Abstract

The emergence of the multicore era has granted unprecedented computing capabilities. Extensively available multicore clusters have influenced hybrid message-passing and multithreading parallel algorithms to become a standard parallelization for modern clusters. However, hybrid parallel applications of portable scalability on emerging high-end multicore clusters consisting of multimillion cores are yet to be accomplished. Achieving scalability on emerging multicore platforms is an enormous challenge, since we do not even know the architecture of future platforms, with new hardware features such as hardware transactional memory (HTM) constantly being deployed. Scalable implementation of molecular dynamics (MD) simulations on massively parallel computers has been one of the major driving forces of supercomputing technologies. Especially, recent advancements in reactive MD simulations based on many-body interatomic potentials have necessitated efficient dynamic n-tuple computation. Hence, it is of great significance now to develop scalable hybrid n-tuple computation algorithms to provide a viable foundation for high-performance parallel-computing software on forthcoming architectures.

This dissertation research develops a scalable hybrid message-passing and multithreading algorithm for n-tuple MD simulation, which will continue to scale on future architectures (i.e. achieving metascalability). The two major goals of this dissertation research are: (1) design a scalable hybrid message-passing and multithreading parallel algorithmic framework on multicore architectures and evaluate it on most advanced parallel architectures; and (2) develop a
computation-pattern algebraic framework to design scalable algorithms for general \( n \)-tuple computation and prove its optimality in a systematic and mathematically rigorous manner.

To achieve the first goal, we have developed and thoroughly analyzed algorithms for hybrid message passing interface (MPI) + open multiprocessing (OpenMP) parallelization of \( n \)-tuple MD simulation, which are scalable on large multicore clusters. Two data-privatization thread scheduling algorithms via nucleation-growth allocation have been designed: (1) compact-volume allocation scheduling (CVAS); and (2) breadth-first allocation scheduling (BFAS). These two algorithms combine fine-grain dynamic load balancing and minimal memory-footprint threading. Theoretical study has revealed decent asymptotic memory efficiency for both algorithms, thereby reducing 75% memory consumption compared to a naïve-threading algorithm. Furthermore, performance benchmarks have confirmed higher performance of the hybrid MD algorithm over a traditional algorithm on large multicore clusters, where 2.58-fold speedup of the hybrid algorithm over the traditional algorithm was observed on 32,768 nodes of IBM BlueGene/P.

We have also investigated the performance characteristics of HTM on the IBM BlueGene/Q computer in comparison with conventional concurrency control mechanisms, using an MD application as an example. Benchmark tests, along with overhead-cost and scalability analysis, have quantified 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 have proposed transaction fusion and spatially compact scheduling techniques to reduce the overhead of HTM with minimal programming. A strong scalability benchmark has shown that the fused HTM has the shortest runtime among various concurrency control mechanisms without extra
memory. Based on the performance characterization, we have derived a decision tree in the concurrency-control design space for general multithreading applications.

To achieve the second goal, we have developed a computation-pattern algebraic framework to mathematically formulate general $n$-tuple computation. Based on translation/reflection-invariant properties of computation patterns within this framework, we have designed a shift-collapse (SC) algorithm for cell-based parallel MD. Theoretical analysis has quantified the compact $n$-tuple search space and small communication cost of SC-MD for arbitrary $n$, which are reduced to those in best pair-computation approaches (e.g. eighth-shell method) for $n = 2$. Benchmark tests have shown that SC-MD outperforms our production MD code at the finest grain, with 9.7- and 5.1-fold speedups on Intel-Xeon and BlueGene/Q clusters. SC-MD has also exhibited excellent strong scalability.

In addition, we have analyzed the computational and data-access patterns of MD, which led to the development of a performance prediction model for short-range pair-wise force computations in MD simulations. The analysis and performance model provide fundamental understanding of computation patterns and optimality of certain parameters in MD simulations, thus allowing scientists to determine the optimal cell dimension in a linked-list cell method. The model has accurately estimated the number of operations during the simulations with the maximum error of 10.6% compared to actual measurements. Analysis and benchmark of the model have revealed that the optimal cell dimension minimizing the computation time is determined by a trade-off between decreasing search space and increasing linked-list cell access for smaller cells.

