J Supercomput (2011) 57: 20–33
DOI 10.1007/s11227-011-0560-1

Exploiting hierarchical parallelisms 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

Published online: 3 February 2011
© Springer Science+Business Media, LLC 2011

Abstract We have developed a scalable hierarchical parallelization scheme for molecular dynamics (MD) simulation on multicore clusters. The scheme explores multilevel parallelism combining: (1) Internode parallelism using spatial decomposition via message passing; (2) intercore parallelism using cellular decomposition via multithreading employing a master/worker model; (3) data-level optimization via single-instruction multiple-data (SIMD) parallelism with various code transformation techniques. By using a hierarchy of parallelisms, the scheme exposes very high concurrency and data locality, thereby achieving: (1) internode weak-scaling parallel efficiency 0.985 on 106,496 BlueGene/L nodes (0.975 on 32,768 BlueGene/P nodes), internode strong-scaling parallel efficiency 0.90 on 8,192 BlueGene/L nodes; (2) in-

L. Peng (✉) · M. Kunaseth · H. Dursun · K.-i. Nomura · W. Wang · R.K. Kalia · A. Nakano · P. Vashishta
Collaboratory for Advanced Computing and Simulations (CACS), University of Southern California, Los Angeles, CA 90089-0242, USA
e-mail: liupeng@usc.edu

M. Kunaseth
e-mail: kunaseth@usc.edu

H. Dursun
e-mail: hdursun@usc.edu

K.-i. Nomura
e-mail: knomura@usc.edu

W. Wang
e-mail: wangweiq@usc.edu

R.K. Kalia
e-mail: rkalia@usc.edu

A. Nakano
e-mail: anakano@usc.edu

P. Vashishta
e-mail: priyav@usc.edu

tercore multithread parallel efficiency 0.65 for eight threads on a dual quadcore Xeon platform; and (3) SIMD speedup around 2 for problem sizes ranging from 3,072 to 98,304 atoms. Furthermore, the effect of memory-access penalty on SIMD performance is analyzed, and an application-based SIMD analysis scheme is proposed to help programmers determine whether their applications are amenable to SIMDization.

Keywords Molecular dynamics simulation · Multicore cluster · Single instruction multiple data · Scalable hierarchical parallelization scheme

1 Introduction

Molecular dynamics (MD) simulation is widely used to study material properties at the atomistic level [1]. Large-scale MD simulations involving multibillion atoms are beginning to address broad material problems [1, 2], however, increasing computing power is needed to satisfy the large spatiotemporal scales of the real world simulations. The advent of multicore paradigm, which provides unprecedented computing power, promises to enable large-scale and long-time simulations, only if we can efficiently harvest the computing power. However, due to the shift from increasing clock speed to increasing number of cores per chip [3], development of efficient parallel applications on these platforms remains a challenge.

To address this challenge, we propose a scalable hierarchical parallelization scheme (SHPF), which exploits multilevel parallelisms of multicore clusters. For internode parallelization, we use an embedded divide-and-conquer (EDC) scheme [4] based on spatial decomposition using message passing, which scales linearly with the number of compute nodes; for intercore level parallelization, we implement cellular decomposition using multithreading without using critical sections working with a master-worker paradigm. Combined with single-instruction multiple-data (SIMD) techniques to exploit data-level parallelism, our hierarchical scheme highly exploits data concurrency and locality and is expected to usable on future multicore platforms. The scheme also includes application-based SIMD analysis, which enables programmers to determine whether their applications are amenable to SIMDization.

This paper is organized as follows. Section 2 describes the linked-list cell MD simulation used in our experiments. Section 3 presents our parallelization scheme, and Sect. 4 shows results and analysis of performance and scalability tests. Conclusions are drawn in Sect. 5.

2 Linked-list cell molecular dynamics simulation

MD simulation follows the phase-space trajectories of an N-atom system, where force fields describing the atomic force laws between atoms are spatial derivatives of a potential energy function $E(\mathbf{r}^N)(\mathbf{r}^N = \{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N\}$ is the positions of all atoms). Positions and velocities of all atoms are updated at each MD step by numerically integrating coupled ordinary differential equations. The potential $E(\mathbf{r}^N)$ considered in this paper consists of two-body $E_2(r_{ij})$ and three-body $E_3(r_{ij}, r_{jk}, \theta_{ijk})$

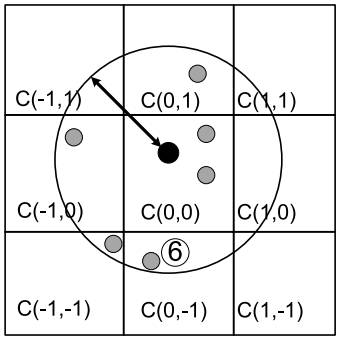

Fig. 1 2D schematic of the linked-list cell method

terms, where r_{ij} is the distance between atomic pair (i, j) and θ_{ijk} is an angle among atom triplet (i, j, k) [2].

