Parallel Scalability Enhancements of Seismic Response and Evacuation Simulations of Integrated Earthquake Simulator

M.L.L. Wijerathne*, Muneo Hori, Tsuyoshi Ichimura, and Seizo Tanaka

Earthquake Research Institute, The University of Tokyo, Tokyo, Japan
{lalith,ichimura,hori,stanaka}@eri.u-tokyo.ac.jp

Abstract. We developed scalable parallel computing extensions for Seismic Response Analysis (SRA) and evacuation simulation modules of an Integrated Earthquake Simulator (IES), with the aim of simulating earthquake disaster in large urban areas. For the SRA module, near ideal scalability is attained by introducing a static load balancer which is based on the previous run time data. The use of SystemV IPC as a means of reusing legacy seismic response analysis codes and its impacts on the parallel scalability are investigated. For parallelizing the multi agent based evacuation module, a number of strategies like communication hiding, minimizing the amount of data exchanged, virtual CPU topologies, repartitioning, etc. are used. Priliminary tests on the K computer produced above 94% strong scalability, with several million agents and several thousand CPU cores. Details of the parallel computing strategies used in these two modules and their effectiveness are presented.

Keywords: multi agent simulations, seismic response analysis, large urban area, HPC, scalability.

1 Introduction

Petascale super computers have opened new avenues for more reliable earthquake disaster predictions compared to the currently used simplified methods. The current earthquake disaster predictions, which are based on the statistical analysis of past events, are less reliable since the built environment has significantly changed since those decades old past events. It is possible to make more reliable predictions by simulating large area earthquake disasters using cutting edge numerical tools from many disciplines like seismology, earthquake engineering, civil engineering, social science, etc. High performance computing is vital to meet the computational demand of simulating high fidelity models of large urban areas, with high spatial and temporal resolutions. Further, the need of stochastic modelling increases this high computational demand by severalfold. Stochastic

* Earthquake Research Institute, the University of Tokyo, 1-1-1, Yayoi, Bunkyo-ku, Tokyo 113-0032, Japan.

M. Daydé, O. Marques, and K. Nakajima (Eds.): VECPAR 2012, LNCS 7851, pp. 105–117, 2013.

modeling is required to improve reliability of the numerical predictions; what decision makers require is the high level of confidence in the predictions.

With the aim of developing a such a system for making for more reliable earthquake disaster predictions, a system called Integrated Earthquake Simulator (IES) is being developed[1]. The objective of IES is to seamlessly simulate earthquake hazards, disasters and aftermaths. Modules for simulating source to site seismic wave propagation, seismic response analysis (SRA) of buildings and underground structures, evacuation and recovery are being developed. Highly scalable parallel extensions are essential for IES, in order to realize more reliable disaster predictions using high fidelity models. While petascale machines provide the necessary hardware resources, it is a challenging task to develop scalable codes for simulating a large urban area with fine detailed models.

Prior to this work, a parallel computation extension for the IES's SRA module has been developed[2]. However, several bottlenecks seriously hinder its scalability to mere 32 CPU cores. The SRA module of IES consists of several simple to moderately advanced seismic response analysis methods. All these seismic response analysis models are implemented as serial programs. With task level parallelism and simple static load balancer, near ideal scalability is attained up to a limited number of CPUs. Runtime of some tasks are excessively large compared to the majority of tasks; large buildings involve much longer run time. These tasks with large runtime limit the scalability, and have to be parallelized to scale over a large number of CPUs.

The emergency evacuation module of IES is based on Multi-Agent Simulation (MAS), in which people are represented by agents that autonomously navigate, and interact with the neighbors and the environment. Even though MAS often advocates the use of simple agents, sophisticated and smart agents are necessary to model complex human behaviors. High performance computing enhancements are necessary to simulate the evacuation of large urban area with millions of smart agents. Most literature focuses on simulation of several ten thousands of simple agents on less than 50 CPU cores, and real time rendering; modeling virtual worlds for games and entertainment is the main objective of the existing studies. Consenza et al.[3] have demonstrated simulation of 100,000 agents on 64 CPUs, but it has limited scalability. With several strategies to hide communications and minimize the volume of data exchanged, our HPC enhanced MAS code attained 94% strong scalability up to 2048 cores on the K-computer, with two million agents.

