Coarse Grain Task Parallelization of Earthquake Simulator GMS Using OSCAR Compiler on Various cc-NUMA Servers

Mamoru Shimaoka[1], Yasutaka Wada[12], Keiji Kimura[1], and Hironori Kasahara[1]

[1] Advanced Multicore Processor Research Institute, Waseda University,
27, Waseda, Shinjuku, Tokyo, Japan
shimaoka@kasahara.cs.waseda.ac.jp,{keiji,kasahara}@waseda.jp
http://www.kasahara.elec.waseda.ac.jp/

[2] Dept. of Information Science, Meisei University,
3-1-1, Hodokubo, Hino, Tokyo, Japan
yasutaka.wada@meisei-u.ac.jp

Abstract. This paper proposes coarse grain task parallelization for a earthquake simulation program using Finite Difference Method to solve the wave equations in 3-D heterogeneous structure or the Ground Motion Simulator (GMS) on various cc-NUMA servers using IBM, Intel and Fujitsu multicore processors. The GMS has been developed by the National Research Institute for Earth Science and Disaster Prevention (NIED) in Japan. Earthquake wave propagation simulations are important numerical applications to save lives through damage predictions of residential areas by earthquakes. Parallel processing with strong scaling has been required to precisely calculate the simulations quickly. The proposed method uses the OSCAR compiler for exploiting coarse grain task parallelism efficiently to get scalable speed-ups with strong scaling. The OSCAR compiler can analyze data dependence and control dependence among coarse grain tasks, such as subroutines, loops and basic blocks. Moreover, locality optimizations considering the boundary calculations of FDM and a new static scheduler that enables more efficient task schedulings on cc-NUMA servers are presented. The performance evaluation shows 110 times speed-up using 128 cores against the sequential execution on a POWER7 based 128 cores cc-NUMA server Hitachi SR16000 VM1, 37.2 times speed-up using 64 cores against the sequential execution on a Xeon E7-8830 based 64 cores cc-NUMA server BS2000, 19.8 times speed-up using 32 cores against the sequential execution on a Xeon X7560 based 32 cores cc-NUMA server HA8000/RS440, 99.3 times speed-up using 128 cores against the sequential execution on a SPARC64 VII based 256 cores cc-NUMA server Fujitsu M9000, 9.42 times speed-up using 12 cores against the sequential execution on a POWER8 based 12 cores cc-NUMA server Power System S812L.

Keywords: earthquake,GMS,OSCAR,task parallelism,compiler,cc-NUMA

1 Introduction

Earthquake simulation that simulates the propagation of seismic waves from hypocenters is important for minimizing the damage by natural disasters. Earthquake wave propagation is often formulated as wave equation, which is approximated by Finite Difference Method (FDM) or Finite Element Method (FEM). The precise simulation usually requires huge calculation time, studies of earthquake simulation have been trying parallelization of the program. Akcelik *et al.*[1] proposed an FEM earthquake simulation method parallelized by MPI. Their parallelized Simulator using 3000 processor cores showed 80% parallel efficiency in weak scaling on the AlphaServer SC at the Pittsburgh Supercomputing Center (PSC). Aoi *et al.*[4] proposed the Ground Motion Simulator (GMS) and parallelized the GMS with GPGPU. They showed the parallelized GMS using 1024 nodes obtained 1028 times speed-up compared to 1 node in weak scaling on the TSUBAME2.0 in Tokyo Institute of Technology. Tiankai *et al.*[3] proposed the parallel octree meshing tool Octor and showed the evaluations of the parallel Partial Differential Equation (PDE) solver using octree mesh by the Octor on the AlphaServer SC at the PSC. They showed the solver using 2000 processor cores could speed-up earthquake simulation 13 times faster than that of using 128 processor cores in strong scaling. Those works achieve high parallel efficiency by hand parallelization. The hand parallelization needs deep knowledge of parallelization and long development periods and costs. Moreover, most existing studies achieve high parallel efficiency with weak scaling, but high parallel efficiency with strong scaling is more desirable than that with weak scaling. In these days, cache coherent Non Uniform Memory Architecture (cc-NUMA) is common architecture, this architecture requires additional tuning compared to Uniform Memory Architecture. Therefore, parallelization that is efficient on cc-NUMA by an automatic parallelizing compiler is expected for productivity and performance.

This paper proposes a parallelization method that includes modifying of a sequential earthquake simulation program into a compiler friendly sequential program to assist automatic parallelization of the OSCAR multigrain parallelizing compiler[5][6]. Unlike the OSCAR multigrain parallelizing compiler, commercial compilers such as Intel Compiler and IBM XL compiler can utilize only loop parallelism. Slight sequential parts prevent us from achieving scalable speed-up in many core architecture. Therefore, multigrain parallelism offered by the OSCAR compiler is important.

