Performance Characteristics of Hardware Transactional Memory for Molecular Dynamics Application on BlueGene/Q: Toward Efficient Multithreading Strategies for Large-Scale Scientific Applications

Manaschai Kunaseth, Rajiv K. Kalia,
Aiichiro Nakano, Priya Vashishta
Collaboratory for Advanced Computing and Simulations
University of Southern California
Los Angeles, CA USA
{kunaseth,rkalia,anakano,priyav}@usc.edu

David F. Richards, James N. Glosli
Lawrence Livermore National Laboratory
Livermore, CA USA
{richards12,glosli1}@llnl.gov

***Abstract*—We have investigated the performance characteristics of hardware transactional memory (HTM) on the BlueGene/Q computer in comparison with conventional concurrency control mechanisms, using a molecular dynamics application as an example. Benchmark tests, along with overhead-cost and scalability analysis, quantify relative performance advantages of HTM over other mechanisms. We found that the bookkeeping cost of HTM is high but that the rollback cost is low. We propose transaction fusion and spatially-compact scheduling techniques to reduce the overhead of HTM with minimal programming. A strong scalability benchmark shows that the fused HTM has the shortest runtime among various concurrency control mechanisms without extra memory. Based on the performance characterization, we derive a decision tree in the concurrency-control design space for multithreading application.**

Keywords-Hardware transactional memory; molecular dynamics; BlueGene/Q; multithreading

I. INTRODUCTION

The rapid growth in computing power enables scientists to address more challenging problems by using high-end supercomputers [1, 2]. However, improvements in computing power are now gained using multicore architectures instead of increased clock speed [3]. The delivery of Sequoia, a 1.6-million core BlueGene/Q supercomputer at Lawrence Livermore National Laboratory (LLNL), marks the end of the free-ride era—legacy parallel applications can no longer increase performance on a new chip without substantial modification. Especially with 64 threads per node of BlueGene/Q, most of the existing distributed-memory parallel algorithms via message-passing are not suitable for the shared-memory model in multicore platform. Thus, an efficient multithreading scheme is needed to exploit the benefits from such large-scale multicore architectures. However, this is a major challenge due to the difficulty of handling race conditions.

Traditionally, race conditions in time-sharing processors and early multicore chips were avoided effectively by conventional concurrency controls such as locks, atomic operations, and data privatization. However, the performance of these techniques tends to be limited on systems with a large number of concurrent multithreads such as BlueGene/Q. Furthermore, the fine-grain race conditions often encountered in real-world applications such as molecular dynamics (MD) [4-6] further limits the utility of traditional concurrency controls. Addressing these issues typically requires extensive coding effort, which limits wide use of massive multithreading platforms. Hence, alternative concurrent controls need to be established.

To improve the programmability on its large multithreading system, BlueGene/Q has become the first commercial platform with transactional memory (TM) implemented in hardware [7, 8]. TM is an opportunistic concurrency control [9], allowing multiple threads to execute through a critical section concurrently by simply defining a critical code block. Since software TM usually suffers considerable overhead [10], hardware TM (HTM) on BlueGene/Q is expected to significantly reduce runtime overhead. This makes HTM on BlueGene/Q a viable option to deal with race conditions effectively in a large multithreading environment. However, the actual performance of HTM on BlueGene/Q compared to other concurrency control mechanisms on real-world applications has not been widely reported. A study of HTM performance along with the traditional mechanisms on BlueGene/Q is of great significance.

In this paper, we characterize the performance of HTM on BlueGene/Q, along with those of traditional concurrency control mechanisms using the MD code, ddcMD [2, 11], as an example application. We also propose transaction fusion and spatially-compact scheduling techniques to reduce the overhead of HTM with minimal programming. Based on the characteristics of concurrency controls, we propose a decision tree in the concurrency-control design space for multithreading applications.

This paper is organized as follows. Section II provides background information on MD and concurrency controls. Section III characterizes each concurrency control mechanism based on a cost model. Section IV analyzes the performance of concurrency controls including HTM for ddcMD code. Section V proposes a decision tree in the multithreading concurrency-control design space. Conclusions are drawn in section VI.

II. Background

Section II-A introduces the MD application ddcMD, which will be used in the benchmark in section IV. Section II-B describes race condition in MD along with several conventional concurrency control mechanisms. Section II-C explains the HTM implemented on BlueGene/Q.

A. *Hybrid MPI/OpenMP Molecular Dynamics*

Molecular dynamics simulation follows the phase-space trajectories of an N-particle system. The forces between particles are computed from the gradient of a potential energy function Φ ($\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N$), where $\mathbf{r}_i$ is the position of the i-th particle. Positions and velocities of all particles are updated at each MD step by numerically integrating coupled ordinary differential equations. The dominant computation of MD simulations is the evaluation of the forces and associated potential energy function. One model of great physical importance is the interaction between a collection of point charges, which is described by the long-range, pair-wise Coulomb field $1/r$ (where r is the interparticle distance), requiring $O(N^2)$ operations to evaluate. Many methods exist to reduce this computational complexity [12-14]. We focus on the highly efficient particle-particle/particle-mesh (PPPM) method [12]. In PPPM, the Coulomb potential is decomposed into two parts: A short-range part that converges quickly in real space and a long-range part that converges quickly in reciprocal space. The division of work between the short-range and long-range part is controlled through a "screening parameter" α, and computational complexity for the force calculation is reduced to $O(N\log N)$.

In this paper, we use a hybrid parallel implementation of the ddcMD code [11], based on the message-passing interface (MPI) and OpenMP. In this program, particles are assigned to nodes using a particle-based domain decomposition and "ghost-atom" information is exchanged via MPI. Parallelization across the cores of a node is accomplished using multithreading via OpenMP, see Fig. 1. The number of particles in each node is denoted by $n = N/P$, where P denotes the number of nodes used in the simulation. Although both short-range and long-range kernels are parallelized using OpenMP, we focus on the more challenging short-range computation.

