Implementation and performance evaluation of SPAM particle code with OpenMP-MPI hybrid programming

Taisuke Boku* Shigehiro Yoshikawa* Mitsuhisa Sato*

Carol G. Hoover† William G. Hoover‡

*Institute of Information Sciences and Electronics, University of Tsukuba
†Lawrence Livermore National Laboratory
‡Department of Applied Science, University of California

Abstract

In this paper, we implement a SPAM (Smooth Particle Applied Mechanics) code in both pure MPI and MPI-OpenMP hybrid manner, then compare and analize the performance of them on an SMP-PC cluster. Our SPAM code is described to handle any mapping of spatial cells on to parallel MPI processes to exploit well load-balancing even with a relatively high communication cost. First we implement a parallelized version of the code with MPI, then parallelize each MPI process with OpenMP parallel-do feature.

In the overall performance results, the performance of MPI-only code exceeds that of MPI-OpenMP hybrid code except with a single node execution. We analized the detail of execution time and cache hit ratio on each processor, and found that the advantage of low communication cost between threads overcomes the disadvantage of low cache miss-hit ratio caused by the data localization problem in OpenMP.

Another issue of this code is load imbalancing caused with the setting in our target problem mode. We consider a better cell mapping suitable for hybrid code and discuss the difficulty to exploit high performance on this kind of hybrid programming.

1 Introduction

SPAM (Smooth Particle Applied Mechanics) is one of the particle simulation methods which is effectively applicable to the analysis of continuum system. The basic concept is very similar to MD (Molecular Dynamics) simulation, which is based on particle-to-particle force calculation with a potential energy rapidly varying with the distance between particles. Therefore, we can introduce the concept of *cut-off radius* to reduce the computational cost for force calculation among particle pairs.

It is well known that such a particle computation with cut-off radius is relatively easy to parallelize with particle decomposition or space decomposition method[1, 2]. Since our problem space size is very large compared with cut-off radius, the spatial decomposition method is more suitable for efficient parallel computing. The problem space is divided into fixed size of cells where the size in a dimension is equal to or greater than cut-off radius. With this method, we can reduce the communication cost among cells because all communications are point-to-point, and able to be localized by certain cell-to-processor mapping. However, when the density of particles in cells varies largely, it is difficult to map cells to processors keeping load balance.

Currently, we are developing a parallelized SPAM code for 2-dimensional shock-wave analysis. The code is mainly described with MPI. However, we target SMP cluster as the platform for a scalable problem with a million of particles. To exploit the parallelism in an SMP node, we describe the program with a hybrid method combining MPI and OpenMP. In this paper, we implement the parallel SPAM code in both MPI-only style and MPI-OpenMP hybrid style. Our target problem causes a serious load imbalancing by its own model, then we apply appropriate cell mapping on MPI and dynamic load balancing on OpenMP.

Basically, the advantage of hybrid code to MPI-only code on SMP clusters can be considered as follows:

Low communication cost: Threads

on the same node can communicate without explicit function call.

Dynamic load balancing: Mapping of parallel tasks on to threads can be controlled dynamically to exploit well load balance. Fine grain control may also be applicable.

Coarse grain communication: Since the number of communication nodes is reduced, the total number of messages is also reduced keeping the total amount of transferred data. It reduces the communication overhead especially for collective communications.

We try to exploit high performance in our hybrid code utilizing above features.

The rest of this paper is constructed as follows. First we describe the target problem model and its code both in MPI-only and hybrid manners. Then, we introduce our target SMP-PC cluster named COSMO, and examine the performance under various conditions. After discussion on the performance comparison, we conclude the paper finally.

2 Target Problem

We apply the SPAM method for 2-dimensional shock-wave analysis. The force affected to the particle-i is represented by the following equations:

$$wp(r_{ij}) = -\frac{\beta}{\pi R^3}\frac{r_{ij}}{R}(1-\frac{r_{ij}}{R})^2,$$

$$f_x(i) = -\sum_{j\neq i} wp(r_{ij})\frac{x_i - x_j}{r_{ij}},$$

$$f_y(i) = -\sum_{j\neq i} wp(r_{ij})\frac{y_i - y_j}{r_{ij}},$$

where r_{ij} represents the distance between particle-i and particle-j, and R is the cut-off radius. Force calculation is applied only with particle-j within the cut-off radius R. The simulation is performed by time-step manner with a certain interval time Δt. To reduce the error on time integration, fourth-order Runge-Kutta method is applied.

