Int J Parallel Prog (2011) 39:717–745
DOI 10.1007/s10766-011-0168-3

A Parallel Implementation of ALE Moving Mesh Technique for FSI Problems using OpenMP

Masroor Hussain · Muhammad Abid · Mushtaq Ahmad · Ashfaq Khokhar · Arif Masud

Received: 1 January 2010 / Accepted: 11 February 2011 / Published online: 26 March 2011

Abstract This paper investigates a high performance implementation of an Arbitrary Lagrangian Eulerian moving mesh technique on shared memory systems using OpenMP environment. Moving mesh techniques are considered an integral part of a wider class of fluid mechanics problems that involve moving and deforming spatial domains, namely, free-surface flows and Fluid Structure Interaction (FSI). The moving mesh technique adopted in this work is based on the notion of nodes relocation, subjected to a certain evolution as well as constraint conditions. A conjugate gradient method augmented with preconditioning is employed for solution of the resulting system of equations. The proposed algorithm, initially, reorders the mesh using an efficient divide and conquer approach and then parallelizes the ALE moving mesh scheme. Numerical simulations are conducted on the multicore AMD Opteron and Intel Xeon processors, and unstructured triangular and tetrahedral meshes are used for the 2D and 3D problems. The quality of generated meshes is checked by comparing the element Jacobians in the reference and current meshes, and by keeping track of the change in the interior angles in triangles and tetrahedrons. Overall, 51 and 72% efficiencies in terms of speedup are achieved for both the parallel mesh reordering and ALE moving mesh algorithms, respectively.

M. Hussain (✉) · M. Ahmad
FCSE, GIK Institute, Topi, Pakistan
e-mail: hussain@giki.edu.pk

M. Abid
FME, GIK Institute, Topi, Pakistan

A. Khokhar
University of Illinois at Chicago, Chicago, IL, USA

A. Masud
University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA

Keywords ALE method · Matrix bandwidth · Mesh reordering · Moving mesh · OpenMP · Parallel

1 Introduction

Adaptive mesh motion techniques constitute a major component of fluid flow problems where the domain of computation changes as a function of time. This situation arises both in internal as well as in external flows. Examples of typical external flows are: free-surface waves interacting with off-shore oil platforms, flows around deformable structures moving through the fluid domain, and flows around propeller of submarines and surface ships that constitute their propulsion mechanisms. Likewise, examples of internal flows are: free-surface oscillations in the liquid storage tanks that lead to flow induced vibrations in the structures, and cardiovascular blood flow through a distensible arterial system.

Various viewpoints have been pursued in the literature to develop good mesh moving techniques. In these works underlying objective is to deliver viable computational grids that not only satisfy the physics based constraints of uniform evolution of the mesh with the fluid continuum at the moving boundaries, but also provide high mesh resolutions in regions of steep gradients in the solution. The methods adopted can be broadly grouped into three classes: h-refinements or adaptive local mesh refinements, p-refinement or the local polynomial enhancements, and r-refinements or relocation of mesh nodal points, while keeping the option of maintaining element connectivity [1–9] and [10]. Following the r-refinement concept, Masud and Hughes [8] proposed a method based on Galerkin/Least-Squares type modification of the Laplace equation that introduces spatially varying scalable-incompressibility effects in the computational domain. This method was applied to a variety of 2D problems by Masud et al. [9] and extended to 3D problems by Kanchi and Masud [10]. Tezduyar and co-workers proposed a solution of modified elasticity equations wherein element Jacobian was excluded in the numerical calculations, thereby introducing variable stiffening effect in the computational domain [11] and [12]. Efficient time-integration techniques for flow problems involving moving and deforming meshes have been pursued by Farhat et al. [13]. Masud [14] investigated the effects of the moving mesh on the stability of fluid flow equations to solve moving boundary flow problems in the Arbitrary Lagrangian Eulerian (ALE) frame of reference. In the rest of this paper moving mesh based on the ALE frame of reference would be termed as ALE moving mesh. Khurram and Masud [15], and Calderer and Masud [16] also employed the ALE moving mesh concept in multiscale/stabilized formulation of the incompressible Navier-Stokes equations. A literature review reveals that ALE moving mesh schemes can be applied to a wide range of FSI problems like human breathing system, blood circulatory system, aerodynamic behavior of aircrafts, flows around ships and submarines, seismic response of liquid storage tanks, propagation of solitary and shock waves and even in the flow over micro structures.

For computational fluid dynamics, the mesh moving techniques need to be integrated with the partial differential equations that govern the flow of fluid. This is accomplished by employing the ALE framework, wherein part of the mesh can be

treated as an Eulerian mesh, while in some other regions the mesh nodal points may be allowed to move independent of the fluid particle motion to accommodate the motion and deformation of the domain of computation. Consequently, in ALE based methods for fluid mechanics, mesh motion is a precursor to the continuously evolving flow problems in the various time steps. For transient calculations that involve millions of elements and thousands of time steps, optimizing the mesh moving scheme for massively parallel architectures can become a formidable task. A literature review shows that most of the work for parallel adaptive meshes is conducted on distributed memory systems [17–20] and [21]. Graph partitioning techniques, using the minimum cut set [19,20] and [21], and octree partition method [17] and [18] have been applied to compute the mesh refinement in parallel. Mitchell [18] presented better results for the dynamic load balancing with adaptive meshes using octree based method over the graph based partitioning, Hilbert space filling curve (HSFC), recursive coordinate bisection (RCB) and recursive inertial bisection (RIB). Multilevel graph partitioning techniques using Metis library have been studied to provide the functionality to reorder and partition the irregular graphs [22,23] and [24].

This paper presents a novel parallel algorithm for the ALE moving mesh problem. The underlying method is based on the r-refinement involving node relocation, mesh reordering and work load re-distribution using sampling approach [25] and [26]. The proposed algorithm is implemented on shared memory machines using OpenMP [27] and performance results are reported. In particular, quadtree and octree (spatial data structures) are employed for the 2D and 3D moving mesh problems, respectively, along with the multi-dimensional quicksort technique to reorder and partition the mesh. For the case where mesh is already partitioned over multiple threads or processors, we have introduced a novel approach to reorder mesh that is based on regular sampling and samplesort techniques [25] and [26]. This approach selects a splitters set from the original mesh and partitions the mesh for dynamic load balancing. The mesh reordering reduces the cache miss rate, particularly for the processors with small cache memory. It also reduces the bandwidth of a matrix in the system of linear equations, which is an NP-hard problem [28] and [29]. Reducing the system matrix bandwidth is an important issue to solve the problems using finite element methods (FEMs) [30] and [31]. Numerical simulations are conducted on four cores of AMD machine and eight cores of Intel Xeon machine. Quality of a mesh is verified by comparing the element Jacobian in the current mesh with the corresponding element Jacobian in the reference or the parent mesh. Another check of the quality of the mesh is facilitated by tracking the change in the interior angles of triangles and tetrahedrons. Interior angles close to that of the reference triangles and tetrahedral elements indicate a good quality element.

The rest of the paper is organized as follows. For the sake of completeness, Sect. 2 outlines the underlying variational structure of the moving mesh problem. The serial algorithm for optimization using preconditioned conjugate gradient method is presented in Sect. 3. Section 4 presents a mesh reordering algorithm based on quadtree, octree, quicksort and sampling. Section 5 discusses the parallelization part of the moving mesh algorithm. Numerical experiments are presented in Sect. 6, discussion on numerical results is presented in Sect. 7, and conclusion is drawn in Sect. 8.

2 Boundary Value Problem for the ALE Moving Mesh Scheme

Boundary value problem (BVP) for the ALE moving mesh method is taken from Masud et al. [8,9] and [10]. The BVP is defined over domain $\Omega \subset R^{n_{sd}}$ with smooth boundary Γ, where n_{sd} denotes the number of spatial dimensions. Here, Γ consists of the fixed Γ_f and the moving Γ_m boundaries; such that

$$\Gamma = \Gamma_f \cup \Gamma_m \tag{1}$$

with

$$\emptyset = \Gamma_f \cap \Gamma_m \tag{2}$$

The underlying equations for mesh motion comprise the Laplace equation defined over domain and the prescribed moving and fixed boundary conditions on parts of the boundary, respectively.

$$\nabla^2 u = 0 \text{ in } \Omega \tag{3}$$
$$u = g \text{ on } \Gamma_m \tag{4}$$
$$u = 0 \text{ on } \Gamma_f \tag{5}$$