The short-range kernel of PPPM computes a sum over pairs: $\Phi = \sum_{i<j} q_i q_j \mathrm{erfc}(\alpha r_{ij})/r_{ij}$, where q_i is the charge of particle i and r_{ij} is the separation between particles i and j. Though this work is focused on this particular pair function, much of the work can be readily applied to other pair functions.

B. *Race Conditions and Traditional Concurrency Control Mechanisms*

A race condition is an unsafe situation where program behavior depends on the relative timing of events. For example, when two or more threads read from and write to the same memory address simultaneously, program results may depend on the order in which the reads and writes are processed. The code section with this possible memory conflict is called a *conflict region*. Memory conflicts can

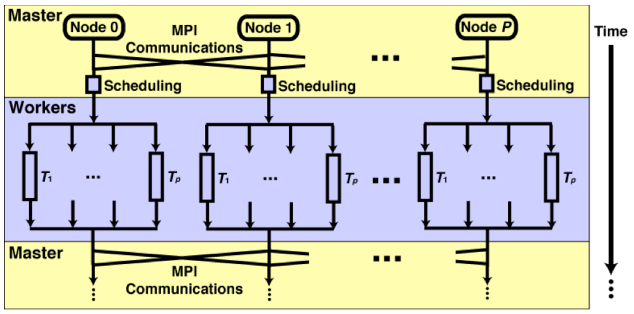


Fig. 1. Schematic diagram of hybrid MPI/OpenMP parallelization.

occur in ddcMD when multiple threads simultaneously update the force of the same particle. Although there is a large number of particle pairs that are prone to memory conflict, the actual occurrence of memory conflict during the force update is rare.

To handle the race condition, several concurrency control mechanisms are available. The conventional concurrency control mechanisms such as lock and atomic operation are available in the OpenMP framework. Alternative data privatization schemes require more coding effort, since automatic array reduction is not yet implemented in general OpenMP distributions (data privatization only involving a scalar reduction variable is available in OpenMP). The detail of these mechanisms is described in the following.

1) OpenMP Critical (OMP-critical): OMP-critical is the simplest form of concurrency control, where the conflict region is wrapped by the `#pragma omp critical` directive. OMP-critical ensures atomicity of the execution in the conflict region by serialization, *i.e.*, allowing only one thread at a time to compute in the conflict region. To achieve this, OMP-critical uses a single global lock to enforce serialization. The major advantage of OMP-critical is its ease of use, but the global lock can cause a severe bottleneck, resulting in tremendous performance degradation.

2) OpenMP Atomic (OMP-atomic): Atomic operations in OpenMP are applied by inserting the `#pragma omp atomic` directive before an atomic update. This causes the compiler to take extra measures to gurantee the atomicity of the scalar update on the following line of code. This mechanism is suitable only for conflict regions that consist solely of a single scalar update.

3) Data Privatization: Data privatization avoids race conditions completely by performing computation on a thread-local buffer instead of reading/writing into the global memory directly. After each thread finishes its computation, partial results from all the threads are combined into the global memory, incurring computational overhead. The main advantage of data privatization is its excellent scalability. However, data privatization needs much more memory than other concurrency controls for local buffer allocation.

C. Hardware Transactional Memory on BlueGene/Q

TM is an opportunistic concurrency control mechanism. It avoids memory conflicts by monitoring a *transaction*, a set of speculative operations in a defined code section. Numerous TM algorithms have been proposed [15-19]. BlueGene/Q is the first commercially available platform that implements TM in the hardware using multi-versioned L2 cache. This hardware feature allows existence of multiple copies of the same physical memory address. When conflict is detected by the hardware, the OS kernel is signaled by the hardware to discard and restart the speculative executions in the conflicted transaction—a process called *rollback*. If no memory conflict is detected, the transaction is committed at the end of the transaction and the change made by the transaction will be made available to all the threads.

To use the HTM, programmers annotate conflict regions with the `#pragma tm_atomic` directive. (This directive is similar to OMP-critical.) This procedure requires minimal coding effort while allowing efficient non-blocking synchronization for multiple threads. However, rollback possibly undermines the potential benefit from HTM. Numerous factors such as algorithms, number of threads, memory layout, and granularity, are likely to influence the number of rollbacks and hence performance. Earlier evaluations of HTM performance on BlueGene/Q are available in [7, 8].

Our HTM benchmarks are performed on Sequoia, the BlueGene/Q cluster at LLNL. The cluster and compiler specifications used in the benchmark are summarized in Table 1. The `-qtm` compiler option is used to enable HTM features. HTM statistics such as numbers of transactions and rollbacks are obtained from HTM-runtime generated report. Details of BlueGene/Q architecture can be found in [20].

TABLE I. SPECIFICATION OF BLUEGENE/Q AT LLNL.

Processor	PowerPC A2
Clock speed	1.6 GHz
Physical cores per node	16
Hardware threads per core	4
Instruction unit	1 integer/load/store OP + 1 floating point OP per clock cycle
L1 data cache	16 KB (per core)
L2 data cache	32 MB (shared by all cores)
Memory per node	16 GB DDR3
Networking	5D torus
Number of nodes/cores	98,304 nodes/1,572,864 cores
Compiler	IBM XLC/C++ version 12.1: mpixlc_r

III. PERFORMANCE CHARACTERIZATION OF CONCURRENCY CONTROL MECHANISMS

In this section, the characteristics of four concurrency-control mechanisms—OMP-critical, OMP-atomic, data privatization, and HTM—are analyzed using a micro benchmark running a single thread on a single core to obtain a cost model.