In this paper, the proposed method parallelizes the earthquake simulator GMS, coarse grain task parallelism, as well as loop parallelism, is used. A locality optimization considering the boundary calculations of FDM, a locality optimization considering First Touch all over the program and an efficient task scheduling on servers using First Touch policy help to us get strong scaling speed-up.

The remainder of this paper is organized as follows. Section 2 introduces the earthquake wave propagation simulator GMS. Section 3 shows the proposed parallelization method. Section 4 gives speed-ups on five different cc-NUMA servers.

The servers consist of the SR16000 VM1 (henceforth SR16000), the BS2000, the HA8000/RS440 (henceforth RS440), the SPARC Enterprise M9000 (henceforth M9000) and the IBM Power System S812L (henceforth S812L). Finally, section 5 provides the conclusion.

2 The Ground Motion Simulator GMS

For effective disaster prevention planning, the importance of precise earthquake simulations is increasing. The Ground Motion Simulator (GMS) is the earthquake simulator developed by Aoi, Fujiwara in the NIED, and the GMS can precisely simulate for Japanese ground structure that we can download at J-SHIS[2]. The GMS consists of parameter generation tools, computation visualization tools and a wave equation solver, and we can download it from the URL in [7].

The GMS solves the wave equations in 3-D heterogeneous structure, and it uses Finite Difference Method to approximate the wave equations. One of the characteristics of the GMS solver is the use of staggered grids. For computation accuracy, grid points for displacement are shifted from grid points for stress a half grid in staggered grids. In staggered grids, second order difference operator is (1).

$$f`_i \simeq \frac{f_{i+1/2} - f_{i-1/2}}{\Delta x} \tag{1}$$

Fourth order difference operator that is higher accuracy than second order difference operator is (2).

$$\begin{aligned} f`_i \simeq \big(&-1/24 f_{i+3/2} + 9/8 f_{i+1/2} \\ &-9/8 f_{i-1/2} + 1/24 f_{i-3/2}\big)/\Delta x \end{aligned} \tag{2}$$

Besides, the GMS solver uses discontinuous grids to accelerate the simulation. In discontinuous grid, as shown in Fig.1, grids of near the earth's surface or Region I is three times smaller than that of a deeper region or Region II. It is because the grid spacing has to be smaller for precisely simulating waves of shorter wavelength. In the grids near the surface, the wavelength is shorter than that of the deeper region. By using discontinuous grid replace of uniform grid, the GMS solver reduces calculation for the deeper region.

In brief, the GMS solver is to calculate velocity and stress of each grid and each step by using external force as inputs.

In the GMS solver, external force can be added as velocity or stress and second order difference operator or fourth order difference operator can be used. This paper deals with the GMS solver in which external force is added as stress and fourth difference operator is used.

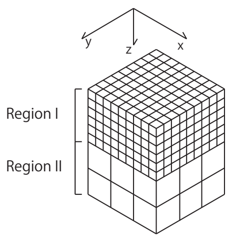


Fig. 1. Discontinuous grid in the GMS

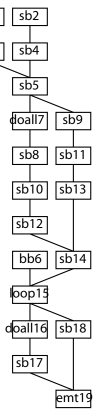


Fig. 2. Macro task graph of the GMS main loop

3 Coarse grain task parallelization of the GMS

This section proposes a parallelization method for the GMS. Before parallelization, the sequential GMS solver written in Fortran 90 is changed into a sequential FORTRAN 77 program. It is because the OSCAR compiler just supports FORTRAN 77 and the GMS uses Fortran 90 to use the I/O library HDF[8] though main parts are written in FORTRAN 77.

3.1 Coarse grain task parallelization

This section shows how the OSCAR compiler[5][6] exploits parallelism in a program. The OSCAR compiler can exploit multigrain parallelism that uses loop parallelism, coarse grain task parallelism and statement level fine grain parallelism considering its parallelism. Coarse grain task parallelism in the OSCAR compiler means parallelism among three kinds of coarse grain tasks, namely Basic Blocks (BBs), Repetition Blocks (RBs) and Subroutine Blocks (SBs).

First, the OSCAR compiler decomposes a sequential source program to macro tasks in each nested level hierarchically. Then it makes macroflow graphs which represent data dependency and control flow among the macro tasks. Next, it analyzes and detects parallelism in the macroflow graphs by using Earliest Executable Condition analysis[5] that analyzes the simplest forms of conditions the macro tasks may start their execution considering control dependencies and data dependencies, and then generates macro task graphs. Next, it analyzes and detects parallelism in the macroflow graphs by using Earliest Executable Condition analysis[5] and then generates macro task graphs. Earliest Executable Condition analysis is to analyze the simplest forms of conditions the macro tasks may start their execution considering control dependencies and data dependencies. Macro task graphs represent parallelism among macro tasks. If the macro task graph has only data dependency edge, the macro tasks are assigned by static scheduling to processors or processor groups that are grouped logically by the compiler for hierarchical coarse grain task parallelization. If the macro task graph has any control dependency edges, the macro tasks are assigned to processors or processor groups at runtime by a dynamic scheduler. The dynamic scheduler is generated by the OSCAR compiler exclusively for the program[5] and embedded into the parallelized program automatically. Finally, the OSCAR compiler generates a parallelized Fortran program using the OSCAR API Ver2.0[11], which the ordinary product OpenMP compilers provided for the target machines can compile.

