A Scalable Hierarchical Parallelization Framework for Molecular Dynamics Simulation on Multicore Clusters

Liu Peng, Manaschai Kunaseth, Hikmet Dursun, Ken-ichi Nomura, Weiqiang Wang, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta
Collaboratory for Advanced Computing and Simulations (CACS)
Department of Computer Science, Department of Physics, Department of Materials Science
University of Southern California, Los Angeles, CA 90089-0242, USA

Abstract - *We have developed a scalable hierarchical parallelization framework for molecular dynamics (MD) simulation on emerging multicore clusters. The framework combines: (1) inter-node level parallelism by spatial decomposition using message passing; (2) intra-node (inter-core) level parallelism through a master/worker paradigm and cellular decomposition using critical section-free multithreading; and (3) intra-core level parallelism via single-instruction multiple-data (SIMD) techniques. Our multithreading scheme takes account of cache coherency to maximize performance. For data-level parallelism via SIMD, zero padding is used to solve the alignment issue for complex data type as array, and simple data-type reformatting is used to solve the alignment issue for data with irregular memory accessing. By combining a hierarchy of parallelism, the framework exposes maximal concurrency and data locality, thereby achieving: (1) inter-node weak-scaling parallel efficiency 0.975 on 32,768 BlueGene/P nodes and 0.985 on 106,496 BlueGene/L nodes; (2) inter-node strong-scaling parallel efficiency 0.90 on 32 dual quadcore AMD Opteron nodes and 0.94 on 32 dual quadcore Intel Xeon nodes; (3) inter-core multithread parallel efficiency 0.65 for the whole program (0.89 for two-body force calculation) for eight threads on a dual quadcore Xeon platform; and (4) SIMD speedup 1.35 for the whole program (1.42 for the two-body force calculation).*

Keywords: Molecular Dynamics simulation, Single Instruction Multiple Data, Scalable Hierarchical Parallelization Framework, Cache Coherency, SIMD Alignment

1 Introduction

Molecular Dynamics (MD) simulation is widely used to study material properties at the atomistic level [1]. Large-scale MD simulations involving million-to-billion atoms are beginning to address broad mechano-chemistry problems such as nanoenergetic reactions [2]. To encompass even larger spatiotemporal scales, however, increasingly larger computing power is needed. Emergence of the multicore paradigm has provided such unprecedented computing power. However, due to the shift from increasing clock speed to increasing number of cores per microchip [3], how to develop efficient parallel applications on these platforms is a challenge.

To address this challenge, we propose a scalable hierarchical parallelization framework (SHPF), which takes advantage of the multilevel feature of multicore clusters through hierarchical parallelization. For inter-node parallelization, we use the embedded divide-and-conquer (EDC) scheme [4] based on spatial decomposition using message passing, which scales linearly among compute nodes; for intra-node (inter-core) level parallelization, we implement cellular decomposition using critical section-free multithreading with a master-worker paradigm. Combined with single-instruction multiple-data (SIMD) techniques to exploit data-level parallelism, our hierarchical framework is expected to maximally expose data concurrency and locality and continue to scale on future multicore platforms.

This paper is organized as follows: Section 2 describes the multiresolution molecular dynamics (MRMD) method used in our experiments. Section 3 presents our parallelization framework, and Section 4 shows results of scalability tests. Conclusions are drawn in Section 5.

2 Multiresolution molecular dynamics

In MD simulation, phase-space trajectories of the system are obtained by numerically integrating

coupled ordinary differential equations to obtain the positions and velocities of all atoms at discretized time steps [2, 5]. Atomic force laws for describing how atoms interact with each other is mathematically encoded in the interatomic potential energy $E(\mathbf{r}^N)$, which is a function of the positions of all N atoms, $\mathbf{r}^N = \{\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_N\}$ [6].

In our MRMD simulation, E $(\mathbf{r}^N)$ consists of two-body (E_2) and three-body (E_3) terms [7]. We use a linked-list cell based EDC algorithm to reduce the computational complexity to $O(N)$. Here, the whole simulation system is divided into spatially localized linked-list cells. In addition, we use various space-filling curves [8] (e.g. Hilbert or Morton curve) to traverse the computational cells in order to optimize the data and computation layouts [9, 10]. We also use multiresolution in time, where temporal locality is utilized by computing forces from further atoms with less frequency with a multiple time-scale method [11-13]. This not only reduces the computational cost but also enhances the data locality, and accordingly is more suitable for parallelization.