Details of the parallel extension of the SRA module and scalability are presented in the section 2. Section 3 summarizes difficulties and the strategies used in parallelizing the multi-agent based evacuation module. Some concluding remarks are given in the last section.

2 Parallel Performance Enhancement of SRA Module

SRA module of IES has a series of simple to advance nonlinear SRA methods like discrete element method (DEM), one component model (OCM), fiber

element model, etc. All these SRA methods are implemented as serial codes with FORTRAN 77. On the other hand, IES is developed with C++; extensive object oriented features make C++ a popular choice for projects developed by large groups. Though FORTRAN 77 is an outdated standard, the reuse of these SRA codes is unavoidable since rewriting in a latest standard, verification and validation require a significant effort and time.

A previous effort to parallelize SRA module [2] based on a master slave model produced much lower scalability than the required; some bottlenecks that hinder the scalability are extensive use of temporary files for inter-process communication and output data saving, uneven workloads assign to CPUs, and the large number of message passing. We eliminated these bottlenecks and achieved high scalability with task parallelism and a static load balancer. The rest of the section provides the details of these improvements.

2.1 Calling Legacy FORTRAN Codes

The existing FORTRAN 77 based SRA codes have to be converted to libraries, so that those can be reused in IES. This conversion is a complex and error prone process since the original codes have been developed under non-standard default compiler settings. The SRA codes have been developed with COMPAQ FORTRAN which used *save* semantics, which makes compiler to allocate all local variables in static storage and initialized to zero. The use of this special compiler setting is not recommended by most of the present day compilers including Intel FORTRAN, which is the successor of COMPAQ FORTRAN. To circumvent this problem, independent SRA executables can be called from IES using *system()* command while using temporary files for data exchnage. Though this is an acceptable solution in serial applications, using temporary files is a serious performance bottleneck in parallel applications.

It is necessary to find a simple and less error prone means of reusing old FORTRAN codes for the future needs. To this end, we explored the applicability of SystemV IPC[4] (commonly abbreviated SysV IPC) in parallel computing environment. In this setting, IES invokes the legacy FORTRAN codes as independent executable and use shared memory segments and semaphores of SysV-IPC to exchange data with IES. Compared with conversion to a library, this process involves fewer modifications to FORTRAN codes. As shown at the end of this section, the use of SysV-IPC is a good alternative to call legacy FORTRAN 77 codes in parallel environments.

One disadvantage of using SysV IPC shared memory segments is that those persist even after termination of the owner processes, unless explicitly cleaned. Under normal operations, the IPC classes introduced to IES take care of the cleaning of the used SysV IPC resources. However, if the main program dies prematurely, the IPC resources used must be manually cleaned, in each node; SysV IPC resources can be released with the command *ipcrm <-s/-m> <shmid/ semid>*. Unless the shared memory segments are manually cleaned, it becomes a system wide persistent memory leak. SysV IPC is a good solution for a cluster dedicated for IES. However, premature exits of IES may cause problems in

clusters with multiple users and automated resource management. Therefore, this SysV IPC approach cannot be used in professional super computers; spawning new process is usually prohibited.

2.2 Saving Large Volume of Data with MPI-IO

When simulating a large urban area, SRA module produces a large volume of output data; around 10GB of binary data per 10,000 structures. IES should organize and save the output data in a ready-to-visualize format. We utilized the MPI-IO functionalities of MPI-2 standard to write the SRA output in ready to visualize format, thereby achieving higher parallel performance and reducing the visualization time. Compared to POSIX I/O, MPI-IO can deliver much higher I/O performance in parallel environment[5]. The level 3 access function *MPI_File_write_all()*, which allows non-contiguous collective access to a file, is used to write SRA output data in a ready to visualize format.