3.2 Modification of the GMS

Fig.2 shows the macro task graph in the main loop of the GMS. The macro task graph was generated by the OSCAR compiler and has 18 macro tasks and one exit task representing the end of the macro task graph. Solid edges in macro task graph represent data dependencies among macro tasks and broken edges in macro task graph represent control dependencies. There is parallelism among coarse grain tasks such as parallelism between SB3 and SB4 in Fig.2. It is because of discontinuous grids of the GMS. In discontinuous grids, We can execute velocity calculation of the near surface grids or SB3 and velocity calculation of the grids in the deeper area or SB4 in parallel. After that, the boundary of the near surface grids and the grids in the deeper area is executed in SB5. There is similar parallelism for stress calculations. We can execute stress calculation of the near surface grids or SB11 and stress calculation of the grids in deeper area or SB12 in parallel.

Next, to increase coarse grain task parallelism, inline expansion is applied to all subroutines, or SBs in Fig.2, in the main loop. Fig.3 is the macro task graph with 131 macro tasks for the main loop after the inline expansion of all subroutines. We extract very large coarse grain task parallelism as shown in Fig.3. It is because coarse grain task parallelism inside the subroutines are taken out to the main loop level. By the inline expansion, task parallelism among the tasks in the SBs with dependency can be used. LOOP3 in Fig.3 is originally in SB1 in Fig.2, and DOALL10 in Fig.3 is originally in SB3 in Fig.2. Though SB1 and SB3 in Fig.2 have dependency among them, LOOP3 and DOALL10 in Fig.3 have no dependency among them. 60 macro tasks are analyzed to be DOALL or parallel loop in in Fig.3. Since we can split each DOALL loop into parallel macro tasks, much larger coarse grain task parallelism can be exploited.

Besides, to enhance loop parallelism and spatial locality, loop interchange and array dimension interchange are applied.

3.3 Data distribution to distributed shared memories using First Touch

In cc-NUMA machines, how to distribute variables to memories is important to get good performance. Usually, cc-NUMA machines use first touch policy[12].

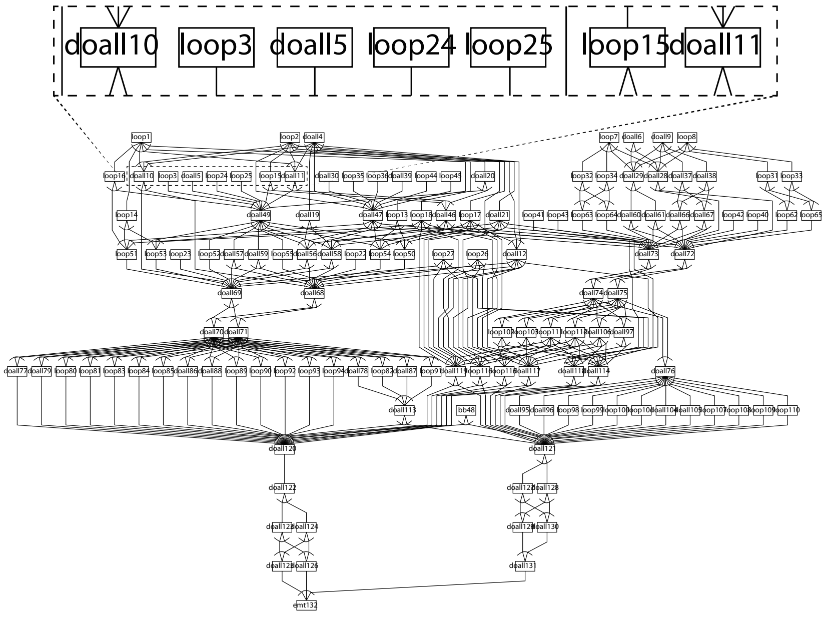


Fig. 3. Macro task graph after inline expansion

On first touch policy, a page is allocated to the memory nearest to the processor that first touched the page.

The GMS solver uses the Hierarchical Data Format (HDF) library[8] for file access. The HDF library is to allow us to manage large and complex data collections. The master thread executing the library first touches all input arrays of the original GMS solver. It forces cc-NUMA machines to assign those arrays to the distributed shared memory near the processor core that execute the master thread. It means that all processor cores access to the distributed shared memory near the processor core executing the main thread, and the heavy memory contention occurs.

