COMMUNICATIONS IN COMPUTATIONAL PHYSICS
Vol. **4**, No. 5, pp. 1086-1105

Commun. Comput. Phys.
November 2008

Parallel Mesh Refinement of Higher Order Finite Elements for Electronic Structure Calculations†

Dier Zhang[1,*], Aihui Zhou[2] and Xin-Gao Gong[1]

[1] *Department of Physics, Fudan University, Shanghai 200433, China.*
[2] *LSEC, Institute of Computational Mathematics and Scientific/Engineering Computing, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing 100190, China.*

Received 29 February 2008; Accepted (in revised version) 29 May 2008

Available online 16 July 2008

Abstract. The finite element method is a promising method for electronic structure calculations. In this paper, a new parallel mesh refinement method for electronic structure calculations is presented. Some properties of the method are investigated to make it more efficient and more convenient for implementation. Several practical issues such as distributed memory parallel computation, less tetrahedra prototypes, and the assignment of the mesh elements carried out independently in each sub-domain will be discussed. The numerical experiments on the periodic system, cluster and nano-tube are presented to demonstrate the effectiveness of the proposed method.

AMS subject classifications: 81Q05, 65L50

Key words: Tetrahedral mesh, adaptive mesh refinement, parallel algorithms, electronic structure calculations.

1 Introduction

The finite-element (FE) method has attracted much attention for electronic structure calculations. Using the compactly supported piecewise polynomials functions, the method allows for variable resolution in real space and produces well structured sparse matrices. Therefore it is very suitable for parallel implementation. On the other hand, the FE method has been so far limited by the huge number of basis functions, which uses much more degrees of freedoms than the traditional plane wave (PW) method [1] based on the Fourier basis. However, the significant strength of the finite element method lies in its

†Dedicated to Professor Xiantu He on the occasion of his 70th birthday.
*Corresponding author. *Email addresses:* dearzhang@fudan.edu.cn (D. Zhang), azhou@lsec.cc.ac.cn (A. Zhou), xggong@fudan.edu.cn (X.-G. Gong)

ability to place adaptive/local refinements in regions where the desired functions vary rapidly while treating the other zones with a coarser description [2–9].

In early works, White *et al.* [10] found that with uniform meshes as many as 10^5 basis functions per atom were required to achieve sufficient accuracy. To decrease the number of basis functions Tsuchida and Tsukada [2] had applied nonuniform hexahedron meshes on H_2 molecules. Beck [5, 11] studied the multigrid method based on such nonuniform hexahedron meshes. In these applications the grid can be made to vary logarithmically near the nuclei, but the smoothness of the wave function is not guaranteed for the non-conforming mesh. Afterward, Tsuchida and Tsukada [3, 4] proposed another approach with the adaptive curvilinear coordinates (ACC's), which was recently applied for the calculations of *ab initio* molecular dynamics. All the above adaptive coordinates are based on hexahedron meshes. In [12], we proposed an adaptive refinement method to generate conforming tetrahedra mesh. This method can create a very flexible mesh which could be locally refined in any interested regions. Moreover, the mesh conforming can be preserved during the refinements. With a posteriori error estimation of the eigenvalue problem [13], the refinement is carried out, automatically paying special attention to the spatial regions where the computed functions vary rapidly, especially near the nuclei. For this implementation the particular technique adopted for mesh refinement is very important. In order to perform large scale electronic structure calculations, we present a parallel algorithm for the adaptive construction of tetrahedral meshes in the finite element computations, which is technically assigned for the physics problem.

The serial refinement algorithms of simplicial meshes have been examined by many authors. However few works concerning parallel mesh refinements have been reported. Jones and Plassmann [14] proposed a parallel algorithm for adaptive local refinement of two-dimensional triangular meshes. Castaños and Savage [15] described the parallel algorithm for local adaptive refinement of tetrahedral meshes used in the PARED package. Zhang [16] proposed another parallel algorithm using the newest vertex approach.

For efficient electronic structure calculations, we will introduce a new direct parallel algorithm for the adaptive refinements. In our method the assignment of the FE nodes is applicable for interpolation during the calculations. Our parallel algorithm follows the simplicial bisection algorithm based on newest vertex approach. This approach was first developed by Bäsch [17] for the local tetrahedral mesh refinement. Later on, Maubach [18] and Horst [19] generalized the method to the case of arbitrary number of dimensions. Kossaczky [20], Liu and Joe [21], and Arnold *et al.* [22] also studied local refinement by bisection with different interpretations. The basic step in the refinement is tetrahedral bisection, as shown in Fig. 1. We adopt the notation and algorithm of the bisection refinement introduced in [22]. In this algorithm, with the data structure named *marked tetrahedron* the tetrahedra are classified into 5 types and the selection of refinement edge depends only on the type and the ordering of vertices for the tetrahedra. The parallelization of the refinement algorithm is based on a natural idea, i.e., to exchange the hanging vertices and edges on the interface of each sub-mesh, as used in the literatures [14–16].

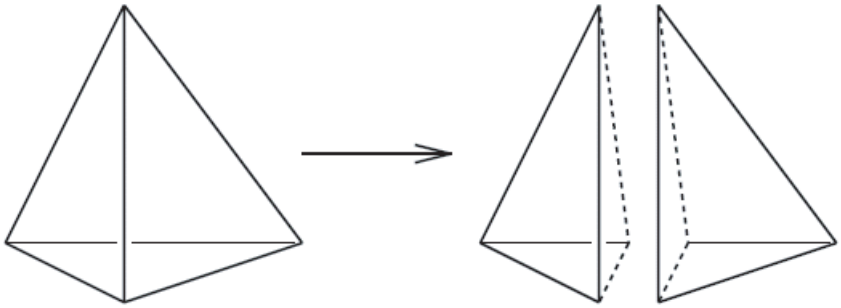

Figure 1: Bisection of a single tetrahedron.

Our method takes care of aspects such as distributed memory parallel computation based on domain subdivision; less tetrahedra prototypes by taking the character of the physics problem into account; the assignment of the mesh elements carried out independently in each sub-domain without any global notation; the data structure fitting for basis functions of higher order by assigning each mesh element (tetrahedron, vertex, edge, face) a number during the refinements.

The paper is organized as follows. The next section provides the domain subdivision for electronic structure calculations. In Section 3 the data structure and algorithm of the parallel mesh refinement are presented. In Section 4 we discuss some available properties of the refined meshes. In Section 5 we discuss the parallel data storage and matrix-vector multiplications. In Section 6 we consider some numerical examples. The final section is devoted to the conclusions.

2 Domain subdivision and mesh partitioning

Although the domains for many FE implementation are irregular, in electronic structure calculations we can always adopt the cuboid-like domains. The main reason is that usually there are only three typical kinds of problems in electronic structure calculations: 1. Super-cell with periodic boundary condition for the crystal systems; 2. Finite systems like clusters or molecules, which need a large enough domain; 3. Transport problems with special boundary condition in a given direction.