A. Concurrency-Control Cost Modeling

Consider a computational kernel, where a conflict region consists of several atomic updates placed inside a loop structure, see Fig. 2(a). The total overhead cost of a control mechanism is estimated from:

$$t_\beta = t_{\text{control}} - t_{\text{serial}} \quad (\beta \in \{\text{critical,atomic,HTM}\}) \quad , (1)$$

where t_{control} is the running time of the kernel with concurrency control, t_{serial} is the running time of the serial kernel. The cost model for each concurrency control mechanism is:

OMP-critical: Here the cost mainly comes from the process of obtaining a single global lock at the beginning of the critical section, and releasing the lock at the end. Since the lock/unlock process is not involved in any calculation inside the conflict region, t_{critical} is independent of the size of calculation within the conflict region. Therefore, the cost model of OMP-critical reads

$$t_{\text{critical}} = mc_{\text{lock+unlock}} \, , \tag{2}$$

where m denotes the number of iterations of the for loop in the benchmark kernel, and $c_{\text{lock+unlock}}$ denotes the overhead cost of each OMP-critical.

OMP-atomic: The cost is associated with an atomic load/store hardware instruction, which is more costly than the ordinary load/store instruction. In BlueGene/Q, high-performance atomic operations are implemented as part of L2 cache access [20]. Since OMP-atomic can only avoid conflict in one atomic update, the cost model of OMP-atomic depends on the number of atomic updates inside the conflict region. The atomic operation cost model reads

$$t_{\text{atomic}} = m\mu c_{\text{atomic}} \, , \tag{3}$$

where μ is the number of atomic updates enforced with OMP-atomic, and c_{atomic} is the overhead cost of each OMP-atomic.

HTM: Each memory address involved in a transaction needs to be monitored for conflict during runtime. This requires additional steps to prepare each transaction in case of rollback. This process incurs a bookkeeping cost (*i.e.*, register check-pointing, operation confinement [7]) associated with every transaction. Furthermore, all speculative executions occur in L2 cache, which incurs considerable latency for L2 accesses. Therefore, the cost of

```
double x[SIZE];
for (int i = 0;i < niter;i++)
{
   for (int j=0;j<=SIZE-8;j+=8*stride)
   {
      x[j         ]+=i*j;
      x[j+  stride]+=i*(j+1);
      x[j+2*stride]+=i*(j+2);
      x[j+3*stride]+=i*(j+3);
      x[j+4*stride]+=i*(j+4);
      x[j+5*stride]+=i*(j+5);
      x[j+6*stride]+=i*(j+6);
      x[j+7*stride]+=i*(j+7);
   }
}
```

```
double x[SIZE];
for (int i = 0;i < niter;i++)
{
   for (int j=0;j<=SIZE-8;j+=8*stride)
   {
      #pragma tm_atomic
      {
         x[j         ]+=i*j;
         x[j+  stride]+=i*(j+1);
         x[j+2*stride]+=i*(j+2);
         x[j+3*stride]+=i*(j+3);
      }
      x[j+4*stride]+=i*(j+4);
      x[j+5*stride]+=i*(j+5);
      x[j+6*stride]+=i*(j+6);
      x[j+7*stride]+=i*(j+7);
   }
}
```

Fig. 2. The benchmark kernel: (a) serial code without concurrency control; and (b) HTM control over 4 updates.

each transaction also depends on the size of conflict region. Hence, the HTM cost model reads:

$$t_{\text{HTM}} = m(c_{\text{HTM\_overhead}} + \mu c_{\text{HTM\_update}}) \,, \tag{4}$$

where $c_{\text{HTM\_overhead}}$ denotes the bookkeeping overhead per transaction, and $c_{\text{HTM\_update}}$ denotes extra cost per atomic update within the transaction. Since the benchmark only involves a single thread, no rollback occurs and thus the cost of rollback is not included in the model in this section.

Data Privatization: The computational overhead is caused by reduction operation in the combining phase. Since the reduction cost is independent of the number of iterations, the cost of data privatization is often negligible compared to the large amount of time spent in the loop. The reduction cost is not applicable to the single thread cost model in this section, and will be included in the ddcMD benchmark in section IV.

B. Model Parameter Fitting Using Microbenchmark

Figure 2(a) shows the serial code as a reference. The serial kernel consists of a loop, in which each iteration contains 8 atomic updates. Figure 2(b) shows an example of HTM-control benchmark codes. To identify the cost parameters depending on the number of atomic updates, we vary the number of atomic updates subjected to concurrency controls. For example in Fig. 2(b), HTM is applied to 4 atomic updates. The benchmark is executed using a single thread on a single core.

Figure 3 shows the average overhead cost of OMP-atomic and HTM per iteration. Benchmark results reveal that the cost of OMP-atomic and HTM mechanisms increase as the number of controlled updates increase. The average cost of OMP-atomic per update c_{atomic} is 393.9 cycles, while the average cost of HTM per update $c_{\text{HTM\_update}}$ is 139.4 cycles. The HTM has additional bookkeeping overhead, $c_{\text{HTM\_overhead}}$ = 970.9 cycles per transaction. The plot shows that OMP-atomic has less overhead than HTM for small number of updates (≤ 4), while HTM has less overhead for larger number of updates. HTM has a large bookkeeping overhead, which is absent in OMP-atomic. However, the cost per update is smaller for HTM than OMP-atomic. This results in the observed crossover of the relative performance advantage between OMP-atomic and HTM.

For OMP-critical, the fitted cost of lock+unlock $c_{\text{lock+unlock}}$ is 418.0 cycles. This cost is independent of the number of atomic updates.

