Parallel Implementation and Optimization of Hybrid-Grid FEM-based Ground Motion Simulation Tool in Large Multicore Cluster Computer

Pher Errol B. Quinay¹, Tsuyoshi Ichimura², Muneo Hori³

¹Member of JSCE, Specially-appointed Assistant Professor, Research Institute for Natural Hazards and Disaster Recovery, Niigata University (8050, Ikarashi 2-no-cho, Nishi-ku, Niigata 950-2181)
E-mail: pequinay@eng.niigata-u.ac.jp
²Member of JSCE, Associate Professor, Earthquake Research Institute, The University of Tokyo (1-1-1 Yayoi, Bunkyo-ku, Tokyo 113-0032)
E-mail: ichimura@eri.u-tokyo.ac.jp
³Member of JSCE, Professor, Earthquake Research Institute, The University of Tokyo (1-1-1 Yayoi, Bunkyo-ku, Tokyo 113-0032)
E-mail: hori@eri.u-tokyo.ac.jp

This paper presents the implementation and optimization of hybrid parallel programming model to an FEM-based simulation tool for earthquake wave propagation. A prepartitioning method is implemented to avoid hardware memory limitation problem in mesh generation of billion order DOF model. Load imbalance is addressed by developing a load balancing procedure based on two-dimensional irregular grid, and by managing the workload of threads in the implementation of hybrid MPI-OpenMP. For a simple (flat surface) model with 1.82 billion DOF, the improvement in parallel efficiency reached 27%, compared with the simple hybrid implementation. For a realistic crust model with about 5.8 billion DOF, the improved code resulted to speed up of up to 6.8 in using 8 cores per compute node, when 2,304 nodes of K computer were used.

Key Words: hybrid parallel programming model, irregular grid domain decomposition, thread workload, hybrid-grid FEM

1. INTRODUCTION

Numerical simulation of all physical processes in earthquake wave propagation contributes in advancing the prediction of the consequence of these processes. Because of the increasing interests to observe each process at finer spatial and temporal resolution, numerical simulation of this problem is pushing the limits of computational resources. For an M8 class earthquake, the length of fault rupture is in the order of 10⁵ meters (m), while earthquake engineering applications require the ground motion distribution in the spatial resolution of the order of 10⁰ ~ 10¹ m. The duration of a whole event of such an earthquake is in the order of 10² seconds (s), and the temporal resolution required by earthquake engineering applications is 10⁻² s at largest. Thus, the expected number of DOF in the numerical simulation easily reaches the order of billions, together with the number of time steps being in the order of ten thousands.

Because of this huge computation cost, a high performance computer is used for the numerical simulation of the earthquake wave propagation. At present, such a computer is designed as a distributed-memory system which is composed of either the combination of multicore processors or the general purpose graphics processing units. A number of past studies in this field used these architectures for developing efficient procedures, which include preprocessing, analysis and postprocessing of a billion order DOF problem.

Achieving efficient use of large computational resources highly depends on optimizations in memory usage and computation time. This study focuses on implementing and optimizing a hybrid parallel programming model to a Finite Element Method (FEM)-based earthquake wave propagation simulator, so that a large distributed-memory computing system with multicore processors can be used. Without optimization, a less tuned end-to-end procedure (that includes preprocessing, time-domain analysis, and post-processing) may result to a serious load balancing problem, which leads to significant reduction of the computational efficiency.

The contents of this paper are as follows: first, we summarize the characteristics of the FEM-based sim-
ulation tool for earthquake wave propagation. We then explained the details of the parallel implementation: the domain decomposition (or partitioning) is improved according to the distribution of finite elements in hybrid grid; and workload of threads are managed at runtime to reduce idle time. To test the improved simulation tool, first, we examined the improvement of the computational efficiency in a simple model of 1.8 billion DOF, varying number of domain decompositions. Then, to fully check the performance of the combination of the improved domain decomposition and the optimized hybrid MPI-OpenMP implementation, analysis of crust model of 5.8 billion DOF was conducted, using up to 2,304 nodes (a total of 18,432 cores) of K computer.

2. BACKGROUND OF SIMULATION TOOL

The simulation tool is used for three-dimensional (3D) modeling of the source process, wave propagation in underground crust structure, interaction of bedrock and sedimentary basin, and topographic effects. Previously, Quinay et al.\textsuperscript{5} used the tool to compute sensitivity functions, and highlight the advantage of surface topography modeling in estimating the underground basin structure. In a different study by Quinay et al.\textsuperscript{5} (heretofore, the previous study), the tool was used to model wave propagation for a large magnitude earthquake, and to provide high resolution input ground motions in a nuclear power plant model for seismic analysis.