One difficulty about MD is that it is a dynamic irregular application, which often suffers considerable performance deterioration during execution. To address this problem, an optimal
data-reordering schedule has been developed for runtime memory-access optimization of MD simulations on parallel computers. Analysis of the memory-access penalty during MD simulations has shown that the performance improvement from computation and data reordering degrades gradually as data translation lookaside buffer misses increase. We have also found correlations between the performance degradation with physical properties such as the simulated temperature, as well as with computational parameters such as the spatial-decomposition granularity. Based on a performance model and pre-profiling of data fragmentation behaviors, we have developed an optimal runtime data-reordering schedule, thereby archiving speedup of 1.35, 1.36 and 1.28, respectively, for MD simulations of silica at temperatures 300 K, 3,000 K and 6,000 K.

The main contributions of this dissertation research are two-fold: Metascalable hybrid message-passing and multithreading parallel algorithmic framework on emerging multicore parallel clusters, and a novel computation-pattern algebraic framework to design scalable algorithm for general n-tuple computation and prove its optimality in a mathematically rigorous manner. We expect that the proposed hybrid algorithms and mathematical approaches will provide a generic framework to a broad range of applications on future extreme-scale computing platforms.
Table of Contents

CHAPTER 1 INTRODUCTION ...........................................................................................................1

1.1 MOTIVATIONS..............................................................................................................................1
1.2 CHALLENGES AND GOALS..........................................................................................................2
1.3 BACKGROUND..............................................................................................................................3

1.3.1 Molecular Dynamics Simulation .........................................................................................4
1.3.2 Multicore Architectures .........................................................................................................4
1.3.3 Hybrid Parallelization ...........................................................................................................6
1.3.4 n-Tuple Molecular Dynamics ...............................................................................................8
1.3.5 Recent Development of Parallel MD Algorithms .................................................................8

1.4 CONTRIBUTIONS.........................................................................................................................9
1.5 STRUCTURE OF THIS DISSERTATION .....................................................................................12

CHAPTER 2 HYBRID MESSAGE-PASSING AND MULTITHREADING PARALLELIZATION .............13

2.1 INTRODUCTION ........................................................................................................................13
2.2 DOMAIN DECOMPOSITION MOLECULAR DYNAMICS ..............................................................14

2.2.1 Internode Operations ..........................................................................................................15
2.2.2 Intranode Operations ..........................................................................................................16
2.2.3 Hybrid MPI/OpenMP Parallelization ..................................................................................18

2.3 DATA PRIVATIZATION SCHEDULING ALGORITHM .................................................................20

2.3.1 Naïve Data Privatization Threading Algorithm ..................................................................20
2.3.2 Nucleation-Growth Allocation Algorithm .........................................................................21

2.4 THEORETICAL ANALYSIS .......................................................................................................26