To fully utilize distributed shared memories on cc-NUMA machines, in the proposed method, the input arrays are copied to new arrays with interchanged indexes to be first touched by each processor element. Fig.4 shows an example of the modification. Originally, an array A is first touched in a subroutine external_library_array_init and is used in a subroutine main_loop. Because the OSCAR compiler can't parallelize external library, a new array A_COPY is created and values of the array A are copied to the array A_COPY. Then, the subroutine main_loop uses the array A_COPY in place of the array A. In the GMS, 33 arrays are copied to be first touched by each processor element.

```
program sample
  integer A(1000)
  integer A_COPY(1000)
  {copied array}
  call external_library_array_init(A)
  {a array is originally first touched here}
  do i=1,1000
    A_COPY(i)=A(i)
  enddo
  {copying the original array to a new array}
  call main_loop(A_COPY)
  {in main loop, the new array is used}
  do i=1,1000
    A(i)=A_COPY(i)
  enddo
  {copying the new array to the original array}
  call output_A(A)
end
```

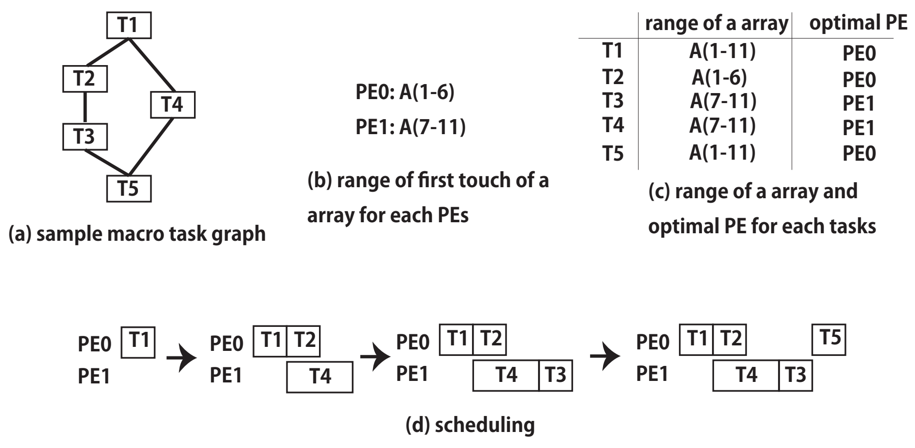
Fig. 4. Example of the array copy for first touch

3.4 Task Scheduling on cc-NUMA

The control dependencies in the macro task graph are represented as broken edges between tasks. There is no control dependency edge in Fig.3. Therefore, the OSCAR compiler chooses static scheduling to schedule the macro tasks to processor elements.

On cc-NUMA machines, access to a remote distributed shared memory is slower than that of a local distributed shared memory. So to improve the efficiency of parallel processing of the program, a scheduler that takes accounts of the first touch information was developed. By first touching the copied new arrays mentioned above, the arrays used for the main loop are first touched at the each copy loop, so the scheduler can know which processor element first touched the array. The static scheduler decides optimal processors to execute for each task using the first touch information, and then schedule ready tasks to its optimal processors in order of critical path length. Critical path length is the length of the longest path from any node to the exit node on a macro task graph.

Fig.5 is an example of the scheduling. Fig5(a) is a macro task graph and (b) shows how PEs first touch variables. Fig5(c) represents the range of arrays used by each task and the optimal PEs to which each task should be assigned considering the information of the first touch showed in (b). Finally Fig5(d) shows processing steps of the scheduling. In the third step of (d), the task T3 is assigned to PE1. The task T3 is not dependent on T4, so if the task T3 is assigned to PE0, the task T3 may start soon after the task T2 ended. But if the task T3 is assigned to PE0, access to a remote distributed shared memory would occur, so the scheduler assigns the task T3 to PE1. The scheduler restricts the tasks to be assigned to the optimal PE considering the first touch to reduce memory access overheads.


T1
T2
T3
T4
T5
(a) sample macro task graph
PE0: A(1-6)
PE1: A(7-11)
(b) range of first touch of a array for each PEs
(c) range of a array and optimal PE for each tasks
PE0
PE1
(d) scheduling

	range of a array	optimal PE
T1	A(1-11)	PE0
T2	A(1-6)	PE0
T3	A(7-11)	PE1
T4	A(7-11)	PE1
T5	A(1-11)	PE0

Fig. 5. An example of the scheduling

3.5 Locality optimization of boundary calculations in FDM

Fig.6 is a source code of velocity calculation of the center grids or DOALL10 in Fig.3 and that of boundary grids or DOALL11 in Fig.3. The GMS use fourth order difference operator for FDM calculations. But the fourth order difference operator can't be used at the boundary of the grids in the GMS. Therefore, second order difference operator is used at the boundary. The DOALL10 and the DOALL11 have no dependency among them, but both loops access the almost same ranges of the arrays taking account of cache lines. Though cache reuse is expected by executing the both loops continuously[9][10], the arrays used by the both loops are too large to be fully stored in L2 or L3 caches.