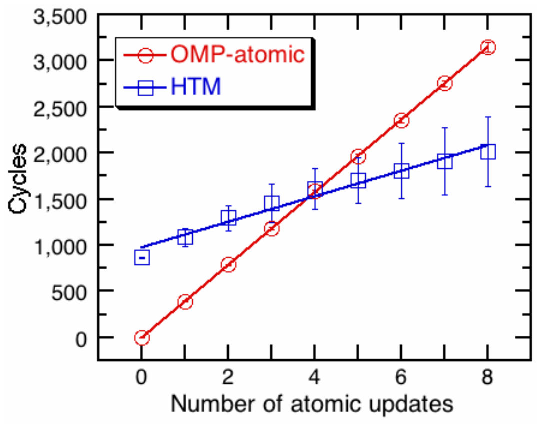


Fig. 3. Overhead cost comparison of OMP-atomic and HTM. The graph indicates a crossover point at μ ~ 4 updates. Error bars for OMP-atomic is too small to be visible.

IV. Molecular Dynamics Benchmark

To evaluate the performance of concurrency control mechanisms in an actual application, we focus on the race conditions in the force update routine of ddcMD. Figure 4 shows the kernel of the ddcMD force update routine, where memory conflict may occur. The code has two conflict regions: (1) the force update in array `fB[j]` in line 28-33; and (2) the force update in array `fA[0]` in line 36-41 in Fig. 4. Concurrency control needs to be applied to these two regions.

Here, we define the *baseline* code as a reference, in which no concurrency control is used (*i.e.*, the best-possible execution time but the result is likely wrong). For OMP-critical or HTM controlled concurrency, the corresponding compiler directive (*i.e.*, `#pragma omp critical` or `#pragma tm_atomic`) substitutes the comments in lines

27 and 35. In case of OMP-atomic, the directive `#pragma omp atomic` is inserted before every statement in both conflict regions.

A. Performance Comparison of Concurrency Controls

In this subsection, we perform a strong-scaling benchmark of ddcMD on 64 BlueGene/Q nodes. The total number of particles used in the benchmark is fixed at 1,024,000, where the particles are uniformly distributed.

Figure 5(a) shows the average runtime per MD step for each concurrency control mechanism as a function of the number of threads per node *p* from 1 to 64. The baseline result (black dashed line) exhibits an excellent scalability with 15.7-fold speedup over a single thread performance for 16 threads. This result is unsurprising since each thread runs on 1 of the 16 physical cores on a BlueGene/Q node. However, the speedup of the baseline code increases to 26.5 when the number of threads is increased to 32 threads. The increased performance from 16 to 32 threads comes from the double-instruction units of the PowerPC A2 processor of BlueGene/Q. We observe further increase of speedup to 40.0 for 64 threads. This additional speedup likely arises from latency hiding enabled by hardware multithreading. These special hardware features on BlueGene/Q thus enable multithreading speedup larger than the number of physical cores. Since the baseline code does not guarantee the correctness of the result, this speedup should be regarded as the upper limit of the actual speedup with concurrency controls.

The data privatization code (blue squares) achieves nearly ideal speedup (*i.e.,* almost the same runtime as the baseline that does not guarantee correctness). The measured speedup is 15.7, 26.5, and 38.9 for 16, 32, and 64 threads, respectively. It is worth mentioning that the data privatization implemented in ddcMD employs spatially-compact scheduling technique, which will be discussed in subsection IV-C. This technique significantly reduces the memory footprint, *i.e.,* memory reduction of a factor 64/5.04 ~ 12.7 compared with naïve data privatization for 64 threads [11].

The OMP-critical runtime (orange open squares) is not monotonically decreasing, but instead starts to increase above 16 threads. This is due to the serialization in the OMP-critical mechanism, which limits the parallel speedup for larger numbers of threads according to Amdahl's law. Thus, OMP-critical is only suitable for a small number of threads. The best speedup that OMP-critical mechanism achieved is 6.3 for 16 threads.

The runtime of OMP-atomic (red circles) is high compared to the baseline. However, it still exhibits an excellent scalability, *i.e.* 40.7-fold speedup for 64 threads.

The HTM result (green diamonds) shows the highest overhead cost among all the tested methods. Nevertheless, the scalability of HTM is still high: the measured speedup is 14.9, 24.1, and 33.5 for 16, 32, and 64 threads, respectively. The slightly reduced scalability above 32 threads likely originates from the rollback cost due to increased frequency of conflict.

```
1  FOUR_VECTOR ftmp;
2  ftmp.v = ftmp.x = ftmp.y = ftmp.z = 0.0;
3  double rA0x, rA0y, rA0z;
4  double qA0 = qA[0];
5
6  rA0x = rA[0].x;
7  rA0y = rA[0].y;
8  rA0z = rA[0].z;
9
10 for (int k=0;k<nPList;k++)
11 {
12    double fs,vij,fxij,fyij,fzij;
13    int j=list[k];
14    double complex ff = fv[k];
15    vij= creal(ff);
16    fs = cimag(ff);
17    fxij=fs*(rA0x  - rB[j].x);
18    fyij=fs*(rA0y  - rB[j].y);
19    fzij=fs*(rA0z  - rB[j].z);
20
21    double qBj = qB[j];
22    ftmp.v +=  qBj*vij;
23    ftmp.x +=  qBj*fxij;
24    ftmp.y +=  qBj*fyij;
25    ftmp.z +=  qBj*fzij;
26
27    //conflict region #1 in fB[j]
28    {
29       fB[j].v +=  qA0*vij;
30       fB[j].x -=  qA0*fxij;
31       fB[j].y -=  qA0*fyij;
32       fB[j].z -=  qA0*fzij;
33    }
34 }//end nPList loop
35 //conflict region #2 in fA[0]
36 {
37    fA[0].v += ftmp.v;
38    fA[0].x += ftmp.x;
39    fA[0].y += ftmp.y;
40    fA[0].z += ftmp.z;
41 }
```

Fig. 4. Code fragment from the force update routine in ddcMD code, which has potential memory conflicts.