2.4.1 Load Imbalance Analysis .....................................................................................................26
2.4.2 Memory Consumption Analysis ..........................................................................................27
2.4.3 Computation Complexity Analysis ....................................................................................35
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>PERFORMANCE EVALUATION ...................................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Scheduling Cost ...........................................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Thread-Level Load Balancing ....................................................</td>
<td>38</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Memory Consumption ......................................................................</td>
<td>39</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Reduction-Sum Operation Cost ....................................................</td>
<td>40</td>
</tr>
<tr>
<td>2.5.5</td>
<td>Strong-Scaling Benchmark ................................................................</td>
<td>41</td>
</tr>
<tr>
<td>2.6</td>
<td>SUMMARY .......................................................................................</td>
<td>44</td>
</tr>
<tr>
<td>3.1</td>
<td>INTRODUCTION ..................................................................................</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>BACKGROUND ...................................................................................</td>
<td>47</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Hybrid MPI/OpenMP Molecular Dynamics .........................................</td>
<td>47</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Race Conditions and Traditional Concurrency Control Mechanisms .......</td>
<td>49</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Hardware Transactional Memory on BlueGene/Q ..................................</td>
<td>50</td>
</tr>
<tr>
<td>3.3</td>
<td>PERFORMANCE CHARACTERISTICS OF CONCURREN CY CONTROL MECHANISMS ....</td>
<td>53</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Concurrency-Control Cost Modeling ..............................................</td>
<td>53</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Model Parameter Fitting Using Microbenchmark ..................................</td>
<td>55</td>
</tr>
<tr>
<td>3.4</td>
<td>MOLECULAR DYNAMICS BENCHMARK ..................................................</td>
<td>57</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Performance Comparison of Concurrency Controls ............................</td>
<td>60</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Rollback Penalty ..........................................................................</td>
<td>64</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Effect of Scheduling Algorithms on Rollback ...................................</td>
<td>65</td>
</tr>
<tr>
<td>3.5</td>
<td>DESIGN SPACE FOR CONCURREN CY CONTROLS ....................................</td>
<td>69</td>
</tr>
<tr>
<td>3.6</td>
<td>CONCLUSION AND DISCUSSION ...........................................................</td>
<td>74</td>
</tr>
<tr>
<td>4.1</td>
<td>EMBEDDED PERFORMANCE PREDICTION MODEL .......................................</td>
<td>75</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Grand Scheme of the Embedded Performance Prediction Model ...............</td>
<td>76</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Tailored Scheme of the Embedded Performance Prediction Model For Molecular Dynamics</td>
<td>79</td>
</tr>
</tbody>
</table>
CHAPTER 5 N-TUPLE COMPUTATION FORMALISM AND SOLUTION FOR RANGED-LIMITED MANY-BODY MOLECULAR DYNAMICS ........................................................................................................91

5.1 PROBLEM STATEMENT ........................................................................................................92

5.2 BACKGROUND .....................................................................................................................92

5.2.1 Many-Body Molecular Dynamics ....................................................................................92

5.2.2 Dynamic Range-Limited n-Tuple Computation .................................................................94

5.3 COMPUTATION-PATTERN ALGEBRAIC FRAMEWORK .......................................................95

5.3.1 Algebraic Formulation of Cell-Based MD .......................................................................96

5.3.2 Shift-Collapse Algorithm ..............................................................................................103

5.3.3 Correctness of SC Algorithm .........................................................................................107

5.4 THEORETICAL ANALYSIS ................................................................................................111

5.4.1 Search-Cost Analysis of SC Pattern .............................................................................111

5.4.2 Communication Analysis of SC Pattern .........................................................................116

5.4.3 Relation to Previous Works .........................................................................................117

5.5 PERFORMANCE BENCHMARK ............................................................................................119

5.5.1 Search Cost of SC and FS Algorithms .........................................................................120

5.5.2 Fine-Grain Parallelism ....................................................................................................121

5.5.3 Strong-Scaling Benchmark ............................................................................................124

5.6 CONCLUSION .....................................................................................................................125

CHAPTER 6 PERFORMANCE ANALYSIS, MODELING, AND OPTIMIZATION OF CELL-BASED MOLECULAR DYNAMICS ......................................................................................................................127

6.1 INTRODUCTION ................................................................................................................128
6.2 CELL-BASE MOLECULAR DYNAMICS SIMULATIONS............................................................................. 129
6.3 PERFORMANCE ANALYSIS................................................................................................................. 131
  6.3.1 Short-Range Interaction Cost Model of the Conventional Linked-List Cell Method.................... 132
  6.3.2 Short-Range Interaction Cost Model of the Reduced Linked-list Cell Method............................. 135
6.4 MODEL VALIDATION.......................................................................................................................... 138
  6.4.1 Validation of the Operation Counts Model .................................................................................... 138
  6.4.2 Parameter Fitting ........................................................................................................................ 140
6.5 PERFORMANCE RESULTS................................................................................................................. 141
6.6 SUMMARY .......................................................................................................................................... 145

CHAPTER 7 MEMORY-ACCESS ANALYSIS AND OPTIMIZATION VIA DYNAMIC DATA REORDERING................................................................................................................................. 146