A cuboid-like domain is suitable for all these kinds of physical problems. Our refinement method is therefore based on such cuboid domains which can be divided into a regular tetrahedra mesh initially. In order to allocate a mesh T on a distributed memory parallel computer, one needs to partition T into P sub-meshes T_i, $(i=0,\cdots,P-1)$, where P is the number of MPI processors. This is easily done by the regular partition, see Fig. 2 as an example, where we simply partition the cube into 8 equal sub-cubes. Each processor treats only one sub-mesh which is similar to the original one. In our implementation, the boundary of each sub-mesh is generated at the very beginning and kept

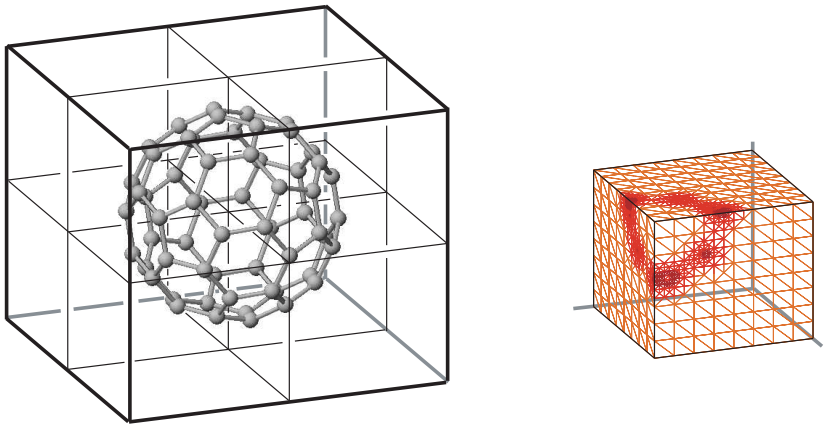

Figure 2: 8 sub-domains for C_{60}, and the mesh of one sub-domain.

unchanged throughout the refinements. In the electronic structure calculation the load balance among different processors can be achieved by putting similar number of atoms in each sub-domain.

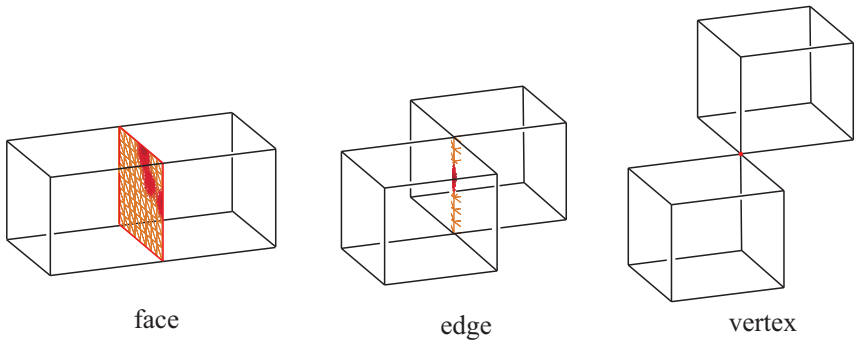

Figure 3: Three kinds of interface-mesh. The interface-meshes on the faces are triangular meshes, which are shared by two processors. The interface-meshes of the edges are 1-dimensional and shared by four processors. The interface-meshes on the vertices are shared by eight processes, which are trivial and never be changed during the refinements.

We name the connection part $T_{ij}=T_i\bigcap T_j$ $(i\neq j)$ between two neighboring sub-meshes T_i and T_j *interface-mesh*. The interface-meshes are also stored just like un-partitioned meshes. There are three kinds of interface-meshes (see Fig. 3): sharing a face, sharing an edge , and sharing a vertex between the neighboring cuboid domains. The face interface-meshes are usually shared by 2 domains, while the edge is shared by 4 domains and the vertex is shared by 8 domains. Each interface-mesh is stored by all its neighbor processors. The interface-meshes of the faces are triangular meshes, the interface-meshes of the edges are 1-dimensional, and the interface-meshes of the vertices are just one point which will never be changed during the refinements.

3 Data structure and parallel refinement algorithm

3.1 Data structure

Our package is written in C language. In this subsection, we describe the data structure. In our implementation, the elements of FE mesh including tetrahedra, faces, vertices, edges, and interpolation nodes are assigned. The tetrahedra are represented below by the structure array `tetra[]` with the members

```
struct tetra{
    int vertex[4];          storing the index of vertices in the tetrahedron,
    int edge[6];            storing the index of edges,
    int face[4];            storing the index of faces,
    int type;               storing the type of the tetrahedron.
}
```

Here, the algorithm described in [22] is used to classify the tetrahedra into 5 types. With a conventional order of the array `tetra[i].vertex[4]` and the type of a tetrahedron, its refinement edge can be uniquely determined [20, 22]. The bisection of a tetrahedron is simply to compute the array and type of its children according to a given set of rules.

Though the array `tetra.vertex[4]` and the type of a tetrahedron are enough for uniquely determining its refinement edge, we also store the indexes of edges and faces for assigning the interpolation nodes in the tetrahedron. With all these indexes we can assign the nodes which occur on the edges or faces during the refinement, it is not necessary to reassign them after the mesh refinement. This yields convenience and natural assignment [23] such that the indexes of new nodes naturally follow the old ones. For example, in the case of quadratic basis functions with the nodes on each vertex and edge, we store the mapping array `vertex[i].node` and array `edge[i].node` to find the nodes. The coordinates of the interpolation nodes are stored in the structure member `node[i].coordination[3]`.

We use β_V, β_E, β_F, β_T and β_N to denote the number of vertices, edges, faces, tetrahedra, and nodes respectively. The number of nodes depends on the order of the basis functions, for 2-order interpolation $\beta_N=\beta_V+\beta_E$. The well-known Euler-Poincare formula states the relationship among them:

$$\beta_V+\beta_F-\beta_T-\beta_E=s, \tag{3.1}$$

where s is the Euler characteristic, $s=1$ for finite meshes and $s=0$ for infinite meshes and the case of periodic boundary condition. As a practical issue, it is important to know the pair-wise relationships to achieve memory efficiency. The relationship (3.1) is apparently not much helpful and it is usually ignored in the literature since the problem is hard to treat for irregular meshes and partition ways. However, the present approach (shown in Section 2) results in the following lemma.

Lemma 3.1. *Let β_V, β_E, β_F, β_T and β_N be the number of vertices, edges, faces, tetrahedra, and nodes respectively. Then*

$$\beta_E \leq 7\beta_V, \quad \beta_F \leq 12\beta_V, \quad \beta_T \leq 6\beta_V. \tag{3.2}$$

The proof of Lemma 3.1 will be given in Section 4. Note that Lemma 3.1 indicates a simple way to initialize these numbers, that is, the upper limits of the number of edges, faces and tetrahedra are determined by the number of vertices. For example, if we choose $\beta_V = 10{,}000$, then $\beta_E = 70{,}000$, $\beta_F = 120{,}000$ and $\beta_T = 60{,}000$ are reasonable choices.