2.3 Load Balancing

The currently implemented task parallelism makes the SRA module an embarrassingly parallel problem. However, the difficulty of predicting the runtime of each task makes it difficult to attain a good load balance; run time for each structure depends on the location and magnitude of an earthquake, etc. Therefore, some form of dynamic load balancing is necessary. A hybrid solution with static load balancing for the majority of the buildings and switching to dynamic load balancing for the remaining is the best solution. However, only static load balancing is implemented in the modified version, since dynamic load balancing requires significant modifications to the present code of IES.

The static load balancer utilizes the runtime information recorded at previous executions to assign nearly equal workloads to each CPU. The simple static load balancer first distributes all the building information and previous run time data to all the CPUs. These data are grouped such that data of one or several GIS (Geographic Information System) tiles are in one set, and each data set is compressed to reduce message size. CPUs independently pick a subset of data so that each has nearly equal work load, estimated based on the previous runtime data. The use of static load balancer is an acceptable solution provided the runtime difference of a given building due to different input Strong Ground Motions (SGM's) of similar magnitudes would not be large. Irrespective of the number of input data files of building shapes and run times, this phase involves only 2 message broadcasts, provided the volume of building shape data and run time data is less than 4GB. Unlike a master slave load balancing, where master CPU decides and distributes the work, this communication independent load balancing does not degenerate the scalability.

2.4 Scalability of the New Parallel Extension of SRA Module

The modified parallel extension of SRA module involves message passing only at the very beginning and the end of a simulation: at the beginning to share some

configuration files and building shapes and run time data of each GIS tile; at the end to save data with collective MPI-IO. In this setting, the modified parallel module should well scale with the number of CPUs, as long as previous run time based load estimation assigns equal workloads to CPUs.

To test the scalability of the new parallel extension, we simulated 125,500 buildings in the Kochi city of Japan, with the OCM model. The buildings are excited with the strong ground motion data observed during the 1995 Kobe earthquake. A DELL cluster with QLogic 12200 InfiniBand switch and 16 computation nodes, each with two hexa-core Intel Xeon X5680 CPUs and 47GB DDR3 memory, was used for these simulation. This cluster has hardware support for MPI-IO. However, it does not have a parallel file system supporting MPI-IO.

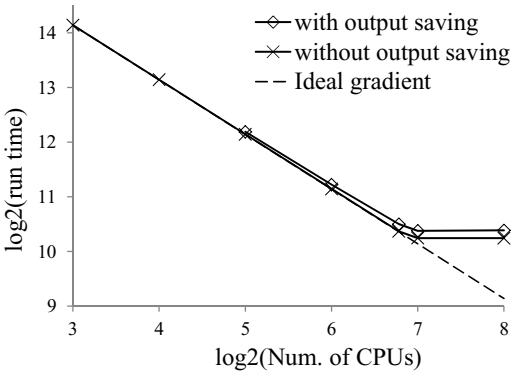


Fig. 1. Scalability of the SRA module

As shown in Fig. 1, the modified parallel extension of SRA module exhibits near ideal scalability, up to 110 CPUs. The same graph shows that saving 84 GB of output data with MPI-IO has little impact on the scalability, even though the cluster does not have supporting file system for MPI-IO. On the same hardware, another simulation is conducted to find the scalability issues due to the use of SysV IPC for inter-process communication; OCM model is called as an independent executable and SysV IPC resources are used for data exchange. It is found that the use of SysV IPC has no noticeable effect on the scalability. This confirms that SysV IPC resources can be used for calling legacy FORTRAN codes in parallel environments, without any impact on scalability. Surely, some work is necessary to introduce SysV IPS resources to FORTRAN codes. However, compared to conversion to a library, it is less error prone and requires much less time.