7.1 INTRODUCTION .................................................................................................................................. 147
7.2 DATA FRAGMENTATION IN PARALLEL MOLECULAR DYNAMICS.................................................. 148
  7.2.1 Temperature Induced Fragmentation ............................................................................................ 150
  7.2.2 Granularity Induced Fragmentation ............................................................................................. 155
7.3 PERFORMANCE MEASUREMENT ....................................................................................................... 158
7.4 REORDERING FREQUENCY OPTIMIZATION .......................................................... .......................... 163
7.5 SUMMARY .......................................................................................................................................... 166

CHAPTER 8 CONCLUSIONS........................................................................................................................ 168

REFERENCES ............................................................................................................................................ 171
List of Tables

Table 3.1: Specification of BlueGene/Q at LLNL .......................................................... 52
Table 3.2: Parallel efficiency estimate resulting from the decision tree ......................... 73
Table 4.1: Benchmark cluster specifications .................................................................. 82
Table 4.2: Parameter-fitting result of MPI-only and hybrid models using bi-square regression . 87
Table 5.1: Specifications of testing platforms .................................................................. 120
Table 6.1: Asymptotic complexity of parameter $P_s'$, $P_c'$, and $P_l'$ over $\mu$ .............. 138
Table 6.2: Accuracy of the operation counts model ....................................................... 139
Table 6.3: Estimated operation costs from various regression algorithms ..................... 141
Table 6.4: The predicted versus measured values of the optimal $\mu$ with their corresponding speedup compared to the case of no cell dimension reduction ........................................ 144
Table 7.1: 3,000 K silica datasets with varying granularities: The number of linked-list cells, the portion of surface cells, and the surface-to-volume ratio of the spatial subsystem are listed for each dataset .......................................................................................................................... 157
Table 7.2: Runtime result with different ordering methods for silica systems at temperature 300K ........................................................................................................................................ 162
Table 7.3: 3,000 K silica datasets varying granularities .................................................. 164
List of Figures

Figure 2.1: (a) 2D schematic of the linked-list cell method for pair computation with the cell dimension $r_c$. Only forces exerted by particles within the cutoff radius (represented by a two-headed arrow) are computed for particle $i$. The blue particles represent particles in the top-half neighbor cells where the force is computed directly, while the gray particles is a particle whose force is computed via reaction force. (b) 2D schematic of a single computation unit $\lambda_k$. The shaded cells $C_j$ pointed by the arrows constitute the half neighbor cells, $nn^*(C_k)$. .................................................................

Figure 2.2: Schematic workflow of a hybrid MPI/OpenMP scheme. The master thread schedules works for the worker threads before entering the parallel threading section. The master thread also handles internode communication, which is explicitly performed outside the threading region. .................................................................

Figure 2.3: Schematic of a memory layout for a naïve data privatization. To avoid data race condition among threads, the entire output array is locally duplicated for each thread. These local arrays are privatized on the thread memory space and thus eliminate race condition during the computation phase. This approach also incurs additional reduction operations to sum all local results to the global array. .................................................................

Figure 2.4: Memory layout and three-step algorithm for nucleation-growth allocation algorithm. Worker threads only allocate the essential portion of the force arrays, corresponding to their assigned workload.................................................................

Figure 2.5: Thread load-balancing algorithm. .................................................................
Figure 2.6: Pseudocode of compact-volume allocation scheduling algorithm ........................................... 24

Figure 2.7: Pseudocode of breadth-first allocation scheduling algorithm ........................................... 26

Figure 2.8: Geometric effects on surface-to-volume ratio. (a) and (b) show the surface volume (shell area) of cubic and spherical shapes with the same main volume. (c) and (d) show the discretized version, which indicates that the spherical shape consumes less volume than the cubic shape. The square grid represents cells in which particles reside. The blue area refers to the main volume while the red area shows the surface volume. Variable l is defined as the side length of main volume created by BFAS while r denotes the radius of CVAS. The surface thickness is r_c. ............................................................................................................................... 29

Figure 2.9: Illustrates a lower-bound memory consumption estimation of CVAS and BFAS algorithms. The graph indicates that CVAS consumes less memory than BFAS. .................. 34