The simulation is performed with the following steps:

1. Set initial condition for position and velocity for all particles,
2. Distribute particles into fixed-size of cells,
3. Distribute cells to processes keeping load balance according to the number of particles per process,
4. Make a list of i-j particle pairs where their distance is lower than cut-off radius (we call it *sort*),
5. Calculate force and acceleration of particles with Runge-Kutta method,
6. Calculate the new position of particles,
7. Exchange particles among cells,
8. Iterate Steps 4-7 for certain times.

Because of the spatial decomposition method for parallelization, Step5 and Step7 require explicit interprocess communication. Since we are developing the code for *any* cell-to-process mapping, a cell contains all information of neighboring cells as well as particle data of the cell. Each cell knows the pointer to the neighboring cells and their owner processes. Even though this coding scheme introduces relatively large overhead, we prefer the flexibility of cell mapping.

In our 2-D shock-wave analysis code, the initial velocity of y-direction is fairly random for all particles. However, the velocity of x-direction is artificially set to make a larger pressure curve on x-direction. Figure 1 shows the initial velocity of x-direction. Actually, certain fraction of randomness is applied to this basic curve. Particles are also distributed initially with the reverse ratio in the space, that is, the density of particles increases with larger x-position. Therefore, cells in right side always have higher density than cells in left side.

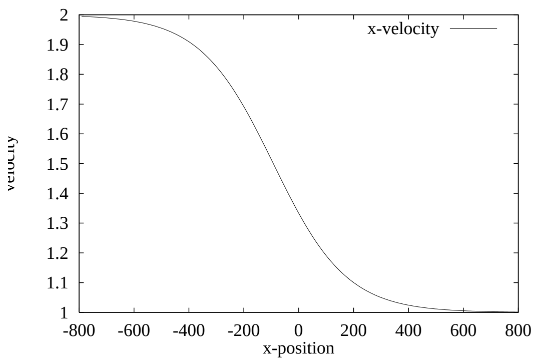

Figure 1: Initial velocity in x-direction

For a long time simulation, it is not efficient to keep the same cell-to-process mapping in this problem, because the actual calculation space continues to shrink in x-direction through computation. After a certain time steps, we go back to Step3 to re-map cells on processes. Generally such a remapping causes a large cost, however we can estimate the efficient timing for this process according to the physical modeling.

3 Programming

Out shock-wave SPAM code is coded in Fortran. Basic MPI programming is not special except the cell mapping is carefully controlled according to the physical model of velocity and particle density curve on x-direction. As mentioned before, we developed the code to separate the simulation engine and mapping process. Then, the simulation engine is very *generic* with self-contained cell information.

To exploit the parallelism on an SMP node, an MPI process is further parallelized by OpenMP coding. The most time consuming part of this problem is the calculation of force and acceleration with Runge-Kutta method. For instance, this part consumes over 70% of CPU time for a problem with 43,200 particles on a workstation in the sequential version of code. The *sorting* step is also relatively heavy, but it is difficult to parallelize avoiding duplication of *i-j* particle pair and *j-i* particle pair on the list. It is important to avoid them because the calculation of potential energy is very heavy, and we can make this cost to the half eliminating these duplications on the list.

Force calculation is applied for all pairs in the sorted-list, however we cannot simply parallelize this loop with *parallel do* directive because there are two problems:

- Particles in cells are not equal, that is, the computation cost varies for each cell.
- After checking of particle distance, a particle pair whose distance is out of cut-off radius is skipped.

Moreover, since multiple threads may update the force of a particle simultaneously, that part is protected by *critical* directive. To reduce the length of critical section, we separate it into two blocks for updating x-direction and y-direction of force individually.

Figure 2 shows the core part (force calculation) which is parallelized with OpenMP directives. Here, `rlucy` is the cut-off radius.

To reduce the load imbalancing problem, we apply dynamic scheduling and GSS (guided self scheduling) to parallelize these loops. We will examine these effects later.