The sudden changes of the graphs in Fig. 1 to a constant, after 110 CPUs, indicates the inadequacy of task level parallelism. For complex SRA methods like OCM and fiber element method, run time for large buildings is 2-4 orders of magnitude larger compared to that for small buildings. Further, these complex

SRA methods involve hours of runtime for large buildings. One of the buildings in this demonstration involves 1211 seconds run time, which is almost equal to the run time with 110 CPUs. Hence the graphs in Fig. 1 flatten.

Parallelizing the SRA codes is necessary to break the above mentioned performance barrier. With parallelized SRA codes, CPU subgroups of different sizes have to be formed such that resources are optimally used. All the small buildings are executed as serial codes in one large CPU group while the other groups execute bigger buildings in parallel. Such strategy makes it possible to simulate a large urban area in a single simulation.

Figure 2 shows a snapshot of seismic response of 145,000 buildings in central Tokyo; assumed structural skeletons and material properties are used. Colors represent the magnitude of displacement vector; red color indicates displacements exceeding $0.5m$.

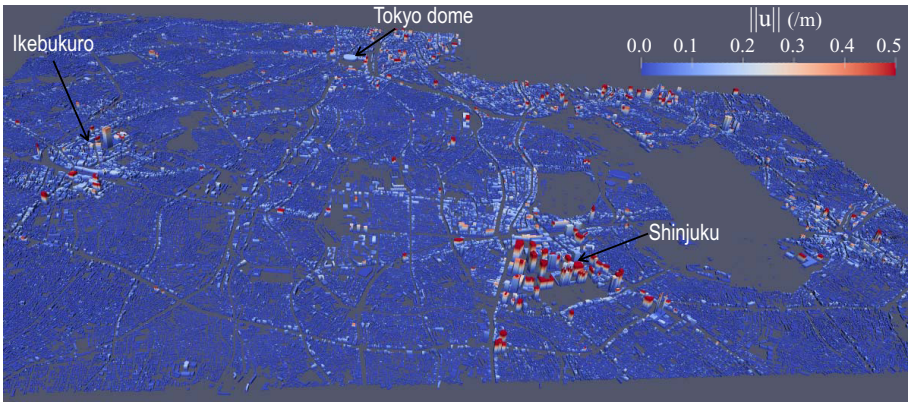


Fig. 2. Snapshot from a seismic response analysis demonstartion of central Tokyo

3 Parallel Performance Enhancements of Evacuation Module

To simulate emergency evacuation of millions of people in a large urban area like Tokyo, it is necessary to develop a scalable parallel extension of the evacuation module. The existing MAS based pedestrian simulation studies have reported low scalability which is limited to a few tens of CPU cores; the objective of majority of the existing studies is simulating virtual worlds, for which a few tens of CPU cores may be sufficient. With several strategies for hiding and minimizing communications, we attained near liner scalability at least up to 2048 CPU cores, with a few millions of agents. The rest of this section provides a short description of the abilities of current agents, difficulties in parallelization and main strategies used to attain high parallel scalability.

3.1 Complexity of Agents

Since the evacuation simulations are concerned with human lives, it is necessary to use smart agents which can reproduce the observed behaviors of real crowds. Modeling smart agents involves complex data structures consisting of a large number of variables. While the current agents are simple, we are implementing the required abilities that are important for tsunami triggered emergency evacuations. To model the heterogeneous real human crowds, a wide range of agents are implemented: agents with different physical abilities like speed, maximum sight distance, etc.; agents with different amount of information related to the environment and the emergency scenario; agents with different levels of responsibilities like police officers, fire fighters, volunteers and common residents. Figure 3 shows a part of the UML diagram for a resident agent. Another two types of agents, officials and non-residents, are implemented similarly by specializing the same _Agent base class.