Figure 2.10: Scheduling cost for CVAS and BFAS algorithms for 128,000-particle system on 64 BG/P nodes. The circles and squares are average scheduling times of CVAS and BFAS algorithms, respectively. .......................................................................................................................... 38

Figure 2.11: Illustrates the load-imbalance factor γ of CVAS as a function of p from theoretical bound, scheduler estimation, and actual measurement................................................................. 39

Figure 2.12: Presents average memory consumption for the private force arrays as a function of p using CVAS compared to the naïve method. Numbers in the legend denote n.................... 40

Figure 2.13: Shows average reduction-sum operation time of the CVAS as a function of p. Numbers in the legend denote n........................................................................................................... 41

Figure 2.14: (a) Thread-level strong scalability of the parallel section on a four quad-core AMD Opteron 2.3 GHz with fixed problem size at n = 8,192 particles. (b) Total running time per
MD steps of 1,024 – 32,768 Power PC 450 850 MHz cores on BG/P for a fixed problem size at $N = 0.84$-million particles and (c) 1.68-million particles. ............................................. 43

Figure 3.1: The benchmark kernel of a serial code without concurrency control. ....................... 56

Figure 3.2: The benchmark kernel with HTM control over 4 updates. ...................................... 56

Figure 3.3: The benchmark kernel with HTM control over 4 updates. ...................................... 57

Figure 3.4: Code fragment from the force update routine in ddcMD code, which has potential memory conflicts. ........................................................................................................... 59

Figure 3.5: (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.............................................................................................................. 63

Figure 3.6: Impact of rollback on runtime, indicating linear correlation between runtime overhead and rollback ratio............................................................................................................. 65

Figure 3.7: 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. ......................................................................................................................... 67

Figure 3.8: Rollback ratio (a) and the runtime overhead (b) as a function of the number of threads for the three scheduling algorithms................................................................................................. 68

Figure 3.9: Typical program structure of loop-oriented applications ......................................... 70

Figure 3.10: Decision tree for concurrency-control design space. We should note that OMP-critical at the bottom has very high performance penalty................................................................. 72

Figure 3.11: Analysis of four concurrency control mechanisms. Circles and triangles denote strong and weak advantages, respectively, while crosses denote disadvantage. .................... 73
Figure 4.1: Illustrates hierarchy architecture of a modern high-end machine. Combinations of distributed and shared memory parallelism are used in each architecture level. 78

Figure 4.2: Optimal region of MPI-only versus hybrid MPI/OpenMP implementations. The crossover line expected to remain constant at $N/P$. 84

Figure 4.3: Validation of (a) MPI-only and (b) hybrid MPI/OpenMP performance models. Blue lines denote the model results while the red dots denote the actual measurement on BlueGene/P. 88

Figure 4.4: The performance model predictions of 1.6 million-particle system of MPI-only versus hybrid MPI/OpenMP implementations, indicating the same result as in terms of optimality region as the actual measurements. 89

Figure 4.5: Extrapolation of the prediction models, fixing granularity per core constant. The predicted model suggested that the crossover is moved up to $\sim 200 - 400$ particles/core. 89

Figure 5.1: Schematic of range-limited $n$-tuples: (a) pair ($n = 2$), (b) triplet ($n = 3$), and (c) quadruplet ($n = 4$). 94

Figure 5.2: Set diagram showing the relationship of a bounding force set $S^{(n)}$ with $\Gamma^{(n)}$ and $\Gamma^{*(n)}$. 95

Figure 5.3: (a) Cell data-structure, where circles are atoms and small squares are cells. Dashed circle denotes the cutoff radius centered at atom $i$. (b) Uniform-cell pattern for pair computation, which generates all pairs between each cell and its neighbor cells. 97

Figure 5.4: $n$-tuple ($n = 3$) generation for cell search-space $S_{cell}$ of cell $c(q)$ (colored in magenta) and computation path $(v_0,v_1,v_2)$. $S_{cell}$ generates all triplets where the 1$^{st}$, 2$^{nd}$, and 3$^{rd}$ atoms in the triplet are in cells $c(q+v_0)$, $c(q+v_1)$, and $c(q+v_2)$, respectively. Dashed line shows one of the triplets $(r_i,r_j,r_k)$. 100

Figure 5.5: Uniform-cell pattern MD algorithm. 101
Figure 5.6: 2D schematic of a shift-collapse (SC) computation pattern. (a) FS computation pattern for \( n = 3 \), containing all possible paths of length 3 originated from the center cell (magenta). (b) SC computation pattern after OC-shift and R-collapse subroutines. Yellow cells denote the cell coverage of the center cell, which for SC is much smaller than that for FS.