3 Scalable hierarchical parallelization framework for molecular dynamics

Our scalable hierarchical parallelization framework (SHPF) combines: (1) inter-node level parallelism by spatial decomposition using message passing; (2) intra-node (inter-core) level parallelism through master-worker pattern with cellular decomposition using critical section-free multithreading; and (3) data-level parallelism via SIMD techniques. The following subsections describe these parallelization levels.

3.1 Inter-node level parallelism

In spatial decomposition, the physical system is partitioned into subsystems of equal volume. Atoms located in a particular subsystem are assigned to one of the compute nodes in the cluster, which are logically arranged according to the topology of physical subsystems (specifically, we use 3D mesh decomposition).

In parallel MD, two events that need communication are implemented using message passing. The first is atom caching: In order to compute interatomic interaction with cut-off length r_c, atomic coordinates of 26 neighbor subsystems, which are located within r_c from the subsystem boundary, are copied to this node, where cache coherence is maintained by copying the latest neighbor surface atoms every time before atomic accelerations are computed. The second is atom migration: After the atomic coordinates are updated according to the time-integration algorithm, some resident atoms may have moved out of the subsystem boundary. Such atoms are moved to proper nodes.

3.2 Intra-node level parallelism

With the spatial decomposition in the previous subsection, each spatial subsystem (or compute node) contains a block of linked-list cells. On multicore clusters, we further decompose the block of cells into small chunks and assign each chunk to a core by multithreading. For portability among broad architectures and operating systems, we adopt the POSIX thread standard. To achieve high parallel efficiency, we have designed a critical section-free algorithm to make interatomic force computations independent at the cost of some computational overhead.

Our multithreading scheme employs a master/worker model: The master thread is in charge of updating the atomic coordinates, constructing neighbor lists, atom caching, atom migration, and coordinating the worker threads, while the worker threads are in charge of force computations. In addition, semaphores are used to synchronize between the master and worker threads as well as to avoid the overhead of thread creation and joining in each MD step.

Our multithreading also takes account of cache coherency. It is known that increasing data locality by packing data together usually increases cache performance. However, this is not always practiced in multithread applications, especially for frequently written data by multiple threads. A typical modern CPU cache architecture implements write-back cache policy, which allows modification of data directly in the cacheline without immediately write modified data back to main memory. However, this could lead to cache-level critical section and race condition if the frequent write destination addresses by several CPUs lie in the same cacheline. A typical example is an array sum[N_T] (N_T is the number of threads) that provides separate accumulators to different threads to avoid a critical section for global sum (see Fig. 1 for $N_T = 2$). Cache-level race condition still occurs when multiple threads simultaneously modify sum[i] laid in the same cacheline. Here, we employ a padding technique that separates sum[i] to different cachelines by defining a new *struct* with array pad[NPAD]. An example of *struct* for a double-precision datum is shown below:

```
#define NPAD 7
struct p {
  double value;      // 1x8 bytes
  double pad[NPAD]; // 7x8 bytes
};                   // total 64 bytes
```

In this example, NPAD has to be chosen to match the cacheline size. Also, better performance can be achieved by placing frequently used variables for individual thread together in the same cacheline.

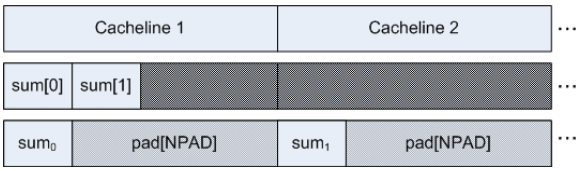

Figure 1. Illustration of a padding technique for $N_T = 2$.

3.3 Intra-core level optimization via SIMD

We exploit data-level parallelism inside each core via SIMD vectorization. A basic SIMD operation packs data into 128-bit vector registers to be operated simultaneously. However, the streaming SIMD extensions (SSE) load and store instructions have a special requirement on alignment, i.e., a load or store instruction must load from or store to a 16-byte aligned memory. Though the SSE intrinsics provides unaligned load instructions, it is at the expense of a large performance penalty, since such instruction may cause cacheline splits. It is thus of great importance to naturally align memory access.