To improve the performance of HTM, we aim to reduce the overhead cost by fusing multiple transactions into a single transaction, so that the number of transactions is reduced. According to Eq. (4), this will reduce the bookkeeping cost $mc_{\mathrm{HTM\_overhead}}$, though the update cost $m\mu c_{\mathrm{HTM\_update}}$ will be unchanged. To do this, the entire `NPList` loop, along with the second conflict region (line 10-41 in Fig. 4) is wrapped by `#pragma tm_atomic` instead of the conflict regions in the loop. Runtime of fused HTM (purple triangles) is significantly decreased from naïve HTM and becomes even smaller than OMP-critical and OMP-atomic. The speedup of fused HTM is 14.7, 26.1, and 38.2 for 16, 32, and 64 threads, respectively. HTM runtime report shows that the number of committed transactions is reduced by a factor of 5.9 by the transaction fusion technique. Although HTM fusion is a useful technique to reduce the overhead cost, it should be noted that the hardware limitation (*e.g.* L2 cache capacity) poses an upper bound for the size of the fused transaction.

Figure 5(b) shows the runtime of each mechanism normalized by that of the baseline for different numbers of

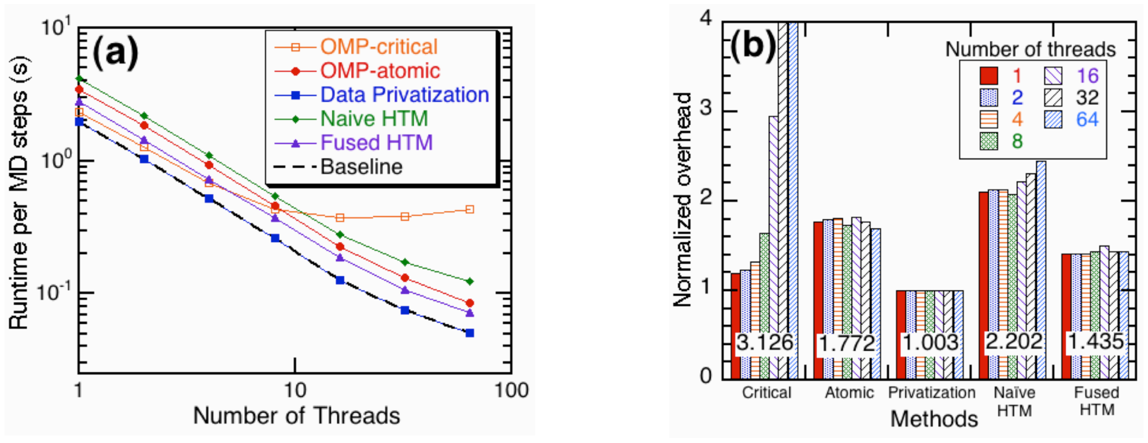


Fig. 5. (a) Strong-scaling runtime of ddcMD comparing the performance of concurrency control mechanisms. (b) Runtime overhead of each concurrency controls normalized by the baseline runtime.

threads, p. The numeral for each set mechanism in the graph indicates the normalized runtime averaged over p: 1.772, 3.126, 1.003, 2.202, and 1.435 for OMP-atomic, OMP-critical, data privatization, naïve HTM, and fused HTM, respectively. Although the runtime of data privatization is the smallest, it incurs a large memory overhead (5.04 times more). Among the concurrency controls without extra memory overhead (OMP-critical, OMP-atomic, naïve HTM, and fused HTM), the fused HTM exhibits the smallest runtime.

B. Rollback Penalty

To quantify the cost penalty and frequency of rollback, we first define the rollback ratio, $f_{\mathrm{rb}} = n_{\mathrm{rb}}/n_{\mathrm{transaction}}$, where n_{rb} and $n_{\mathrm{transaction}}$ denote the total number of rollbacks and that of committed transactions, respectively. These two quantities can be obtained from the HTM-runtime generated report.

Measuring rollback cost is challenging since we need to compare runtime of two HTM codes with/without the race condition, which have almost the same computational pattern. To achieve this, we quantify the rollback cost by measuring runtime difference between normal HTM code (*i.e*., fused HTM from subsection IV-A) and non-conflict HTM code. The non-conflict HTM code is constructed by combining data privatization and HTM mechanisms within a single run. In particular, the force computation is performed in the conflict-free local buffer of each thread. Then, the HTM is applied on top of the local buffers. This still involves the HTM overhead cost but without memory conflict.

In this benchmark, 1,024,000 particles are simulated on 64 nodes of BlueGene/Q using fused HTM code and data privatization/HTM combined code. The number of threads on each node p varies from 1 to 64. In each run, extra runtime from non-conflict code compared to the normal HTM runtime are measured for different numbers of threads. Figure 6 shows the extra runtime as a function of rollback ratio. The total number of committed transactions in all 64 nodes is $n_{\mathrm{transaction}} = 9.85\times10^7$ per MD step, in all test cases. We observe a linear relation between the increased runtime and the rollback ratio. From the slope of the plot, we estimate that the runtime increases by 0.69% per 1% increase of rollback ratio. This indicates insignificant performance penalty caused by rollback in MD. Typical rollback ratio in ddcMD is less than 8% according to the HTM runtime report.

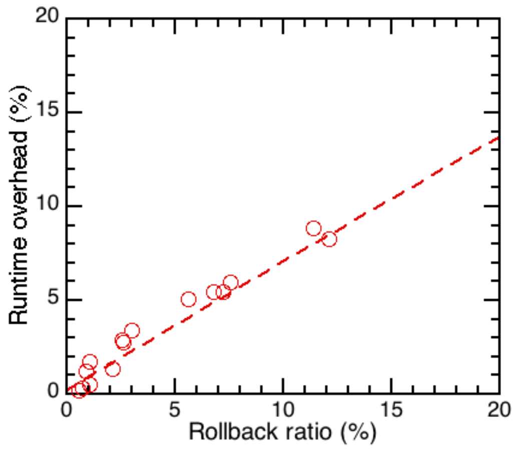


Fig. 6. Impact of rollback on runtime, indicating linear correlation between runtime overhead and rollback ratio.