In this section, the loop fusion is applied to the both loops to optimize the locality. To focus on ux in Fig.6, the DOALL10 uses ux(2 : nk-2, 2 : nj-1, 2 : ni-1), and the DOALL11 uses ux(1, 2 : nj-1, 2 : ni-1) and ux(nk-1, 2 : nj-1, 2 : ni-1). Though the ranges of the array ux of the first loop don't overlap with that of the second loops, it is expected that ux(1,j,i) and ux(2, j, i) are allocated in the same cache line. The same is true of ux(nk-2, j, i) and ux(nk-1, j, i). By the loop fusion taking account of cache lines, memory access of the boundary calculation in FDM is expected to be sharply optimized.

Fig.7 is the macro task graph of the main loop after loop fusion. The proposed method fuses 12 loops into the four loops.

3.6 Generated compiler friendly sequential program and its parallel compilation

The proposed method applies above-mentioned modifications to the sequential GMS program. The modified sequential program is compiled by the OSCAR compiler and changed into parallelized Fortran program using the OSCAR API Ver2.0[11]. The OSCAR API is compatible with OpenMP. Therefore, compilers

```
{calculation of the center area}
do i=2,ni-1
 do j=2,nj-1
  do k=2,nk-2
   ux(k,j,i)=(ux(k,j,i)+bbx*(
            +dtdx*(c0*(sxx(k,j,i+1)-sxx(k,j,i))
                 - c1*(sxx(k,j,i+2)-sxx(k,j,i-1)))
            +dtdy*(c0*(sxy(k,j,i)-sxy(k,j-1,i))
                 - c1*(sxy(k,j+1,i)-sxy(k,j-2,i)))
            +dtdz*(c0*(sxz(k,j,i)-sxz(k-1,j,i))
                 - c1*(sxz(k+1,j,i)-sxz(k-2,j,i))))
            )*aaqq
  enddo
 enddo
enddo
{calculation of the boundary}
do i=2,ni-1
 do j=2,nj-1
  do k=1,nk-1,nk-2
   ux(i,j,k)=( ux(i,j,k)+bbx*(
             +dtdx*(c0*(sxx(k,j,i+1)-sxx(k,j,i))
                  - c1*(sxx(k,j,i+2)-sxx(k,j,i-1)))
             +dtdy*(c0*(sxy(k,j,i)-sxy(k,j-1,i))
                  - c1*(sxy(k,j+1,i)-sxy(k,j-2,i)))
             +dtdz*(sxz(k,j,i)-sxz(k-1,j,i))  )
             )*aaqq
  enddo
 enddo
enddo
```

Fig. 6. Example of center and boundary calculations

provided for target cc-NUMA machines can compile the program with the OSCAR API to the executable binary. In this paper, IBM XL Fortran compiler, Intel Fortran compiler and Sun Studio Fortran compiler compile the generated parallel programs.

4 Performance of the parallelized GMS

This section evaluates speed-up of the parallelized GMS on five different cc-NUMA machines.

4.1 Evaluation Environments

The authors use the SR16000, the BS2000, the RS440, the M9000 and the S812L for the evaluations. Table 1 summarizes the specifications of the five servers.

The SR16000 is a POWER7 based 128 cores cc-NUMA machine. The SR16000 consists of four boards and the fully-connected network connects the four boards. Each board has four processors and the fully-connected network connects the four processors. The evaluations in Section.4.2,4.3, 4.4 use the SR16000. The authors bind the paralelized programs to the processor cores by the compact manner. The compact manner is to use processor cores in core number order.

The BS2000 is a Xeon E7-8830 based 64 cores cc-NUMA machine. The special feature of BS2000 is that it consists of four ordinary blade servers, however,

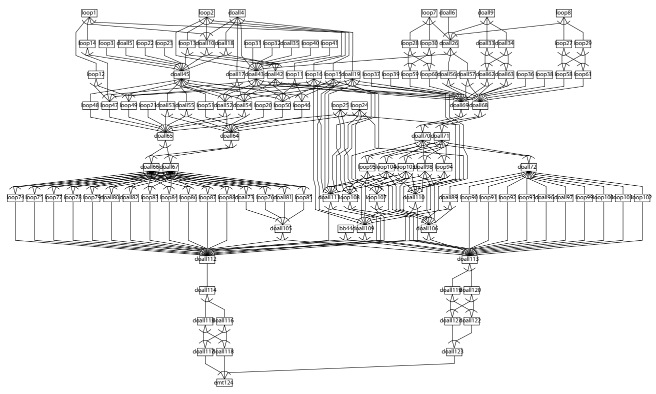


Fig. 7. Macro task graph after loop fusion

just attaching the inter-blade coherent control module connecting the blades, the blades is changed into a cc-NUMA server. Because each processor can use three QPIs for inter-processor connection, some pairs of the processor are connected directly and the other pairs are connected with one hop or two hops. The evaluations in Section.4.3 use the BS2000. The authors bind parallelized programs to the processor cores by the compact manner.

The RS440 is a Xeon X7560 based 32 cores cc-NUMA machine. The RS440 consists of four processors each of which has eight cores, and QPIs fully connect each processor. The evaluations in Section.4.3 use the RS440. The authors bind parallelized programs to the processor cores by the compact manner.