Figure 1 shows a schematic of the computational kernel of MD, which employs a linked-list cell method to compute interatomic interactions in *O(N)* time based on Embedded Divide and Conquer [2, 4]. The center cell $C(0, 0)$ is surrounded by eight neighbor cells. The cell dimensions are often chosen to be the cutoff radius (represented by the two-heads arrow) of interatomic interaction. Only atoms (grey color) within the cutoff radius from the dark atom are shown. Periodic boundary condition is applied to the system in three Cartesian dimensions. Here, a simulation domain is divided into small rectangular cells, and the linked-list data structure is used to organize atomic data (e.g., coordinates, velocities, and atom type) in each cell. Traversing the linked list, one retrieves all atom information belonging to a cell and thereby computes interatomic interactions. The dimensions of the cells are usually determined by the cutoff radius r_c of the interatomic interaction, so that the search for interacting atomic pairs is restricted to the nearest-neighbor cells. MD is an archetype of irregular memory-access pattern applications due to atom diffusion, which imposes great challenge on efficient parallelization and performance optimization.

Since MD simulation is one of the most prominent applications in material science, there exist a number of publications on accelerating MD simulation. For example, special hardware accelerators including MD-GRAPE [5], Anton [6], and reconfigurable computers [7] promise to reach millisecond-level simulations. Erez et al. [8, 9] implemented an MD application, GROMACS, on Standford's streaming supercomputers, Merrimac [10]. Our goal is instead to investigate how to improve MD performance and scalability on a low-cost cluster platform, which is available to individual research groups. George et al. [11] conducted similar research at the initial stage of the IBM BlueGene architecture but did not discuss hierarchical optimization. There are other projects like NAMD [12] mainly targeting supercomputing systems composed of conventional clusters. However, there are few to our knowledge that reports large-scale internode, intercore scalability as well as fine-grained data-level optimization. The following details our parallelization scheme.

3 Scalable hierarchical parallelization scheme for molecular dynamics

Our scalable hierarchical parallelization scheme (SHPF) combines: (1) internode level parallelism by spatial decomposition using message passing; (2) intranode (intercore) level parallelism through master-worker pattern with cellular decomposition via multithreading without using critical sections; and (3) data-level parallelism via SIMD techniques. The following subsections describe these parallelization levels.

3.1 Inter-node level parallelism

Our internode level parallelism is based on spatial decomposition, where 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 the physical subsystems (specifically, we use 3D mesh). In parallel MD, two communication operations are implemented using message passing. The first is atom caching: In order to compute interatomic interaction within cut-off radius r_c at each MD step, atomic coordinates of 26 neighbor subsystems, which are located within r_c from the subsystem boundary, are copied to each node, where data coherence is maintained by copying the latest neighbor surface atoms every time before atomic accelerations are computed. The second communication operation 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, and such atoms are moved to proper nodes. We implement the internode spatial decomposition using the message passing interface (MPI) standard.

3.2 Intercore level parallelism

With spatial decomposition, each spatial subsystem (or compute node) contains a set of linked-list cells. On multicore clusters, we further decompose the set 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 computation of each thread independent of those of the other threads at the cost of some computational overhead [2]. Our multithreading scheme employs a master/worker model [2] (Fig. 2): The master thread is in charge of updating the atomic coordinates, constructing neighbor lists, atom caching,

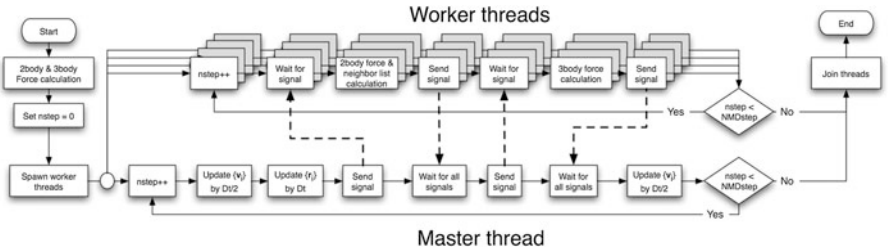

Fig. 2 Flow chart of master/worker force computation

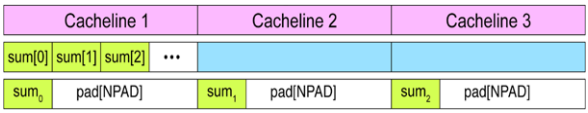

Fig. 3 Illustration of padding for $N_T = 3$

atom migration, and coordinating the worker threads, while the worker threads are in charge of force computations. In addition, semaphores are used to synchronize the master and worker threads as well as to avoid the overhead of thread creation and joining at each MD step.