Currently, the simulation tool is based on linear-elastic analysis of wave propagation. FEM is used in spatial discretization, and Newmark-\(\beta\) (with \(\beta = \frac{1}{4}\)) method is used in time integration. A damping force term is included to model frequency independent attenuation. The working equation to solve per time step, \(n\), is given as,

\[
(K + \frac{2}{\Delta t} C + \frac{4}{\Delta t^2} M)u^{n+1} = (\frac{2}{\Delta t} C + \frac{4}{\Delta t^2} M)u^n + (C + \frac{4}{\Delta t} M)v^n + Ma^n + f^{n+1},
\]

where, \(K\), \(C\), \(M\), are matrices for stiffness, damping, and mass, and \(u\), \(v\), \(a\), and \(f\) vectors for displacement, velocity, acceleration, and force, respectively. \(\Delta t\) is the time increment.

The flow of the solving process using the simulation tool is illustrated in Fig. 1. As shown in preprocessing step, domain decomposition follows 3D model setting, and is performed before mesh generation (prepartitioning). The main reason is to avoid insufficient memory problem that can be encountered in mesh generation of high fidelity models. The mesh generation procedure is based on the method of Ichimura et al.\textsuperscript{7} that uses a hybrid (structured and unstructured) grid as reference for the FEM mesh. The main advantage of this procedure is that it highlights the irregular geometries while reducing the memory storage cost. The analysis step includes a flat MPI parallel implementation of preconditioned conjugate gradient (PCG) method to solve Eq. 1. Matrix-vector product (MVP) operation is performed using element-by-element (EBE) method:

\[
Ku = \sum_i (B_i^T D_i^T B_i^T u^n) + \sum_i (B_i^T D_i^T B_i^T u^n),
\]

where, superscripts \(t\) and \(v\) corresponds to tetrahedral and hexahedral elements, \(B\) is displacement differentiation matrix, \(D\) is elasticity matrix, and \((.)^T\) means transpose of (.) Postprocessing step includes application of bandpass filter to synthetics at selected observation points, and generation of data sets for offline visualization.

3. PARALLEL IMPLEMENTATION

The procedure shown in Fig. 1 can be improved to achieve high scalability by adopting an end-to-end procedure (i.e. executions of all the steps in a computer system are tightly-coupled). In the study of Tu et al.\textsuperscript{2}, the advantages of such procedure were highlighted, mainly in reducing the bottlenecks in multi-terabyte I/O. In computer systems such as the K computer wherein file-staging is performed for job execution, unnecessary I/O operations leads to significant amount of data storage and stage-in/out time. Because of the efficiency of an end-to-end procedure, the
solving process shown in Fig. 1 can be extended for use in iterative computations, for example, in inverse modeling studies. Inverse modeling requires multiple forward modeling executions to compute for sensitivity functions. However, FEM-based simulation tools are generally not employed to provide for sensitivity functions, even though FEM is accurate in terms of satisfying surface boundary and interface conditions. The main reason for this is the huge computation cost both in model generation and analysis. In using an FEM-based simulation tool for large scale forward or inverse modeling, computational efficiency is a major parameter to consider. In this paper, preprocessing and analysis steps were improved. In the preprocessing step, domain decomposition based on irregular grid partitioning is proposed. For the analysis step, a hybrid MPI-OpenMP implementation that balances computation time by managing workload of threads within each compute node is proposed. The target computer system for this application is distributed-memory system with multicore processors.

(1) Domain decomposition using two dimensional prepartitioning on irregular grid

The conventional approach to model generation is to first execute the meshing algorithm and then perform domain partitioning. This procedure, along with the use of an available partitioning library can generate high quality partitioning of mesh data that minimizes load imbalance in the parallel analysis solver. However, meshing a large model with high resolution would require large memory that may exceed the memory limitation of a single compute node (for example, K computer has a maximum memory limit of only 16 GB per compute node). Moreover, the conventional method outputs mesh data to be later processed offline by a partitioning library. In an end-to-end procedure for forward or inverse modeling, this I-O operation may lead to reduced overall computational efficiency as problem size grows larger. As an alternative that can address the said problems, a method for parallel mesh generation called prepartitioning, is used in this study.

Prepartitioning is a method that subdivides a domain before executing the meshing algorithm. It is used to circumvent the problem of handling large data structures (for example, high fidelity adaptive meshes) in preprocessing step (see the earlier works of Shostko and Löhner8), Galtier and George9, and Galtier10 for related papers on this subject). The subdomains that results from this partitioning can be meshed independently, leading to a parallel mesh generation procedure. Following the previous study5, the prepartitioning method implemented in this study suits the hybrid grid meshing procedure7, wherein the resulting subdomains automatically achieve node compatibility at subdomain interfaces. Because of less workload in the meshing procedure, this method results to fast preprocessing. However, the major disadvantage is the resulting load imbalance, since the distribution of the finite elements is not known beforehand. To minimize this load imbalance, we introduced a simple improvement to the partitioning method.

