket 171: 169, 170, 179, 300; Store node 507 into list *Candlist*.

The adjacent buckets of bucket 179: 169, 171, 177, 180, 334; Store nodes 557, 607, 657 into list *Candlist*.

4) The serial number of the third layer buckets: 161, 162, 177, 165, 170, 300, 180, 344. That is all of the adjacent buckets of the second layer buckets except the second layer buckets and the first layer buckets. The following nodes are stored into list *Candlist*:

Nodes 559, 560 in bucket 161.

Node 459 in bucket 162.

Nodes 558, 608, 609, 658, 659 in bucket 177.

Node 358 in bucket 165.

Nodes 407, 457 in bucket 170.

Nodes 506, 555 in bucket 300.

Node 707 in bucket 180.

Nodes 594, 595, 644, 645, 695 in bucket 344 .

5) The shaded parts of Fig. 2, which is composed of three layers of buckets, show the local region of central node 508 and the 26 nodes in candidate satellite-node set *Candlist*. For convenience, the shaded parts of Fig. 2 are separated, and shown in Fig. 6.

Example 2: Mesh parallel generation

For the rectangular region as shown in Fig. 1, because its nodal distribution is relatively uniform, we build the data structure of uniform buckets, the mesh parallel experiment is carried out in the large scale cluster system, according to FLSMUB proposed in this paper. Its mesh generation result is shown in Fig. 7.

For the region as shown in Fig. 2, because its nodal distribution is nonuniform, we build the data structure of multilayer buckets, the mesh parallel experiment is

carried out in the large scale cluster system, according to FLSMMB proposed in this paper. Its mesh generation result is shown in Fig. 8.

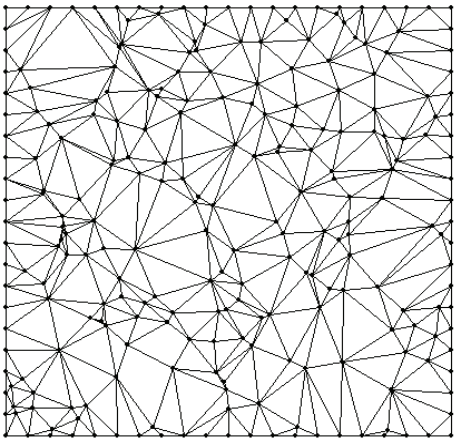

Figure 7: Apply FLSMUB to mesh parallel generation

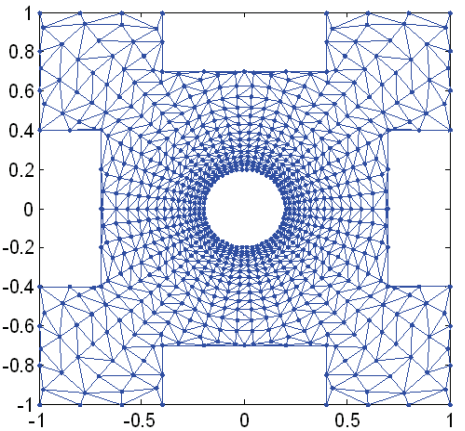

Figure 8: Apply FLSMMB to mesh parallel generation

Example 3: The computational cost of FLSMUB and FLSMMB

The time costs of uniform bucket and multilayer bucket in the computational region are different and can be test with the serial time. For uniform bucket, the number of buckets can reflect accurately the topological relations of the data structure. Because the sizes of multilayer buckets are different, the number of buckets can't reflect the topological relations of the data structure. So during the construction of multilayer bucket by recursion, the terminal condition of the recursion is that the buckets contain no more than p nodes.

Here two types of nodal distribution are employed. 10000 uniformly distributed nodes and 10000 unevenly distributed nodes(including 5000 uniformly distributed nodes and 5000 nodes obeying normal distribution). The computational cost of FLSMUB or FLSMMB is obtained and compared in Fig. 9 and Fig. 10.