Figure 5.7: Shift-Collapse algorithm.

Figure 5.8: FS generation algorithm.

Figure 5.9: OC-shift algorithm.

Figure 5.10: Reflective-collapse algorithm.

Figure 5.11: Schematic of shell methods for pair computation: (a) full-shell, (b) half-shell, and (c) eighth-shell patterns.

Figure 5.12: Average number of triplets as a function of domain size. Error bars are too small to be visible in the plot.

Figure 5.13: Runtime of SC-MD (red), FS-MD (green), and Hybrid-MD (blue) as a function of the granularity on (a) 48 Intel Xeon nodes and (b) 64 BlueGene/Q nodes. The plot shows that SC-MD is the fastest for \( N/P < 2,095 \) and 425 on Intel Xeon and BlueGene/Q platforms, respectively.

Figure 5.14: Strong scaling speedup of SC-MD, Hybrid-MD, and FS-MD on (a) Intel Xeon cluster and (b) BlueGene/Q.

Figure 6.1: 2D schematic of the linked-list cell method. (a) Illustration of conventional linked-list cell method with cell dimension \( R_s \geq R_c \). (b) 2D schematic of the reduced linked-list cell method. Atoms in the shaded region will be excluded from the force computation of
center atom i. Only forces exerted by atoms within the cutoff radius (represented by a two-
headed arrow) are computed............................................................... 129

Figure 6.2: Force computation algorithm. .......................................................... 130

Figure 6.3: The running time from the prediction and the measurement for $R_c = 10 \, \text{Å}$. The
numerals are the number of atoms............................................................... 143

Figure 6.4: Comparison of speedup gained from predicted and actual $\mu^*$ for $R_c = 6.6$ and 10 Å.
......................................................................................................................... 143

Figure 7.1: Schematic of memory layout for atom data in (a) a fully disordered state and (b) a
fully ordered state, where $C(i,j,k)$ is the linked list for the cell with indices $i$, $j$, and $k$ in the
$x$, $y$, and $z$ directions, respectively, $r[a]$ is the data associated with the $a$-th atom. ........... 149

Figure 7.2: Time variation of data fragmentation ratio over 3,000 MD steps. Three datasets of
different initial temperatures (300 K, 3,000 K, and 6,000 K) are plotted............................. 152

Figure 7.3: Mean squared displacement (MSD) of 300 K, 3,000 K, and 6,000 K datasets. The
MSD of 300 K dataset remains constant at 0.23 Å$^2$ (too small to be seen in the figure). The
inset shows the diffusion coefficients at the three temperatures........................................... 153

Figure 7.4: Data fragmentation ratio during aluminum combustion simulation over 2,000 MD
steps. The dataset involves 15 million atoms on 1,024 processors................................. 154

Figure 7.5: (a) Time variation of the data fragmentation ratio of 12,288-atom silica at 3,000 K
over 3,000 MD steps, for surface cells, core cells, and both combined. (b) Time variation of
the data fragmentation ratio of 3,000 K silica with varying granularities over 3,000 MD
steps..................................................................................................................... 156

Figure 7.6: Time variation of the DTLB miss rate in 300 K, 3,000 K and 6,000 K dataset over
3,000 MD steps................................................................................................. 159
Figure 7.7: Relation between the DTLB miss rate and the data fragmentation ratio in 300 K, 3,000 K and 6,000 K datasets. ................................................................. 160

Figure 7.8: Running time per MD step as a function of MD steps at temperatures 300K, 3,000K, and 6,000K for 98,304 silica atoms. ................................................................. 161