Equations (3), (4) and (5) are governing equations of the moving mesh problem with both the moving and fixed boundary conditions, respectively. Spaces relevant to the BVP are defined as:

$$S = \{u \mid u \in ((H^1(\Omega))^{n_{sd}}, u = g \text{ on } \Gamma_m \text{ and } u = 0 \text{ on } \Gamma_f\} \tag{6}$$
$$V = \{w \mid w \in (H_0^1(\Omega))^{n_{sd}}\} \tag{7}$$

where $H^1(\Omega)$ denotes the space of functions in $L_2(\Omega)$ with generalized derivatives also in $L_2(\Omega)$. $H_0^1(\Omega)$ is a subset of $H^1(\Omega)$, whose members satisfy the zero boundary conditions and w is an arbitrary weight function [9]. The Laplace equation for the unknown displacement field works well for problems where the meshes are composed of approximately equal-sized elements, and the motion of the interface boundary is of the order of the size of the elements. If the motion of the interface boundary is larger than the size of the elements adjacent to the moving boundary then employing equation (3) results in overturning of the elements which results in algorithm breakdown. Typical fluid meshes invariably have higher resolution close to the moving boundaries than in the far field. In order to prevent the overturning of the smaller elements we add a constraint on the gradient of the displacement field:

$$|\nabla u^h| \leq \alpha \tag{8}$$

where $\alpha \in [0, 1)$ is a tolerance parameter for the element distortion. In order to impose this constraint, we design an element based weight function τ^e in Ω^e, that is designed such that it imposes the constraint condition strongly over the smaller elements as

compared to that over the larger elements. It thus introduces a stiffening effect that is inversely proportional to the size of the elements. A simple definition of τ^e is:

$$\tau^e = \frac{1 - \Delta_{\min}/\Delta_{\max}}{\Delta_e/\Delta_{\max}} \tag{9}$$

where Δ_e, $\Delta_{\min}$ and $\Delta_{\max}$ are the current, minimum and maximum areas (2D problem) or volumes (3D problem) of an element. Finite element form of the problem is defined as [9]:

$$B(w^h, u^h) = 0 \tag{10}$$

$$B(w^h, u^h) = (\nabla w^h, \nabla u^h) + \sum_{e=1}^{n_{el}} \tau^e (\nabla w^h, \nabla u^h)\Omega^e \tag{11}$$

where n_{el} is the total number of elements and $(\cdot, \cdot)$ denotes L_2 inner product, which is bilinear, continuous and symmetric. In general, the L_2 inner product of two real functions f and g on any space X is defined as:

$$(f, g)_{L_2} = \int_X fg dx \tag{12}$$

Preconditioned conjugate gradient (PCG) solver is employed to solve the finite element optimization problem defined in the Eq. (10) and (11) [8,9] and [10]. Both the tolerance and iterative variables are used to check the convergence of the solution.

3 Description of Serial ALE Moving Mesh Algorithm

The serial ALE moving mesh algorithm uses the following steps to generate a new mesh using Eqs. (10) and (11):

1. Generate the element stiffness matrix.
2. Move the boundary nodes subject to given constraints.
3. Generate the new nodal values of the mesh using PCG.
4. Update the existing nodal values to generate a new mesh.

Algorithm 1 depicts the serial algorithm to generate the new mesh using PCG method, described in [8] and [32]. A system of linear equations is defined by $Av = b$, where A is the element stiffness matrix. The values of b (vector) are equal to the displacement of the boundary nodes on the Γ_m, and on internal nodes, values of the respective b are equal to 0.

Lines 1–10 initialize the uncoupled equations. Line 3 computes the preconditioned matrix P through the diagonal scaling of the element stiffness matrix A to solve the system of algebraic equations. Here, P is the nodal matrix and A is the element matrix. Lines 11–21 iterate j to the specified number of steps to achieve convergence of the solution that is integrated with the line search method in line 12. Line 13 updates

Algorithm 1 Precondition conjugate gradient algorithm (Ref [32])

Require: Element stiffness matrix A

Step 1: Initialize the uncoupled equations
1. $r_0 \leftarrow b$
2. $v_0 \leftarrow 0$
3. $P \leftarrow diagonal(A)$
4. **for** $j \leftarrow 0$ to n_{el} **do**
5. **if** $Aij = 0 \,\forall\, i \neq j$ **then**
6. $v_j \leftarrow r_j P_{jj}^{-1}$
7. $r_j \leftarrow 0$
8. **end if**
9. $q_1 \leftarrow z_1 \leftarrow P^{-1} r_0$
10. **end for**

Step 2: Iterate j (user defined) to achieve convergence and to update the solution
11. **for** $j \leftarrow 1$ to $given\_steps$ **do**
12. $\alpha_j \leftarrow \frac{r_{j-1} \cdot z_j}{q_j \cdot A q_j}$
13. $v_j \leftarrow v_{j-1} + \alpha_j q_j$
14. $r_j \leftarrow r_{j-1} - \alpha_j A q_j$

Check for the convergence (δ is a user defined tolerance)
15. **if** $\|r_j\| \leq \delta \|r_0\|$ **then**
16. **return**
17. **end if**

Compute a new conjugate direction
18. $z_{j+1} \leftarrow P^{-1} r_j$
19. $\beta_{j+1} \leftarrow \frac{r_j \cdot z_{j+1}}{r_{j-1} \cdot z_j}$
20. $q_{j+1} \leftarrow z_{j+1} + \beta_{j+1} q_j$
21. **end for**

the vector v of the algebraic equations and line 14 calculates the new residual for jth iteration. Lines 15–17 check the convergence of the solution. Finally, lines 18–21 compute the new conjugate direction to update the solution for the next iteration.

A two step methodology is adopted in order to solve the ALE moving mesh problem in parallel. In the first step, an un-deformed mesh is spatially ordered and a new mesh is created using the parallel ALE moving mesh generation method.

4 Algorithm for Mesh Reordering

The reordering of mesh nodes and elements is needed to ensure efficient memory mapping of neighboring elements and nodes. This is achieved by re-assigning the global numbering to both the elements and nodes of the mesh such that the associated nodes of all the elements are stored in contiguous memory locations. Such a mapping would improve cache miss rate thus reducing the overall execution time. In the case of a parallel implementation this reordering would also facilitate an efficient partitioning of the mesh computations over multiple threads. Consequently, mesh reordering also minimizes the bandwidth of the element stiffness matrix A used in the system of linear

equations $Av = b$, which is solved in Algorithm 1. In the following we first present a serial mesh reordering algorithms and study it performance. This algorithm can be used prior to mesh partitioning. Subsequently, we present a parallel mesh reordering algorithm that can be used in situations where mesh has already been partitioned, and moving mesh algorithm is just a frequent intermediate process.

4.1 Serial Mesh Reordering

A recursive divide and conquer algorithm is developed to reorder the mesh based on a greedy approach. This algorithm uses Peano-Hilbert space filling curve, recursive orthogonal bisection/octree (spatial data structures), and quicksort to reduce the matrix bandwidth. First, this greedy approach calculates the elements' centroid and creates a quadtree (for the 2D case) and octree (for the 3D case) using the quicksort based technique. Then it re-numbers both the elements and nodes of the mesh based on the geometric locality created by the quadtree/octree method.

This algorithm recursively divides the original mesh m spatially into $2^{n_{sd}}$ sub-meshes based on centroid values, where n_{sd} denotes the number of spatial dimensions. The centroid of a 3D mesh C_m can be calculated using the following formula:

$$C_m = \left\{ \left(\frac{x_{\min} + x_{\max}}{2} \right), \left(\frac{y_{\min} + y_{\max}}{2} \right), \left(\frac{z_{\min} + z_{\max}}{2} \right) \right\} \tag{13}$$

The mesh is divided recursively until the number of elements n_{el}^{sm} of a sub-mesh (sm) is less or equal to a user defined threshold value (t). The value of t should be equal to or a multiple of the size of cache-line of the processor.

Algorithm 2 shows the reordering algorithm of a mesh using the above described divide and conquer approach. The resultant formulation/output is a tree data structure, which is either a quadtree (for a 2D mesh) or an octree (for a 3D mesh). Line 3 calls the *rearrangemesh* function that partitions the original mesh from the centroid C_m. Lines 4–6 call the *reorder* function recursively for all the created sub-meshes. In an initial call of the *reorder (Mesh)*, the *Mesh* structure contains the centroid values and original indices of all the elements, e.g. centroid of a triangular element C_{tri} can be calculated by the following formula:

$$C_{tri} = \left\{ \left(\frac{x_1 + x_2 + x_3}{3} \right), \left(\frac{y_1 + y_2 + y_3}{3} \right) \right\} \tag{14}$$

Figure 1 graphically shows a 2D mesh of an oscillating beam that is divided into four sub-meshes: left top (LT), right top (RT), left bottom (LB) and right bottom (RB). In the same way a 3D mesh is divided into eight sub-volumes. Figure 2 shows the possible final state of a 2D oscillating beam after applying the reordering algorithm. This algorithm also organizes the 2D mesh data into a single dimensional array using Peano-Hilbert order.

Algorithm 3 shows the *Partition* function written in Fortran language, which is similar to the quicksort partitioning algorithm [33]. This function partitions the mesh into

Algorithm 2 A divide and conquer algorithm for reordering of a mesh

```
1. procedure reorder (Mesh)
2. if Mesh.n_el ≥ threshold then
3.    partitionedmesh ← rearrangemesh (Mesh)
4.    for i ← 1 to 2^n_sd do
5.       call reorder (partitionedmesh (i))
6.    end for
7.    Mesh ← partitionedmesh
8. end if
9. end procedure

10. function rearrangemesh (Mesh)
11. XMeshPar ← partition (Mesh, x-axis) {It divides a mesh into two partitions}

12. if Mesh = 1D then
13.    return XMeshPar
14. end if

15. YMeshPar ← partition(XMeshPar(left), y-axis)
16. YMeshPar ← YMeshPar ∪ partition(XMeshPar(right), y-axis)

17. if Mesh = 2D then
18.    return YMeshPar
19. end if

20. ZMeshPar ← ∅
21. if Mesh = 3D then
22.    for i ← 1 to 4 do
23.       ZMeshPar ← ZMeshPar ∪ partition(YMeshPar(i), z-axis)
24.    end for
25.    return ZMeshPar
26. end if
27. return ∅
28. end function
```

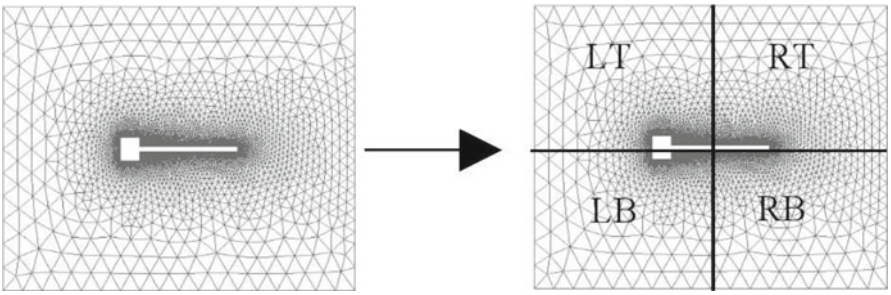

Fig. 1 Division of a 2D Mesh into four sub-meshes (LT, RT, LB and RB)

upper and lower spaces based on the centroid values. An initial call of the *Partition* function is *Partition* (*CentroidofAllElements, IndexofAllElements, 1,* n_{el}*, min, max, 1, mid*). The following are the description of the variables used in the *Partition* function:

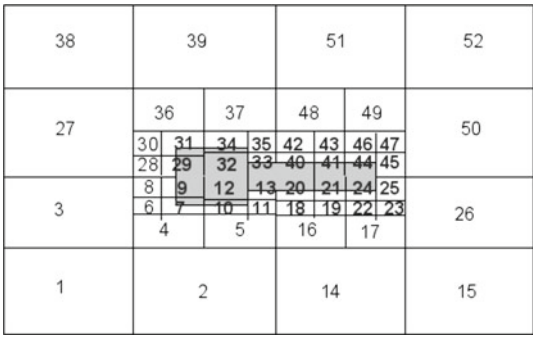

Fig. 2 A 2D beam is partitioned into 52 sub-areas. The number of elements of all the sub-areas is less or equal to a user defined threshold value

Algorithm 3 Fortran implementation of the partition function that partitions a mesh into two halves in a particular spatial dimension

```
subroutine
Partition(eleval, eleid, p, r, min, max, axis, mid)
c all the variables are called by reference
   implicit none
   integer eleid(1)
   real*8  eleval(3, 1)
   integer p, r, axis, i, j
   real*8 min, max, pivotvalue
   integer mid
   i = p − 1
   pivotvalue = (min + max)/2.0
c  axis is equal to 1, 2 and 3 for the x, y and z axis, respectively
   do j = p, r
      if (eleval(axis, j) .lt. pivotvalue) then
         if (i .ne. j) then
            i = i + 1
            call swap(eleval(1, i), eleval(1, j))
            call swap(eleval(2, i), eleval(2, j))
            call swap(eleval(3, i), eleval(3, j))

            call swap(eleid(i), eleid(j))
        endif
      endif
   end do
   mid = i + 1
   return
end
```

1. The vector *eleval* contains the centroid of all the elements.
2. The vector *eleid* contains the values of original indices of all the respective elements.
3. The variable *p* contains the starting index of the called mesh.
4. The variable *r* contains the ending index of the called mesh.

Table 1 Comparative speedup results between the ordered and un-ordered mesh of the 2D big beam using AMD Opteron with 1MB cache

PEs	Ordered mesh	Un-ordered mesh
2	1.99	1.71
3	2.96	2.16
4	3.92	2.43

5. The variable min contains the minimum value of all the element centroids for x, y and z-axis.
6. The variable max contains the maximum value of all the element centroids for x, y and z-axis.

Initially, all the elements are re-numbered according to the spatial information given in Fig. 2 and then the nodes are re-numbered anticlock wise. After the re-numbering, all the boundary conditions associated with the previous node ordering are adjusted to the newly assigned numbered nodes without loss of information. For this purpose previous node numbers are stored in an ID vector.

In order to study benefits of the proposed reordering technique, we have executed the mesh moving scheme on a large 2D mesh corresponding to the oscillating beam problem, with and without renumbering of the nodes/elements. We have also experimented with two different multicore platforms: AMD Opteron and Intel Xeon based machines. Table 1 shows the speedup enhancement of the mesh reordering algorithm with the processing elements (PEs). This experiment is conducted on the four cores of AMD Opteron machine containing 1MB of cache memory. Our results show that reordering indeed improves performance of the parallel implementation. On the quad core Intel Xeon based machine, speedups of 3.56 and 3.48 are achieved with and without the reordering algorithm, respectively. The Intel machine contains 12MB of cache memory. The speedup enhancement is less in the Intel Xeon machine as compared to the AMD Opteron machine. The difference of the speedups indicates that the mesh reordering algorithm might be more useful for processors containing small cache memory. Small cache memory leads to more cache miss rate. Hussain et al. [34] presented the 2D ALE moving mesh by applying the serial mesh reordering algorithm using OpenMP.

Remark 1 The mesh reordering algorithm (Algorithms 2–3) based on the divide and conquer approach would not terminate if all the elements have the same coordinates/nodal values. But in reality no two elements can occupy the same space. Therefore the algorithm is guaranteed to terminate. If hypothetically in such a case the algorithm is invoked, where all the elements have the same coordinates then extra termination condition should be added, based on the mid variable. Where $mid = \frac{(l+r)}{2}$ instead of $mid = i + 1$ in the Partition function.

Remark 2 If both the total spatial dimension and threshold value are equal to 1, then the algorithm behaves like quicksort. Similarly, for 2D and 3D cases, if the threshold value is equal to 1, it behaves like quicksort in 2D and 3D cases. The time complexity of the reordering algorithm is equal to $O(n\ lg(n))$, which is equal to an average case

of the quicksort algorithm [33]. Here, n is equal to the total number of elements in a mesh.

Remark 3 As a byproduct of the proposed reordering technique, the problem is partitioned into small areas or cubes using a top-down approach. These partitions can also be used to distribute the mesh in a distributed memory system.

4.2 Parallel Mesh Reordering

For the case when a mesh is already partitioned over multiple threads or processors, and moving mesh is an intermediate step, this section presents a new parallel mesh reordering algorithm based on the parallel bucket sort [35], and sampling approach [25] and [26]. Initially, Frazer and McKeller [25] proposed a new serial samplesort based on the sampling concept, which has been observed to be very useful for the parallel implementations due to its non overlapped computation (see [26] and references therein). In this approach a set of $s \times p$ elements from the original input data, called splitters (Sp), are determined. These splitters partition the n input elements into p groups of elements $sp_0 \ldots sp_{p-1}$. In particular, every element in the set sp_i is in the lower space as compared to every element in the set sp_{i+1}. The partitioned sets are then sorted independently to achieve the overall sorted sequence. In our case, the input is an un-ordered mesh consisting of n elements, assumed to be distributed evenly over p threads or processors such that each thread is assigned n/p elements. In the following we outline the proposed parallel mesh reordering algorithm.

1. Calculate the centroid of all the distributed elements in parallel. This step takes $O(n/p)$ time.
2. In parallel, all the threads locally reorder sub-meshes by incorporating the spatial reordering algorithm presented in Algorithm 2. This step also linearizes the multidimensional data into one dimension. Figure 3 explains the linearization part by assigning the index number to each leaf node of a quadtree/octree. These index numbers are stored in the Sp along with the spatial centroid values, which are latter used in the comparisons. Here, partition 1 contains the leaf nodes $[1 \ldots n/p]$, partition 2 contains the leaf nodes $[n/p + 1 \ldots 2n/p]$ and so on. In this way the partition vector is created in step 6. Expected time of this step is $O(n/p \lg (n/p))$.
3. Create the splitters set Sp of $s \times p$ elements in parallel from the locally ordered sub-meshes. Each thread selects evenly spaced s elements from its n/p associated centroids along with their respective indices. Here, s is a user defined constant, normally it is assigned the value n/p^2. Each element of the set Sp contains three

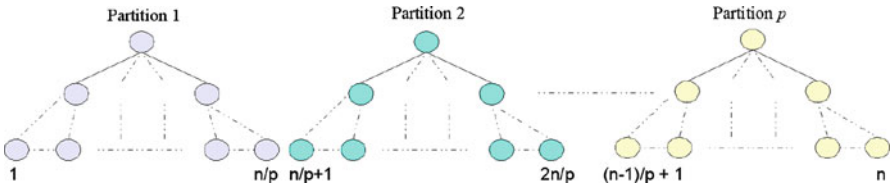

Fig. 3 Local reordering of a mesh and linearization of all *centroids*

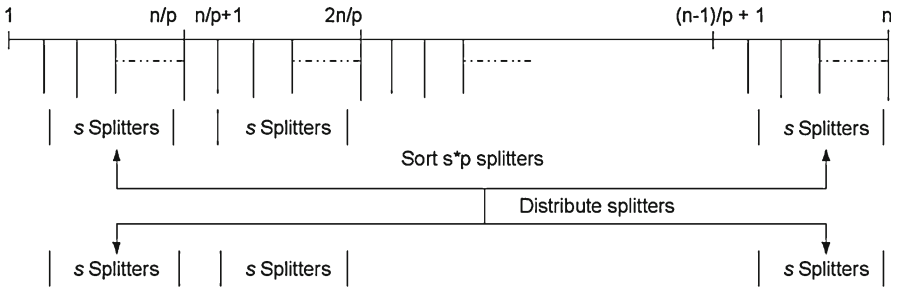

Fig. 4 Selection, reordering and distribution of *splitters*

values: Centroid, original index and partition value (the partition value is calculated in step 6 to reduce the number of exchange operations). Time to do this step is $O(n/p^2)$ in parallel.