For each processor, the neighboring interface-meshes are stored like a 2-D mesh (on the faces) or 1-D mesh (on the edges). Each of them has a set of mapping arrays between the sub-meshes and interface-meshes. The interface-meshes and their mapping arrays are created when the mesh is partitioned, and updated during mesh refinement. Two neighboring sub-meshes T_i and T_j share the vertices, edges and faces; all of them are the elements on their interface-meshes $T_{ij} = T_i \bigcap T_j$. All the information exchanging between the neighboring sub-meshes will be realized through the interface-mesh. The mesh information of T_{ij} is stored in the processor i and j as $T_{ij}^{(i)}$ and $T_{ij}^{(j)}$ respectively, where the superscript denotes the processor id:

$$\mathcal{T}_i(z) \xleftrightarrow{mapping} \mathcal{T}_{ij}^{(i)}(z) \xLeftrightarrow{data\ exchange} \mathcal{T}_{ij}^{(j)}(z) \xleftrightarrow{mapping} \mathcal{T}_j(z). \tag{3.3}$$

During the refinement, the new vertices, edges and faces will be added up to the interface-meshes. The interface-meshes are updated synchronously with the sub-meshes, but they are not be regenerated after the refinement. We suppose that the indexes of vertices, edges and faces are the same on the shared interface-meshes $T_{ij}^{(i)}$ and $T_{ij}^{(j)}$, for that we need not to create a mapping between them. It is easy to generate an initial regular mesh satisfying this condition, and we demand that the indexes keep the same order throughout the refinement. We will discuss how to achieve this in Section 3.2.

Compared with the method in [16], our data structure helps us to assign the interpolation nodes more naturally for high order basis functions. In our method the data exchange during the mesh refinement is easy to be implemented in the case that the boundary of the sub-domain is fixed.

3.2 The parallel refinement algorithm

The refinement procedure consists of two phases. In the first phase, each sub-mesh is refined independently, while the shared faces, edges and vertices of the interface-meshes are treated as if they were on the boundary. In the second phase, the new elements on interface-meshes are exchanged between the neighboring sub-meshes.

In step 1 of Algorithm 3.1, `BisectTets` is a serial procedure for bisecting every tetrahedron in set S,

$$\texttt{BisectTets}(T,S) = (T \backslash S) \bigcup \Big(\bigcup_{\tau \in S} \texttt{BisectTet}(\tau) \Big),$$

Algorithm 3.1: Parallel Refinement Algorithm

$(T_i', \{T'^{(i)}_{ij}\}) = \texttt{ParallelRefine}(T_i, \{T^{(i)}_{ij}\}, S_i)$, where i is the processor index, and $T_i \bigcap T_j \neq \emptyset$

Input: The sub-meshes on each sub-domain $\{T_i'\}$, the set of interface-meshes $\{T^{(i)}_{ij}\}$, the set of the tetrahedra to be refined S_i , $S_i \subset T_i$.

1. Bisect the tetrahedron $\tau \in S_i$ in each processor i , $\bar{T}_i = \texttt{BisectTets}(T_i, S_i)$,
2. Conform the sub-meshes $(T_i', \{T'^{(i)}_{ij}\}) = \texttt{ParallelConform}(\bar{T}_i, \{T^{(i)}_{ij}\})$.

Output: The refined sum-mesh T_i' given by $T' = \bigcup_{i=1}^{n} T_i'$.

Algorithm 3.2: Parallel Conforming

$(T_i', \{T'^{(i)}_{ij}\}) = \texttt{ParallelConform}(T_i, \{T^{(i)}_{ij}\})$

Input: The sum-mesh T_i has not been conformed and its neighboring interface-meshes $\{T^{(i)}_{ij}\}$

1. Exchange the data between every two neighboring processors, renew the interface-meshes and make the sub-meshes coupled with each neighbor meshes
   $(\tilde{T}_i, \{T'^{(i)}_{ij}\}) = \texttt{Exchange}(T_i, \{T^{(i)}_{ij}\})$.
2. Let $S_i = \{\tau \in \tilde{T}_i | \text{with hanging vertices on } \tau\}$.
3. if $(\bigcup_j S_j \neq \emptyset)$ then
   $\bar{T}_i = \texttt{BisectTets}(\tilde{T}_i, S_i)$,
   $(T_i', \{T'^{(i)}_{ij}\}) = \texttt{ParallelConform}(\tilde{T}_i, \{T'^{(i)}_{ij}\})$
4. else
   $T_i' = \tilde{T}_i$

Output: The conformed sum-meshes of T_i' and interface-meshes $\{T'^{(i)}_{ij}\}$.

where $\texttt{BisectTet}(\tau)$ is the bisection of a single tetrahedron. This is the basic refinement step in the algorithm of [22]. After that, in step 2 the mesh is conformed by clearing up the hanging vertices and edges.

Algorithm 3.2 is a recursive procedure which stops when no more hanging vertices and edges. The difference with the serial algorithm is that we need to exchange the hanging vertices and edges on the interface of each neighboring sub-mesh. Because the result of the serial algorithm is independent of the sequence of the hanging vertices and edges, we can ensure that the parallel algorithm produces the same result as the serial one and terminates in same finite number of steps (see the proof in [16]). In this sense, our algorithm is essentially the same as that in [16], except the exchange mode of the hanging vertices and edges on the interface and the data structures. In our method the exchange of the hanging vertices and edges is based on the shared interface-mesh between two neighboring sub-meshes. It helps us to assign the indexes of the sub-mesh elements, naturally distributed over every processor to avoid a global assignment.

In Algorithm 3.3 each processor sends and receives the hanging vertices and edges on

Algorithm 3.3: Exchange the Hanging Vertices and Edges

$(\tilde{T}_i,\{T'^{(i)}_{ij}\})=\texttt{Exchange}(T_i,\{T^{(i)}_{ij}\})$

Input: The sum-mesh T_i, and its neighboring interface-meshes $\{T^{(i)}_{ij}\}$

1. Let $H^{(i)}_{ij}$ be the set of the hanging vertices and edges on the interface of T_i to be added to the interface-mesh $T^{(i)}_{ij}$.
2. Send the data of $H^{(i)}_{ij}$ to each neighboring processor j. The positions of the hanging vertices and edges are assigned with the index of $T^{(i)}_{ij}$.
3. Receive the preceding data from all the neighboring processors.
4. Update the sub-mesh T_i, $\tilde{T}_i=T_i\bigcup\left(\bigcup_j\tilde{H}_{ij}(j)\right)$.
5. Update the interface-meshes $\{T^{(i)}_{ij}\}$ to $\{T'^{(i)}_{ij}\}$.