C. Effect of Scheduling Algorithms on Rollback

In ddcMD, particles assigned to each node are sorted into cubic cells with side length of r_c (the cutoff radius of the short-range interactions) based on their spatial coordinate. This reduces the complexity of short-range force computation from $O(N^2)$ to $O(N)$. Here, the cells in each node are assigned to threads by the load-balance scheduler. In each scheduling step, scheduler picks a cell from the unassigned-cell pool and assigns it to the least-loaded thread to maintain load balance. This scheduling algorithm works well in most cases. However, the choice of the selected cell in each scheduling iteration affects the number of particle pairs with potential memory conflict, which in turn influences the rollback ratio of HTM. Here, we propose a spatially-compact scheduling algorithm, which minimizes the number of rollbacks by preserving the spatial proximity of the cells assigned to each thread. The cell-selection algorithms used in ddcMD (including the proposed one) are described in the following.

Baseline scheduling sweeps cell indices in the *x*, *y*, and *z* directions in turn. This scheduling algorithm has similar characteristic to the naïve `#pragma omp parallel for` with dynamic scheduling. It considerably increases memory conflicts since each thread is likely overlapping with at least two other threads that are assigned nearest-neighbor cells.

Random scheduling picks a cell from the unassigned-cell pool randomly. Compared to the baseline scheduling, this reduces the chance that two or more threads are assigned nearest-neighbor cells; see Fig. 7(a).

Spatially-compact scheduling assigns each cell to the least-loaded thread, while preserving the spatial proximity of the cells assigned to each thread, see Fig. 7(b). This reduces the surface area of the cluster of cells assigned to each thread, which is prone to memory conflicts. More information on the spatially-compact scheduling for MD can be found in [11].

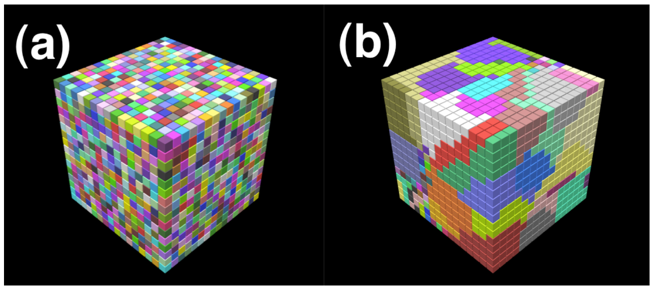


Fig. 7. Workload distribution in a physical space, where cells assigned to different threads are distinguished by different colors. (a) Random scheduling; and (b) Spatially-compact scheduling.

To quantify the effect of scheduling algorithms on the number of rollbacks, we perform a strong-scaling benchmark involving 8-million particles on 64 BlueGene/Q nodes using fused HTM. Here, the cell-selection algorithm used in the thread-level scheduling is varied between baseline, random, and spatially-compact algorithms. Due to the decreasing thread granularity in strong-scaling benchmark, higher rollback ratio is expected for larger numbers of threads.

Figure 8(a) shows the rollback ratio as a function of the number of threads *p*. The graph shows that the rollback ratios of all scheduling algorithms increase almost linearly with *p*. The rollback ratios for baseline, random, and compact schedulings are 32.9%, 11.4%, and 7.3%, respectively for 64 threads. Figure. 8(b) shows the increase of the runtime as a function of *p*, corresponding to the rollback profiles in Fig. 8(a). Here, the extra runtime of baseline scheduling is much higher than the random and compact schedulings, which is in accord with their rollback characteristics. For 64 threads, the extra runtimes of 21.4%, 8.9%, and 5.4% are observed for baseline, random, and compact schedulings, respectively. The ratio of extra runtime in Fig. 8(b) and the rollback ratio in Fig. 8(a) is consistent with that in subsection IV-B. This result demonstrates the importance of thread scheduling in MD when HTM is used. The number of rollbacks can be reduced by the proposed spatially-compact scheduling algorithm.

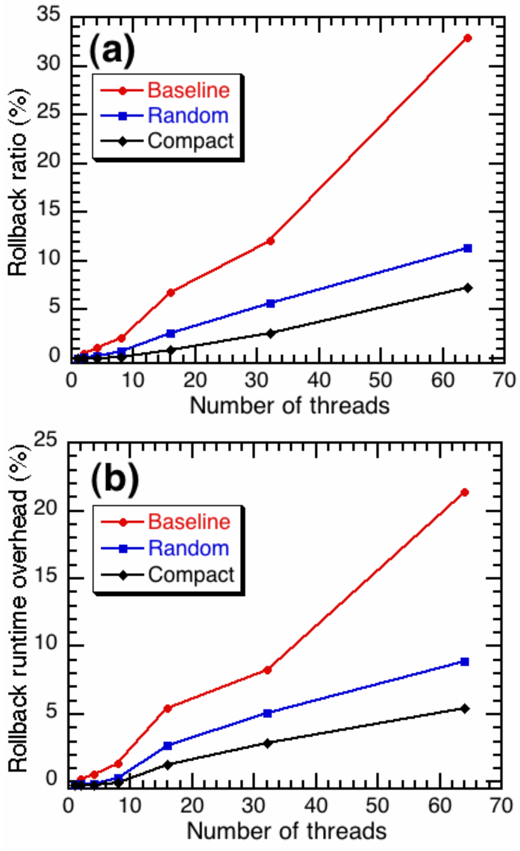


Fig. 8. Rollback ratio (a) and the runtime overhead (b) as a function of the number of threads for the three scheduling algorithms.