There are mainly two issues for achieving high SIMD speedup: One is to exploit as many SIMDizable statements to maximally reduce the computation and loading; the other is to arrange data naturally aligned to gain optimal performance. To address the first issue, we have previously proposed translocated statement fusion and vector composition techniques, thereby achieving 3.5-fold SIMD speedup out of 4. To address the second issue, we here propose two key techniques: zero padding for complex data type (e.g. array); and simple data-type reformatting for irregular memory accessing.

3.3.1 Zero padding to align complex data types

Zero padding is an effective way to solve the alignment issue in SIMD vectorization for complex data type like array with an appropriate padding size.

This subsection illustrates the use of zero padding for SIMD vectorization of MRMD.

Here, the original code is doubly nested for loops, where the inner loop traverses the x, y and z Cartesian dimensions to perform certain computation:

```
for (i=0; i<N; i++)
  for (a=0; a<3; a++)
    r[i][a] = r[i][a]+DeltaT*rv[i][a];
```

Our SIMD solution (see Fig. 2) redefines the array r[N][3] and rv[N][3] (N is the number of atoms) to array r[N][4] and rv[N][4] by padding zero to each row of both arrays, i.e., pad 0 to each {r[i][0], r[i][1], r[i][2]} to make {r[i][0], r[i][1], r[i][2], 0}, and each {rv[i][0], rv[i][1], rv[i][2]} to {rv[i][0], rv[i][1], rv[i][2], 0}. After this reformatting, we can unroll the inner loop, pack data {r[i][0], r[i][1], r[i][2], 0} to *rvec* vector, {rv[i][0], rv[i][1], rv[i][2], 0} to *rvvec* vector, and {DeltaT, DeltaT, DeltaT, DeltaT} to *DeltaTvec* vector, and multiply and add them simultaneously to obtain the result. The pseudo-code of the above SIMDization is given below:

```
Data reformatting by zero padding
/* Number of zeros for padding */
#define NPAD 1
int zeros[NPAD] = {0,0,…0};
r[i] ← {r[i][0],r[i][1],r[i][2],
  zeros[NPAD]};
rv[i] ← {rv[i][0],rv[i][1],
  rv[i][2],zeros[NPAD]};
DelaTvec ← {DeltaT,DeltaT,DeltaT,
  DeltaT}
for (i=0; i<N; i++){
  Data packing and loading
  rvec ← load{r[i][0],r[i][1],
    r[i][2],zeros[NPAD]};
  rvvec ← load{rv[i][0],rv[i][1],
    rv[i][2],zeros[NPAD]};
  Computation
  rvec ← mul_add(rvec,rvvec,
    DeltaTvec);
  Data storing
  {r[i][0],r[i][1],r[i][2],0} ←
    store(rvec);
}
```

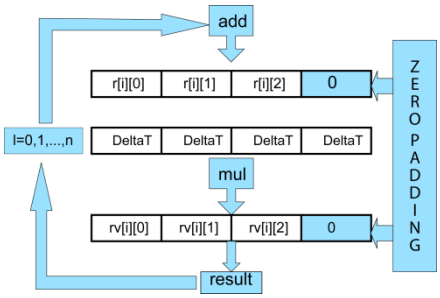

Figure 2. Zero padding for SIMDization.

The effect of this SIMDization can be analyzed as follows. For the computation, it reduces the computation from 3*N* to *N* with an ideal speedup of 3 (*N* is the number of atoms). For the memory

accessing part in the ideal case (i.e., if all data are stored in the cache), it reduces the memory accessing from 9*N* to 3*N*. (Due to the zero padding, all vector load and store operations start from an aligned address, as the stride is 16 bytes for each r[i][0] and rv[i][0].) Therefore, the ideal memory accessing speedup is 3. However, the SIMDization introduces some memory overhead, since it increases the size of each array by one third, which could cause cache miss.

3.3.2 Data-type reformatting for alignment of irregular memory accessing

While the zero padding is effective for array reformatting, simple data-type reformatting works for the alignment of irregular memory accessing. Most instruction sets provide conversion between simple data types (e.g. float and double), which allows a simple way to circumvent unaligned memory accessing. This subsection illustrates the use of data-type reformatting for SIMD vectorization of MRMD.