Output: The sub-mesh $\tilde{T}_i$ matched with all of its neighbor sub-meshes and its neighbor interface-meshes $\{T'^{(i)}_{ij}\}$ matched with $\tilde{T}_i$.

the interface, and then adds them to sub-mesh and interface-meshes. The data exchange is based on the shared interface-meshes. Let the index of the edge on interface-meshes $T^{(i)}_{ij}$ be $\mathcal{T}^{(i)}_{ij}(\text{edge}_z)$. Because the index is equal to that of the shared interface-meshes

$$\mathcal{T}^{(i)}_{ij}(\text{edge}_z)\equiv\mathcal{T}^{(j)}_{ij}(\text{edge}_z), \tag{3.4}$$

as we send the index to the neighboring processor j, the position of the hanging vertex is determined on the neighboring sub-mesh T_j, and the processor j will add up the new vertex using the information. In the same way, the hanging edge is denoted by the index of face and vertex where it is located to help the neighboring processors to find the position.

In step 5 we add the new vertices and edges to the interface-meshes. The problem is how to keep the indexes of the interface-meshes $T'^{(i)}_{ij}$ and $T'^{(j)}_{ij}$ in the same order, which are stored and updated in different processors. The natural idea is to add the new vertices and edges in same order. In our method the new vertices and edges are denoted by their original processor. The precedence for adding them is determined by their original processor id. As shown in Fig. 4, the interface-mesh T_{14} is a plane which has 6 neighbor sub-meshes $0,1,\cdots,5$. It was stored by two processors 1 and 4 as $T^{(1)}_{14}$ and $T^{(4)}_{14}$, and the new vertices and edges will come from all the 6 neighbor sub-meshes. The processor 1 or 4 will treat the new vertices and edges in the same order according to the number of their original processor. More precisely, the new vertices and edges coming from the sub-mesh 0 will be first added to the interface-mesh $T^{(1)}_{14}$ and $T^{(4)}_{14}$, then from the sub-mesh 1, and so on. Note that the overlapping new vertices and edges should not be reassigned redundantly. In this way the indexes of $T^{(1)}_{14}$ and $T^{(4)}_{14}$ always keep in the same order.

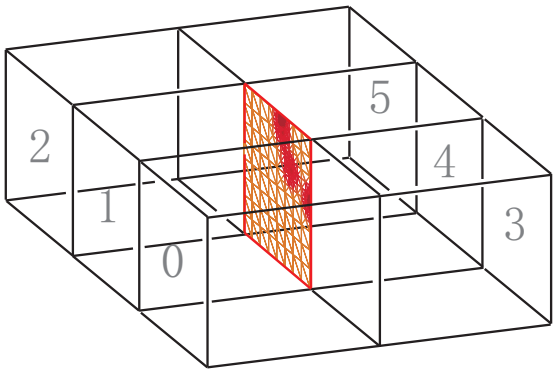

Figure 4: The neighboring sub-meshes of the interface-mesh.

3.3 Assigning indices to new nodes

We assign the interpolating nodes with a local set of indexes on each sub-mesh, $\{0,1,\cdots,\beta_N^{(i)}-1\}$, where $\beta_N^{(i)}$ is the number of nodes on sub-mesh i. Different from the method in [16], we have no global index for the nodes, so each processor assigns the nodes independently. For each quadratic basis function there are 10 interpolating nodes in one tetrahedron located on each vertex and on the middle point of each edge. In this case, these arrays are stored in

- vertex$[i]$.node,$(i=0,1,\cdots,\beta_V-1)$, store the indexes of the nodes located on vertices, β_V is the number of vertices;
- edge$[i]$.node,$(i=0,1,\cdots,\beta_E-1)$, store the indexes of the nodes located on edges, β_E is the number of edges.

When a new node is created, a unique index is assigned to it. In a step of bisecting a tetrahedron, 4 new nodes will be created at most, if the nodes have not been created by the bisection of neighboring tetrahedra (see Fig. 5). The new vertex will inherit the node originally belonging to the edge bisected. Each new edge will get a new node assigned as β_N, then $\beta_N \to \beta_N+1$.

The nodes on the interface-meshes are assigned by the same way. We demand that the indexes of the nodes on the interface-meshes $T'^{(i)}_{ij}$ and $T'^{(j)}_{ij}$ also keep in the same order throughout the refinement just like that for the vertices and edges. In the finite element implementation, a vector is presented by the values on the mesh nodes, which are stored by an array in β_N order. We store $\beta_N^{(i)}$ array in ith processor after dividing the array into P parts, each of which corresponds to one sub-domain. The overlapping parts will be stored repeatedly in all neighboring processors. With the mapping between the nodes of sub-mesh and interface-meshes we can find the overlapping parts in the array, and exchange them between the neighboring processors.

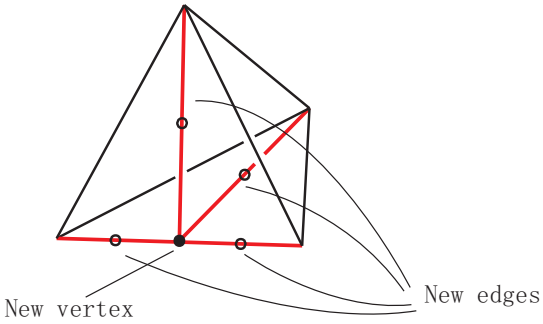

Figure 5: In a step of bisecting a tetrahedron, 4 new nodes will be created at most for a quadratic basis function.

4 Regular shapes of tetrahedra in refined mesh

Starting from the regular mesh the tetrahedra in the refined mesh keep some of the regular shapes throughout the refinements. In the initial regular mesh all the tetrahedra are congruent and marked the same type, type A. Obviously, all the bisected tetrahedra are marked in three types A, P_u, P_f, as shown in Fig. 6. It can be seen that all tetrahedra with the same type are similar to each other in the refined mesh from the initial regular mesh.

Definition 4.1. Let type A^0 be the mark of the initial tetrahedra for the regular mesh. Let type P_f^i be the mark of the son tetrahedra of type A^i, type P_u^i be the mark of the son tetrahedra of P_f^i tetrahedra, type A^{i+1} be the mark of the son tetrahedra of P_u^i, $(i = 0,1,2,\cdots)$. Then A^i, P_f^i, P_u^i is the i-th generation of bisected tetrahedra,

$$A^0 \to P_f^0 \to P_u^0 \to A^1 \to \cdots \to P_f^i \to P_u^i \to A^{i+1} \to \cdots. \tag{4.1}$$

Lemma 4.1. *The tetrahedra of the same type and generation are congruent with each other. The tetrahedra of the same type but different generation are similar to each other.*

Proof. It is easy to verify that A^0 type tetrahedra are congruent with each other. As in Fig. 7, when an A^0 type tetrahedron is bisected into two P_f^0 type tetrahedra, $AE = CE,\ AC = CA,\ AD = CB,\ EC = EA,\ ED = EB,\ CD = AB \Rightarrow T_{AECD} \cong T_{CEAB}$. Therefore the tetrahedra of type P_f^0 are all congruent.