4. Spatially reorder the set Sp by calling the serial reordering algorithm. The expected time of this step is $O(n/p \lg (n/p))$.
5. Distribute the ordered set Sp evenly over all the threads, such that ith thread gets s elements of a sub-splitter (sp_i) set from the Sp, where $sp_0 = Sp[1 \ldots s]$, $sp_1 = Sp[s+1 \ldots 2s]$, $\ldots$, $sp_{p-1} = Sp[((n-1)s)+1 \ldots n\,s]$. The creation, reordering and distribution of the splitters is graphically depicted in Fig. 4.
6. Calculate the partition vector using the original index value for all the sp_i in parallel, which gives the information about the partition index, defined in step 3. The time complexity of this step is $O(n/p^2)$.
7. Two splitters cannot be compared spatially, so the partition vector is utilized to compare the splitter within each thread, which is calculated in the previous step. The ith thread sorts its sp_i set in parallel based on the partition vector using a stabilized sorting algorithm e.g. counting sort. This step takes $O(n/p^2)$ time.
8. Select a set of $p-1$ evenly spaced elements from all the sp_i of the subset assigned to ith thread and store them into a $\bar{Sp}$: $\bar{Sp} = \{Sp[s], Sp[2s], \ldots, Sp[(p-1) \times s]\}$. The $\bar{Sp}$ is sent to the master thread or written down in the shared memory. This step takes $O(1)$ time.
9. Create the local partition (lpv_i) and index vectors (liv_i) for all the ith thread of size p, in parallel. These vectors are obtained by incorporating the maximum index value of ith partition of respective sp_i and $\bar{Sp}$. This step takes $O(p)$ time.
10. Thread i sorts the liv_i vector in ascending order by calling insertion sort, where lpv_i is used as the satellite data. This step is done in parallel. Insertion sort is used because most of the values of these vectors are already spatially ordered, thus less insertions are required during the sort operation. In the worst case, this step takes $O(p^2)$ time. Quicksort algorithm can be employed as an alternative to reduce the worst case scenario to $O(p\,lg(p))$ time. All these vectors are combined row wise to construct a global partition and index (GPI) matrix. The dimension of the GPI matrix is $p \times p$, where each row represents the current processor ID and each column represents the initial partition number. Figure 5 graphically elaborates the sorting of all the sp_i sets and the creation of GPI matrix using the sp_i sets.

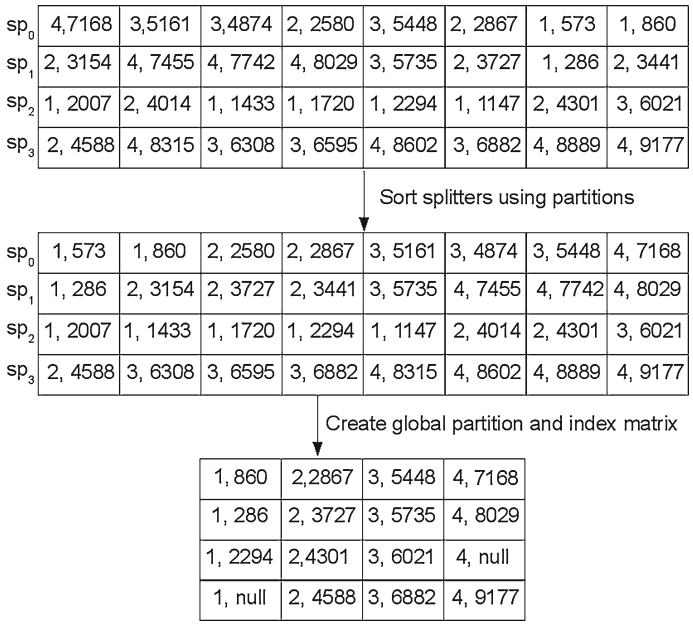

Fig. 5 Assume spatially ordered total eight splitters are selected for submarine mesh using four threads. *First step* shows the sorting of all sub-splitter sets (sp_i) and *next step* shows the creation of global partition and index matrix from sorted sub-splitters. ith row of this *matrix* represents the lpv_i and liv_i vectors. Both partition and index values are separated by ','

11. Calculate the new distribution of all the sub-meshes using the sorted GPI matrix in serial. This step also takes $O(p^2)$ time. Overall, step 6–11 are used to calculate the criteria to distribute the centroids over all the threads based on the Sp. It is worth mentioning that the splitter value cannot be compared spatially to the centroid value. The key point here is to utilize both the index and partition numbers that make possible to compare the splitter with all the centroids.
12. Copy data to the local memory of all the threads in parallel using the new distribution, which is calculated in the pervious step. This step takes $O(p^2) + O(n/p)$ time.
13. Reorder the newly distributed partition in parallel by calling the spatial tree based reordering algorithm. Expected time of this step is $O(n/p \lg(n/p))$.

The threshold value in Algorithm 2 is set to 1 for all the above stated steps. The total time complexity of the parallel mesh reordering T_p is defined as:

$$T_p = O\left(\frac{n}{p}\right) + O\left(p \lg(p)\right) + O\left(p^2\right) + O\left(\frac{n}{p} \lg\left(\frac{n}{p}\right)\right) \tag{15}$$

Since the complexity of the serial spatial mesh reordering algorithm is $O(n \lg(n))$, so the overall isoefficiency function [36] of the parallel spatial mesh reordering is $O(p^2 \lg(p)) + O(p^3)$. Thus, the problem size should be at least of the order of $\Omega(p^3)$, in order to maintain a fixed efficiency.

5 Description of Parallel ALE Moving Mesh

Initially, the parallel ALE moving mesh takes an un-deformed ordered mesh as an input and partitions the mesh element-wise over all the PEs. Next, the parallel algorithm calculates the element stiffness matrix and then solves the under study BVP in parallel using the PCG method.

Load balancing is an important issue in parallel algorithms. OpenMP preprocessor directives are used to partition the problem for all the PEs of a shared memory system. Most of the time, simple OpenMP preprocessor directives with the static scheduling are used to parallelize the code such as reduce functions and parallel do loops.

5.1 Communication

We note that intensive inter-processor communication is involved in solving the equation given in line 12 of Algorithm 1. Here, initially an element solution is computed for all the elements of the given mesh. Then, the computed solution is transformed to their nodal solution. This process requires communication of the nodal solution of all the shared nodes among the different threads. A similar procedure is required to compute P^{-1} (inverse of P) in line 6 of Algorithm 1. The rest of the equations are solved using parallel loops over all the nodes or all the elements independently. The result is communicated using the reduce function of OpenMP library.

The communication among different threads is achieved using an extra temporary bin vector with the memory size of $O(n_{sn} * n_{pe})$, where n_{pe} and n_{sn} are the total number of threads and shared nodes of the given mesh, respectively. In this concept, a process or thread has its own local list of bins against each shared node, where it stores the particular nodal result. Here, n_{sn} represents the total number of bins. All the threads complete their tasks and store their result in their own local bins, which are later combined in parallel using a reduction operation. Finally, the result is stored in a resultant data structure for further processing. Figure 6 shows a scenario in which two threads generats a new mesh and communicate the information related to the shared nodes. Main reason to use the said concept is that OpenMP does not perform well in the tree data structures instead of the array.

Algorithm 4 depicts the segment of the Fortran code that combines the results stored in all the bins by calling the simple addition function. It also stores the final result in the resultant vector (q), where n and i are the indices variables. This method is used to increase the speedup, without using a synchronization directives like *omp lock* available in OpenMP library.

5.2 Time Complexity Analysis

The total serial computation time of the ALE moving mesh (T^s_{ale}) includes the time to calculate the element stiffness matrix (T^s_{esm}) and the PCG solver (T^s_{pcg}). Assuming n to be the total number of elements in the mesh, the total serial computation time is:

$$T^s_{ale} = T^s_{esm} + T^s_{pcg} \tag{16}$$

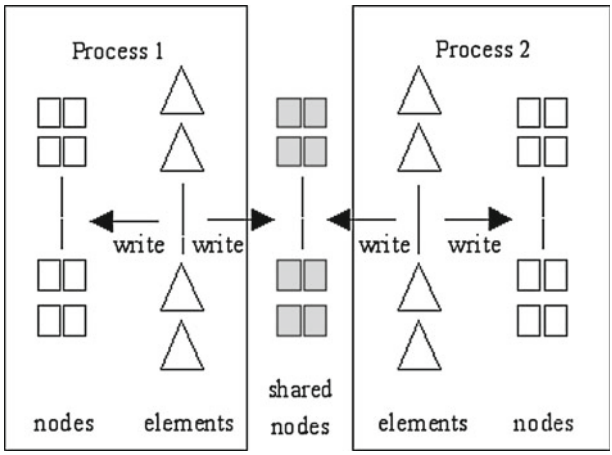

Fig. 6 Communication of shared nodes

Algorithm 4 Parallel do loop for the synchronization using the bins' concept.

```
!$omp do
   do n = 1, TotalSharedNodes
         do i = 1, TotalProcessors
               q(n) = q(n) + bin(n, i)
         enddo
   enddo
```

where

$$T^s_{esm} = O(n^2) \tag{17}$$

$$T^s_{pcg} = O(k\, n^2) \tag{18}$$

Here, k is the total number of iterations in the PCG solver. Hence, the total serial computation time is:

$$T^s_{ale} = O(n^2) + O(k\, n^2) \tag{19}$$

The total parallel computational time of the ALE moving mesh (T^p_{ale}) is:

$$T^p_{ale} = T^p_{esm} + T^p_{pcg} + T^p_{oh} \tag{20}$$

where T^p_{esm} and T^p_{pcg} are the parallel time of the calculation of the element stiffness matrix and PCG solver, respectively, and T^p_{oh} is the parallel overhead time including the extra process and communication:

$$T^p_{oh} = O(n) + O(n_{sn}) \tag{21}$$

and

$$T_{esm}^{p} = O\left(\frac{n^2}{p}\right) \tag{22}$$

$$T_{pcg}^{p} = O\left(\frac{k\,n^2}{p}\right) \tag{23}$$

Thus, total time for the parallel ALE moving mesh is:

$$T_{ale}^{p} = O\left(\frac{n^2}{p}\right) + O\left(\frac{k\,n^2}{p}\right) + O(n) + O\left(n_{sn}\right) \tag{24}$$

Where n_{sn} is the total number of shared nodes and k is the total number of iterations used in the PCG solver. The isoefficiency function [36] of the parallel ALE moving mesh is $O(p^2)$. Thus, the problem size should be increased with the number of processors with the order of $\Omega(p^2)$, in order to maintain a fixed efficiency.