Our multithreading scheme also takes account of cacheline false sharing conditions among threads. A typical example is an array $sum[N_T]$ (N_T is the number of threads) that provides separate accumulators to different threads for global sum. Though this eliminates a critical section at data level, cache-level racing 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 (Fig. 3). Furthermore, better performance is achieved by placing frequently used variables for individual thread together in the same cacheline. Nearly constant speed up (between 2.1 and 2.2) is observed for various granularities (i.e., the number of atoms per thread ranging from 12,288 to 98,304) by avoiding overhead due to the cache coherence protocol on Intel Xeon platform.

3.3 Data level parallelism via SIMD

We exploit data-level parallelism inside each core via SIMD vectorization. There are mainly two issues for achieving high SIMD speedup: One is to exploit as many SIMDizable statements as possible via data packing and fusion [13]; and the other is to arrange data to avoid unaligned memory accessing [14]. In the following subsections, we introduce inner-loop branch translocation and loop unrolling [15] to enhance the data packing, and padding to address memory alignment issues [16, 17]. And all the following discussions are for float data type and SSE extension.

3.3.1 Data packing for SIMD via inner loop branch translocation and loop unrolling

For data-level parallelization, we employ SIMD techniques based on loop branch translocation and loop unrolling. To explain these techniques, Fig. 4 shows the flowcharts of branch translocation of interatomic force calculation. The cyan block represents the most compute-intensive code segment, where the conditional statement inside it (colored in red) is used to avoid duplicating force calculations among different threads.

Since it is well known that branches block streaming processes, and thus seriously degrade SIMD performance, it is important to translocate the conditional statement from inside a loop to the outside to make the most compute-intensive part more SMIDizable. Noting the descending order of atom indices in each cell according to the linked lists, our SIMD solution translocates the branch statement on condition, $j < n$, to the outside by decomposing the innermost loop into two blocks: First, $j \geq n$, and then $j < n$. The branch-translocated innermost loop has thus become

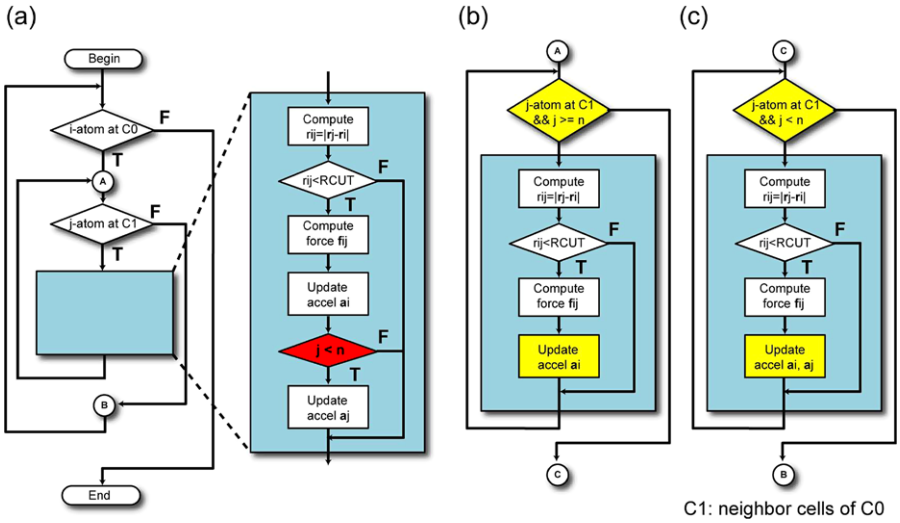

Fig. 4 Flowcharts of branch translocation of force calculation

two consecutive blocks of SIMD-friendly code segments with different conditional statements checked at the beginning of each block as shown in Figs. 4(b) and (c). Figure 4(b) shows the segment to calculate the distance between resident atom i and non-resident atom j, only updating the force on atom i, whereas Fig. 4(c) handles a pair of resident atoms, updating the forces on both atoms i and j.