The M9000 is a SPARC64 VII based 256 cores cc-NUMA machine. The M9000 consists of 16 boards each of which has 16 cores. Two crossbar switches connect eight boards to make a cluster, and then two clusters are connected to compose the M9000. The evaluations in Section.4.3 use the M9000. The evaluations use up to 128 cores of 256 cores for the OSCAR compiler can cope with up to 128 cores at present. The authors bind parallelized programs to the every other processor core to utilize L2 cache memory and main memory fully.

The S812L is a POWER8 based 12 cores cc-NUMA machine. The S812L has a Dual Chip Module (DCM) and a Dual Chip Module includes two chip each of which has six cores[13]. The evaluations in Section.4.3 use the S812L. Though S812L has eight slots for DIMM modules, the authors equipped four 16GB DIMM modules to S812L.

The evaluations use three data sets such as Unit00420, Unit01680 and Unit06720. Table 2 summarizes the number of grids in the data sets. The Unit01680 is medium size among them and used for Section.4.2,4.3. The Unit00420 is the smallest data set among them and used for Section.4.4. The Unit06720 is the biggest data set among them and used for Section.4.4.

Table 1. Server Specifications

	SR16000	BS2000	RS400
CPU	POWER7	Xeon E7-8830	Xeon X7560
Frequency	4GHz	2.13GHz	2.27GHz
cores per 1 processor	8	8	8
L2 cache	256KB(1core)	256KB(1core)	256KB(1core)
L3 cache	32MB(1processor)	24MB(1processor)	24MB(1processor)
Processors	16	8	4
CPU cores	128	64	32
Memory	1TB	256GB	128GB
OS	RedHat Linux	RedHat Linux	Ubuntu
Version	6.4	6.1	14.04.1
Linux kernel version	2.6.32	2.6.32	3.13.0
Compiler	XL Fortran	Intel Fortran compiler	Intel Fortran compiler
Version	13.1	12.1.5	12.1.5

	M9000	S812L
CPU	SPARC64 VII	POWER8
Frequency	2.88GHz	3.026GHz
cores per 1 processor	4	12(1 DCM),6(1 chip)
L2 cache	6144KB(1processor)	512KB(1core)
L3 cache	none	96MB(1DCM),48MB(1chip)
Processors	64	1(DCM),2(chip)
CPU cores	256	12
Memory	512GB	64GB
OS	Solaris	RedHat Linux
Version	10	7.1
Linux kernel version		3.10.1
Compiler	Sun Studio Fortran compiler	XL Fortran
Version	12.1	15.1.1

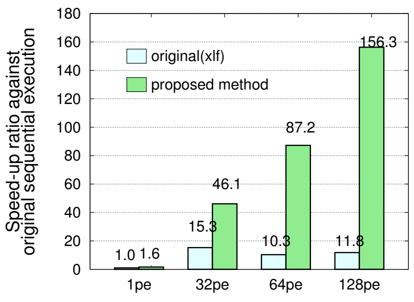
Table 2. Number of grids in Datasets

	Unit00420	Unit01680	Unit06720
Number of Grids in RegionI	$420 \times 420 \times 100$	$1680 \times 1680 \times 100$	$6720 \times 6720 \times 100$
Number of Grids in RegionII	$140 \times 140 \times 200$	$560 \times 560 \times 200$	$2240 \times 2240 \times 200$
Total Memory	0.8GB	12.2GB	195.2GB

4.2 Comparison of Commercial Compilers and the Proposed Method

The comparisons among the original GMS parallelized by commercial compilers provided for the servers, such as IBM XL Fortran compiler, Intel Fortran compiler and Sun Studio and the GMS parallelized by the proposed method are shown.

Fig.8 shows a summary of the comparison between XL Fortran compiler and the proposed method on the SR16000. On a one processor core, the sequential execution time by the proposed method is 1.65 times faster than the original sequential program. Speed-ups of the original GMS parallelized by XL Fortran compiler were 15.3 times using 32 cores, 10.3 times using 64 cores and 11.8 times using 128 cores. It means that XL Fortran compiler can find loop parallelism in the GMS, but it can't give us scalable speed-up for the GMS on 64PEs and 128PEs in the SR16000. Speed-up of the GMS parallelized by the proposed method using 128 cores was 156.3 times against the original sequential execu-



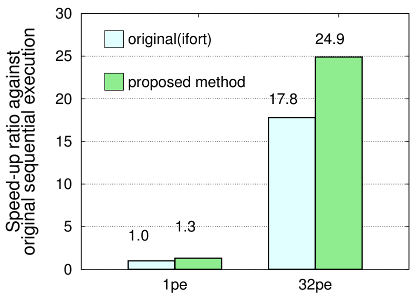


Fig. 8. XL Fortran compiler vs the proposed method on the SR16000 (Unit01680)

Fig. 9. Intel Fortran compiler vs the proposed method on the RS440 (Unit01680)

tion. Higher speed-up by the proposed method using 128 PEs is obtained. The first reason is that the parallelization by XL Fortran compiler can only utilize loop parallelism, besides the proposed method can utilize multigrain parallelism. The second reason is that the master thread first touches the most of arrays and those arrays are assigned to distributed shared memory near processor core that execute the master thread. Therefore, remote memory accesses of parallel execution by XL Fortran compiler occur frequently and the execution time gets long.