First, we briefly review the meshing procedure. At the start, a uniform background cell (created by intersection of 3D structured grids) is underneath in the model. The size of background cell is set as the finest resolution in the model. The octree technique11 is then performed recursively to combine small cells to form larger cells (see Ref. 7 for criteria, such as ratio of shear wave velocity). This results to multiresolution of cells. Here, the structured grids that coincide with the edges of the largest cells become candidates for a new set of grids that will be used for the partitioning. This new set is called irregular or I-grid, because the horizontal distance between adjacent grids is allowed to vary. After performing the octree technique, mesh generation is conducted, and mesh sizes are determined based on the multiresolution of cells. Figure 2 illustrates the setting of I-grid. The objective is to balance the computation load (for example, distribution of number of elements in the grids) by adjusting the grid locations in a given direction.

In 3D, the I-grid forms vertical planes, called separators that are perpendicular to a horizontal axis. The procedure for determining the candidate locations of separators is explained (the previous study5 and a recent study by Quinay et al.12 also explained these details for regular partitioning and I-grid, but no comparison of results was made). Suppose, $L_x$ is the dimension of the model along $x$-axis direction, and $V_{s_{max}}$ and $V_{s_{min}}$ are the maximum and minimum shear wave velocity. $L_x$ is chosen such that:

$$\text{mod}(L_x, d_{s_{max}}) = 0 \quad \text{where,}$$

$$d_{s_{max}} = d_{s_{min}} \times 2^\lfloor\log_2(V_{s_{max}}/V_{s_{min}})\rfloor,$$

If origin at the $x$-axis direction is placed at the center of $L_x$, separators may be located at the following coordinates that do not intersect the face of any $d_{s_{max}}$:

$$a_m = \pm m\left(L_x \frac{2^k}{a_m}\right),$$

where,

$$1 \leq k \leq \lceil\log_2(L_{min})\rceil,$$

$$\text{mod}(L_x \frac{2^k}{a_m}, d_{s_{max}}) = 0.$$

In the above equations, $a_m$ is a coordinate in $x$-axis, $m$ is any integer value (greater than 0), from 1 to $2^{(k-1)} - 1$, and $L_{min}$ is smaller of $L_x$ and $L_y$ (dimension of the model along $y$-axis direction). For the case
perpendicular to the y-axis, \( L_x \) is changed to \( L_y \) in Eqs. 3, 5 and 7.

Now, the procedure for setting the separators in the model is as follows: first, a regular partitioning grid based on the previous study\(^5\) is used for the 3D model (i.e., all subdomains have the same dimensions respectively in \( x \)- and \( y \)-axis directions). Then, mesh generation\(^7\) is conducted, and the number of hexahedral and tetrahedral elements in each subdomain and for the whole domain are determined. **Equations 5, 6, and 7** are then used to determine the candidate separator locations. A time parameter, \( t_{\text{ave}} \) is introduced which will serve as a reference for adjusting the regular grid according to candidate separator locations. \( t_{\text{ave}}^{\text{I-grid}} \) is estimated using predetermined \( \mathbf{K}_u \) computation times for one hexahedral and tetrahedral element, and the total number of subdomains (see Eq. 8). Then, to adjust grids that are perpendicular to an axis, a second time parameter, \( t_{\text{ave}}^{\text{I-grid}} \), is introduced. \( t_{\text{ave}}^{\text{I-grid}} \) is an average for all the subdomains in a particular grid (see Eq. 9). The grids are then adjusted by minimizing per grid the difference between \( t_{\text{ave}}^{\text{I-grid}} \) and \( t_{\text{ave}}^{\text{I-grid}} \).

\[
\begin{align*}
\hat{t}_{\text{ave}}^{\text{I-grid}} &= \left[ \left( \sum_i n_{i,\text{vox}}^{\text{I-grid}} \times t_{i,\text{vox}}^{\text{I-grid}} \right) + \left( \sum_i n_{i,\text{tet}}^{\text{I-grid}} \times t_{i,\text{tet}}^{\text{I-grid}} \right) \right] / n_{\text{subd}}, \\
\hat{t}_{\text{ave}}^{\text{I-grid}} &= \left[ \left( \sum_i n_{i,\text{vox}}^{\text{I-grid}} \times t_{i,\text{vox}}^{\text{I-grid}} \right) + \left( \sum_i n_{i,\text{tet}}^{\text{I-grid}} \times t_{i,\text{tet}}^{\text{I-grid}} \right) \right] / n_{\text{subd}}.
\end{align*}
\]

In Eqs. 8 and 9, \( n_{\text{subd}} \) and \( n_{\text{subd}} \) are the number of subdomains in total and in a grid, \( n_{i,\text{vox}}^{\text{I-grid}} \) and \( n_{i,\text{tet}}^{\text{I-grid}} \) are the number of hexahedral and tetrahedral elements in subdomain \( i \), and \( t_{i,\text{vox}}^{\text{I-grid}} \) and \( t_{i,\text{tet}}^{\text{I-grid}} \) are the \( \mathbf{K}_u \) computation times for a single hexahedral and tetrahedral element. In adjusting the grids, the total dimensions of the 3D model and the initial number of partitioning must remain unchanged. It is noted that the time parameters can be chosen to suit the target model (for example, instead of average time, maximum time to compute Eq. 2 can be used). Average time was used in this study because when applied to a complicated model, the resulting I-grid settings always satisfied the initial number of partitioning, and at the same time reduced the difference in distribution of maximum number of elements of subdomains (as shown later in the performance test on realistic model).