6 Performance Evaluation

Numerical experiments are conducted on AMD as well as Intel multicore processor based machines. The AMD machine contains two Dual Core AMD Opteron Processors (285) 2.6 GHz with 667 MHz FSB and 1MB L2 cache memory. The Intel machine contains two Quad Core Intel Xeon Processors (E5405) 2.0 GHz with 1,333 MHz front side bus (FSB) and 12MB L2 cache memory.

The mesh database used in our experiments is comprised of three 2D meshes and three 3D meshes for the different problems and sizes (see Table 2 for details). The mesh generation code is written in Fortran programming language and OpenMP preprocessor directives. GNU Fortran compiler (GCC) 4.2.3 (Gentoo 4.2.3 p1.0) is used for the compilation of the code. The code was compiled without using an optimization flag to get an unbiased speedup on a shared memory system. Optimization flags utilize the vector processors available in AMD and Intel processors, e.g. sse2, ssse3 and 3dnow. The mesh visualization code is written in C++ programming language using an OpenGL library.

Table 2 Mesh database for the ALE moving mesh

Dimensions	Mesh name	No. of elements	No. of nodes
2D	Big beam	230,806	116,518
	Small beam	9,509	5,024
	Small submarine	9,177	4,683
3D	Sphere	30,930	5,861
	Flexible cylinder	43,648	9,132
	Rigid cylinder	43,648	9,132

Unstructured triangular and tetrahedral elements are employed for both the 2D and 3D problems, respectively. Four noded bi-linear quadrilateral shape functions are used for the triangular element, where 3rd and 4th nodes coalesced to create a triangle:

$$N_1^{(e)} = \frac{1}{4}(1-\xi)(1-\eta) \tag{25}$$

$$N_2^{(e)} = \frac{1}{4}(1+\xi)(1-\eta) \tag{26}$$

$$N_3^{(e)} = \frac{1}{4}(1+\xi)(1+\eta) \tag{27}$$

$$N_4^{(e)} = \frac{1}{4}(1-\xi)(1+\eta) \tag{28}$$

The linear shape functions are used for four noded tetrahedral element and defined as:

$$N_1^{(e)} = \xi \tag{29}$$

$$N_2^{(e)} = \eta \tag{30}$$

$$N_3^{(e)} = \zeta \tag{31}$$

$$N_4^{(e)} = (1-\xi-\eta-\zeta) \tag{32}$$

where $N_i^{(e)}$ represents the shape function of ith node of the element e and ξ, η and ζ are used to map the values from the physical domain $\Omega^e(x, y, z)$ to the reference domain $\Omega^e(\xi, \eta, \zeta)$. Numerical integration is done using Gauss quadrature formula to calculate the element stiffness matrix using the shape functions. The Gauss quadrature formulas for the reference elements in 2D and 3D are defined in Eqs. (33) and (34), respectively:

$$\int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta)d\xi d\eta = \sum_{Q_1=1}^{n_{Q_1}}\sum_{Q_2=1}^{n_{Q_2}} w_{Q_1} w_{Q_2} f(\xi_{Q_1}, \eta_{Q_2}) \tag{33}$$

$$\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta,\zeta)d\xi d\eta d\zeta = \sum_{Q_1=1}^{n_{Q_1}}\sum_{Q_2=1}^{n_{Q_2}}\sum_{Q_3=1}^{n_{Q_3}} w_{Q_1} w_{Q_2} w_{Q_3} f(\xi_{Q_1}, \eta_{Q_2}, \zeta_{Q_3}) \tag{34}$$

where the values of the weights w_{Q_i} and the coordinates ξ_{Q_1}, η_{Q_2} and ζ_{Q_3} of the quadrature points n_{Q_i} are taken from [37]. Two point Gauss quadrature formula is used for the numerical integration of the triangular element to obtain *full quadrature*, and is also known as 2×2 integration rule [9]. One point Gauss quadrature formula is used for the numerical integration of the tetrahedral element, because its derivative is a constant [10].

7 Experimental Results and Discussion

7.1 Mesh Reordering

A total of seven different 2D and 3D meshes are used to evaluate the speedup and execution time of the parallel spatial reordering algorithm based on the sampling and quicksort algorithms. Two additional meshes of 2D biunit square are included in addition to the meshes used in Table 2. The dimensions of the biunit square meshes are specified in Table 3.

The numerical experiments are repeated 10 times to get an average behavior of the speedup. Table 4 shows the maximum speedup factor and minimum time in seconds for the parallel reordering algorithm executed on different meshes using the eight cores of the Intel Xeon machine. The maximum speedup is obtained for the 2D mesh problems. This is due the fact that less communication and exchanges of data occur for the 2D case, compared to the 3D case. On an eight core machine, the parallel mesh reordering yielded 51% more efficiency in terms of speedup compared to the serial mesh reordering, where

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Total processors}} \times 100 \tag{35}$$

7.2 2D ALE Moving Mesh

As a case study for the 2D problems, *oscillating beam* and *submarine maneuvering* are used in our experiments to study the performance of the proposed parallel ALE moving mesh algorithm. The oscillating beam is a classical example of FSI problems. It covers a wide range of applications in the field of civil and mechanical engineerings

Table 3 Mesh database used for the parallel spatial reordering algorithm

Dimensions	Mesh name	No. of elements	No. of nodes
2D	Small biunit square	22,910	11,656
	Big biunit square	92,006	46,404

Table 4 Mesh results in terms of the speedup and average time in seconds using eight PEs

Dimensions	Mesh name	Speedup	Time in seconds
2D	Big beam	4.53	4.93E-02
	Small beam	4.71	1.70E-03
	Small submarine	4.61	1.80E-03
	Small biunit square	3.96	4.80E-03
	Big biunit square	3.96	2.01E-02
3D	Sphere	3.39	4.80E-03
	Multiple cylinders	3.45	1.19E-02

Table 5 The speedup for 2D problems on a four core AMD machine

PEs	Big beam	Small beam	Small submarine
2	1.99	1.87	1.87
3	2.97	2.66	2.71
4	3.95	3.44	3.4

Table 6 The speedup for 2D problems on a eight core Intel machine

PEs	Big beam	Small beam	Small submarine
2	1.86	1.82	1.83
3	2.81	2.59	2.64
4	3.57	3.29	3.24
5	4.37	3.86	3.95
6	5.12	4.42	4.5
7	5.76	4.79	4.85
8	6.37	5.23	5.22

and has major utilization in the microelectromechanical systems (MEMS). Similarly, the submarine maneuvering is an important case study for flows around propeller of submarines and surface ships that constitute their propulsion mechanisms. A detailed discussion about these problems is given in [3] and [9].

Table 5 shows the speedup results of the 2D problems using two dual core AMD Opteron processors. A maximum efficiency of 98.64% in terms of speedup is obtained for the larger mesh (corresponding to the oscillating beam), which contains 230,806 number of elements and 116518 number of nodes. Whereas, 84.89 and 85.9% efficiencies are recorded for the maneuvering of the small submarine and oscillation of the small beam problems, respectively.

These efficiencies are reduced due to less computational data in the small meshes as compared to the big mesh, as the total number of equations are a multiple of the total number of elements. A similar behavior is also observed in the Intel Xeon processor, see Table 6 for details. In addition, better speedup is produced on the AMD Opteron as compared to the Intel Xeon whereas the Intel Xeon has larger L2 cache memory as compared to the AMD Opteron. The difference in the speedup might be due to the difference in their architectures.

The zoomed views for maneuvering of the submarine and oscillation of the beam problems are shown in Figs. 7 and 8, respectively.

7.3 3D ALE Moving Mesh

As a case study for the 3D problems, meshes corresponding to flexible and rigid deformation of multiple cylinders and oscillation of sphere are used in our experiments. These selected problems have great utilization in heat transfer applications [10].

Tables 7 and 8 show the speedup results of 3D problems using four AMD Opteron processors and eight Intel Xeon processors, respectively. The maximum efficiency of

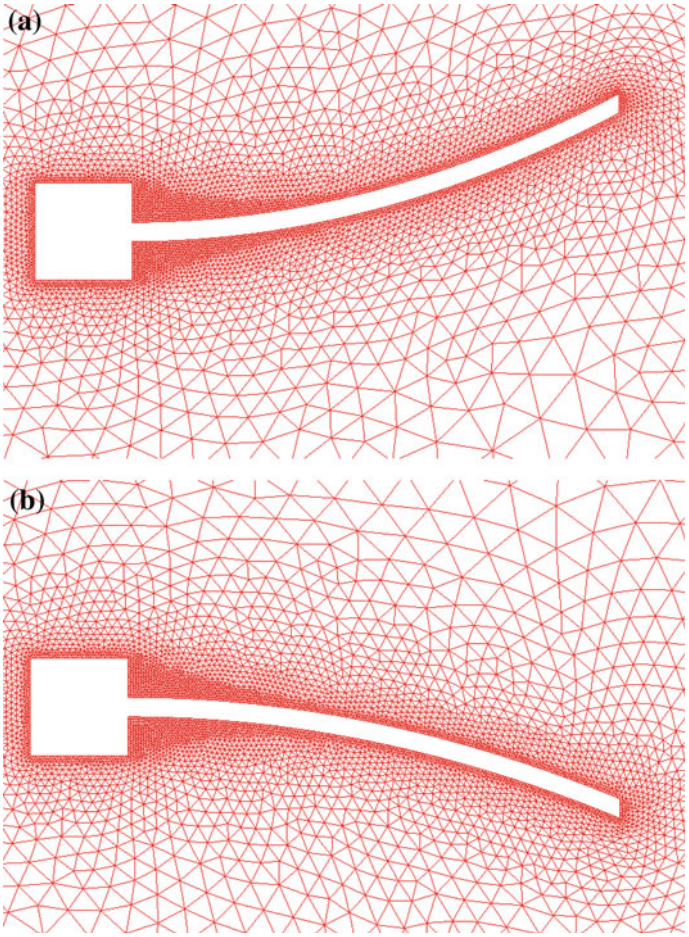

Fig. 7 Zoomed view of the deformed meshes for the 2D oscillating beam at two different instants. **a** Beam is moved *up*, **b** Beam is moved *down*

93.55% in terms of speedup is obtained for the flexible multiple cylinders using four AMD Opteron processors. Whereas, 92.87 and 92.88% efficiencies are recorded for both the oscillating sphere and rigid multiple cylinders, respectively. The maximum of 89.82, 88.03 and 92.19% efficiencies are obtained for the flexible cylinders, rigid cylinders and oscillating sphere problems, respectively, using four PEs of the Intel machine. On the average 74% efficiency is recorded for eight Intel processors. Overall, the AMD Opteron produces slightly better speedup as compared to the Intel Xeon using four PEs, in conformance to the result of 2D problems.