We program the code with both MPI and OpenMP paradigms. If we run the program with multiple processes assigning only one thread for each process (OMP_NUM_THREADS=1), it

```
!$OMP parallel do private(i,j,xij,yij,rr,
                          rij,w,wp)
      do ij=0,npair-1
         i = ipairs(1,ij)
         j = ipairs(2,ij)
         xij = x(i) - x(j)
         yij = y(i) - y(j)
         if(yij.gt.+0.5*ny) yij = yij - ny
         if(yij.lt.-0.5*ny) yij = yij + ny
         rr = xij*xij + yij*yij
         if(rr.lt.rlucy*rlucy)then
            rij = sqrt(rr)
            call pot(rij,w,wp)
!$OMP critical (FX)
            fx(i) = fx(i) - wp*xij/rij
            fx(j) = fx(j) + wp*xij/rij
!$OMP end critical (FX)
!$OMP critical (FY)
            fy(i) = fy(i) - wp*yij/rij
            fy(j) = fy(j) + wp*yij/rij
!$OMP end critical (FY)
         endif
enddo
```

Figure 2: Core part of force calculation

is referred as MPI-only program. On the other hand, if we run the program with a single process assigning multiple threads for the process (OMP_NUM_THREADS=n), it is referred as OpenMP-only program. We control the number of physical processors to be used with the combination of number of MPI-processes and OMP_NUM_THREADS.

4 Performance Evaluation and Analysis

4.1 Models and Platform

The target problem on our preliminary performance evaluation is a 2-D shock-wave analysis with 43,200 particles. The problem space size is 1600×18 where the unit size is the average distance of all particles. The velocity ratio of the fastest particle (on the left edge) and the slowest particle (on the right edge) is set to 2:1 (Figure 1). The periodic boundary condition is applied for y-direction. On x-direction, the left edge is open while the right edge has mirror boundary condition since all particles move to right on x-direction.

The cut-off radius is set as 3.0, then the cell size is set to 3.1×3.1, where the problem space is divided into 518×6 cells. The remapping of cell-to-process is made in every 50 time steps of calculation, however we evaluate the first 50 steps only in this experiment. Therefore there is no overhead for cell remapping process in this evaluation.

Out target platform is an SMP-PC cluster named COSMO (Cluster Of Symmetric Memory prOcessor) with four nodes. Each node is DELL PowerEdge 6300 which contains four 450MHz Pentium-II Xeon processors connected by a shared bus. Each processor has 512KB of second cache, and DRAM is 4-way interleaved to provide high throughput. Nodes are connected with two classes of network, 800Mbps Myrinet and 100base-TX Ethernet. In this study, we use the latter one.

Basic characteristics of COSMO is shown in Table 1. To achieve the best performance on SMP-PC cluster with MPICH, the MPICH device for intra-node communication is memory-to-memory copy, and that for inter-node communication is TCP/IP(p4).

Table 1: Basic feature of COSMO

CPU	Intel Pentium-II Xeon (450MHz)
L2 Cache	512KB / processor
Node	4-way SMP (DELL PowerEdge 6300)
# of nodes	4
Network	800Mbps Myrinet & 100base-TX Ethernet
MPI library	MPICH-1.2
Compiler	PGI Fortran 3.1
OS	Linux 2.2.6 SMP kernel

4.2 Cell-to-process mapping in MPI-only code

In our target problem, the calculation load is imbalanced in x-direction while there is no problem on y-direction. Therefore, a simple blocked mapping for cell-to-process mapping (we call it as *flat mapping*) cannot achieve a good performance. On the other hand, a cyclic (or block-cyclic) cell-to-process mapping for x-direction may increase the communication traffic between contiguous cells. To examine these trading-off, first we compare the performance of both mapping strategies.

As a factor of calculation time, we counted the number of actual force calculations (inside `if` statement in the code of figure 2). Here, the number of threads for OpenMP execution is limited to 1, then whole calculation is parallelized by MPI only. Table 2 shows the result with four processes. It is shown that the cyclic mapping makes well-balanced computation.

Table 2: Force calculation load balancing

process-ID	flat mapping	cyclic mapping
0	366,719	972,417
1	387,360	974,227
2	1562,349	962,881
3	1535,350	954,058

Figure 3 shows the execution time per step with flat and cyclic mappings. Hereafter in graphs of total execution time and calculation time, we present several combinations of the number of nodes and the number of processors. The horizontal axis always shows the total number of processors. In any case, the number of processors per node is the same. For instance, a curve connecting the points of 1, 2, 3 and 4 processors represents the case using one node and varying the number of processors as 1, 2, 3 and 4, respectively. A curve connecting the points of 4, 8, 12 and 16 processors represents the case using four nodes and varying the number of processors per node as 1, 2, 3 and 4, respectively.