When a P_f^0 type tetrahedron is bisected into two P_u^0 type tetrahedra, $AC = AB,\ EC = EB,\ FC = FB \Rightarrow T_{AEFC} \cong T_{AEFB}$. Therefore the tetrahedra of type P_u^0 are all congruent.

When a P_f^0 type tetrahedron is bisected into two P_u^0 type tetrahedra, $AE = BE,\ AF = BF,\ AG = BG \Rightarrow T_{EFGA} \cong T_{EFGB}$. Therefore the tetrahedra of type A^1 are all congruent also. Obviously, $T_{ABCD} \simeq T_{GBFE}$, so that the tetrahedra of type A^1 are similar to that of A^0. □

Lemma 4.1 ensures that there are only 3 types of tetrahedra throughout the refinements.

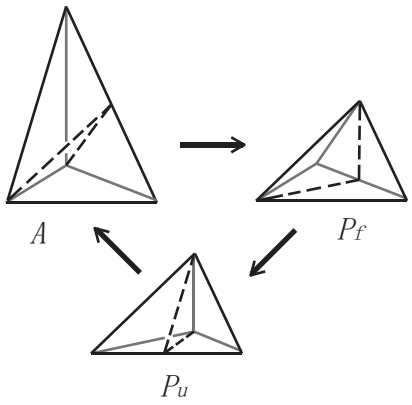

Figure 6: 3 types of tetrahedra.

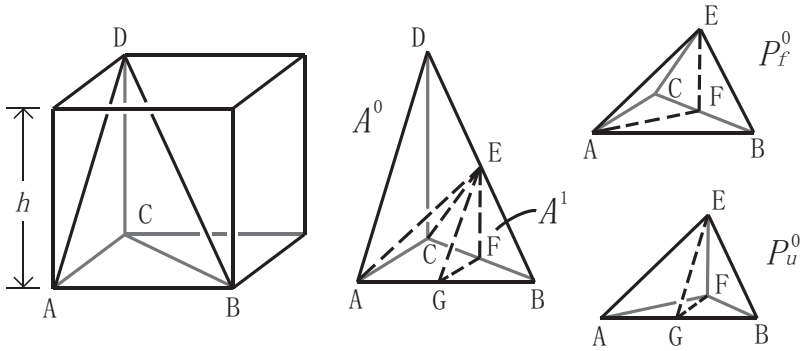

Figure 7: A^1 type tetrahedra are obtained by 3 successive bisections of an A^0 type tetrahedron. The right part of the figure shows the sequence of the bisections $A^0 \to P_f^0 \to P_u^0 \to A^1$.

Corollary 4.1. The tetrahedra of the same type in i-th generation and j-th generation satisfy

$$l_i = 2^{(i-j)} l_j, \quad V_i = 8^{(i-j)} V_j, \tag{4.2}$$

where l_i is the edge length of i-th tetrahedra and V_i is the volume of i-th tetrahedra.

Lemma 4.1 and Corollary 4.1 ensure the regular shape of the tetrahedra in the refined mesh. In finite element calculations these properties provide a fast algorithm to generate stiffness matrix (see [12]).

Suppose that in the initial mesh the length of the cube edge is h. Let type $\mathcal{A}^i$ be the mark of the edges with edge length $\sqrt{3}h/2^i$, type $\mathcal{P}_f^i$ be the mark of the edges with edge length $\sqrt{2}h/2^i$, type $\mathcal{P}_u^i$ be the mark of the edges with edge length $h/2^i$, where i is the generation number of the edges, $(i=0,1,2,\cdots)$.

When bisecting a tetrahedron, the particular edge to be refined is called the *refinement edge* [22].

Lemma 4.2. *The refinement edge of an A^i type tetrahedron is of type $\mathcal{A}^i$, the refinement edge of a P_f^i type tetrahedron is of type $\mathcal{P}_f^i$, and the refinement edge of a P_u^i type tetrahedron is of type $\mathcal{P}_u^i$.*

Proof. As in Fig. 7, the refinement edge in A^0 type tetrahedron T_{ABCD} is BD, the refinement edge in P_f^0 type tetrahedron T_{CEAB} is BC, and the refinement edge in P_u^0 type tetrahedron T_{AEFB} is AB. Suppose in the initial mesh the length of the cube edge AB is h. Then according to Corollary 4.1, the refinement edge length of the i-th generation tetrahedron is

$$l_i = \begin{cases} \sqrt{3}h/2^i & \text{type } A, \\ \sqrt{2}h/2^i & \text{type } P_f, \\ h/2^i & \text{type } P_u. \end{cases} \tag{4.3}$$

Thus the lengths of refinement edges belonging to different type of tetrahedra never equal to each other; conversely if two tetrahedra shared their refinement edge they must belong to the same type and generation. □

Lemma 4.2 ensures that the types of tetrahedron correspond to the types of their refinement edges. If an edge shared by two tetrahedra is the refinement (longest) edge on both tetrahedra, the two tetrahedra belong to the same type and generation.

We close this section by giving a proof of Lemma 3.1.

Proof. First, for a regular mesh with $a\times b\times c$ elements there hold

$$\begin{aligned} &\beta_V = (a+1)(b+1)(c+1), \\ &\beta_T = 6abc < 6\beta_V, \\ &\beta_E = (2a+1)(2b+1)(2c+1) - (a+1)(b+1)(c+1) < 7\beta_V, \\ &\beta_F = 12abc + 2ab + 2ac + 2bc < 12\beta_V. \end{aligned} \tag{4.4}$$

In an infinite regular mesh each cubic element corresponds to 6 tetrahedra, 12 faces, 7 edges, and 1 vertex, namely

$$\beta_E = 7\beta_V, \quad \beta_F = 12\beta_V, \quad \beta_T = 6\beta_V. \tag{4.5}$$

Then, we will prove that the refined meshes starting from the regular mesh also satisfy those inequalities in (3.2). Whatever the refinements proceed, any refined mesh can be generated from the initial regular mesh by bisecting the edges in the order

$$\mathcal{A}^0 \to \mathcal{P}_f^0 \to \mathcal{P}_u^0 \to \mathcal{A}^1 \to \cdots \to \mathcal{P}_f^i \to \mathcal{P}_u^i \to \mathcal{A}^{i+1} \to \cdots. \tag{4.6}$$

In each bisection step, all the neighboring tetrahedra will be bisected. This order verifies the bisection by the tetrahedron generation, and no hanging vertex exists during the generation. In a step of bisecting an edge and all of its neighboring tetrahedra, one new vertex will be added and the increasing numbers of the edges, faces and tetrahedra are dependent on the number of neighboring tetrahedra. In Fig. 8, suppose the refined edge