The 3D problems produced better speedup as compared to the 2D problems, since computation to communication (CC) ratio increases in the higher dimension space. For instance, $2^2 \times 4 = 16$ equations need to be solved for one iteration of one element inside the loop of Gauss quadrature points for the 2D problems, and $3^2 \times 4 = 36$ equations need to be solved for the 3D problems. At the most, results of four nodes may be

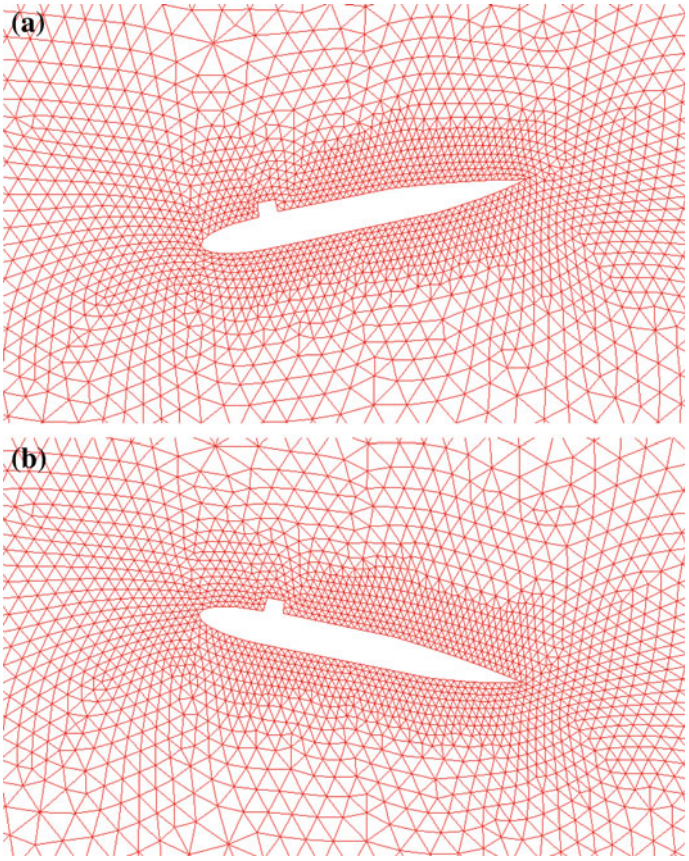

Fig. 8 Zoomed view of the deformed meshes for the 2D submarine at two different instants. **a** Submarine is moved *up*, **b** Submarine is moved *down*

Table 7 The speedup for 3D problems on a four core AMD machine

PEs	Flexible cylinder	Rigid cylinder	Sphere
2	1.97	1.93	1.96
3	2.88	2.9	2.89
4	3.72	3.74	3.71

communicated among different PEs for the triangular and tetrahedral elements. In the worst case scenario, CC ratios turn out to be 4:1 and 9:1 for the 2D and 3D problems, respectively. In general CC ratio will be $2^{n_{sd}} : n_{ne}$, where n_{ne} denotes the total number of nodes per element.

Table 2 depicts that the ratio of the number of elements to the number of nodes are approximately 2:1 and 4:1 in the 2D and 3D problems, respectively. This shows that less information needs to be communicated in the 3D problems. This causes a decrease in the CC ratio for the 3D problems as compared to the 2D problems, especially in the PCG solver.

Table 8 The speedup for 3D problems on a eight core Intel machine

PEs	Flexible cylinder	Rigid cylinder	Sphere
2	1.90	1.87	1.91
3	2.81	2.75	2.87
4	3.59	3.52	3.69
5	4.34	4.3	4.48
6	5.02	4.93	5.3
7	5.73	5.66	5.99
8	5.93	5.83	6.07

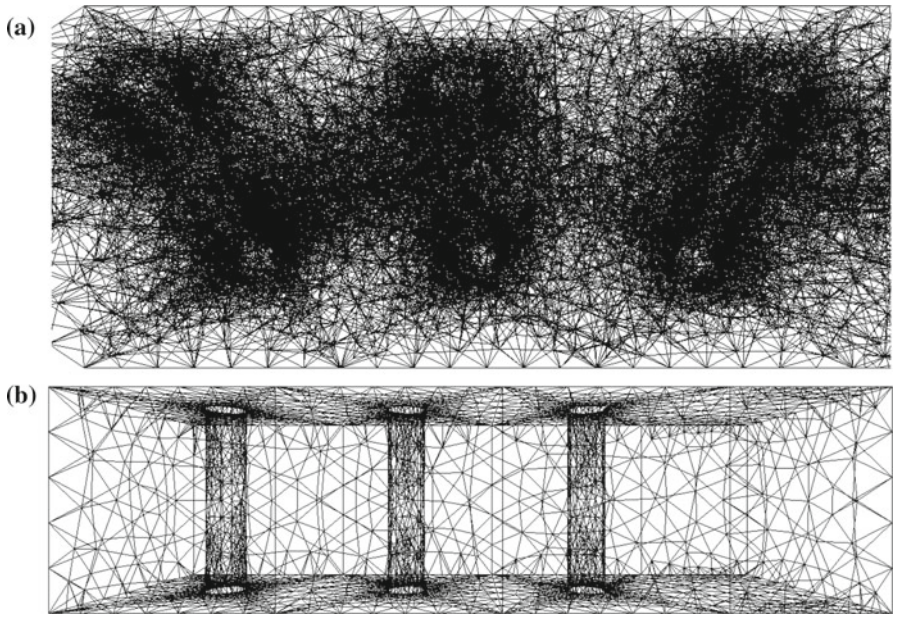

Fig. 9 The 3D multiple cylinders with the different views. **a** Element view, **b** Surface view

To visualize the working of the developed parallel code, the results of flexible deformation of the multiple cylinders and oscillation of the sphere problems are shown in Figs. 9 and 10, respectively.

7.4 The Improvement in Results using Reordering

In order to see the effectiveness of the developed reordering algorithm for the smaller cache memory processor, the biased speedup is utilized and defined as follows:

$$\text{Biased speedup} = \frac{\text{Serial time without reordering}}{\text{Parallel time}} \tag{36}$$

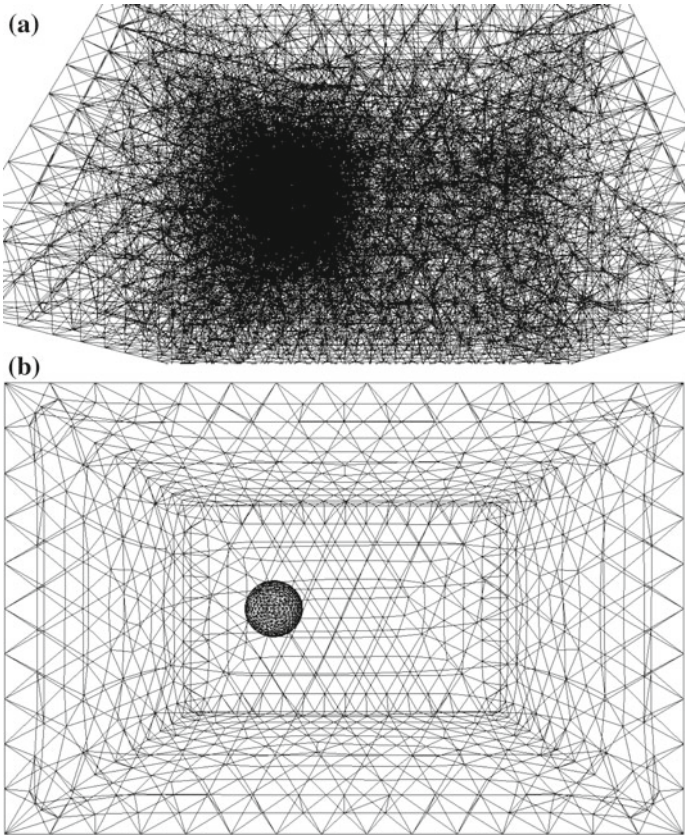

Fig. 10 The 3D oscillating sphere with the different views. **a** Element view, **b** Surface view

Table 9 Increase in the biased speedup with four cores of AMD machine for 3D problems

PEs	Flexible cylinder	Rigid cylinder	Sphere
1	1.02	1.04	1.02
2	2	2.05	1.98
3	2.95	2.99	2.97
4	3.78	3.85	3.83

Tables 7 and 9 explain the difference of calculated speedup of the 3D problems on the AMD Opteron machine by dividing the sequential time of the ordered and unordered meshes, respectively. It clearly shows that reordering produces better results for the serial computation, as shown in the first row of Table 9. It also shows that the serial time of the ALE moving mesh is decreased due to the reordering of mesh, which decreases the cache miss rate and bandwidth of the matrix. The result depicts that the reordering overhead is less dominant over the whole process of the ALE moving mesh generation due to decrease in the cache miss rate and bandwidth of the matrix.