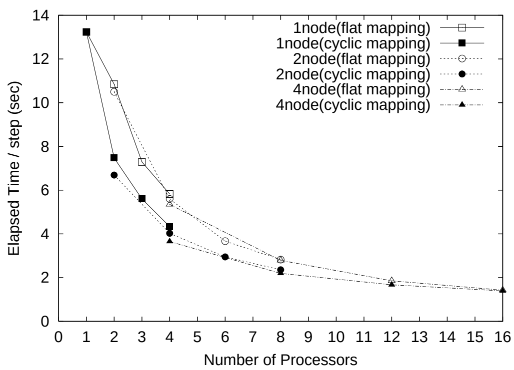

Figure 3: Execution time comparison on cell mapping methods

From these results, the cyclic mapping achieves about 30% higher performance than the flat mapping. However, the difference of them reduces with increasing the total number of processors. It is considered that the advantage of load balancing with the cyclic mapping and the advantage of low communication overhead with the flat mapping balances with 16 processors in this problem.

When the calculation steps increased, the load imbalancing problem becomes serious in our problem. Therefore, we use the cyclic mapping for the performance evaluation hereafter.

4.3 Load balancing among OpenMP threads

In the above section, we evaluated the *static* load balancing among MPI processes. In the hybrid code, the calculation load in an MPI process is farther distributed to multiple threads on SMP. In our code, it is performed in another level, that is, the force calculation among all *i-j* particle pairs in a process is performed in parallel. Before this process (Figure 2), all combinations of particle pairs are recorded in index arrays `ipairs()` and `jpaires()` in *sorting* phase. The load imbalancing problem is caused because the actual force calculation is executed only within the pairs which distance is less than or equal to the cut-off radius. Therefore, such *i-j* particle pairs are randomly distributed on these index arrays. It means the calculation time for each iteration in the loop may vary.

To solve this load imbalancing problem, we control the scheduling of threads with `schedule` option in `parallel do` directive. Table 3 shows the force calculation time on each thread for a time step with various scheduling policies, and Figure ?? also shows the number of actual force calculations in each thread with various scheduling policies. Here, we used a single node and four processes.

Table 3: Comparison of thread scheduling policies (exec. time)

policy	force calc. time (sec)
static	6.80
cyclic	6.78
dynamic	6.77

Table 4: Comparison of thread scheduling policies (*i-j* pairs)

thread-ID	static	cyclic	dynamic
0	924,895	942,257	960,329
1	970,685	955,141	960,285
2	972,495	970,682	956,265
3	972,915	972,910	964,111

From these results, the execution time for various scheduling policies are almost the same. Generally, the dynamic scheduling (like GSS - guided self scheduling) policy requires heavier overhead than static ones, and it is important to select an appropriate basic *chunk* size to achieve enough performance to overcome the overhead. We also vary the chunk size according to the number of *i-j* particle pairs assign to an MPI process. There is a slight gain of dynamic thread scheduling. We consider that it is the best to apply dynamic scheduling always. In this case, it is easy to handle even with static scheduling, however more complicated and imbalanced load is considerable after long time steps.

4.4 Performance of hybrid SPAM code

Finally, we compare the performance of MPI-only code and hybrid code. Figure 4 shows the execution time per step in both methods. To minimize the communication traffic among node, we distribute the same number of processes on to all nodes fairly and map the neighboring processes on to the same node as much as possible.

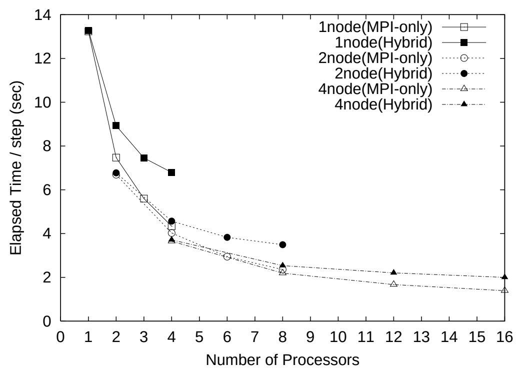

Figure 4: Performance comparison between MPI-only and hybrid codes

From these results, the performance of MPI-only code always exceeds that of hybrid code, however the difference becomes smaller with increasing the number of nodes. To examine the reason of these differences, we measured the execution time for force calculation only. The result is shown in Figure 5.

Figure 5 shows the force calculation time still dominates the total execution time in all cases, and the main reason of the performance difference exists here. However, as described in above sections, the computation load is carefully distributed both for MPI-only code and hybrid code. It means that there is no serious load balancing problem at this point in all cases.