Here, the original code is a complex doubly nested while loops with an if statement. It involves irregular memory accessing to array v[ic][jc][ir+1][0] via linked list lscl[i]. The innermost loop performs computation using a large four-dimensional array, where all data types are float.

```
while (i != EMPTY) {
  ...
  j = head[c1];
  while (j != EMPTY) {
    if (rr < rcij2[ic][jc]) {
      v0 = (1.0-fr)*v[ic][jc][ir][0]
      +fr*v[ic][jc][ir+1][0];
      v1 = (1.0-fr)*v[ic][jc][ir][1]
      +fr*v[ic][jc][ir+1][1];
    }
    ...
    j = lscl[j];
  }
  i = lscl[i];
 }
```

The complex iteration structure of the code will likely cause irregular accessing to array v[M][N][2][2], which precludes us from finding four float elements to be SIMDized. Our solution for the alignment of this code is simply to redefine array v[M][N][2][2] from float to double. By simply redefining the array to double type (and variable fr to double), we can SIMDize the innermost loop as follows (see Fig. 3): First, pack data {fr,fr} to *frvec*, {1–fr,1–fr} to *1mfrvec*, {v[ic][jc][ir][0], v[ic][jc][ir][1]} to *v0vec*, {v[ic][jc][ir+1][0], v[ic][jc][ir+1][1]} to *v1vec*; then multiply *frvec* by *v1vec*, multiply *1mfrvec* by *v0vec*, and sum the products; finally use abstract operation to get v0 and v1. The pseudo-code of the above SIMDization is given below:

```
Data reformatting:
double v[M][N][2][2];
double fr;
Data packing and loading:
frvec ← {fr, fr};
1mfrvec ← {1-fr, 1-fr};
V0vec ← load{v[ic][jc][ir][0],
  v[ic][jc][ir][1]};
v1vec ← load{v[ic][jc][ir+1][0],
  v[ic][jc][ir+1][1]};
Computation:
tmp1 ← mul(frvec,v1vec);
tmp2 ← mul(1mfrvec,v0vec);
tmp1 ← add(tmp1,tmp2);
v0 ← abstract(tmp1);
v1 ← abstract(tmp1);
```

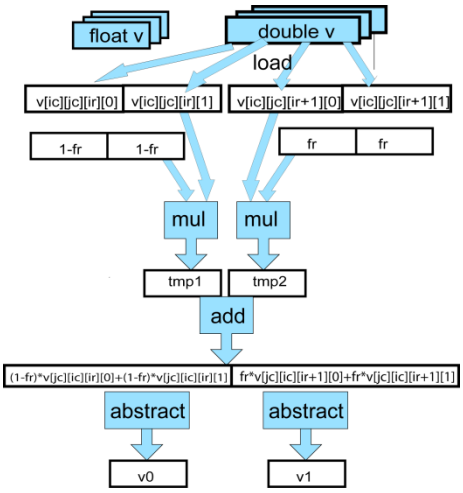

Figure 3. Example of data-type reformatting for SIMDization.

The effect of this SIMDization can be analyzed as follows. It reduces the number of floating-point operations from 8 to 6 with an ideal computational speedup of 1.33. It also reduces memory accessing from 6 to 4. (Due to the data reformatting, all vector load operations are from aligned address as the stride is 16 bytes for each rv[ic][ic][ir][0].) Therefore, the ideal memory-accessing speedup is 1.5.

4 Performance tests and analysis

The scalability of the SHPF applied to MRMD has been tested on various multicore clusters:

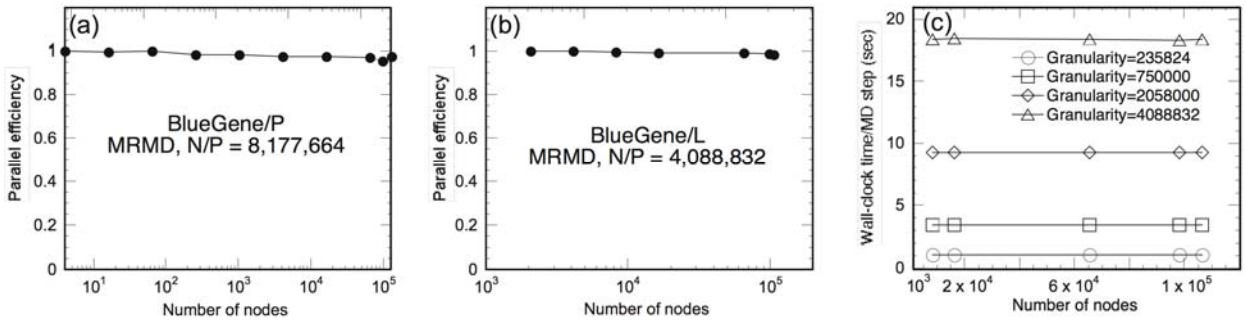

Figure 4. Inter-node weak-scaling parallel efficiency on (a) BlueGene/P and (b) BlueGene/L. (c) Total running time per MD step on BlueGene/L as a function of the number of nodes.

106,496 IBM BlueGene/L nodes (each with two IBM PowerPC 440 processors at 700 MHz clock) at the Lawrence Livermore National Laboratory, 32,768 IBM BlueGene/P nodes (each with four 450 POWER PC processors at 850 MHz clock) at the Argonne National Laboratory, and dual quadcore Intel Xeon (2.33 GHz clock) and dual qaudcore AMD Opteron (2.30 GHz) based clusters at the High Performance Computing and Communications facility of the University of Southern California (HPCC-USC).

4.1 Inter-node weak scalability

We have implemented the inter-node spatial decomposition using the Message Passing Interface (MPI) standard and test the inter-node weak scalability, where the problem size (i.e., the number of atoms N) is scaled linearly with the number of nodes P. Here, we define the inter-node weak-scaling parallel efficiency as the ratio between the running time on one node and that on P nodes. Figure 4(a) shows the inter-node weak-scaling parallel efficiency for 8,177,664P atoms as a function of the number of nodes P on BlueGene/P, and Fig. 4(b) shows the efficiency for 4,088,832P atoms on BlueGene/L. Our framework achieves excellent weak-scaling efficiency on both platforms: 0.975 on 32,768 BlueGene/P nodes and 0.985 on 106,496 BlueGene/L nodes based on the speedup over 2,048 nodes.

To quantify the effect of granularities (i.e., the number of atoms per node, N/P), we have tested the weak scalability with different granularities. Figure 4(c) shows the execution time of the MRMD algorithm as a function of the number of nodes P over a wide range of granularity. The figure shows that the running time varies only slightly as a function of P independent of the granularity. Thus the MRMD algorithm achieves nearly perfect inter-node weak-scaling parallel efficiency independent of the value of N/P.

4.2 Inter-node strong scalability

Next, we test the strong scalability for inter-node parallelism. Strong-scaling speedup S_P on P nodes is the running time on one node divided by that on P nodes, while the efficiency E_P is defined as S_P/P. We fix the problem size at 3.15 million atoms, whereas P varies up to 128 nodes (with 8 processors per node, the total number of processor is 1,024 processors). Figure 5(a) shows the strong-scaling parallel efficiency as a function of P on dual quadcore Xeon and Opteron clusters. We observe superlinear speedup of approximately 2.6% on the Xeon platform and 0.8% on Opteron platform. The superlinear speedup may be explained as follows. Since the workload per node (granularity), N/P, decreases as P increases, the data becomes less scattered among paged memory, resulting in less TLB misses. We analyze this effect using the Intel Vtune performance analyzer, which shows that the penalty from TLB misses on Xeon reduces from 2.94% of total CPU clock cycles for P = 32 nodes to 2.49% for P = 64, and eventually to 1.64% for P = 128. When the granularity is further reduced until it is small enough to fit into L2 cache, cache miss ratio would be reduced as well, and accordingly, the running time would be further reduced.

However, less granularity also introduces larger sequential overhead. Figure 5(b) shows the ratio of sequential segment as a function of P. We see that the sequential bottleneck increases for larger P. For the smallest granularity on Xeon, the sequential section accounts for 54% of total running time. The sequential bottleneck offsets the speedup gained from the cache effect, and therefore, there exists a tradeoff between the cache effect and the sequential bottleneck. This tradeoff causes the strong-scaling parallel efficiency to be peaked at certain P.