Figure 7.9: Schematic ordering of (a) sequential ordering (b) 3rd-order 2D Hilbert-curve ordering (c) 3rd-order 2D Morton-curve ordering................................................................. 163

Figure 7.10: Total running time after 3,000 MD steps as a function of reordering periods. The optimal period for 300 K, 3,000 K and 6,000 K datasets are 69, 5 and 3 steps, respectively. ................................................................. 165

Figure 7.11: Comparison of total running time of silica MD over 3,000 steps achieved by model prediction and actual measurement of periodic reordering compared with that of the original code without ordering. (All tests are conducted with 98,304 atoms per core.) ............... 166
Chapter 1

Introduction

1.1 Motivations

Computer simulation has become vital tools to solve a broad range of scientific and engineering problems such as studying properties of nanomaterials at the atomistic scale, designing efficient wind-turbine farms for high-yield clean energy production, simulating catastrophic earthquakes for disaster preparedness, and modeling drug for curing fatal diseases. However, larger and longer simulations, along with sophisticated modeling of real-world problems, are required in order to characterize realistic systems. One simulation model of great significance is molecular dynamics (MD) simulation, which has broad applications in diverse fields such as physics, chemistry, biology, and materials science. Recent advancements in chemically reactive MD simulations have driven the need of vast computing power. Thanks to the advent of multicore architectures, the rapid increase in computing power has enabled scientists to confront these grand challenges for mankind.
Nevertheless, efficiently harvesting computing power from the modern architectures is challenging for application developers due to the sophisticated structure of computing platforms. Research has focused on evolving legacy MD algorithms to efficiently execute on modern architectures. Especially, recent advancements in reactive MD simulations based on many-body interatomic potentials have necessitated efficient dynamic $n$-tuple computation. An aspiration to have scalable and efficient parallel frameworks along with systematic understanding of the reactive MD problems has motivated us to find solutions to these challenges. This will enable software developers to continually enhance MD applications to cope with the advent of novel architectures in the future.

### 1.2 Challenges and Goals

The emergence of the multicore era has granted unprecedented computing capabilities. However, the longstanding message-passing programming model tends to underutilize the full potential of modern architectures. This has posed difficulties to software developers since the legacy programs significantly relied on the increased processing power of a single processor instead of the increased number of cores [1-3]. Thus, many algorithms currently in use are not suitable for the shared-memory model, thereby demanding substantial alterations in software [4, 5]. Also, performance of the traditional concurrency techniques tends to be limited on a large-scale concurrent system. Addressing these issues typically requires extensive coding effort, which limits wide use of massive shared-memory platforms. Hence it is of great challenge to develop a parallel algorithm that possesses “metascalability” (*i.e.*, design once, continue to scale on future architectures) [6] for applications on continuously evolving platforms.
Despite remarkable progresses in parallel algorithms for simple MD, parallel dynamic $n$-tuple computation of reactive MD is still in its infancy. The major challenge emerges from the lack of fundamental understanding of general $n$-tuple computation characteristics: How can we understand the fundamental characteristics of $n$-tuple computation such that we can apply the benefits from advanced techniques in simple MD into arbitrary dynamic $n$-tuple computations?

To address these challenges, this dissertation develops a metascaleable parallel algorithmic framework, along with mathematical formulations and optimal solutions for general $n$-tuple computation. This allows scientists to fully utilize the computational capacity of high-end multicore machines to solve $n$-tuple computation, encompassing even larger spatiotemporal barrier of the physical simulations on the current and forthcoming platforms.

1.3 Background

In this section, we provide background information on our research. First, we introduce MD, a prototypical scientific application that is characterized by a number of complications toward parallelization on the multicore architecture. Second, we provide background information on emerging multicore architectures. Third, we discuss our hierarchical parallelization, specifically, a hybrid message-passing and multithreading algorithm for scientific applications without loss of generality. Next, we explain $n$-tuple computation in many-body MD, which is used to perform reactive MD simulation. Finally, we provide a description of state-of-the-art scalable parallel MD algorithms.
1.3.1 Molecular Dynamics Simulation

MD simulation has become an important tool [7] to study a broad range of scientific problems at the atomistic level [8-14]. MD simulation follows the phase-space trajectories of an $N$-particle system, where the forces between particles are given by the gradient of a potential energy function. 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 potential energy function and associated forces.