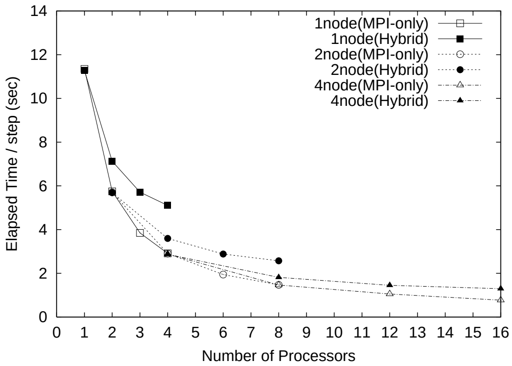

Figure 5: Comparison of force calculation time between MPI-only and hybrid codes

4.5 L2 cache miss-hit ratio

Since the *i-j* pairs to be actually calculated seem to be distributed fairly to all processors in both codes, the calculation time for each particle pair must differ in MPI-only and hybrid codes. To examine this, we measured the L2 cache miss-hit ratio on all processors. Pentium-II Xeon processor provides two performance counters which can be utilized to count various events in the processor. We counted the number of total memory references (DATA_MEM_REFS) and the number of memory bus transactions (BUS_TRAN_ANY), and measured L2 cache miss-hit ratio from the ratio of them.

Table 5: L2 cache miss-hit ratio

# nodes	# processors	MPI-Only	Hybrid
1	1	0.118	0.119
	2	0.119	0.552
	3	0.121	0.723
	4	0.130	0.854
2	2	0.124	0.120
	4	0.122	0.555
	6	0.125	0.726
	8	0.132	0.861
4	4	0.122	0.123
	8	0.128	0.563
	12	0.138	0.747
	16	0.142	0.899

Table 5 shows the L2 cache miss-hit ratio for both codes. It is clear that there is a serious cache miss-hit for hybrid code and the ratio is 6 times higher at maximum than MPI-only code. Especially for large number of processes per node, it is really serious.

We consider the reason why such a big difference was made. In the hybrid code, the force calculation for *i-j* particle pairs listed in *sorting* phase is parallelized with threads. To make the structure of this list clear, we explain the *sorting* phase in detail.

1. For a particle-i in a cell, list all other particles in the same cell as particle-j, and record all *i-j* pairs in the list. Here, the duplication of the combinations of *i-j* and *j-i* is eliminated.

2. For a particle-i in a cell, list all particles in the surrounding cells as particle-j, and record all *i-j* pairs in the list. To eliminate the duplication of *i-j* and *j-i* combinations, only four surrounding cells (east, north-west, north and north-east) are scanned (except the case of boundary region).

3. Process above two steps for all particles in a cell.

4. Process above steps for all cells assigned to the process.

As described above, the list of *i-j* particle pairs is generated based on cell-oriented manner. In step 4, the selection of targeting cell is performed in y-direction order first, then x-direction.

When processing force calculation between particle pairs with the list generated in above sequences, access to the particle data is repeated among five cells as shown in Figure 6. Here, the dominant data arrays to be referred are particle's position (`x,y`) and its force (`fx,fy`). In our target problem, the number of particles in a cell does not exceed 100. Then, the data amount to be scanned in these five cells can be estimated as

$$8(bytes) \times 100 \times 5 \times 4 = 16000(bytes).$$

Since the scanning of targeting cells is performed y-direction order, the next cell's data (north for the current one) has been already exist in L2 cache. Moreover, the number of cells in y-direction is only 6 in this problem. The total amount of data for contiguous three cells in x-direction is about 56KB, which is small enough to be contained in L2 cache. Therefore, the data for x-direction's next cell (east for the current one) is also kept in L2 cache. As a result, all data except the cells in east and north-east direction from current cell have already been loaded into L2 cache always. It means L2 cache works relatively well in any size of problem.

Next, let's consider the case of parallel execution. In MPI-only code, the basic process is the

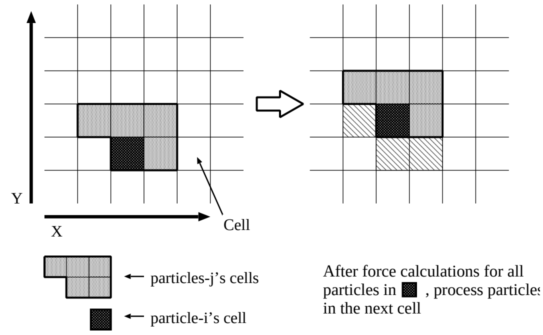

Figure 6: Data access pattern in force calculation

same as sequential version, that is, each process makes its own *i*-*j* pair list and processes particle pairs as described above. Then, L2 cache hit ratio is high and basically constant with any number of parallel processes. On the other hand, in the hybrid code, tasks (*i*-*j* pairs) are dispatched to parallel threads independent from the cell scanning sequence. As a result, the data access pattern does not follow to the above scenario, and the L2 cache hit ratio is degraded. Since we introduced dynamic thread scheduling with chunk size control which causes smaller chunk with increasing the number of processors (threads), this irregular access becomes serious for larger number of threads.