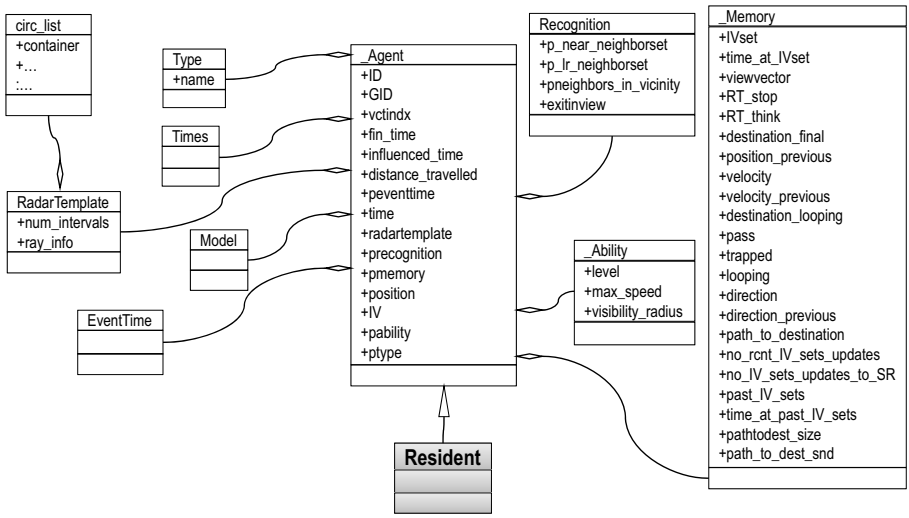


Fig. 3. Class hierachy of a resident agent

All the types of agents have common navigation behavior, though the different type of agents may take different actions depending on the role they play. As an example, official agents seek for the other agents requiring support while resident agents move to nearest evacuation center. With *See*() functionality, an agent makes a high resolution scan of his visible environment like a radar, and identify the boundary of visibility and visible neighboring agents. Next, with *Think*() functionality, he analyzes the area and the boundary of his visibility and identify the available paths and choose the closest path to his destination direction. Finally, with *Move*() functionality, he navigates avoiding collisions with neighboring agents and other obstacles.

3.2 Difficulties in Parallelization

The major steps in parallelizing the multi agent code are the same as that of other particle type simulations like SPH. However, compared to other particle type methods, parallelization of the multi agent code involves several additional difficulties, some of which are listed below.

1. Complexity of data structures and the amount of data involved
   (a) The volume of agent's data is 50 times or more compared to SPH.
   (b) Agents have dynamically growing data like memory of their experiences, etc.
   (c) Implementing smart agents requires the use of complex data structures like graphs, maps and trees, which grow in size.
2. Objects of different types of agents, like officials, residents, etc., have to be stored in non-contiguous locations in different vectors.
3. Require maintaining a wide ghost layer of thickness at least equal to the maximum visibility distance of an agent.
4. Amount of computations depends on the type and surrounding conditions of an agent.

While all the first three items increase the communication time, the last item leads to load imbalance. The agent data to be communicated are located in non-contiguous memory locations; the hierarchical data structure shown in Fig. 3 requires byte padding for the alignment of base class objects and for the sake of performance. Item 2 makes the data to be communicated become further fragmented and non-uniform; it is not possible to store agents to be sent and received to each CPU in a continuous memory stretches, and one may prefer not to delete inactive agents for performance reasons. In order to preserve the continuity, it is necessary to maintain a ghost layer of width at least equal to the largest sight distance of agents. In a dense urban area, ghost layer of $50m$ may contain a large number of people. Communicating a larger volume of fragmented data always is associated with increase in communication time[6]. Hence the first three items increase the communication time. The presence of dynamically growing data further increases the communication time; it requires at least two messages, memory allocation at the receiving end and packing and unpacking of data.

3.3 Strategies for Enhancing Scalability

The basic strategies used in parallelization of the multi agent code are more or less the same as that of other particle type simulations: the domain is decomposed such that each has equal work loads; ghost or overlapping layer is maintained and updated with the neighboring CPUs to preserve the continuity; agents are moved to neighbor CPUs when they enter the domain of a neighbor CPU; domain is repartitioned when the agent movements bring significant load imbalance. For domain decomposition, kd-tree is used. Although kd-tree does not minimize the volume of data being communicated, its simple geometry

makes it possible to easily detect the movements of agents between different domains. In addition to these common strategies, the following strategies are used to deal with the above mentioned additional difficulties.