From these figures, we can conclude:

1) Fig. 9 shows that when FLSMMB is used for two types of nodes, their computational cost has little difference, when the situation of unevenly distributing nodes is slightly better. As shown in Fig. 10, when FLSMUB is used for two types of nodes, it performs much better for the case of uniformly distributed nodes than for the case of unevenly distributed nodes. But FLSMMB is suitable for both uniformly and unevenly distributed nodes cases.

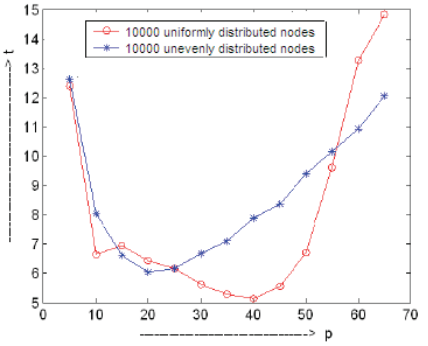

Figure 9: Single processor serial time corresponding to applying FLSMMB

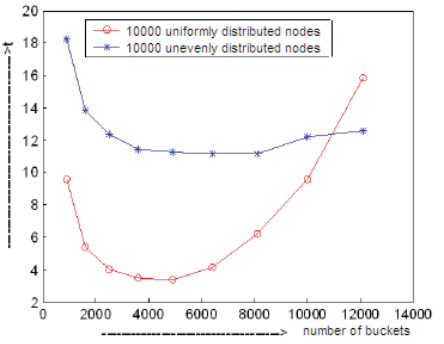

Figure 10: Single processor serial time corresponding to bucket number applying FLSMUB

2) The variety of computational cost of FLSMMB decreases firstly and then increases with the value p, and the variety of serial time of FLSMUB follows the same trend as that of FLSMMB. For two types of node distribution, whether by FLSMUB or by FLSMMB, the serial time cost of uniformly distributing nodes is sensitive to the change of the parameter p: decreases quickly, and increases quickly. Certainly, there are optimal parameters to lead to the lowest computational cost.

Example 4: The performance in the large-scale parallel computation

Different with example 3, in this study, there are 100000 nodes in a rectangular domain. The serial time costs of FLSMUB and FLSMMB are listed in Table 3. The data in Table 3 prove again that FLSMUB is only suitable to the uniformly distributed nodes, but FLSMMB is more effective for both cases.

Table 4 and Table 5 present the parallel time cost of FLSMUB and FLSMMB in the cluster system, including computational time, speed-up ratio and parallel efficiency. The parallel computation model in this paper is the manager/worker-based dynamic allocation node model in combination with manager participating in computation[Nie, Chang and Fan (2007)]. Considering the problem scale, 2, 4, 6 and 8 processors are used respectively to test the parallel performance. It has proven that FLSMUB and FLSMMB are powerful for the large-scale parallel computation. This is due to the two better data structure, local searching method with high speed, and NLMG algorithm suitable for parallel computation naturally.

It should be noted that serial algorithm is used to construct the data structure, that is, each processor builds the same data structures. This affects the efficiency of parallel computation. However, because the time cost of building the data structure

Table 3: Serial computing time applying FLSMUB and FLSMMB

	FLSMUB	FLSMMB
Distribute uniformly 100000 nodes	44.4971	64.8614
Distribute non-uniformly 100000 nodes	144.8948	78.7826

Table 4: The parallel computing results for the case that nodes are distributed uniformly

The Number of processers	computational time(s)		Speed-up ratio		parallel efficiency	
	FLSMUB	FLSMMB	FLSMUB	FLSMMB	FLSMUB	FLSMMB
2	21.504	32.86	1.97190	1.97387	98.60	98.69
4	11.4463	17.5586	3.70459	3.694	92.61	92.35
6	8.25344	12.4227	5.13771	5.2212	85.63	87.02
8	6.623231	9.6843	6.40228	6.69758	80.03	83.72

is relatively smaller, this effect can be neglected in most of cases. Only when the computation scale is small, or the processors are too many, the effect is distinct.