SIMDization of our program is then implemented in conjunction with loop unrolling. We unroll the inner loop four times and do data packing, i.e., for each atom i, a group of four atoms j_0, j_1, j_2, j_3 are fetched together to concurrently perform pair-interaction computations with atom i. If the four pairs are within the cutoff, we fully SIMDize them (except for the force table lookup, which is not SIMDizable), else we just use the original code to process them one by one. By eliminating branches through decomposition of the inner loop, the program becomes much more suitable for SIMDization. Furthermore, by loop unrolling, our solution achieves better performance, as we mostly pack four data together to fully take advantage of SIMD.

3.3.2 Memory alignment for SIMD via zero padding

A basic SIMD operation packs data into 128-bit vector registers to be operated simultaneously. However, the load and store instructions in the streaming SIMD extensions (SSE) have a special requirement on alignment, i.e., an instruction must load from or store to a 16-Byte aligned memory. Although the SSE intrinsics provide unaligned load instructions, it is at the expense of a large performance penalty, since such an instruction may cause cacheline splits. It is thus of great importance to naturally align memory accessing. This subsection illustrates the use of zero padding to achieve this, using the velocity update function in our MD code as an example. Here, the code is doubly nested for loops, where the outer loop traverses N atoms, and the inner loop

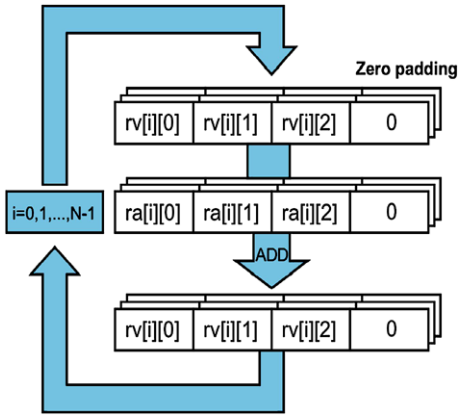

Fig. 5 SIMD alignment via padding for velocity update function

is over the three Cartesian dimensions:

$$\begin{aligned}&\text{for } (i = 0;\ i < N;\ i{+}{+})\\&\text{for } (a = 0;\ a < 3;\ a{+}{+})\\&rv[i][a] = rv[i][a] + ra[i][a];\end{aligned}$$

Our SIMD solution redefines the velocity array $rv[N][3]$ and acceleration array $ra[N][3]$ (N is the number of atoms) to array $rv[N][4]$ and $ra[N][4]$ by padding zero to each row of both arrays. For example, each $\{rv[i][0], rv[i][1], rv[i][2]\}$ is made $\{rv[i][0], rv[i][1], rv[i][2], 0\}$ by padding 0. Subsequently, we unroll the inner loop, pack each $\{rv[i][0], rv[i][1], rv[i][2], 0\}$ to vector *rvvec* and $\{ra[i][0], ra[i][1], ra[i][2], 0\}$ to vector *ravec*, and add them concurrently to obtain the result. The scheme of the above SIMDization are given in Fig. 5. The effect of this SIMDization can be analyzed as follows. The computation is reduced from $3N$ to N with an ideal speedup of 3. For the memory accessing part in the ideal case (i.e., if all data are stored in the cache), it reduces the number of memory accesses from $9N$ to $3N$. (Due to the zero padding, all vector load and store operations start from an aligned address, as the stride is 16 bytes for each $rv[i][0]$ and $ra[i][0]$). Therefore, the ideal memory accessing speedup is 3. However, there are some overhead because the padding introduces a 33% increase of memory bandwidth as the padded array is 33% larger. More detailed analysis will be given in the following subsection.

4 Performance test and analysis

The performance and scalability of the SHPF applied to MD has been tested on various multicore clusters: 106,496 IBM BlueGene/L nodes (each with two IBM PowerPC 440 processors at 700 MHz clock) at Lawrence Livermore National Laboratory, 32,768 IBM BlueGene/P nodes (each with four 450 POWER PC processors at 850 MHz clock) at Argonne National Laboratory, and dual quadcore Nehalem Intel Xeon (2.33 GHz clock) and dual quadcore AMD Opteron (2.3 GHz) based clusters at the High Performance Computing and Communications facility of the University of Southern California (HPCC-USC).