V. Design Space for Concurrency Controls

Although the implementation of HTM on BlueGene/Q provides more options to deal with race conditions in massive-multicore multithread programming, choosing the optimal concurrency control mechanism for a particular application is still a difficult issue. In this section, we propose a concurrency-control decision tree to handle race condition in multithread programming, based on the performance characteristics from the previous sections.

To discuss the concurrency-control design space, we introduce several parameters that are associated with program runtime and conflicts. First, we describe parameters that are obtained from pre-profiling of the serial code. Figure 9 shows a typical program structure of loop-oriented applications such as MD. Let *conflict region* (CR) be a part with a race condition in the *target loop* (TL) for multithreading. Pre-profiling is performed to obtain the following parameters: (1) the total time in serial code spent in target loop t_{TL}; and (2) the total time spent in the conflict region t_{CR}. Based on these parameters, we define the fraction of time spent in the conflict region as

$$f_{\mathrm{CR}} = \frac{t_{\mathrm{CR}}}{t_{\mathrm{TL}}} . \quad (5)$$

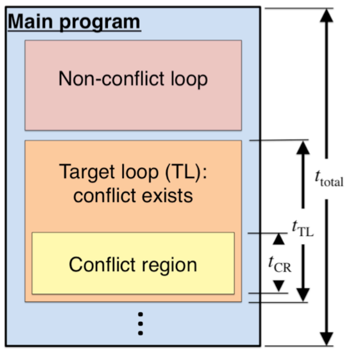


Fig. 9. Typical program structure of loop-oriented applications

In addition to the pre-profiling parameters, we define the number of threads p and the number of atomic updates inside the conflict region μ. The user specifies p, while μ can be obtained easily from code inspection. M_{avail} is the memory available for data privatization.

Figure 10 shows the decision tree to choose the optimal concurrency control mechanism among OMP-critical, OMP-atomic, HTM, and data privatization. The decision tree focuses on providing the best performance, *i.e.*, high scalability and low overhead cost, based on the analysis and benchmark results in the previous sections. In the following, we elaborate the reasoning for each decision.

Data Privatization: According to the benchmark result in section IV, data privatization has the best performance. Namely, it has the smallest overhead among all mechanisms and the decent thread scalability up to 64 threads. Thus, data privatization is the first choice to be considered. Although data privatization includes the reduction overhead (*e.g.* $O(n\log p)$ for hypercube reduction), it is usually negligible. The major condition that restricts the use of data privatization is the available memory, since the simple application of data privatization could require up to p times more memory. If the available memory is not enough for data privatization, we should consider the other mechanisms that do not require extra memory.

OMP-critical: The low overhead cost and the simplicity are the main advantages of OMP-critical. However, its scalability degrades significantly when p is large. Thus, OMP-critical is only suitable for small p (*e.g.* $p \le 8$ in Figs. 5(a) and 5(b)). The critical number of threads $p_{\text{crit}} = 1/f_{\text{CR}}$ denotes the largest number of threads that can theoretically run concurrently without a lock contention for the serialized fraction of f_{CR}. In the decision tree, we recommend $p < p_{\text{crit}}/2$ to ensure reasonable scalability.

OMP-atomic and HTM: Simplicity without compromising scalability is the major advantage of OMP-atomic and HTM. To choose between these two mechanisms, we use the cost models in section III to find the one with the lower overhead. If OMP-atomic is applicable (*i.e.*, only atomic updates are present in the conflict region) and the OMP-atomic cost is lower, then OMP-atomic should be used. Otherwise, the decision depends on conflict rate f_{rb}.

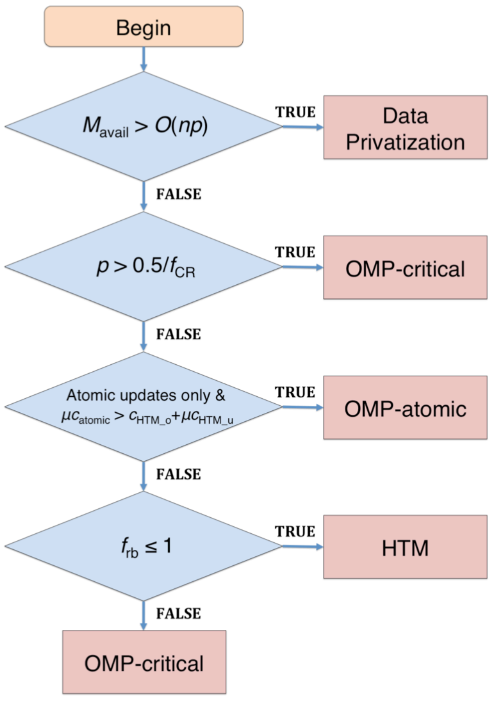


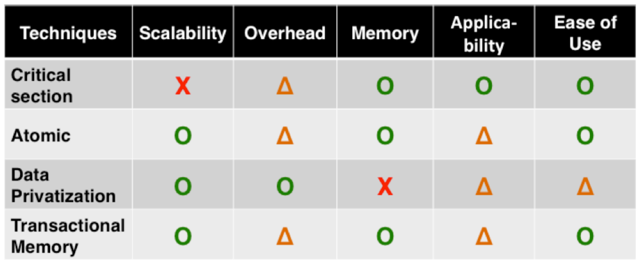
Fig. 10. Decision tree for concurrency-control design space. We should note that OMP-critical at the bottom has very high performance penalty.

If the conflict rate is low, HTM should be used. (Note that the overhead cost of HTM can be further reduced by using HTM fusion technique.) According to Schindewolf *et al*. [8], the threshold for the usability of HTM is $f_{\text{rb}} \le 1$. For $f_{\text{rb}} > 1$, OMP-critical can be used to avoid memory conflicts although it has high penalty. We found that in some cases, minor code structural change could improve the performance of concurrency controls.