Table 10 The average bandwidth values of different meshes with the standard deviations

Problem	Reordering	Average	SD
Small beam	Yes	70.54	244.74
	No	372.74	598.29
Big beam	Yes	350.24	2, 705.10
	No	1, 654.46	7, 587.01
Submarine	Yes	90.93	290.97
	No	204.82	639.02
Oscillating sphere	Yes	383.12	714.70
	No	2, 087.52	1, 304.15
Multiple cylinders	Yes	425.58	828.64
	No	3, 385.55	2, 186.28

Table 11 The average bandwidth values of different meshes for eight PEs

PEs	Small beam		Big beam		Submarine		Sphere		Cylinders	
	O	UO	O	UO	O	UO	O	UO	O	UO
1	72.2	361.8	339.8	1, 637.4	90.2	206.1	367.7	2, 080.4	430.6	3, 345.5
2	69.4	381.4	348.1	1, 671.7	98.2	209.6	385.3	2, 067.4	422.4	3, 406.3
3	77.9	363.4	334.2	1, 647.7	91.8	213.6	385.4	2, 065.8	426.4	3, 326.5
4	68.1	384.1	374.9	1, 664.0	97.2	170.3	377.2	2, 082.6	422.7	3, 365.9
5	76.1	365.6	349.6	1, 642.6	88.9	213.6	381.8	2, 082.6	425.7	3, 454.6
6	61.9	383.5	357.6	1, 660.1	88.6	197.5	378.1	2, 072.4	425.4	3, 423.0
7	65.5	355.8	343.4	1, 644.1	92.2	207.1	378.3	2, 067.4	425.9	3, 384.8
8	73.2	386.3	354.3	1, 668.1	80.3	220.8	371.4	2, 082.1	425.4	3, 377.8

O ordered mesh, *UO* un-ordered mesh

An average bandwidth $B(M)$ is utilized for a non-symmetric matrix M (for unstructured meshes), which is equal to the average value of all $B(M_i)$, where the M_i is ith row of a matrix. The $B(M_i)$ (bandwidth of ith row of the matrix) is the maximum difference between two non-zero values. It is formally defined as $|l - m|$ such that $M_{i\ k_1} = 0$ and $M_{i\ k_2} = 0, \forall\ k_1, k_2$, where $l \leq m, k_1 < l$ and $k_2 > m$. The bandwidth of a diagonal matrix is zero.

Table 10 shows the average bandwidth of the different meshes along with their respected standard deviation values. It clearly shows that the bandwidth of all these meshes is decreased due to the proposed reordering algorithm. In summary, the maximum of 7.96 times average bandwidth is decreased for the 3D multiple cylinders problem and the minimum of 2.25 times average bandwidth is decreased for the 2D submarine problem. Table 11 shows the average bandwidth of distributed meshes of the different problems over eight PEs. It shows that the bandwidth of each PE's matrix for the ordered mesh is also less than the bandwidth of the un-ordered mesh.

Metis is a well known library that is employed to order and partition a mesh over different PEs. This library is based on a k-way graph partitioning algorithm using a

Table 12 The average bandwidth values of different meshes for eight PEs using Metis library

PEs	Small beam	Big beam	Submarine	Sphere	Cylinders
1	86.60	926.92	32.23	368.85	407.96
2	123.17	1, 065.61	58.06	333.07	534.23
3	107.27	1, 106.77	92.74	398.86	662.56
4	93.15	1, 377.44	145.74	587.72	451.94
5	98.19	1, 548.31	68.18	502.98	1, 006.89
6	266.03	1, 824.93	144.16	754.34	1230.93
7	207.77	1, 153.89	120.55	1, 004.98	968.79
8	176.67	1, 399.07	258.14	891.71	1, 175.34

minimum cut set [22] and [23]. The resultant partitions are mainly used on a distributed memory system. Table 12 shows the average bandwidth values of the different meshes using Metis library for eight PEs. It indicates that for these meshes on a shared memory system, the tree based reordering algorithm is more effective than the k-way graph partitioning algorithm using Metis library.

7.5 Reliability of the Parallel ALE Moving Mesh Algorithm

The mesh generated by the parallel ALE moving mesh scheme should be similar to the mesh generated by the serial algorithm to ensure its reliability. This is concluded from the measured average absolute error of 10^{-6} by taking the difference of the nodal values of both the serial and parallel algorithms.

The reliability is also ensured by checking both the element Jacobian and internal angles of all the elements, as an element Jacobian is only a scalar function of area (in 2D) or volume (in 3D) of an element and therefore does not account for better element distortion. Therefore, an angle information is also used for the reliability of a mesh using law of cosines and the following threshold variable:

$$T_{\min} = A_{\min} - \frac{1}{2}\sigma \tag{37}$$

$$T_{\max} = A_{\max} + \frac{1}{2}\sigma \tag{38}$$

where $T_{\min}$ and $T_{\max}$ are the threshold minimum and maximum angles, $A_{\min}$ and $A_{\max}$ are the actual minimum and maximum angles and σ is the standard deviation of all angles in the given mesh. If the internal angle of any element is less than $T_{\min}$ or greater than $T_{\max}$, a new mesh is generated for the current time step in a transient analysis.

Figures 11 and 12 show the quality of meshes for the 2D oscillating beam at tip and the 2D submarine, respectively using the serial and parallel ALE moving mesh generation algorithms under the maximum deformation. The quality of the mesh around

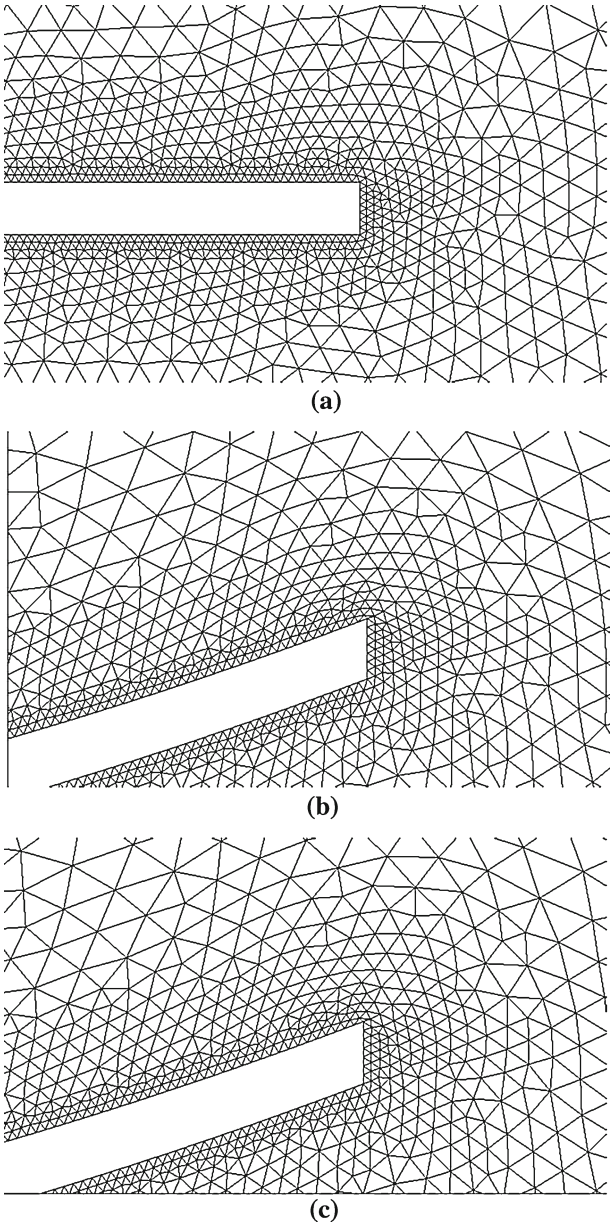

Fig. 11 Comparison of the ALE moving mesh for the 2D oscillating beam. **a** Undeformed mesh, **b** Mesh generated in serial, **c** Mesh generated in parallel using four PEs

the submarine and tip of the oscillating beam is also comparable to the quality of these regions in the initial un-deformed mesh. These un-deformed meshes are passed as an input to the parallel ALE moving mesh algorithm.