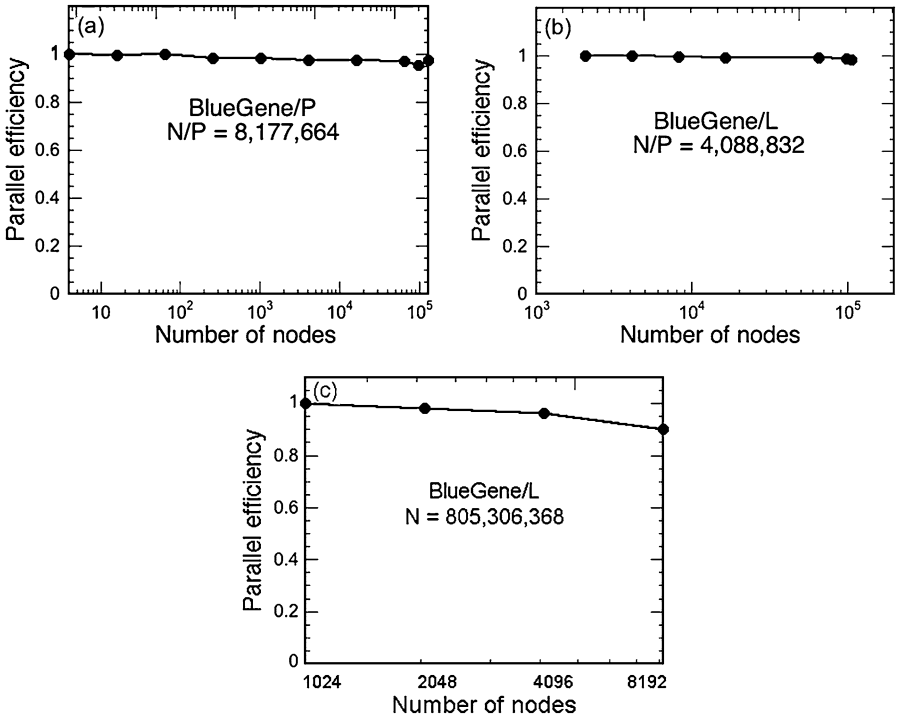

Fig. 6 (**a**) and (**b**) inter-node weak-scalability test, (**c**) inter-node strong-scalability test

4.1 Inter-node scalability

We first test the internode weak scalability on 106,496 IBM BlueGene/L nodes (each with two IBM PowerPC 440 processors at 700 MHz clock) at Lawrence Livermore National Laboratory, and 32,768 IBM BlueGene/P nodes (each with four 450 POWER PC processors at 850 MHz clock) at Argonne National Laboratory. Here, the problem size (i.e. the number of atoms N) is scaled linearly with the number of nodes P, and the internode weak-scaling parallel efficiency of P nodes over Q nodes is defined as $(Time_{p\_cores}/P)/(Time_{Q\_core}/Q)$. Figure 6(a) shows the internode weak-scaling parallel efficiency for 8,177,664P-atom silica systems as a function of P on BlueGene/P, and Fig. 6(b) shows that for 4,088,832P-atoms on BlueGene/L. Our scheme 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.

We also test the strong scalability of internode parallelism. Strong-scaling speedup S_P on P nodes is the running time on one node divided by that on P nodes, and the efficiency E_P is defined as S_P/P. We fix the problem size as 805 million atoms, whereas P varies up to 8,192 nodes. Figure 6(c) shows the strong-scaling parallel efficiency as a function of P on the BlueGene/L cluster based on the speedup of 1,024 nodes. The figure shows that our parallelization scheme maintains a decent

strong-scaling parallel efficiency over 0.90 up to 8,192 nodes based on the speedup over 1,024 nodes on BlueGene/L cluster.

4.2 Inter-core scalability

Next, we test the intercore multithreading strong-scalability of MD on a dual quad-core 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 kept constant. We then define the intercore (intranode) multithreading parallel efficiency E_{NT} as S_{NT}/N_T. Figure 7 shows the multithreading parallel efficiency as a function of the number of worker threads ranging from 1 to 8. The program scales well up to 8 threads on the 8-core platform for the two-body force calculation with efficiency 0.89, while for the entire program the efficiency is 0.65.

It is important to identify the major 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 be due to the overhead for maintaining cache coherency when there is false sharing of cacheline as discussed in Sect. 3.2. Here, we use the Intel Vtune performance Analyzer 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 code without padding and that the modified data sharing ratio is 9 times less, which indicates a large performance gain from padding. Our experiments show that

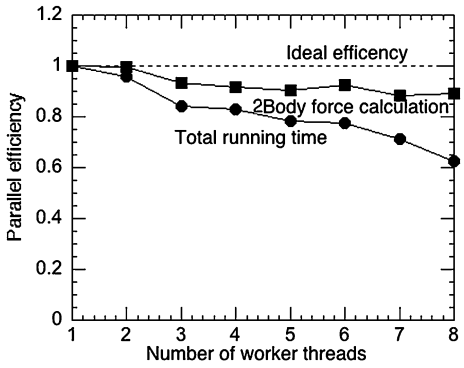

Fig. 7 Intranode multithreading parallel efficiency on Xeon platform

Table 1 Profiling of cache coherence transactions. The granularity N/P is 3072 atoms in both results with 4 threads