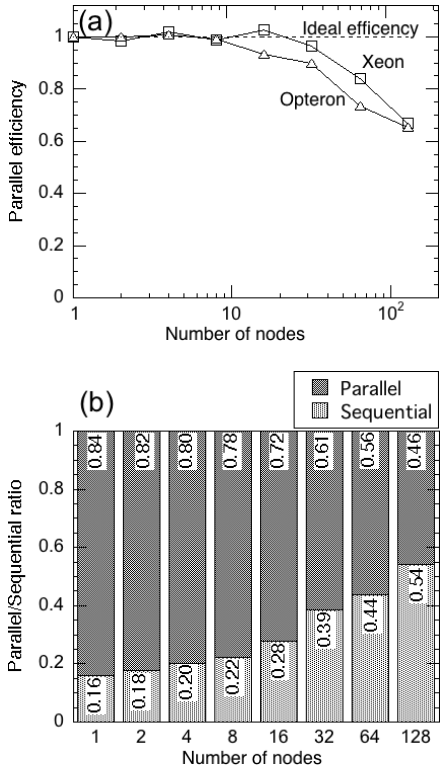

Figure 5. (a) Inter-node strong-scaling parallel efficiency on Xeon and Opteron as a function of the number of nodes *P*. (b) The parallel/sequential ratio of the running time on Xeon as a function of *P*.

The cache size also impacts the strong-scaling parallel efficiency. Figure 5(a) shows that the efficiency on Xeon is higher than that on Opteron for P larger than 8, because Xeon has larger L2 cache (6 MB per chip and 12 MB per multi-chip module) compared to only 2 MB on Opteron. For larger granularity, when P is less than 8 (i.e., $N/P >$ 370,000), the cache size has less effect, and thus Figure 5(a) shows no significant difference between Xeon and Opteron.

4.3 Intra-node multithreading scalability

We test the multithreading scalability of MRMD on a dual quadcore Intel Xeon platform. Here, we define the multithreading speedup with n_t threads, S_{nt}, as the running time of the program with one thread divided by that with n_t threads, while the problem size is the kept constant. We then define the intra-node (inter-core) multithreading parallel efficiency E_{nt} as S_{nt}/n_t. Figure 6 shows the multithreading parallel efficiency as a function of the number of worker threads from 1 to 8. We see that the code scales rather well up to 8 threads on the 8-core platform for the two-body force calculation with efficiency 0.89, while scales less for the whole program (efficiency 0.65).

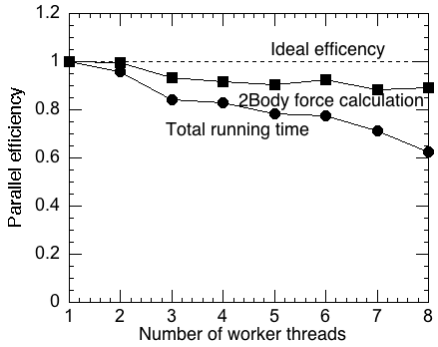

Figure 6. Intra-node multithreading parallel efficiency on Xeon platform for the whole program and two-body force computation.

It is important to identify the real factor causing the decrease of efficiency when the thread number increases. This is partly due to the redundant computation introduced for three-body computations for eliminating critical sections. It could also due to the overhead for maintaining cache coherency when there is a cacheline racing as discussed in section 3.2. Here, we use the Shark 4 profiler on a dualcore Xeon 2.66 GHz to observe the impact of better cache coherency. Large value of request-for-ownership (RFO) transactions and modified data sharing ratio indicate frequent races among threads on using and modifying data laid in the same cacheline. Table 1 shows that the padding technique reduces more than 98% of RFO transactions compared to the naïve code and that the modified data sharing ratio is 9 times less. This analysis indicates a large performance gain from cache coherency. To quantify the performance improvement, Figure 7 compares the running time of the padding code to that of the naïve code. The padding technique gets average speedup of 53% for all granularities compared to the naïve code.

Table 1. Cache coherency profiling. The granularity *N/P* is 3072 atoms in both results.

Code	Clock per Instruction retired (CPI)	RFO to clock ratio	Modified data sharing ratio
Naïve	0.83	0.000645	0.0009
Cache coherency	0.67	0.000012	0.0001

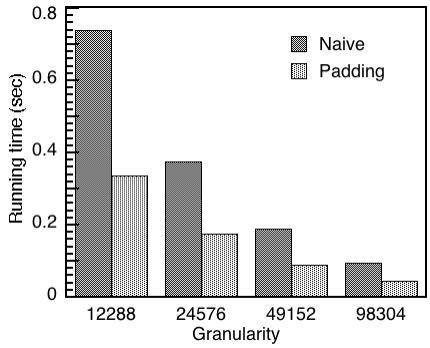

Figure 7. Running times per MD step of naïve and padding codes.