Virtual CPU Topologies. With 2D-tree based partitioning we cannot take the advantage of communication topologies of underlying hardware like hypercube, torus, etc.; the resulting communication patterns of kd-tree is too irregular to be mapped to these structured hardware topologies. Distributed graph topology interfaces of MPI-2.2 address this problem[7]. We used *MPI_Dist_graph_create*() to map MPI process ranks to processing elements, to better match the communication pattern of the partitions to the topology of the underlying hardware. However, we could not test the effectiveness of virtual topology due to the unavailability of a cluster supporting this feature.

Algorithm 1.

```
comm_freq = ghost_update_interval;
for k = 1 to n do
    Execute send agents;
    if (!(k%comm_freq)) then
        Initialize ghost update;
    end
    Execute inner agents;
    if (!(k%comm_freq)) then
        Finalize ghost update;
    end
end
```

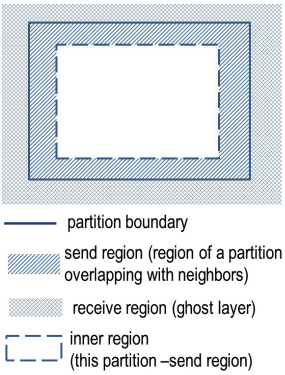


Fig. 4. Subdivisions of a partition

Hiding Communications and Minimizing Volume of Data Exchanged. As mentioned in the Section 3.2, communication of large volume of fragmented data is time consuming. However, most of the communication time can be hidden behind calculations by processing the agents in a certain order (see Algorithm 1). To this end, agents in each CPU are divided into three sub groups (see Fig 4); agents to be received from other CPUs, agents to be sent to other CPUs and the rest of the active agents which are named inner most agents hereafter. To deal with the large amount of data to be communicated, only the necessary agent data is exchanged. The three agent sub groups are stored in *std::map<>* C++ containers instantiated with *std::pair< global ID, _Agent * >*. The map data structure makes it efficient to manage agent movements among CPUs and repartitioning.

The presence of dynamically growing data makes it difficult to eliminate the communication overhead completely. Exchanging all the dynamic data requires at least two messages and packing and unpacking of large amount of data. Instead, in a single message, only the newly added contents are exchanged when updating ghost boundaries; new updates are packed to a small temporary buffer in agent objects, sent with the static data members in one message, and unpacked at the receiving end. The explicit packing and unpacking makes it impossible to hide the communication overhead completely. However, a significant portion of communication overhead is hidden, making the code to attain high parallel scalability.

Reduction of the Frequency of Ghost Layer Updates. Even if the ghost update communications are hidden, packing and unpacking dynamically growing data, etc. incur some time. Therefore, further gain in scalability is possible by reducing the frequency of ghost layer updates. This introduces small error to the simulation. However, the error is negligibly small; the time increment used is $0.2\,s$. Table 1 shows that the reduction of ghost update frequency has a significant advantage only with smaller number of CPUs. With the increasing number of CPUs, its effect diminishes; the overhead of packing and unpacking data goes down with the decreasing number of agents. Therefore, this approach is effective only when CPUs have a large number of agents in the ghost regions.

Table 1. Comparison of runtimes with *ghost_update_interval*'s of 1 and 2

CPUs	Runtimes with /(s)		Difference
	comm_freq=1	*comm_freq*=2	
4	16701.7	15103.2	1598.5
8	7765.6	7025.6	740.0
16	3243.5	3170.5	73.0
32	1701.3	1694.4	6.9