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

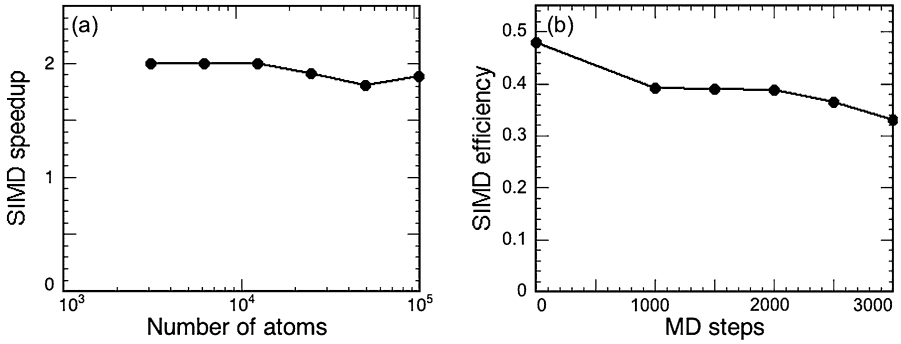

Fig. 8 SIMD performance test. (**a**) SIMD speedup for various problem sizes. (**b**) SIMD efficiency as a function of MD time steps

the padding technique gains average speedup of 53% for all granularities compared to the code without padding.

4.3 SIMD performance tests and analysis

We test our data-level optimization scheme by measuring SIMD speedup and SIMD efficiency averaged over MD steps for various problem sizes on Intel Core i7 E920 processor. Here, the SIMD speedup is defined as the ratio of the running time of the program without SIMD optimization to the optimized running time; and the SIMD efficiency is the ratio of the measured SIMD speedup over the theoretical peak value (i.e. 4 for SSE). Performance tests are performed for silica system with the initial temperature of 6,000 K. Figure 8(a) shows that the SIMD speedup is around 2 for problem sizes varying from 3,072 to 98,304 atoms.

To study the dynamic nature of SIMD performance during MD simulation, Fig. 8(b) shows the SIMD efficiency as a function of MD time steps for the 98,304-atom silica system. The SIMD efficiency is found to degrade from 0.5 to 0.33 within 3,000 steps. Profiling reveals that the dynamic degradation of SIMD efficiency is a consequence of the increased memory access penalty caused by atom diffusion and migration as the simulation progresses. To understand this effect, below we provide a theoretical analysis of SIMD speedup as well as the effect of memory access penalty on SIMD speedup.

Analysis of SIMD speedup: Theoretical SIMD speedup is related to data type, i.e., 4 and 2 for float and double, respectively. In real applications, however, this ideal SIMD speedup is rarely achieved due to the presence of unSIMDizable code segments. To explain the discrepancy between the theoretical SIMD speedup of 4 and the measured value around 2 in Fig. 8(a), we here use a model that is based on pre-SIMD profiling data.

SIMDizability of our application is characterized by the SIMDizable factor, $F_{\text{SIMDizable}} = T_{\text{SIMDizable}}/T_{\text{all}}$, where $T_{\text{SIMDizable}}$ is the running time of the SIMDizable part in the original code and T_{all} is that for the entire program. We then introduce an estimation of application-based ideal SIMD speedup, $E_{\text{App}} = [F_{\text{SIMDizable}}/N_{\text{type}} +$

Table 2 Tests of the estimated application-based ideal SIMD speedup model for MD

Problem size	$T_{\text{SIMDizable}}$ (clock cycles)	T_{all} (clock cycles)	$F_{\text{SIMDizable}}$	E_{App}	Real SIMD speedup	Relative error (%)
6,144	180	261	0.689	2.07	2.00	3
12,288	357	514	0.695	2.09	2.00	4
24,576	735	1,040	0.705	2.12	1.91	10
98,304	308	427	0.722	2.18	1.98	12

$(1 - F_{\text{SIMDizable}})]^{-1}$, where N_{type} is the data type-dependent ideal SIMD speedup (i.e., 4 for float and 2 for double). We can estimate E_{App} by obtaining its parameters from pre-SIMD profiling of the original program. The pre-SIMD profiling data of our MD application with various problem sizes are estimated by measuring $T_{\text{SIMDizable}}$ and T_{all} of the most time-consuming force calculation part using the Intel VTune Performance Analyzer on Intel Xeon Quadcore processor (the major unSIMDizable part of the force calculation in our MD program due to the force table lookup operation). Table 2 lists the measured $T_{\text{SIMDizable}}$ and T_{all} with different problem sizes, together with the corresponding SIMDizable factors $F_{\text{SIMDizable}}$ and the application-based ideal SIMD speedups E_{App}. Table 2 also shows that the actual SIMD speedups obtained by SIMDization of the code agree well with the estimated ones (with a maximum difference of only 12%).