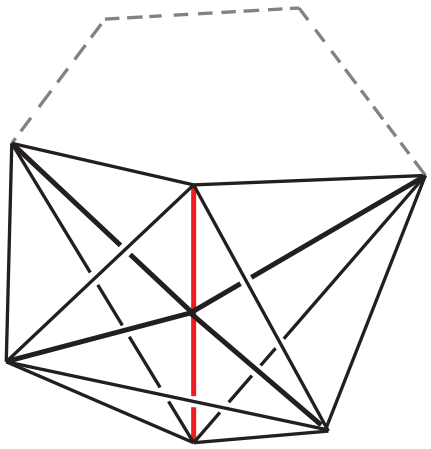

Figure 8: The refinement edge has n neighboring tetrahedron. The bisection of this edge will result in $n+1$ new edges, $2n$ new faces and n new tetrahedra.

to be bisected is not on the boundary and has n neighboring tetrahedra, the increasing numbers of the vertices, edges, faces and tetrahedra are

$$\begin{aligned} &\delta\beta_V=1, \quad \delta\beta_E=n+1, \\ &\delta\beta_F=2n, \quad \delta\beta_T=n, \end{aligned} \tag{4.7}$$

where n is determined by the type of the refinement edge,

$$n=\begin{cases} 6 & \text{the edge of type } \mathcal{A}, \\ 4 & \text{the edge of type } \mathcal{P}_f, \\ 8 & \text{the edge of type } \mathcal{P}_u. \end{cases} \tag{4.8}$$

If the refined edge is on the boundary, less edges, faces and tetrahedra will be added:

$$\begin{aligned} &\delta\beta_V=1, \quad \delta\beta_E<n+1, \\ &\delta\beta_F<2n, \quad \delta\beta_T<n, \end{aligned} \tag{4.9}$$

where n satisfies (4.8).

Each step of bisecting a $\mathcal{P}_f$ type edge results in 8 P_u type tetrahedra generated. Each step of bisecting a $\mathcal{P}_u$ type edge will consume 8 P_u type tetrahedra, and all the P_u type tetrahedra are produced by bisecting the $\mathcal{P}_f$ type edges. For that the times of bisecting $\mathcal{P}_u$ type edges is less than or equal to those of bisecting $\mathcal{P}_f$ type edges. From Eq. (4.8) the average number of n, $\bar{n}$ satisfy

$$\bar{n}\leq 6. \tag{4.10}$$

From (4.7) and (4.9), the average numbers of increasing vertices, edges, faces and tetra-

hedra in each step of bisecting an edge satisfy

$$\begin{aligned}
&\overline{\delta\beta}_V = 1,\\
&\overline{\delta\beta}_E = \bar{n}+1 \le 7 = 7\overline{\delta\beta}_V,\\
&\overline{\delta\beta}_F = 2\bar{n} \le 12 = 12\overline{\delta\beta}_V,\\
&\overline{\delta\beta}_T = \bar{n} \le 6 = 6\overline{\delta\beta}_V.
\end{aligned} \tag{4.11}$$

The desired inequality (3.2) follows from (4.11), (4.4) and (4.5). This completes the proof. □

5 Distributed data storage and parallel matrix-vector multiplications

In our method, with the local assignment of the index of FE interpolating nodes the generation and storage of matrices are distributed locally on the processors. Each processor i constructs the local matrix M_i serially on its own sub-mesh T_i without exchanging any data with others. In this way the holistic matrix is divided into P local matrices, P is the number of sub-domains. The nodes shared by the neighboring sub-meshes result in the overlap of the local matrices. These overlapping parts need not to be summed up, each processor treats its own local matrix only. The holistic matrix $M = \sum_i M_i$ is not stored. The storage of vectors is also distributed over all processors, each for its own sub-domain, repetitively for the overlapping part. Note that the array elements of the vector on the shared nodes should be saved repetitively in each processor. Fig. 9 shows a sketch for the storage of the matrix and vector for 2 processors.

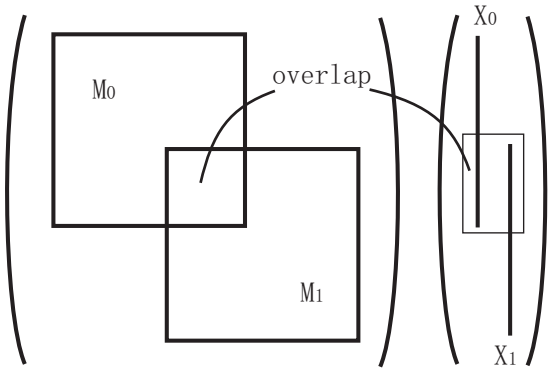

Figure 9: The matrices and vectors are stored locally on each processor, with the overlapping parts of the shared nodes of the neighboring sub-domains.

The matrix-vector multiplication is also localized. Suppose the holistic matrix $M = \sum_i M_i$ and the vector x is distributed over all processors as x_i. The matrix-vector multiplication $y = Mx = \sum_i M_i x_i$ is calculated in 2 steps:

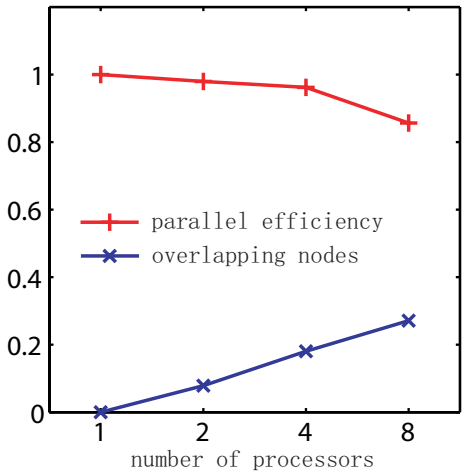

Figure 10: The parallel efficiency in H_2 calculation.

1. Calculate the serial matrix-vector multiplication, that is, $\tilde{y}_i = M_i x_i$;
2. Exchange the vector values of $\tilde{y}_i$ on the overlapping nodes between the neighboring processors and sum them up, that is $y_i(z) = \sum_i \tilde{y}_i(z)$ for any node $z \in T_i$.

In this way, the cost of the data exchange is reduced. The repetitive calculations of matrix-vector multiplication are determined by the number of the overlapping nodes. Here an example for parallel efficiency and the proportion of overlapping nodes depending on the number of sub-domains in the calculation of H_2 molecular is shown in Fig. 10. In such a calculation matrix-vector multiplication in solving the Kohn-Sham equation [24] consumes about 90% of the resource.

6 Numerical experiments

In this section, three typical examples are presented to show the efficiency of our parallel adaptive refinement algorithm: face centered cubic (fcc) super-cell with 32 carbon atoms (Fig. 11) demonstrating the periodic system, the C_{60} cluster (Fig. 2) demonstrating the finite system, and the carbon nano-tube with 72 atoms (Fig. 12) demonstrating the transport problem respectively.

In practice, the first example deals with the electronic properties of crystal, i.e. the periodic structure. One super-cell consisting of 32 carbon atoms, together with periodic boundary condition, is adopted. A $2\times2\times2$ super-cell is taken as a domain. In the second example, the domain is a large cuboid to model a cluster in vacuum. The size of the domain is optional but large enough. In the last example, the carbon nano-tube is considered with special boundary condition along the axis of the tube. The size of the domain along this direction is determined by the boundary condition, while along other directions, the size is optional as for the second example.