4.4 Intra-core level SIMD vectorization tests

We use SSE3 intrinsics (which is supported by the Intel Platform) to implement SIMD for MRMD. Here, we SIMDize the two-body force computation that consumes most computation. We test the SIMD speedup and its scalability with different problem sizes, while the SIMD speedup is defined as the ratio of the running time for unSIMDized program over that of the SIMDized program with the same problem size. Figure 8 shows that the SIMD speedup for the whole program is 1.35, while for the two-body force calculation (for which we have implemented SIMDization) is 1.42. This is much lower than the theoretical estimate in the section 3.3.

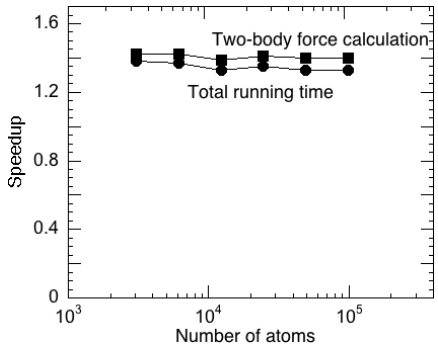

Figure 8. SIMD speedup with various problem sizes for whole program and two-body force computation.

To identify the cause of the low performance, we have profiled two code segments similar to the code in section 3.3.1. For one code segment, the running time fraction (defined as the running time of the code divide by the total running time) is reduced from 0.09% to 0.05% by SIMDization, while for the other from 0.18% to 0.11 %. For both, the SIMD speedup is lower than 2. This is mainly due to the memory overhead, as we introduce 1/3 overhead when redefining the array. Data layout reordering is thus of great significance for making the MRMD application more SIMDizable.

Figure 8 also points out some positive trend as the SIMD speedup remains more or less the same for both the whole program and two-body computation: 1.35 for the whole program and 1.42 for the two-body computation with one worker thread. This indicates that the SIMDization scales perfectly for different problem sizes varying from 3,072 to 98,304.

5 Conclusions

In summary, we have developed a scalable hierarchical parallelization framework for molecular dynamics simulation, thereby achieving almost ideal weak scalability on BlueGene L and P clusters as well as good strong scalability on dual quadcore Xeon and Opteron clusters. Within a node, multithreading has achieved reasonable inter-core parallel efficiency combined with intra-core level data parallelization via SIMD vectorization. We have also quantified the degradation of the scalability through performance profiling. Future work will address better data layout to improve the SIMD performance. This work was supported by NSF-ITR/PetaApps/EMT, DOE-SciDAC/BES, ARO-MURI, and DTRA.

References

[1] M.P. Allen and D.J. Tildesley, Computer Simulation of Liquids, Oxford Science Publication, Oxford, 1987.
[2] S. J. Plimpton. *J. Comput. Phys*, **117**:1, 1995.
[3] J. C. Phillips *et al.*, *Proc. Supercomput. 2002* (CA: IEEE/ACM)
[4] J. Dongarra *et al.*, *CTWatch Q.* **3**:11, 2007.
[5] A. Nakano *et al.*, Comput Mater Sci **38**, 642, 2007.
[6] A. Nakano *et al.*, *Proc. Supercomput. 2001* (NY: IEEE/ACM)
[7] H. Kikuchi *et al.*, *Proc. IPDPS* 2003 (IEEE).
[8] B. Moon *et al.*, *IEEE T. Knowl Data Eng.*, **13**: 124 2001.
[9] J. Mellor-Crummey, D. Whalley, and K. Kennedy, *Int'l J. Parallel Prog.* **29**:217, 2001.
[10] M. M. Strout, and P. D. Hovland, in *Proc. Workshop Memory System Performance*, 2004.
[11] Nakano. *Comput. Phys. Comm.*, **105**:139, 1997.
[12] A. Nakano. *Int. J. High Perform. Comput. Appl.*, **13**:154, 1999.
[13] M. E. Tuckerman *et al*. *Comput. Phys.Comm.*, **128**:333, 2000.