**2) Implementation of hybrid MPI-OpenMP**

In the previous study\(^5\), the simulator is based on a flat MPI implementation of the analysis step of Fig. 1. In a simple model problem, this implementation achieved satisfactory computation performance for up to 124 processors. However, the performance may easily degrade with increasing number of processors due to simple domain decomposition, and absence of any optimization. In this study, we aim to achieve good performance of the simulator in using thousands of multicore processors.

The operations of the analysis that consumes the most wall clock time are first identified. Consider a 4.6 billion degree-of-freedom problem (DOF) solved using 3,120 processors. Table 1 summarizes the computation time of a single step of Eq. 1 for operations in one PCG iteration. As shown, all MVP operations consumed about 43% of the total time, while message passing (or MP) operations consumed about
Table 1 Sample computation time of a single step of Eq. 1:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Computation time (s)</th>
<th>% of total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time</td>
<td>1.87483</td>
<td>-</td>
</tr>
<tr>
<td>2 MVP (residual eq)</td>
<td>0.5406</td>
<td>29.072</td>
</tr>
<tr>
<td>2 MP (residual eq)</td>
<td>0.40484</td>
<td>21.593</td>
</tr>
<tr>
<td>1 MVP (PCG iter)</td>
<td>0.27253</td>
<td>14.536</td>
</tr>
<tr>
<td>1 MP (PCG iter)</td>
<td>0.20242</td>
<td>10.796</td>
</tr>
<tr>
<td>3 inner products</td>
<td>0.08180</td>
<td>4.3631</td>
</tr>
</tbody>
</table>

32% of the total time (here, the maximum percentage of nodes used for communication to the total nodes in a subdomain is 9.7%). These two operations combined consumed a total of about 75% of the total time. Thus, optimizing these operations will have significant impact on the computational performance. It should be noted that the computation time of MVP operation is not influenced by the workload of other processors, unlike MP operation.

First, the choice of parallel programming model is justified. Available models are flat programming (implementing pure MPI) and hybrid programming (implementing MPI-OpenMP). Since the target computing environment is a distributed-memory computer with multiple cores, it is more efficient to utilize the shared-memory and use a coarse domain decomposition. Thus, the hybrid (MPI-OpenMP) programming model is the more suitable choice.

Algorithm 1 A flat MPI code for MVP and MP operations to update a vector with length, n_oshubd

Required: Update vector, r_o
1: Initialize r_o to null vector
2: r_o ← Eq. 2
3: Perform MP operation

Algorithm 2 A simple hybrid MPI-OpenMP implementation for Algorithm 1

Required: Update vector, r_o
1: Initialize r_o to null vector
2: Start OpenMP parallel region
3: Initialize a private vector r_p to null vector
4: r_p ← Eq. 2
5: Start OpenMP critical region
6: r_o=r_o+∑_{i} (r_p)_i
7: End OpenMP critical region
8: End OpenMP parallel region
9: Perform MP operation
- where nth is the total number of OpenMP threads

A typical flat MPI algorithm for updating the values of a given vector by the selected operations is given in Algorithm 1. Implementation of a hybrid MPI-OpenMP to Algorithm 1 is straightforward as shown in Algorithm 2. However, several problems related to performance can arise from this simple implementation. First, at the start of the parallel region (see Algorithm 2: line 3), initialization of each thread’s private vector, r_p, with length of n_oshubd (total number of nodes in a subdomain) is required. At runtime, however, a single thread will only update several elements of this vector. This also occurs in the critical region (Algorithm 2: line 6), wherein all threads loop the private vector to update the global vector, r_o. However, for the critical region, the problem is worse since only one thread at a single time can update r_o. If OpenMP directives are implemented simply as Algorithm 2, the workload of each thread is automatically set at runtime. Thus, using the dimension, n_oshubd of r_o is necessary for r_p. But this can lead to reduction in efficiency because of these operations inside the parallel region with workloads that are comparable to a serial code. Next, as shown in Algorithm 2: line 9, the MP operation needs to wait until the OpenMP parallel region is finished. But MP operation can already be initiated once all FE nodes used for communication (or active nodes) have been updated. But since the workload is automatically set at runtime, this operation must wait until the threads looped through all elements to update r_o. Moreover, since MP operation is executed by a single thread, all other threads are idle until the message-passing has been completed. Clearly, without optimization, such simple implementation leads to inefficient computation when large number of cores are used.

In order to address these problems, this study proposes a more efficient approach wherein the workload of each thread within a compute node is managed at runtime to achieve balanced computation time among all threads. The following are the proposed measures:
1. Isolate the finite elements that have local node(s) which is/are active node(s), and perform MVP operation of these elements separately.
2. Perform MP operation simultaneously with MVP operation of finite elements that have local nodes which are not used for communication (inactive nodes).
3. Make the code manage each thread’s workload.
4. Reduce the dimension of the private vector according to the set workload for each thread.