Once the decision of concurrency control mechanism is made, we can estimate the theoretical parallel efficiency for each mechanism. Table 2 summarizes the parallel efficiency estimate based on the profiling parameters. In Table 2, t_{total} is the total runtime of the serial code and $c_{\text{reduction}}$ denotes the reduction cost per particle per hypercube dimension. Figure 11 summarizes the advantages/disadvantages of the concurrency control mechanisms considered in this paper.

TABLE II. PARALLEL EFFICIENCY ESTIMATE RESULTING FROM THE DECISION TREE.

Concurrency control mechanism	Parallel efficiency
OMP-critical	$e = \min(\frac{1}{pf_{\text{CR}}}, 1)$
OMP-atomic	$e = \frac{t_{\text{total}}}{t_{\text{total}} + m\mu c_{\text{atomic}}}$
Data privatization	$e = \frac{t_{\text{total}}}{t_{\text{total}} + c_{\text{reduction}} n \log p}$
HTM	$e = \frac{t_{\text{total}}}{t_{\text{total}} + m(c_{\text{HTM\_overhead}} + \mu c_{\text{HTM\_update}})}$



Techniques	Scalability	Overhead	Memory	Applica-bility	Ease of Use
Critical section	X	Δ	O	O	O
Atomic	O	Δ	O	Δ	O
Data Privatization	O	O	X	Δ	Δ
Transactional Memory	O	Δ	O	Δ	O

Fig. 11. Analysis of four concurrency control mechanisms. Circles and triangles denote strong and weak advantages, respectively, while crosses denote disadvantage.

VI. Conclusion and Discussion

We have investigated the performance of HTM and conventional concurrency control mechanisms on BlueGene/Q using microbenchmarks and the MD application ddcMD. The microbenchmark indicates the overhead cost of HTM is smaller than OMP-atomic for a large number of threads. The strong scaling benchmark shows excellent scalability of HTM as well as OMP-atomic and data privatization. The proposed fused-HTM and spatially-compact scheduling techniques have sped up naïve HTM by a factor of 1.53. An analysis of rollback penalty showed minor impact of rollback on the performance of ddcMD. Finally, we have derived a decision tree in the concurrency-control design space for multithreading application. This will help application developers choose the concurrency control mechanism with the best performance for their applications.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-585958). The work at USC was partially supported by DOE BES/EFRC/SciDAC/SciDAC-e/INCITE, NSF PetaApps/CDI, and ONR.

References

[1] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. Towles, "Millisecond-scale molecular dynamics simulations on Anton," in *Proceedings of Supercomputing (SC09)*, Portland, Oregon, 2009.

[2] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz, "Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability," in *Proceedings of Supercomputing (SC07)*, Reno, Nevada, 2007.

[3] S. H. Fuller and L. I. Millett, "Computing performance: game over or next level?," *Computer,* vol. 44, pp. 31-38, 2011.

[4] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kale, J. C. Phillips, and C. Harrison, "Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime," in *Proceedings of Supercomputing (SC11)*, Seattle, Washington, 2011.

[5] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," *Journal of Chemical Theory and Computation,* vol. 4, pp. 435-447, 2008.

[6] K. Nomura, H. Dursun, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, P. Vashishta, F. Shimojo, and L. H. Yang, "A metascalable computing framework for large spatiotemporal-scale atomistic simulations," in *International Parallel and Distributed Processing Symposium (IPDPS09)*, Rome, Italy, 2009.

[7] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *Procceeding of Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, Minnesota, 2012.

[8] M. Schindewolf, B. Bihari, J. Gyllenhaal, M. Schulz, A. Wang, and W. Karl, "What Scientific Applications can Benefit from Hardware Transactional Memory?," in *Proceedings of Supercomputing (SC12)*, Salt Lake City, Utah, 2012.

[9] M. Herlihy and J. E. B. Moss, "Transactional memory - architectural support for lock-free data-structures," *Proceedings of International Symposium on Computer Architecture,* pp. 289-300, 1993.

[10] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?," *Communications of the ACM* vol. 51, pp. 40-46, 2008.

[11] M. Kunaseth, D. F. Richards, J. N. Glosli, R. K. Kalia, A. Nakano, and P. Vashishta, "Scalable data-privatization threading for hybrid MPI/OpenMP parallelization of molecular dynamics," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, 2011.

[12] D. York and W. Yang, "The fast Fourier Poisson method for calculating Ewald sums," *The Journal of Chemical Physics,* vol. 101, pp. 3298-3300, 1994.

[13] R. Hockney and J. Eastwood, *Computer simulation using particles*. New York: McGraw-Hill, 1981.

[14] T. Darden, D. York, and L. Pedersen, "Particle mesh Ewald: An N log(N) method for Ewald sums in large systems," *The Journal of Chemical Physics,* vol. 98, pp. 10089-10092, 1993.

[15] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, "ATLAS: A chip-multiprocessor with Transactional memory support," *Design, Automation & Test in Europe Conference & Exhibition,* vol. 1-3, pp. 3-8, 2007.

[16] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance sparc CMT processor," *IEEE Micro,* vol. 29, pp. 6-16, 2009.

[17] J. T. Wamhoff, T. Riegel, C. Fetzer, and P. Felber, "RobuSTM: A robust software transactional memory,"

Stabilization, Safety, and Security of Distributed Systems, vol. 6366, pp. 388-404, 2010.

[18] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems,* vol. 21, pp. 1793-1807, 2010.

[19] T. Riegel, M. Nowack, C. Fetzer, P. Marlier, and P. Felber, "Optimizing hybrid transactional memory: The importance of nonspeculative operations," *Proceedings of Symposium on Parallelism in Algorithms and Architectures,* pp. 53-64, 2011.

[20] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, G. L. T. Chiu, P. A. Boyle, N. H. Christ, and C. Kim, "The IBM Blue Gene/Q compute chip," *IEEE Micro,* vol. 32, pp. 48-60, 2012.