Analysis of the effect of memory-access penalty on SIMD performance: We also analyze and quantify the effect of memory-access penalty on SIMD speedup in order to explain the dynamic degradation of SIMD performance during MD simulation in Fig. 8(b). We again use the same velocity-update example as in Sect. 3.3.2, but with different stride sizes to mimic the effect of different memory access patterns as the result of atom migration. Since the positions of all atoms are stored in one large one-dimensional array, different patterns of accessing the position array are associated with different cache and DTLB misses. Below, we model the SIMD performance including these penalties.

Let T_{c} denote the running time for each computation (e.g., add, subtract, and multiply), N_{c} the number of computations, T_{LS} the load/store cost, and N_{LS} the number of load/store under ideal condition as from L1 cache for both X86 instruction set and SIMD instruction set (SSE3) (throughput of 1 for both computation and load/store). We further define a memory access penalty function P, while P_{orig} and P_{SIMD} represent the penalty before and after SIMDization, respectively. Then the total memory accessing time T_M can generally be expressed as

$$T_M = T_{\text{LS}} N_{\text{LS}} + P. \tag{1}$$

In the case of the original velocity-update subroutine, each inner loop involves three memory accesses and one computation, resulting in the running time of $T_{\text{c}} + 3T_{\text{LS}}$. Thus, the estimated total running time ET_{orig} is given by

$$ET_{\text{orig}} = 3NT_{\text{c}} + 9NT_{\text{LS}} + P_{\text{orig}}. \tag{2}$$

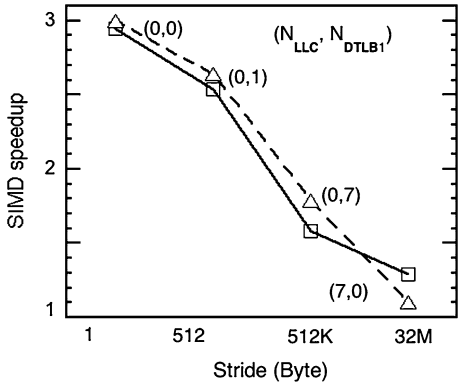

Fig. 9 SIMD analysis: squares, triangles show actual, estimated speedup for various strides

Similarly, the estimated total running time for the subroutine after SIMDization is

$$ET_{\mathrm{SIMD}} = NT_{\mathrm{c}} + 3NT_{\mathrm{LS}} + P_{\mathrm{SIMD}} \tag{3}$$

Relationship between the memory accessing penalties of the SIMDized and un-SIMDized velocity-update subroutines can be represented by $\alpha = P_{\mathrm{orig}}/P_{\mathrm{SIMD}}$, which is the reduced memory accessing penalty due to the data-packing preparation for SIMD. (In our case, α is around 1, since the packed data for SIMD use are from very close memory sections.) The SIMD speedup $S_{\mathrm{SIMD}} = ET_{\mathrm{orig}}/ET_{\mathrm{SIMD}}$ is then given by

$$S_{\mathrm{SIMD}} = \frac{3NT_{\mathrm{c}} + 9NT_{\mathrm{LS}} + P_{\mathrm{orig}}}{NT_{\mathrm{c}} + 3NT_{\mathrm{LS}} + P_{\mathrm{SIMD}}} = \begin{cases} \frac{3(NT_{\mathrm{c}}+3NT_{\mathrm{LS}})/P_{\mathrm{orig}}+1}{(NT_{\mathrm{c}}+3NT_{\mathrm{LS}})/P_{\mathrm{orig}}+\alpha^{-1}} & (P_{\mathrm{orig}} \neq 0) \\ 3 & (P_{\mathrm{orig}} = 0) \end{cases}. \tag{4}$$

According to (4), if the memory accessing is all from L1 cache so that the penalty is extremely small (i.e., $P = 0$), then the SIMDized program will achieve an ideal speedup of 3. On the other hand, when the memory accessing is extremely poor, i.e., when P is large, the SIMD speedup decreases to 1.