These proposed measures would require a preprocessing of finite elements and nodes lists, and an explicit control of each thread’s workload. Thus, suitable MPI and OpenMP directives are implemented. First, from MPI-2.0 standard, several levels of thread support are available (see Lusk and Chan3). In this study, MPI_THREAD_FUNNELED is used, which allows only the main (or master) thread to initiate MPI calls. While other thread support can be used, MPI_THREAD_FUNNELED is easier to man-
age since the master thread is given a unique thread id number inside the multithreaded region.

For the OpenMP setting, a directive-pair !$OMP SECTIONS/$!$OMP END SECTIONS was used. Inside this directive pair, multiple number of !$OMP SECTION can be placed wherein each section will be assigned a single thread. !$OMP MASTER/$!$OMP END MASTER directive pair was also used for $MVP$ and $MP$ operations of the master thread. With these settings, workload can be balanced and distributed to each thread. By setting the master region plus the number of sections equal to the number of available threads, all threads are kept busy and the workload can be optimized such that all threads reach the synchronization region (before entering the critical region) at almost the same time.

To control the workload for each thread, new elements and node lists are created. As mentioned, the elements (hexahedral and tetrahedral) associated with the active nodes (heretofore, active elements) are separated from the other elements. We call the size of this list, $L^a$. The rest of the elements (heretofore, inactive elements) in total has size, $L^b$. The workload of thread $i$ for the active and inactive elements are vectors with size $L^a_{e,i}$ and $L^b_{e,i}$, respectively. For the node lists, vectors with size $L^a_{n,i}$ and $L^b_{n,i}$ are created. Setting the workloads based on computation time, $t$, is as follows for the active elements:

$$\min_{L^a_{e,i}}[t_{MVP}(L^a_{e,i}) - \bar{t}_{MVP}(L^a_{e,1}, L^a_{e,2}, \ldots, L^a_{e,nth})] = \left(1 \over n^{th}\right) \times \sum_{i=1}^{nth} \bar{t}_{MVP}(L^a_{e,i}) \ ,$$

(10)

where, thread, $i = 1 \text{ to } nth$, and,

$$\bar{t}_{MVP}(L^a_{e,1}, L^a_{e,2}, \ldots, L^a_{e,nth}) = \left(1 \over n^{th}\right) \times \sum_{i=1}^{nth} \bar{t}_{MVP}(L^a_{e,i}) \ .$$

For inactive elements, the setting is slightly different from the active elements since $MP$ operation is to be conducted simultaneously. First, the workload of the master thread ($L^b_{e,1}$) is determined for several time steps while the workload of other threads are equally divided:

$$\min_{L^b_{e,i}}[t_{MVP}(L^b_{e,i}) + \bar{t}_{MP} - \bar{t}_{MVP}(L^b_{e,2}=L^b_{e,3}=\ldots=L^b_{e,nth})] = \left(1 \over n^{th}-1\right) \times \sum_{i=2}^{nth} \bar{t}_{MVP}(L^b_{e,i}) \ .$$

(11)

where,

$$\bar{t}_{MVP}(L^b_{e,2}=L^b_{e,3}=\ldots=L^b_{e,nth}) = \left(1 \over n^{th}-1\right) \times \sum_{i=2}^{nth} \bar{t}_{MVP}(L^b_{e,i}) \ .$$

(12)

After getting $L^b_{e,1}$, the workload of all other threads, $i = 2 \text{ to } nth$, are then set using the value of

$$\min_{L^b_{e,i}}[t_{MVP}(L^b_{e,i}) - \bar{t}_{MVP}(L^b_{e,2}=L^b_{e,3}=\ldots=L^b_{e,nth})] = \left(1 \over n^{th}-1\right) \times \sum_{i=2}^{nth} \bar{t}_{MVP}(L^b_{e,i}) \ .$$

(13)

Here, the total workload must remain constant:

$$\sum_{i=1}^{nth} L^b_{e,i} = L^b \ .$$

(14)

At the first time step, initial values for $L^a_{e,i}$ and $L^b_{e,i}$ are set, and minimization based on Eqs. 10 and 11 is performed using computation times recorded in the time step. New values of $L^a_{e,i}$ and $L^b_{e,i}$ are then used in the next time step, and the procedure is repeated. Because the total workload per subdomain may vary (depending on the results of grid partitioning), different values of $L^a_{e,i}$ and $L^b_{e,i}$ may be obtained among subdomains. The procedure can be stopped when the computation times of all threads are balanced, or when there is no further computation time reduction.

Algorithm 3 describes the improved MVP and MP operations using the workloads determined by Eqs. 10 to 14. Lines 3 to 9 perform MVP operations for active elements. In line 10, thread synchronization is set to ensure that $r_n$ of active elements are fully updated before $MP$ operation. Lines 11 to 16 and Lines 18 to 21 are performed simultaneously ($MP$ operations and $MVP$ operations of inactive elements) by all threads. In the initialization and critical sections, each thread will call on $L^a_{e,i}$ and $L^b_{e,i}$. It should be noted that all operations in each thread updates only private variables, except for $MP$ operation (i.e. race condition is avoided).