Minimizing Data Exchange in Repartitioning. Migration of agents from a partition to another brings load imbalance. When a significant load imbalance occurs, repartition is necessary to maintain equal workloads. Repartitioning is an expensive step since sophisticated agents have large amount of data. With 2D-tree, it is observed that most of the agents remain in the same CPU even after repartitioning, unless *MPI_Dist_graph_create*() maps a partition to a different CPU. The repartitioning algorithm detects whether a partition is assigned to the same CPU and exchanges only the newly assign agents. This drastically reduces the communication overhead involved with repartitioning, effectively lowering any performance degeneration due to repartitioning. Table 2 compares run times without any repartitioning and with 4 repartitioning (once in 80 steps). As is seen, even with the current serial 2d-tree algorithm, the gain due to repartitioning is significantly increasing with the number of CPUs. Figure 6, about

which a detailed explanation is given later, demonstrate the gain due to repartitioning. Instead of calling repartitioning at fixed intervals, load in each CPU has to be monitored and repartitioning has to be called based on the level of load imbalance.

Table 2. Effectiveness of repartitioning

CPUs	Runtime /(s)		Gain /(%)
	no repartition	4 repartitions	
32	1733.9	1656.7	4.6
64	939.1	866.5	8.4
128	459.8	434.1	5.9
256	250.9	220.6	13.7

3.4 Scalability

In order to test the effectiveness of the above major strategies, we conducted a series of simulations with 500,000 agents in a part of Kochi city environment. The same cluster used for the scalability test of the SRA module is used for these simulations. Ghost layer updating at each time step and repartitioning at an interval of 80 steps are considered for these simulations. As Fig. 5 indicates, the above strategies have produced near linear scalability; super linear behavior is due to the nonlinear time reduction of the neighbor search algorithm with the decreasing number of agents.

Further, preliminary tests in the K-computer produced 94% strong scalability with 2048 CPU cores; strong scalability is defined as $\frac{(\frac{T_m}{T_n})}{(\frac{n}{m})}$, where T_k is the time with k number of CPU cores and $n \geq 2m$. For the tests on the K-computer, 400 time steps with 2 million of agents are considered. Further the ghost boundaries are updated at each iteration, movements of agents between CPU cores are checked at each 10 iterations and domain is repartitioned at each 100 iterations for load balancing. Figures 6a and 6b show the runtime for 400 iterations, except the repartitioning time. These figures clearly show the significant performance gain due to repartitioning. As is seen, at each iteration, the run time with 2048 cores is nearly the half of that of 1024 cores.

Figure 7a shows the history of total run time with 2048 CPU cores. As is seen the major bottleneck in the current code is repartitioning, which is still a serial code. Further, as shown in Fig. 7b, migration of agents (detecting agent movements from the domain of one CPU to another and dispatch those agents to the new CPU) is relatively time consuming. In future developments, these two bottlenecks are to be addressed to further increase the scalability.

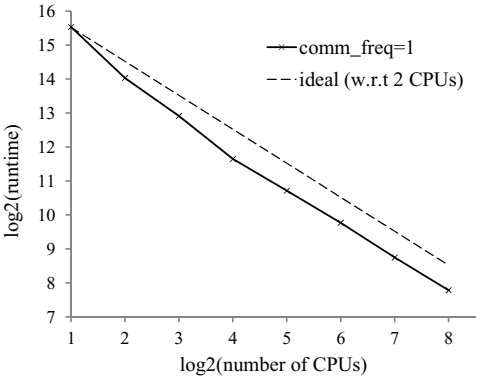
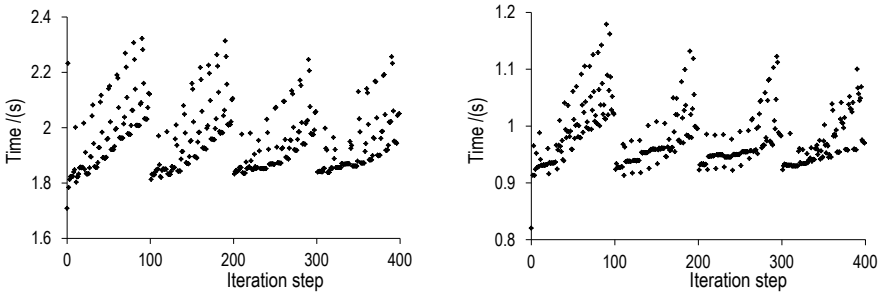
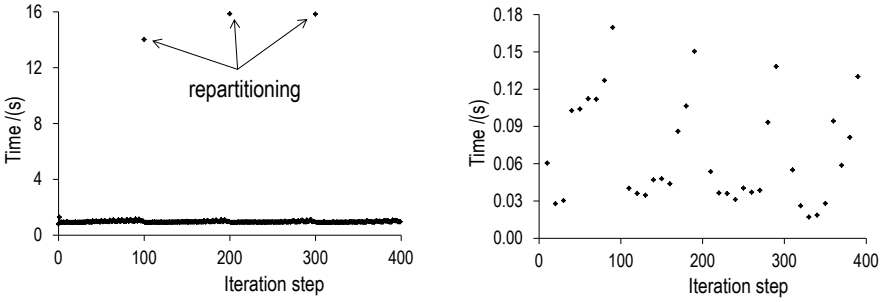