As mentioned before, we modify the original velocity-update subroutine by adding a stride parameter. Here, we define the stride as the distance between two neighboring elements in accessing. We perform the experiments on an Intel Core i7 platform, where the major memory accessing penalties are from DTLB miss introducing around 7 cycle penalty and last level cache (LLC) miss introducing around 80 cycle penalty (last level cache latency + bus transaction). For simplicity, we use the sum of LLC miss penalty and DTLB miss penalty to approximate P_{orig} in (4). Figure 9 shows the results of our experiments with stride length ranging from 4 Bytes to 32 MBytes. The memory accessing is characterized by the pair $(N_{\mathrm{LLC}}, N_{\mathrm{DTLB}})$, where N_{LLC} denotes the number of LLC misses and N_{DTLB} that of DTLB misses. We use the Intel VTune Performance Analyzer to collect N_{LLC} and N_{DTLB}, which are averaged over 1,000 iterations before rounded off to an integer. In Fig. 9, the squares denote the actual SIMD speedup, while the triangles show the estimated SIMD speedup calculated from (4) using the parameters estimated from pre-SIMD profiling data. Our application-based ideal SIMD speedup model is accurate with

relative errors of 3–12% from the measured values. The figure also shows that the SIMD speedup decreases considerably due to LLC or DTLB misses, which is likely the main reason of the SIMD performance degradation in MD simulations (Fig. 8(b)), pointing out that further memory optimization is needed to enhance the SIMD performance.

5 Conclusions

In summary, we have developed a scalable hierarchical parallelization scheme for molecular dynamics simulation, thereby achieving almost ideal weak scalability on BlueGene/L and P clusters as well as good strong scalability BlueGene/P cluster. Within each compute node, multithreading has achieved reasonable intercore parallel efficiency combined with intracore 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.

Acknowledgements This work was supported by NSF-ITR/PetaApps/EMT, DOE-SciDAC/BES/EFRC, ARO-MURI, and DTRA.

References

1. Vashishta P, Bachlechner ME, Nakano A, Campbell TJ, Kalia RK, Kodiyalam S, Ogata S, Shimojo F, Walsh P (2001) Multimillion atom simulation of materials on parallel computers-nanopixel, interfacial fracture, nanoindentation, and oxidation. Appl Surface Sci 182:258–264
2. Nomura K et al (2009) A metascalable computing scheme for large spatiotemporal-scale atomistic simulations. In: Proceedings of the 2009 international parallel and distributed processing symposium. IEEE Press, New York
3. Dongarra J et al (2007) The impact of multicore on computational science software. In: CTWatch
4. Nakano A et al (2001) Scalable atomistic simulation algorithms for materials research. In: SuperComputing
5. Ohno Y (2007) A 128 Tflops calculation for x-ray protein structure analysis with special-purpose computers MD-GRAPE3. In: SuperComputing
6. Shaw DE (2007) Anton, a special-purpose machine for molecular dynamics simulation. In: ISCA
7. Scrofano R, Prasanna VK (2005) Preliminary investigation of advanced electrostatics in molecular dynamics on reconfigurable computers. In: SuperComputing, New York, NY
8. Erez M et al (2004) Analysis and performance results of a molecular modeling application on Merrimac. In: SuperComputing, Washington, DC
9. Erez M et al (2007) Executing irregular scientific applications on stream architectures. In: ICS, New York, NY
10. Dally WJ et al (2003) Merrimac: supercomputing with streams. In: SuperComputing, Washington, DC
11. Almasi GS et al (2001) Demonstrating the scalability of a molecular dynamics application on a petaflop computer. In: ICS, New York, NY, 2001
12. Phillips JC et al (2002) NAMD: Biomolecular simulations on thousands of processors. In: Proceedings of supercomputing (SC2002). IEEE/ACM, New York
13. Peng L et al (2009) High-order stencil computations on multicore clusters. In: Proceedings of the 2009 international parallel and distributed processing symposium

14. Chang H, Sung W (2008) Efficient vectorization of SIMD programs with non-aligned and irregular data access hardware. In: Proceedings of the 2008 international conference on compilers, architectures and synthesis for embedded systems, Atlanta, GA, USA
15. McKinley KS et al (1996) Improving data locality with loop transformations. ACM Trans Program Lang Syst 18:424–453
16. Darte A, Robert Y (1994) On the alignment problem. Parallel Process Lett 4:259–270
17. Eichenberger AE et al (2004) Vectorization for SIMD Architectures with alignment constraints. ACM SIGPLAN Not 39:82–93