4. PERFORMANCE TESTS

This section describes the performance tests on the improved simulation tool. The first test deals with a simple two-layered model, and the second test deals with a realistic crust model. For these tests, we used the computational resources of the K computer of the Advanced Institute for Computational Science in Kobe, Japan. The K computer consists of SPARC64 VIII/Hx processors with 2.0 GHz clock speed. Each compute node is allocated with one CPU with 8 cores and a maximum available memory of 16 GB.

(1) Simple model

The setting of the simple two-layered model is shown in Fig. 3. The top and bottom layers are characterized by flat surface and interface. A single layer of tetrahedral elements is placed at the surface and a double layer is placed at the interface. The remaining parts are meshed with regular hexahedral elements. The element size at the top layer is set at
Algorithm 3 An improved hybrid MPI-OpenMP implementation for Algorithm 1

Required: Update vector, $r_o$

1. Initialize $r_o$ to null vector
2. Start OpenMP parallel region
3. Start OpenMP sections for active elements
4. (for each thread, $i$, initialize private vector $r_{p,i}^a$ (with size $L_{n,i}$) to null vector
5. $r_{p,i}^a \leftarrow \text{Eq. 2}$ (where workload of each thread is determined by Eq. 10)
6. End OpenMP sections for active elements
7. Start OpenMP critical region for active elements
8. $o_o = r_o + \sum_i^{nth} (r_{p,i}^a)$
9. End OpenMP critical region for active elements
10. Set OpenMP barrier (for thread synchronization)
11. if $i$ is master thread then
12. Start OpenMP master region (for thread, $i=1$)
13. Perform $MP$ operation
14. Initialize private vector $r_{p,i}^b$ (with size $L_{n,i}$) to null vector
15. $r_{p,i}^b \leftarrow \text{Eq. 2}$ (where workload is determined by Eq. 11 or 13)
16. End OpenMP master region
17. else
18. Start OpenMP sections for thread, $i=2$ to $nth$ (one section for each thread)
19. (for each thread, $i=2$ to $nth$, initialize private vector $r_{p,i}^b$ (with size $L_{n,i}$) to null vector
20. $r_{p,i}^b \leftarrow \text{Eq. 2}$ (where workload of each thread is determined by Eq. 11 or 13)
21. End OpenMP sections for thread, $i=2$ to $nth$
22. end if
23. Start OpenMP critical region for inactive elements
24. $r_o = r_o + \sum_i^{nth} (r_{p,i}^b)$
25. End OpenMP critical region for inactive elements
26. End OpenMP parallel region

100 m, while the element size at the bottom layer is set at 200 m. This arrangement results to a larger distribution of hexahedral elements than tetrahedral elements in the model.

By applying a regular partitioning, each subdomain contains the same number of nodes, tetrahedral and hexahedral elements. However, the number of active nodes varies depending on the location of the subdomain in the grid. In this setting, we can isolate and check the performance of the improved hybrid MPI-OpenMP implementation without influence of imbalance in the distribution of nodes and elements.

For the same target model, we considered four cases (see Table 2) which vary in number of domain decompositions and grid dimensions. Table 3 provides a summary of the distribution of nodes, elements, and active nodes. For all cases, each subdomain is assigned to one compute node. The total number of unknowns in the model is about 1.82 billion.

First, we checked the strong-scaling performance of a flat MPI implementation (Algorithm 1) for the above cases. Table 4 provides a summary of the results. Nearly ideal scalability was achieved because of coarse domain decomposition. Here, MVP operation did not incur any overhead because it operated as a serial code within each compute node. On the other hand, MP operation incurred overhead, but since coarse domain decomposition was used, the overhead was almost negligible. For a hybrid implementation, the performance is expected to drop due to additional overhead introduced in increasingly finer granularity (i.e. smaller task for computation with respect to communication).

Next we checked the performance of the improved hybrid MPI-OpenMP for Case 2, Case 3, and Case 4. Figure 4 plots the comparison of the optimized and unoptimized code for a strong scaling test. The optimized code refers to the code based on Algorithm 3, while the unoptimized code refers to simple hybrid parallel implementation shown in Algorithm 2. Using the flat MPI result for the 64 domain decomposition in Table 4 as reference, the strong scaling efficiency for each case was computed. This flat MPI result was also used to draw the ideal line. The hy-
Table 3 Computation loads of different cases for test on simple model

<table>
<thead>
<tr>
<th>Case</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of tetrahedra per subdomain</td>
<td>373,248</td>
<td>165,888</td>
<td>93,312</td>
<td>23,328</td>
</tr>
<tr>
<td>No. of hexahedra per subdomain</td>
<td>9,310,464</td>
<td>4,137,984</td>
<td>2,327,616</td>
<td>581,904</td>
</tr>
<tr>
<td>No. of nodes per subdomain</td>
<td>9,524,325</td>
<td>4,262,277</td>
<td>2,414,037</td>
<td>620,157</td>
</tr>
<tr>
<td>Active nodes (max) / subdom nodes (%)</td>
<td>1.41</td>
<td>2.10</td>
<td>2.80</td>
<td>5.52</td>
</tr>
</tbody>
</table>