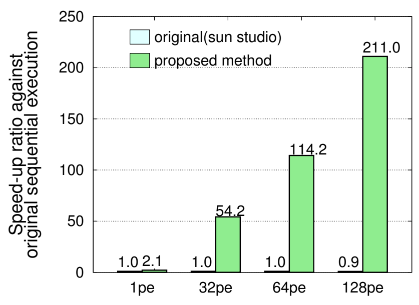
Fig.9 shows a summary of the comparison between Intel Fortran compiler and the proposed method on the RS440. The proposed method works 1.3 times faster than the original sequential execution. The speed-up ratio of Intel Fortran compiler using 32PEs is 17.8 times, it means that Intel Fortran compiler can also find loop parallelism in the GMS. On the RS440, loop parallelization works well. But on cc-NUMA with the bigger number of cores like the SR16000 and the M9000, the distance between the core and the remote memory becomes farther. The parallelization of the initialization of the arrays and the coarse grain task parallelization which consider First Touch is thought to be indispensable on cc-NUMA with the big number of cores.

Fig.10 shows a summary of the comparison between Sun Studio and the proposed method on the M9000. The proposed method gives us 2.1 times faster execution than the original sequential execution. Moreover, the proposed method using 128PEs gets 211 times speed-up from the original sequential execution.

In Fig.8, Fig.9, Fig.10, the sequential executions of the proposed method get speed-up from the original sequential executions. This is because the locality optimization by the loop fusion described in Sectuion.3.5.

4.3 Performance on the five different cc-NUMA servers

Speed-ups of the GMS parallelized by the proposed method from the sequential execution of the proposed method on the five different cc-NUMA servers are shown in Fig.11. Speed-ups of the GMS on the SR16000 was 94.9 times using



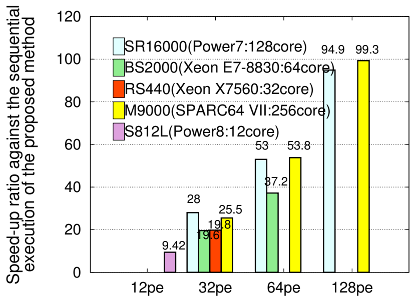


Fig. 10. Sun studio vs the proposed method on the M9000(Unit01680)

Fig. 11. SR16000 vs BS2000 vs RS440 vs M9000 vs S812L(Unit01680)

128 cores, that with 64 cores on the BS2000 was 37.2 times, that with 32 cores on the RS440 was 19.8 times, that with 128 cores on the M9000 was 99.3 times, and that with 12 cores on the S812L was 9.42 times.

The BS2000 and the RS440 are relatively inexpensive servers compared to the SR16000 and the M9000, memory bandwidth of the former two servers are relatively narrow compared to the latter two servers. Therefore, speed-ups by parallelization on the former two servers tend to be limited by the memory bandwidth.

The speed-up of S812L is 9.42 times using 12 cores against sequential processing. The parallel efficiency of the S812L using maximum core is $9.42 \div 12 = 78.5\%$, and it is higher than that of the RS440(61%) and that of the BS2000(58%).

On the SR16000 and the M9000, near 100 times speed-up using 128 cores can be obtained. It means that the proposed method successively utilize cc-NUMA machines.

4.4 Evaluations with various data sizes

Fig.12 summarizes the results of the evaluation with the various data sizes on the SR16000. The speed-ups on the Unit00420, a relatively small data set, were 25.0 times using 32 cores, 43.7 times using 64 cores and 75.7 times using 128 cores. Even on the smallest data set, over 64 times speed-up or half number of the cores used can be obtained. The speed-ups on the Unit06720 or the biggest data set were 21.7 times using 32 cores, 58.7 times using 64 cores and 110.7 times using 128 cores. Naturally, the results show that the bigger data size give us better speed-ups because of the smaller ratio of remote memory access in the whole execution.