5 Discussion

5.1 Data localization

As described in the end of previous section, we consider the main reason of performance degradation in the hybrid code is caused by heavy L2 cache miss-hit ratio. In another word, it is caused by process-data missmatching, or generalized as data localization problem with OpenMP.

On of the ways to exploit data locality is to block the first half and latter half of loops to fit the same data location. However, such a solution breaks the advantage of dynamic load balancing. Another way is to describe explicit thread execution which contains both loops in it. However we have to describe so *naive* code for this, and it seriously reduces the easiness on parallel programming with OpenMP.

Hybrid programming with OpenMP may reduce the communication overhead within a node and make the inter-node communication grain coarse. In our code, however, the problem on data localization overcomes these advantages. If any explicit data localization directive is introduced, there may be a solution.

5.2 Cell mapping suitable for hybrid code

Since the aspect ratio of the rectangular problem space is very high in our target problem, we divided blocks of cells only in x-direction and mapped them to MPI processes. It is difficult to divide cells in y-direction because there are only 6 cells on y-direction and the parallelism is limited. Therefore, we cannot introduce such a mapping even it solves the problem on load imbalancing in x-direction.

In our hybrid code, however, it is possible to apply this mapping method since the number of processes is reduced from original MPI code. In another word, it is to consider that the problem has two levels of parallelism in x- and y-directions where the degree and characteristics of parallelization differ. As seen in the above section, it is difficult to exploit higher performance in hybrid code generally with a straight forward expansion from original MPI code. Problems with such multi-level parallelism may be one of the suitable candidates against it.

Currently, we are implementing a new version of hybrid code based on this idea. Even there is another problem of large amount of communication data caused by long edges in the spatial region, it is attractive to exploit higher performance in the hybrid code.

6 Conclusions and Future Works

In this paper, we described the design, implementation and performance evaluation of SPAM parallel code with MPI-OpenMP hybrid programming. To exploit the advantage of OpenMP in a sense of dynamic load balancing, we tuned the program in several ways; a static load balancing among MPI processes and dynamic load balancing among threads within a node. However, the preliminary performance evaluation on SMP-PC cluster results in lower performance of hybrid code than MPI-only code. We analyzed the detailed performance data and found the main reason is L2 cache miss according to irregular data access patterns in OpenMP parallelization.

Current OpenMP description does not allow effective data localization, and in our code, it is a serious problem. To exploit more performance, another cell-to-process mapping which is suitable for hybrid code rather than MPI-only code may be one of solutions. More detailed performance check is also required on cache miss-hit. After the considerable performance tuning, our final goal is to run this code on ASCI machines to perform very large scale simulation with more than a million particles.

Acknowledgment

The authors truly thank Dr. Antony DeGroot of Lawrence Livermore National Laboratory for his valuable comments in the early stage of this research. This study is partly supported by the Grant-in-Aid of Ministry of Education, Culture, Sports, Science and Technology in Japan (C-12680327).

References

[1] D.M.Beazley, P.S.Lomdahl : *Message-passing multi-cell molecular dynamics on the Connection Machine 5*, Parallel Computing 20, pp.173-195, 1994.

[2] S.Plimpton : *Fast Parallel Algorithms for Short-Range Molecular Dynamics*, Journal of Computational Physics, vol.117, pp.1-19, 1995.

[3] D.S.Henty : *Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling*, Proc. of SC'00, Dallas, USA, Nov. 2000.

[4] F.Cappello, D.Etiemble : *MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks*, Proc. of SC'00, Dallas, USA, Nov. 2000.

[5] T.Boku, K.Itakura, S.Yoshikawa, M.Kondo, M.Sato : *Performance Analysis of PC-CLUMP based on SMP-Bus Utilization*, Proc. of Workshop on Cluster Based Computing 2000, Santa Fe, May 2000.