Table 4 Results of flat MPI run: computing time of different cases for test on simple model

<table>
<thead>
<tr>
<th>Case (X)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation time, (s)</td>
<td>3.9494</td>
<td>1.7585</td>
<td>0.9651</td>
<td>0.2393</td>
</tr>
<tr>
<td>Case 1/ Case (X)</td>
<td>1.0000</td>
<td>2.2458</td>
<td>4.0922</td>
<td>16.50</td>
</tr>
<tr>
<td>Parallel efficiency (%)</td>
<td>-</td>
<td>99.813</td>
<td>102.30</td>
<td>103.15</td>
</tr>
</tbody>
</table>

Table 5 Comparison of maximum memory usage (in Gigabytes) in a single compute node

<table>
<thead>
<tr>
<th>Case</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>2.26</td>
<td>1.30</td>
<td>0.38</td>
</tr>
<tr>
<td>Optimized</td>
<td>2.40</td>
<td>1.37</td>
<td>0.41</td>
</tr>
</tbody>
</table>

The bridge MPI-OpenMP parallel settings are as follows: for Case 2, a total of 1,152 cores (144 MPI threads × 8 CPU cores) was used; for Case 3, a total of 2,048 cores (256 MPI threads × 8 CPU cores) was used; and for Case 4, a total of 8,192 cores (1,024 MPI threads × 8 CPU cores) was used (Case 4 was introduced to check the performance of the code for fine granularity). As shown in the figure, parallel efficiency was improved in all cases. The poor performance of the unoptimized cases was due to parallel overhead (imbalance due to communication time) and unoptimized vector initialization and updating. In the optimized case, the increase in efficiency is between 21% to 27%. Also, it can be observed that the efficiency for the optimized case is above 80% for Case 2 and Case 3, but drops to 73% for Case 4. In Case 4, MP operation consumed about 28% of the total time, which is significantly higher than in the other cases (12.5% for Case 2, and 16.1% for Case 3). Although the proposed optimization distributes the MP operation time across all available threads, it was found that the granularity of Case 4 resulted to significant reduction in efficiency. Thus, in order to achieve parallel efficiency above 80% for this example, the ratio of active nodes with total nodes in subdomains must be set below 3.0% (see last row of Table 3). In terms of maximum memory usage, Table 5 shows that the optimized and unoptimized codes are comparable, although new variables were introduced for each thread in the optimized approach. These results suggest that the proposed optimization can lead to significant improvement in performance of the simulator.

(2) Realistic crust model

We selected a region in Niigata Prefecture, Japan, and generated a model of the crust structure based on the data obtained from Japan Seismic Hazard Information Station (J-SHIS)\(^\text{14}\). The model has a dimension $57.6 \times 57.6 \times 45$ km and composed of eight layers. Figure 5 shows the model domain. This region is chosen for this test because of its high liquefaction hazard (based on the damage to Niigata city by the 1964 Niigata earthquake), presence of active fault zones (FZ) near the city, and highly varying topography and underground structure. Thus, it is relevant for both seismology and engineering study. Figure 6 shows the cross-sections of the model along latitude $38^\circ$ and along longitude $139^\circ15'$ (shown in Fig. 5).
Table 6 lists the shear wave velocity properties for each layer. It can be observed that along the east-west direction (EW), low shear wave velocity layers are dominant on the west side, while high shear wave-velocity layers are dominant on the east side. Because the element size is tailored with respect to the shear wave velocity, this setting will introduce imbalance in the distribution of nodes and elements in the multiresolution model. In the north-south (NS) direction, the layer arrangement is less varying than in EW, thus less imbalance in the distribution of nodes and elements is expected in this direction.

The prepartitioning and mesh generation were conducted in K computer. The minimum and maximum element sizes are 15 m and 60 m. With reference to Table 6, maximum mesh size for each layer is set as follows: for layers L1 and L2: 15 m; layers L3 and L4: 30 m; and L5 to L8: 60 m. In the initial regular grid partitioning, the 3D model was decomposed into 2,304 subdomains with a symmetric grid shape of $48 \times 48$. The horizontal dimension of each subdomain was 1200 m. The total computation load is summarized in Table 7. In setting the $I$-grid, the grid adjustments were first performed in EW direction since the load imbalance was expected to be more severe in this direction. Computation times $t_{ave}$ and the $t_{ave}$ were estimated and grid locations were then...
In order to check the effect of using $I$-grid, three cases were considered: Case 1 is using regular grid; Case 2 is using irregular grid dimensions in $EW$ direction only (i.e. grid dimensions in $NS$ direction are equal); and Case 3 is using fully irregular grid with setting given in Figs. 7-A and 7-B. Figure 8 shows the distribution of maximum number of elements of a subdomain in each grid. Although parameters based on average time, $t_{ave}^i$ and $t_{ave}^e$, were used in setting the $I$-grid, the severe imbalance in distribution of maximum number of elements per grid of Case 1 was reduced in Case 2 and Case 3. Note that in Case 2, the maximum number of elements in $NS$ grids were also changed since adjustments in $EW$ direction changed the distribution of elements in each subdomain. Since only minor adjustments were made in $NS$ grids, the plots of Case 2 and Case 3 are close to each other. As shown, there were still unbalanced grids even after Case 3. This is caused by the limitation of using average time (to account for the effect of maximum element distributions) as well as the settings used in 2D partitioning method (using planes as separators, and constraints in separator locations).