5 Conclusions

This paper has proposed a parallelizing optimization method of the earthquake simulator GMS. We can use earthquake simulations for damage predictions of

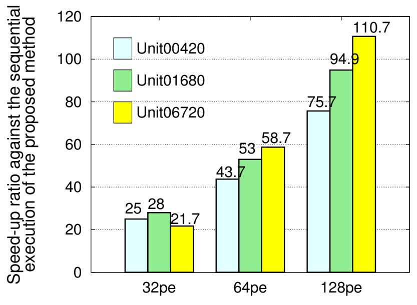


Fig. 12. Speed-up ratios of the proposed method with various data sets on the SR16000

earthquakes. By accelerating the earthquake simulations, it is expected that more exact damage prediction required for protecting more lives from disaster become possible. The proposed method modifies an original sequential Fortran program into parallelizing compiler friendly sequential Fortran program by hand to increase coarse grain task parallelism and data locality. The modifications by hand are the loop interchange and the array dimension interchange described in Section.3.2 and the array duplication described in Section.3.3 and the loop fusion described in Section.3.5. By the simple modifications, the OSCAR compiler can analyze coarse grain parallelism and data dependency among coarse grain tasks and generate a portable parallel program. In the proposed method, once users modify the original program into parallelizing compiler friendly sequential program, no further work is required to port to another shared memory servers.

The performance evaluations show 110.7 times speed-up using 128 cores against the sequential execution on the POWER7 based 128 cores cc-NUMA server Hitachi SR16000 VM1, 37.2 times speed-up using 64 cores against the sequential execution on the Xeon E7-8830 based 64 cores cc-NUMA server BS2000, 19.8 times speed-up using 32 cores against the sequential execution on the Xeon X7560 based 32 cores cc-NUMA server HA8000/RS440, 99.3 times speed-up using 128 cores against the sequential execution on the SPARC64 VII based 256 cores cc-NUMA server Fujitsu M9000, 9.42 times speed-up using 12 cores against the sequential execution on the POWER8 based 12 cores cc-NUMA server Power System S812L. Besides, the performance evaluation shows that the proposed method succeeded to obtain 13.2 times speed-up against the parallel execution by XL Fortran compiler using 128 cores on the SR16000 and 1.4 times speed-up against the parallel execution by Intel Fortran compiler using 32 cores on the RS440 and 211.0 times speed-up against the parallel execution by Sun Studio Fortran compiler using 128 cores on the M9000.

The proposed method is effective for programs with simple array access order like Finite Difference Method. Additional optimizations may improve the performance of programs with complex array access order parallelized by the proposed method. Finite Element Method often uses complex array access order.

This paper has shown the proposed parallelization method of the GMS using the OSCAR multigrain parallel compiler gives us scalable speed-ups with strong scaling on five different cc-NUMA servers.

Acknowledgment

The authors would like to thank the members of the Hitachi-Waseda collaborative research project and the Hitachi, Ltd. and the NIED for their support.

References

1. Akcelik, V., Bielak, J., Biros, G., Epanomeritakis, I., Fernandez, A., Ghattas, O., Kim, E.J., Lopez, J., O'Hallaron, D.R., Tu, T. and Urbanic, J.: Highresolution forward and inverse earthquake modeling of terascale computers. Proc. *ACM/IEEE SC2003*, (2003)
2. Aoi, S., Fujiwara, H.: 3-D finite difference method using discontinuous grids. Bulletin of the *Seismological Society of America*, vol. 89, pp.918–930 (1999)
3. Tiankai, T., David, R.O., Omar,G.,: Scalable Parallel Octree Meshing for Terascale Applications. Proc. *ACM/IEEE SC2005*, (2005)
4. Aoi, S., Nishizawa, N., Aoki, T.: Large Scale Simulation of Seismic Wave Propagation using GPGPU. Proc. *THE FIFTHTEENTH WORLD CONFERENCE ON EARTHQUAKE ENGINEERING*, (2012)
5. Kasahara, H., Obata, M., Ishizaka, K.: Automatic coarse grain task paralell processing on smp using openmp. Workshop on Languages and Compilers for parallel Computing (2001) 1-15
6. Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K., Kasahara, H.: Hierachical Parallelism Control for Multigrain Parallel Processing. Lecture Notes in Computer Science 2481 (2005) 31-44
7. GMS Homepage, http://www.gms.bosai.go.jp
8. The HDF Group, http://www.hdfgroup.org/
9. Monica, D.L., Edward, E.R., Michael E.W.,: The cache performance and optimizations of blocked algorithms. Proceedings of the fourth international conference on Architectural support for programming languages and operating systems. pp.63–74 (1991)
10. Apan, Q., Ken, K.,: A Cache-Consciout Profitability Model for Empirical Tuning of Loop Fusion. 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, pp. 106–120 (2005)
11. OSCAR API 2.0, http://www.kasahara.elec.waseda.ac.jp/api2/ regist_en.html
12. Jaswinder P.S., Truman J., Anoop G., John L.H.,: An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors. Supercomputing '93 Proceedings of the 1993 ACM/IEEE conference on Supercomputing pp. 214–225 (1993)
13. Cahill J.J., Nguyen T., Vega M., Baska D., Szerdi D., Pross H., Arroyo R.X., Nguyen H., Mueller M.J., Henderson D.J., Moreira J.: IBM Power Systems build with the POWER8 architecture and processors. IBM Journal of Research and Development, Volume:59, Issue: 1, pp. 4:1–4:10 (2015)