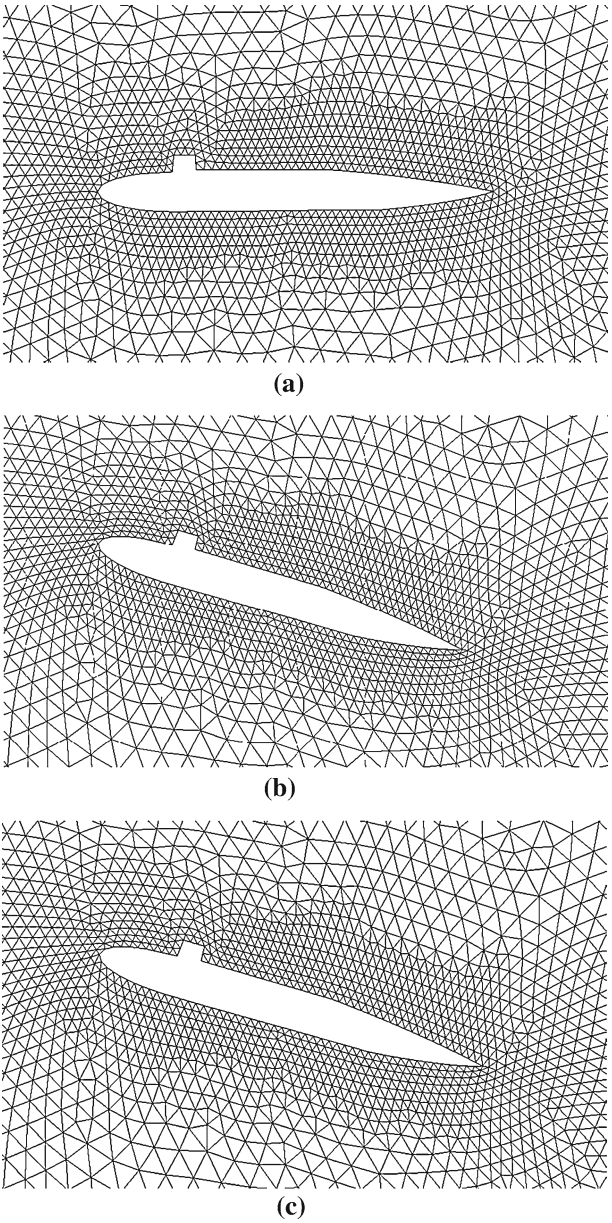

Fig. 12 Comparison of the ALE moving mesh for the 2D submarine. **a** Undeformed mesh, **b** Mesh generated in serial, **c** Mesh generated in parallel using four PEs

8 Conclusions

We have presented a parallel adaptive mesh motion and mesh generation algorithm for solving 2D and 3D FSI problems using OpenMP on shared memory systems. The proposed algorithm incorporates a novel tree based mesh reordering using

sampling approach, combined with recursive bisections and quicksort methods. The mesh reordering algorithm improved the cache memory performance due to the geometric memory locality of both the element and nodes via their global re-numbering. The result of the reordering algorithm can also be incorporated to partition a mesh on a distributed memory system. In addition, this algorithm can be applied to higher order elements for both 2D and 3D problems. Overall, the parallel mesh reordering produced 51% average efficiency as compared to the serial mesh reordering using eight PEs for all the meshes employed herein.

A new mesh is generated by employing the un-deformed ordered mesh using the parallel ALE moving mesh technique. The numerical results in terms of speedup showed that the AMD Opteron based multicore machine produced better results for the parallelism as compared to the Intel Xeon for all the mesh formulations studied. Both the element Jacobian and angles of triangle and tetrahedral elements are used to verify the correctness of the parallel ALE moving mesh code. Overall, 72% average efficiency of the parallel ALE moving mesh is achieved for both the 2D and 3D problems using eight PEs. On the average, absolute error of 10^{-6} is measured using the difference of the nodal values of both the serial and parallel algorithms for the reliability.

Acknowledgments Support for this work was provided by the collaborative Pak-US project under HEC Pakistan and NSF USA (2006–2009). Support of KICS at UET, Lahore for the experimental work is acknowledged. Support of the colleagues at FCSE, GIKI is also acknowledged in completing this work.

References

1. Li, R., Tang, T., Zhang, P.: A moving mesh finite element algorithm for singular problems in two and three space dimensions. J. Comput. Phys. **177**, 365–393 (2002)
2. Tang, T.:, Moving mesh methods for computational fluid dynamics. In: Shi, Z.C., Chen, Z., Tang, T., Yu, D. (eds.) Recent Advances in Adaptive Computations (Providence USA), Contemporary Mathematics, vol. 383, pp. 141–174. American Mathematical Society, Zhejiang University, Hangzhou, China (2005)
3. Bhanabhagwanwala, M.: A Mesh Pre-Processing Scheme for Moving Boundary Flows and a Novel Graphics Post-Processor for DGM, MS. Thesis, Mechanical Engineeering, University of Illinois at Chicago (2004)
4. Perot, J.B., Nallapati, R.: A moving unstructured staggered mesh method for the simulation of incompressible free-surface flows. J. Comput. Phys. **184**(1), 192–214 (2003)
5. Tourigny, Y., Hülsemann, F.: A new moving mesh algorithm for the finite element solution of variational problems. SIAM J. Numer. Anal. **35**, 1416–1438 (1998)
6. Donea, J.: Arbitrary Lagrangian-Eulerian finite element methods. In: Belytschko, T., Hughes, T.J.R. (eds.) Computational Methods for Transient Analysis, pp. 473–516. North-Holland, Amsterdam (1983)
7. Hughes, T.J.R., Liu, W.K., Zimmerman, T.K.: Lagrangian-Eulerian finite element formulation for incompressible viscous flows. Comput. Methods Appl. Mech. Eng. **29**, 329–349 (1984)
8. Masud, A., Hughes, T.J.R.: A space-time Galerkin/least-squares finite element formulation of the Navier-Stokes equations for moving domain problems. Comput. Methods Appl. Mech. Eng. **146**, 91–126 (1997)
9. Masud, A., Bhanabhagwanwala, M., Khurram, R.A.: An adaptive mesh rezoning scheme for moving boundary flows and fluid-structure interaction. Comput. Fluids **36**, 77–91 (2005)
10. Kanchi, H., Masud, A.: A 3D adaptive mesh moving scheme. Int. J. Numer. Methods Fluids **54**(4), 21–34 (2007)
11. Johnson, A.A., Tezduyar, T.E.: Mesh update strategies in parallel finite element computations of flow problems with moving boundaries and interfaces. Comput. Methods Appl. Mech. Eng. **119**(1–2), 73–94 (1994)

12. Johnson, A.A., Tezduyar, T.E.: Advanced mesh generation and update methods for 3D flow simulations. Comput. Mech. **23**, 130–143 (1999)
13. Farhat, C., Geuzaine, P.: Design and analysis of robust ALE time-integrators for the solution of unsteady flow problems on moving grids. Comput. Methods Appl. Mech. Eng. **193**(39–41), 4073–4095 (2004)
14. Masud, A.: Effects of mesh motion on the stability and convergence of ALE based formulations for moving boundary flows. Comput. Mech. **38**(4), 430–439 (2006)
15. Khurram, R.A, Masud, A.: A multiscale/stabilized formulation of the incompressible Navier-Stokes equations for moving boundary flows and fluid-structure interaction. Comput. Mech. **38**(4), 403–416 (2006)
16. Calderer, R., Masud, A.: A multiscale stabilized ALE formulation for incompressible flows with moving boundaries. Comput. Mech. **46**, 185–197 (2010)
17. Flaherty, J.E., Loy, R.M., Shephard, M.S., Szymanski, B.K., Teresco, J.D., Ziantz, L.H.: Adaptive local refinement with Octree load-balancing for the parallel solution of three-dimensional conservation laws. J. Parallel Distrib. Comput. **47**, 139–152 (1998)
18. Mitchell, W.F.: A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. J. Parallel Distrib. Comput. **67**(4), 417–429 (2007)
19. Balman, M.: Tetrahedral mesh refinement in distributed environments. In: Proceedings Supercomputing'93, pp. 497–504. IEEE, Aug. (2006)
20. Walshaw, C., Cross, M.: Mesh partitioning: a multilevel balancing and refinement algorithm. SIAM J. Sci. Comput. **22**(1), 63–80 (2000)
21. Karypis, G.: Multi-constraint mesh partitioning for contact/impact computations. In: Proceedings Supercomputing'03, pp. 497–504. IEEE Computer Society, Washington, DC, USA (2003)
22. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. In: Proceedings ICPP, pp. 113–122 (1995)
23. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. J. Parallel Distrib. Comput. **48**, 96–129 (1998)
24. Gupta, A.: Fast and effective algorithms for graph partitioning and sparse-matrix ordering. IBM J. Res. Develop. **41**, 171–183 (1996)
25. Frazer, W.D., McKellar, A.C.: Samplesort: a sampling approach to minimal storage tree sorting. J. ACM **17**, 496–507 (1970)
26. Chen, J.C.: Efficient sample sort and the average case analysis of PEsort. Theoret. Comput. Sci. **369**, 44–66 (2006)
27. The OpenMP API specification for parallel programming. www.openmp.org
28. Diaz, J., Petit, J., Serna, M.: A Survey of graph layout problems. ACM Comput. Surveys **34**(3), 313–356 (2002)
29. Lai, Y.L., Williams, K.: On bandwidth for the tensor product of paths and cycles. Discrete Appl. Math. **73**(2), 133–141 (1997)
30. George, P.L.: Automatic Mesh Generation: Application to Finite Element Methods. Wiley, New York (1991)
31. Sadeghi, S., Mashadi, M.M.: A new semi-automatic method for node numbering in a finite element mesh. Comput. Struct. **58**, 183–187 (1996)
32. Hughes, T.J.R., Ferencz, R.M., Hallquist, J.O.: Large-scale vectorized implicit calculations in solid mechanics on a Cray X-MP/48 utilizing EBE preconditioned conjugate gradients. Comput. Methods Appl. Mech. Eng. **61**(2), 215–248 (1987)
33. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. pp. 145–164. 2nd edn. Printice-Hall of India, New Delhi (2003)
34. Hussain, M., Ahmad, M., Abid, M., Khokhar, A.: Implementation of 2D parallel ALE mesh generation technique in FSI problems using OpenMP. In: Proceedings FIT'09. Abbottabad, Pakistan, no. 18, December (2009)
35. Wilkinson, B., Allen, M.: Parallel Programming—Techniques and Applications and Parallel Computers. pp. 145–164. 2nd edn. Printice-Hall of India, New Delhi (2003)
36. Grama, A.Y., Gupta, A., Kumar, V.: Isoefficiency: measuring the scalability of parallel algorithms and architectures. IEEE Parallel Distrib. Technol **1**(3), 12–21 (1993)
37. Belytschko, T., Liu, W.K., Moran, B.: Nonlinear Finite Elements for Continua and Structures. Wiley, New York (2006)