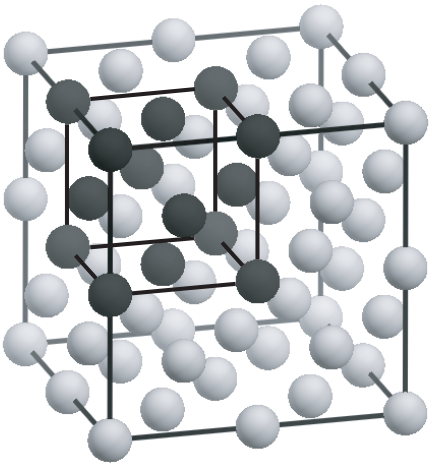

Figure 11: Domain containing $2\times2\times2$ super-cell of *fcc* crystal.

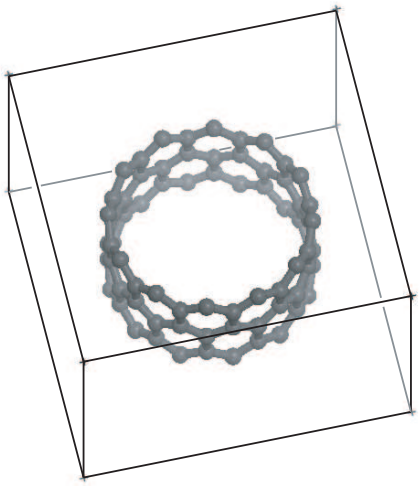

Figure 12: Carbon nano-tube with 72 atoms in a cuboid domain.

All the numerical experiments were carried out on SGI O300 machine with 16 processors. The efficiency of parallel calculation is tested by increasing the number of processors from 1 to 2, 4, 8, 12 and 16. The refinement of the mesh is based on an error estimator [13] for the wave functions. The wave functions are the eigenvectors of a nonlinear Schrödinger equation [24, 25],

$$\left[-\frac{1}{2}\nabla^2+V_{\text{eff}}(\boldsymbol{r})\right]\psi_i(\boldsymbol{r})=\epsilon_i\psi_i(\boldsymbol{r}), \tag{6.1}$$

$$V_{\text{eff}}(\boldsymbol{r})=\sum_s V^s_{\text{ion}}(\boldsymbol{r}-\boldsymbol{R}^s)+V_{\text{Hartree}}(\rho)+V_{\text{xc}}(\rho(\boldsymbol{r})), \tag{6.2}$$

where the first, second, and third terms on the right side of Eq. (6.2) are the ionic potential, Hartree potential, and exchange-correlation potential respectively. Moreover,

$$\rho(\mathbf{r})=\sum_i^M n_i|\psi_i(\mathbf{r})|^2, \tag{6.3}$$

where n_i is the occupation number of state i. Only the first M smallest eigenvalues need to be calculated, where M is half of the valence electron number. In our previous works [12, 26], we presented the FE solutions for (6.1)-(6.3) on adaptive mesh.

The Hartree potential is the Coulomb interaction of electrons,

$$V_{\text{Hartree}}(\boldsymbol{r})=\int\frac{\rho(\boldsymbol{r}')}{|\boldsymbol{r}-\boldsymbol{r}'|}d\boldsymbol{r}' \tag{6.4}$$

satisfying the Poisson equation

$$-\Delta V_{\text{Hartree}}=4\pi\rho. \tag{6.5}$$

The computational procedure of our examples is sketched in the following:

1. Generate the initial mesh.
2. Calculate the wave functions.
3. Estimate the local error indicators.
4. If the error estimated is not small enough, then refine the mesh and go to Step 2.

For all the three examples, the domain is partitioned into P sub-domains, where P is the number of processors. Each sub-domain is a unit cuboid. All the initial sub-meshes are regular. The boundary of each sub-mesh is unchanged throughout the refinements.

Table 1: Test results from SGI Origin 300 computer.

fcc super-cell	217,728 tetrahedra generated		
No. processor	tetrahedra/Sec.	wall time (Sec.)	efficiency
1	62,565	3.48	1.00
2	127,007	1.71	1.02
4	250,260	0.87	1.00
8	471,490	0.46	0.94
12	603,627	0.36	0.80
16	820,853	0.27	0.82
C_{60} cluster	711,412 tetrahedra generated		
No. processor	tetrahedra/Sec.	wall time (Sec.)	efficiency
1	63,095	11.28	1.00
2	123,540	5.76	0.98
4	251,118	2.83	0.99
8	469,932	1.51	0.93
12	595,718	1.19	0.79
16	710,601	1.00	0.70
nano-tube	717,776 tetrahedra generated		
No. processor	tetrahedra/Sec.	wall time (Sec.)	efficiency
1	65,001	11.04	1.00
2	122,019	5.56	0.94
4	262,864	2.73	1.01
8	494,163	1.45	0.95
12	705,443	1.02	0.90
16	894,102	0.80	0.86

The number of tetrahedra in the final mesh, the parallel refinement time, and the average number of tetrahedra created per second with different processor are listed in Table

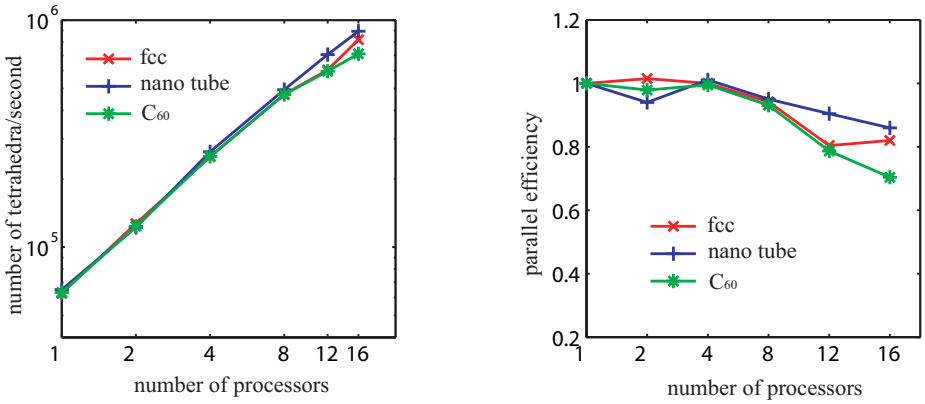

Figure 13: Number of tetrahedra/second and parallel efficiency versus number of processors.

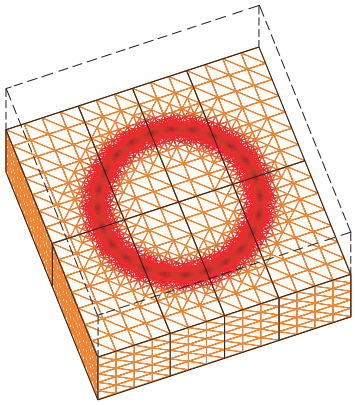

Figure 14: Domain subdivision of carbon nano-tube with 16 ($=4\times2\times2$) sub-meshes. The dark region shows the more refined part of the mesh where the nano-tube is located.