Fig. 5. Scalability of the multi agent code



(a) run time with 1024 CPU cores (b) run time with 2048 CPU cores

Fig. 6. History of run time with 1024 and 2048 cores. Repartitioning time is exculded.



(a) history of total run time (b) time for moving agents among domains

Fig. 7. History of total run time and agent migration time with 2048 CPU cores

4 Summary

Scalable parallel extensions for the seismic response analysis and multi-agent based evacuation modules of an Integrated Earthquake Simulator (IES) are being developed. Task parallelism is considered for the SRA module and near ideal scalability is attained with a static load balancer. In future, parallelization of the SRA codes is necessary to address the scalability limitations due to the presence of long time consuming tasks (i.e. large buildings). Further, out of a given number of CPUs, forming CPU subgroups of different sizes to execute the parallel SRA codes with minimum waste of CPU time and task scheduling have to be considered.

The multi agent based evacuation simulation module produced high strong scalability with a few thousands of CPU cores in the K-computer. In future, further scalability improvements of evacuation module are planned with direct remote memory access features of MPI. Such enhancements are necessary to cope with complexities arising from planned sophisticated agent features and introduction of different types of agents.

Acknowledgements. This work was supported by JSPS KAKENHI Grant Number 24760359. Part of the results is obtained by using the K computer at the RIKEN Advanced Institute for Computational Science.

References

1. Hori, M., Ichimura, T.: Current state of integrated earthquake simulation for earthquake hazard and disaster. J. of Seismology 12(2), 307–321 (2008)
2. Sobhaninejad, G., Hori, M., Kabeyasawa, T.: Enhancing integrated earthquake simulation with high performance computing. Advances in Engineering Software 42(5), 286–292 (2011)
3. Cosenza, B., Cordasco, G., De Chiara, R., Scarano, V.: Distributed Load Balancing for Parallel Agent-Based Simulations. In: 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 62–69 (2011)
4. Richard Stevens, W.: UNIX Network Programming. Interprocess Communications 2 (1999) ISBN 0-13-081081-9
5. Latham, R., Ross, R., Thakur, R.: The impact of file systems on MPI-IO scalability. In: Kranzlmüller, D., Kacsuk, P., Dongarra, J. (eds.) EuroPVM/MPI 2004. LNCS, vol. 3241, pp. 87–96. Springer, Heidelberg (2004)
6. Balaji, P., Buntinas, D., Balay, S., Smith, B., Thakur, R., Gropp, W.: Nonuniformly Communicating Noncontiguous Data: A Case Study with PETSc and MPI. In: Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, March 26-30 (2007)
7. Hoefler, T., Rabenseifner, R., Ritzdorf, H., de Supinski, B.R., Thakur, R., Traff, J.L.: The Scalable Process Topology Interface of MPI 2.2. Concurrency and Computation: Practice and Experience 23(4), 293–310 (2010)