Similar to the previous example, we aimed to observe the improvement in parallel performance due to optimization. Speed-up values (averaged per grid) were computed by comparing the time in using 18,432 cores (2,304 MPI threads $\times$ 8 CPU cores) and in using 2,304 cores (2,304 MPI threads $\times$ 1 CPU core). The ideal speed up is 8.0 (based on using all 8 CPU cores per node). All times were measured after a global synchronization (MPI_Barrier). The results for Cases 1, 2, and 3 are plotted in Fig. 9, grouped into optimized (labelled "O") and unoptimized (labelled "U") subcases. Due to optimization, for Case 1, the increase in speed up is between 27% to 32%; for Case 2, the increase is about 30%, and for Case 3, the increase is between 28% to 30%.

Now, we examine the effect of using irregular grid. For a given direction, it can be observed that each case gives a similar trend for with and without optimization. Thus, the trend is considered to be largely influenced by the model partitioning. In Case 1 (speed up results are in green solid and open circles in Fig. 9), the speed up decreased from south towards the north grids, and from west towards the east grids. Here, the grids on the north part of $NS$ direction obtained lower speed up compared to the grids on the east part of the $EW$ direction). The limitation in speed up was caused by the difference in distribution of number of elements of subdomains within a grid, i.e. large difference results to low speed up (for example, based on the plot for Case 1 in Fig. 9-A, we can infer that for a particular grid number along $NS$ direction, the included subdomains in this grid follow this distribution of elements). In Case 2 (speed up results are in red solid and open triangles in Fig. 9-A, we can infer that for a particular grid number along $NS$ direction, the included subdomains in this grid follow this distribution of elements). In Case 3 (speed up results are in yellow solid and open squares in Fig. 9-A, we can infer that for a particular grid number along $NS$ direction, the included subdomains in this grid follow this distribution of elements).
Fig. 8 Distribution of maximum number of elements per grid for Cases 1, 2, and 3 for (A) EW and (B) NS directions. Square plots correspond to hexahedral elements only; triangle plots correspond to combined hexahedral and tetrahedral elements. Lines connect plots of the same case.

9), the elements were redistributed in EW direction which reduced these differences and resulted to significant improvement in speed up (Fig. 8-A shows the increase in combined tetrahedral and hexahedral elements in east part). In Case 3 (speed up results are in black solid and open squares in Fig. 9), only small adjustments were introduced in the NS direction resulting to small change in distribution of maximum number of elements (see Fig. 8-B). Hence, the plots of speed up for Case 2 and Case 3 are almost identical.

5. CONCLUSION

This paper presented improvements to the 2D grid partitioning and implementation of hybrid MPI-OpenMP, so that an FEM model with billion order unknowns can be solved efficiently in large multicore computer. In analyzing a realistic model, results showed that these improvements increased the speed up by about 30% in comparison with simple hybrid parallel implementation. This led to achieving speed up value of up to 6.8 when 8 cores of each node of K computer were used. Here, the current limitation in performance can be directly attributed to the sim-
plicity of load balancing procedure based on 2D partitioning. Thus, as part of future work of this study, we aim to develop a load balancing procedure based on 3D partitioning that also accounts for the distribution of active nodes. Moreover, since the efficiency in memory access is only partially improved (inherent to EBE method is accessing and updating noncontiguous memory locations which was not addressed in this study), we are now looking at the advantage of implementing a pure MPI implementation based on MPI 3.0 standard with support for shared memory programming (this standard allows for efficient noncontiguous memory access by mapping memory segments of processes into noncontiguous locations\footnote{Hoefler, T., J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, R. Thakur: MPI+MPI: a new hybrid approach to parallel programming with MPI plus shared memory, \textit{Computing}, \textbf{95}, 1121-1136, 2013.}). It is expected that these proposed improvements will address the current limitation in performance.

\textbf{ACKNOWLEDGEMENTS:} This study is part of research work under the Next Generation Supercomputer project: Study on Urban Area Simulation of Earthquake and other Natural Disasters. The K computer of RIKEN Advanced Institute for Computational Science and computers of Earthquake Information Center (EIC) of Earthquake Research Institute, The University of Tokyo were used for this study.

\textbf{REFERENCES}


\textit{(Received June 20, 2014)}