6 Conclusions

NLMG algorithm developed successfully is the precondition to ensure the parallel efficiency of NLFEM. Here both FLSMUB and FLSMMB are developed and applied successfully to the NLMG algorithm, and thus the algorithm becomes practice.

FLSMUB performs well in the case of uniform distribution nodes. FLSMMB is suit to much more situations compared with FLSMUB. To improve calculational efficiency, according to different problems, we should select FLSMUB, FLSMMB or their combination.

Acknowledgement: This research was supported by Program for New Century Excellent Talents in University of China and National Natural Science Foundation of China.

Table 5: The parallel computing results for the case that nodes are not distributed uniformly

The Number	computational time(s)		Speed-up ratio		parallel efficiency	
of processers	FLSMUB	FLSMMB	FLSMUB	FLSMMB	FLSMUB	FLSMMB
2	73.2099	40.0074	1.9804	1.9692	99.02	98.46
4	38.3233	21.2444	3.7832	3.7084	94.58	92.71
6	26.7687	14.7649	5.4162	5.3358	90.27	88.93
8	21.0416	11.5449	6.8904	6.8240	86.13	85.30

References

Atluri, S.N. (2004): The Meshless Method (MLPG) for Domain & Bie Discretizations, *Tech Science Press*

Bern, M.; Eppstein, D.; Teng, S.H. (1999): Parallel construction of quadtrees and quality triangulations, *International Journal of Computational Geometry and Applications* , vol.9, no.6, pp. 517-532

Bielak, J.; Ghattas, O.; Kim, E.J. (2005): Parallel Octree-Based Finite Element Method for Large-Scale Earthquake Ground Motion Simulation , *CMES: Computer Modeling in Engineering & Sciences*, vol. 10, no. 2, pp. 99-112

Bowyer, A. (1981): Computing Dirichlet tessellations, *The Computer Journal*, vol. 24, no. 2, pp. 162-166

Cartwright, C.; Oliveira, S.; Stewart, D.E. (2001): A parallel quadtree algorithm for efficient assembly of stiffness matrices in meshfree Galerkin methods, *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, pp. 1194-1198

Chang, S.; Nie, Y.F. (2005): Node-Based Local Mesh Generation Algorithm Within an Arbitrary 2D Domain, *Acta Aeronautica ET Astronautica Sinica*, vol. 26, no. 5, pp. 556-561

Danielson, K.T.; Adley, M.D. (2000): Parallel construction of quadtrees and quality triangulations, *Comput. Mech.* , vol.25, pp. 267-273

Danielson, K.T.; Hao, S.; Liu, W.K.; Aziz, R.; Li, S. (2000): Parallel computation of meshless methods for explicit dynamic analysis, *Inter. J. Numer. Methods* , vol. 47, no.6, pp. 1323-1341

Du, Q.; Wang, D.S. (2006): Recent progress in robust and quality Delaunay mesh generation, *Journal of Computational and Applied Mathematics* , vol. 195, no.1-2,

pp. 8-23

Fan, X.K.; Nie, Y.F.; Chang, S. (2008): Search algorithm for initial exploring circle radius in NLMG, *Chinese Journal of Computational Mechanics*, vol. 25, no. 2, pp. 188-194

Fortune, S. (1987): A sweepline algorithm for Voronoi diagrams, *Algorithmica*, vol. 2, pp. 153-174

Guibas, L.; Stolf, J. (1985): Primitives for manipulation of general subdivisions and the computation of voronoi diagram, *Transations on Graphics*, ACM, vol. 4, no. 2, pp. 74-123

Griebel, M.; Schweitzer, M. A. (2000): A Particle-Partition of Unity Method for the solution of Elliptic, Parabolic and Hyperbolic PDE, *SIAM J. Sci. Comp.*, vol.22, no.3, pp. 853-890

Ha, T.; Seo, S.; Sheen, D. (2006): Parallel iterative procedures for a computational electromagnetic modeling based on a nonconforming mixed finite element method, *CMES: Computer Modeling in Engineering & Sciences*, vol. 14, no. 1, pp. 57-76