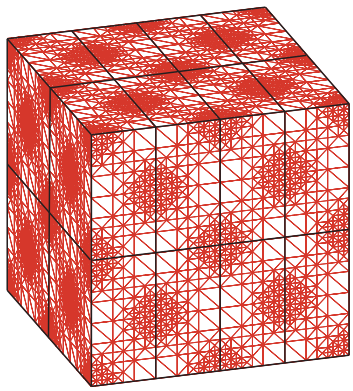

Figure 15: Domain subdivision of *fcc* super-cell with 16 ($=4\times2\times2$) sub-meshes. The refined mesh shows more dense grids around the atoms.

1, while the speedup and parallel efficiency obtained are shown in Fig. 13. In the first example, the initial mesh contains 384 tetrahedra, while the final number of tetrahedra increases to 218,112. The parallel refinements are efficient until the number of processors reaches 16. Other two examples give similar results.

Compared with the regions far from the atoms, the regions around the atomic nuclei and those between atoms connected by chemical bonds are more important. The self-adaptive method based on posterior-error analysis of the wave functions places more nodes within such critical regions. As shown in Fig. 14, the dark region shows that more refined meshes are presented in the area where the nano-tube is located.

The domain subdivision takes the character of the specific physical problems under consideration into account by putting roughly the same number of atoms in each sub-domain. The final meshes for 16 processors are demonstrated in Figs. 14 and 15 respectively. It is observed that the resulting subdivisions are balanced.

7 Conclusion

In this paper, we present a parallel algorithm for adaptive mesh refinement in finite element calculations of electronic structure. The algorithm allows simultaneous refinement of sub-meshes before synchronization between sub-meshes, without the need of a global index assignment and a central coordinator processor. Numerical experiments are carried out to demonstrate the efficiency of the proposed algorithm.

Acknowledgments

The authors are thankful to the discussions with Prof. Wei-Guo Gao, Prof. Lin-bo Zhang, Prof. Pingwen Zhang, Prof. Lihua Shen and Dr. Xiaoying Dai. This work is partially supported by NSF of China, the National Basic Research Program of China, MOE and Shanghai basic research project.

References

[1] R. M. Martin, Electronic Structure Basic Theory and Practical Methods, Cambridge University Press, 2004, 236.
[2] E. Tsuchida and M. Tsuukada, Electronic-structure calculations based on the finite-element method, Phys. Rev. B, 52 (1995), 5573–5578.
[3] E. Tsuchida and M. Tsuukada, Adaptive finite-element method for electronic-structure calculations, Phys. Rev. B, 54 (1996), 7602–7605.
[4] E. Tsuchida and M. Tesukada, Large-scale electronic-structure calculations based on the adaptive finite element methods, J. Phy. Soc. Jpn., 67 (1998), 3844–3858.
[5] T. L. Beck, Real-space mesh techniques in density-function theory, Rev. Modern Phys., 72 (2000), 1041–1080.
[6] J. E. Pask, B. M. Klein, P.A. Sterne and C.Y. Fong, Finite-element methods in electronic-structure theory, Comput. Phys. Commun., 135 (2001), 1–34.
[7] E. Tsuchida, Ab initio molecular-dynamics study of liquid formamide, J. Chem. Phys., 121 (2004), 4740–4746.
[8] J. E. Pask and P. A. Sterne, Finite element methods in ab initio electronic structure calculations, Modelling Simul. Mater. Sci. Eng., 13 (2005), R7–R96.
[9] T. Torsti, T. Eirola, J. Enkovaara1, T. Hakala, P. Havu, V. Havu, T. Höynälänmaa, J. Ignatius, M. Lyly, I. Makkonen, T. T. Rantala, J. Ruokolainen, K. Ruotsalainen, E. Räsänen, H. Saarikoski, and M. J. Puska, Three real-space discretization techniques in electronic structure calculations, Phys. Stat. Sol., B243 (2006), 1016-1053.
[10] S. R. White, J.W. Wilkinsand and M.P. Teter, Finite-element method for electronic structure, Phys. Rev. B, 39 (1989), 5819–5833.
[11] T. L. Beck, Multigrid high-order mesh refinement techniques for composite grid electrostatics calculations, J. Comput. Chem., 20 (1999), 1731–1739.
[12] D. Zhang, L. Shen, A. Zhou and X. G. Gong, Finite element method for solving Kohn-Sham equations based on self-adaptive simplex mesh, Phys. Lett. A., to appear.

[13] D. Mao, L. Shen and A. Zhou, Adaptive finite element algorithms for eigenvalue problems based on local averaging type a posteriori error estimates, Adv. Comp. Math., 25, (2006), 135–160.
[14] M. T. Jones and P.E. Plassmann, Parallel algorithms for adaptive mesh refinement, SIAM J. Sci. Comput., 18 (1997), 686–708.
[15] J. G. Castaños and J.E. Savage, Parallel refinement of unstructured meshes, Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, November 3–6, (1999), MIT, Boston, USA.
[16] L. Zhang, A parallel algorithm for adaptive local refinement of tetrahedral meshes using bisection, Preprint ICM–05–09, (2005).
[17] E. Bänsch, An adaptive finite-element strategy for the three dimensional time dependent Navier-Stokes equations, J. Comput. Appl. Math., 36 (1991), 3–28.
[18] J. M. Maubach, Local bisection refinement for N-simplicial grids generated by reflection, SIAM J. Sci. Comput., 16 (1995), 210–227.
[19] R. Horst, On generalized bisection of n-simplices, Math. Comp., 66 (1997), 691–698.
[20] I. Kossaczky, A recursive approach to local mesh refinement in two and three dimensions, J. Comput. Appl. Math., 55 (1994), 275–288.
[21] A. Liu and B. Joe, Quality local refinement of tetrahedral meshes based on bisection, SIAM J. Sci. Comput., 16 (1995), 1269–1291.
[22] D. N. Arnold, A. Mukherjee and L. Pouly, Locally adapted tetrahedral meshes using bisection, SIAM J. Sci. Comput., 22 (2000), 431–448.
[23] A. Khamayseh and G. Hansen, Use of the spatial kD-tree in computational physics applications, Commun. Comput. Phys., 2 (2007), 545–576.
[24] W. Kohn, Noble lecture: electronic structure of matter wave functions and density functionals, Rev. Mod. Phys., 71 (1999) 1253.
[25] R. O. Jones and O. Gunnarsson, Density-functional formalism: Sources of error in local-density approximations, Phys. Rev. Lett. 55 (1985) 107.
[26] X. G. Gong, L. Shen, D. Zhang and A. Zhou, Finit element approximations for schrodinger equations with applications to electronic structure computations, J. Comput. Math., 26 (2008), 310–323.