Hajjar, J.F.; Abel, J.F. (1989): On the Accuracy of Some Domain-by-Domain Algorithms for Parallel Processing of Dynamics, *Inter. J. Numer. Methods,*, vol. 28, no.6, pp. 1855-1874

Hassan, O.; Morgan, K.; Jones, J.; Larwood, B.; Weatherill, N. P. (2004): Parallel 3D Time Domain Electromagnetic Scattering Simulations on Unstructured Meshes, *CMES: Computer Modeling in Engineering & Sciences*, vol. 5, no. 5, pp. 383-394

Liu, G.; Qiu, J.; Yang, R. (2004): 8-neighborhood Finding Algorithm for Binary Image Represented by Linear Quadtrees, *Computer and Information Technology*, vol. 12, no. 4, pp. 31-34, 59

Liu, G.; Yan, M. (1997): Neighbor Finding Techniques For Binary Image, *Computer Applications and Software*, vol. 14, no. 5, pp. 32-36

Maus, A. (1984): Delaunay triangulation and the convex hull of n points in expected linear time, *BIT*, vol. 24, no. 2, pp. 151-163

Moulinec, C.; Issa, R.; Marongiu, J.C.; Violeau, D. (2008): Parallel 3D Time Domain Electromagnetic Scattering Simulations on Unstructured Meshes, *CMES: Computer Modeling in Engineering & Sciences*, vol.25, no.3, pp.133-147

Nie, Y.F.; Chang, S. (2006): Node-based local mesh generation algorithm, *Chinese Journal of Computational Mechanics*, vol. 23, no. 2, pp. 252-256

Nie, Y.F.; Chang, S.; Fan, X.K. (2007): The Parallel Mechanism of Node-based Seamless Finite Element method, *CMES: Computer Modeling in Engineering & Sciences*, vol. 19, no. 2, pp. 135-144

Nie, Y.F.; Fan, X.K.; Yuan, Z.B. (2006): A New and Efficient Node-Based Local Mesh Generation Parallel Algorithm, *Journal of Northwestern Polytechnical University*, vol. 24, no. 6, pp. 735-739

Rao, A. R. M.; Rao, TVSRA; Dattaguru, B. (2004): Parallel 3D Time Domain Electromagnetic Scattering Simulations on Unstructured Meshes, *CMES: Computer Modeling in Engineering & Sciences*, vol. 5, no. 3, pp. 213-234

Saad, Y. (1996): Iterative Methods for Sparse Linear Systems, *PWS Publishing*

Santiago, E.D.; Law, K.H. (1996): A Distributed Implementation of an Adaptive Finite Element Method for Fluid Problems, *Computers and Structures* , vol. 74, pp. 97-119

Shephard, M. S.; Flaherty, J. E.; Bottasso, C. L.; de Cougny, H. L.; Ozturan, C.; Simone, M. L. (1997): Parallel automatic adaptive analysis, *Parallel Computing* , vol. 23, no. 9, pp. 1327-1347

Shirazaki, M.; Yagawa, G. (1999): Large-scale parallel flow analysis based on free mesh method: A virtually meshless method, *Comput. Methods Appl. Mech. Eng.* , Vol. 174, pp. 419-431

Sladek, J.; Sladek, V. (2006): Advances in Meshless Methods, *Tech Science Press*

Su, P.; Robert, L.; Scot, D. (1995): A Comparison of Sequential Delaunay Triangulation Algorithms, *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, ACM, pp. 61-70

Watson, D.F. (1981): Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes, *The Computer Journal*, vol. 24, no. 2, pp. 167-172

Xiao, L.; Gong, J.; Xie, C. (1998): A New Algorithm for Searching Neighbors in The Linear Quadtree and Octree, *Aata Geodaetica et Cartographica Sinica*, vol. 27, no. 3, pp. 195-203

Yagawa, G. (2004): Node-by-node parallel finite elements: A virtually meshless method, *International Journal for Numerical Methods in Engineering*, vol. 60, no. 1, pp. 69-102