Many potential energy models were proposed in order to describe numerous properties of various materials. Some of these impose significant computational challenges [15-19]. Many methods exist to reduce this computational complexity [17-21]. Another example of important potential models is one consisting of many-body terms [16]. The dominant computation in such MD simulation is the evaluation of many-body terms involving summation over particle $n$-tuples.

1.3.2 Multicore Architectures

The rapid growth in computing power enables scientists to address more challenging problems by using high-end supercomputers [22, 23]. However, recent improvements in computing power are now gained using multicore architectures instead of increased clock speed [2, 24]. Application scientists at the forefront of high-performance computing (HPC) have been investigating the use of hybrid architectures to improve performance of scientific codes. Hierarchical architectures attempt to improve application performance by combining conventional processors with any of a variety of more specialized processors such as graphics processing units (GPUs) [25, 26], manycores [27-29], and Cells [30, 31]. This marks the end of
the free-ride era, where legacy applications cannot increase performance on a newer chip without substantial modification. Furthermore, the number of cores per chip is expected to grow continuously, which deepens the performance impact on the legacy software.

Multicore processors with sophisticated computational and memory organizations have been developed as a result of the pursuit for high performance with low energy usage. At the low-level of memory hierarchy, most of the recent multicore chip designs adopt heterogeneous cache access at the cache level. This design has local level 1 and level 2 caches for individual core, while subsets of cores share its last-level cache (LLC). The cache-coherency concept is usually adopted, which guarantees data consistency among all cores in a node. At a node level, there are two major approaches for memory hierarchy: (1) symmetric multiprocessing (SMP), where all the compute cores have the same cost accessing memory; and (2) non-uniform memory access (NUMA), where each processors has different cost to access each memory module. NUMA paradigm requires more attention from software developers to maintain maximal performance [32, 33] (e.g. requires thread pinning to avoid thread switching between cores). However, SMP paradigm tends to have higher latency and is more difficult in circuit design. Nevertheless, the shared-memory concept is usually implemented on a multicore chip. This raises the issues of race conditions, and load balancing when multithreading is used.

As multicore becomes the standard of modern supercomputers, two significant issues have emerged: (1) the performance of traditional parallel applications, which are solely based on message passing interface (MPI), is expected to degrade substantially [34]; and (2) the memory available per core tends to decline, since the growth of the number of cores per chip is considerably faster than that of the available memory. To address these complications, hierarchical parallelization frameworks, which integrate several parallel methods to provide
different levels of parallelism, have been proposed as a solution to this scalability problem on multicore platforms [6, 35-37].

In addition, a recent trend in HPC also focuses on heterogeneous computing. A combination of generic multicore chips and accelerators such as general-purpose graphics processing unit (GPGPU) [38-42] and manycore processor [43] (i.e. light-weight general processing component such as Intel Many Integrated Cores (MIC) chip) are used. Although accelerators are powerful for specific types of problems such as highly efficient floating-point operations in GPGPU, they usually do not execute every instruction efficiently, if it is able to run at all. Therefore, generic multicore chips are still needed to run non-specialized code sections in the program.

1.3.3 Hybrid Parallelization

Hybrid parallelization is a parallelization approach that combines multiple parallelization techniques in order to achieve the best parallelization performance. Hybrid parallelization has been widely used for applications to execute on emerging multicore platforms, using a hybrid scheme reflecting the two-level hierarchical structural designs of multicore clusters. The most common schemes for hybrid parallelization is distributed memory for inter-node communications via message passing, and shared memory for intra-node communication via multithreading.

At cluster-level, divide-and-conquer parallel algorithms based on spatial decomposition implemented with MPI are widely used, since MPI is the de-facto standard for high-end parallel computing for decades. MPI provides a moderate-transparency communication framework for parallel applications, by allowing software developers to explicitly code communication routines
from elementary communication building blocks without being concerned with sophisticated network protocols [44]. The basic communication functions provided by the MPI standard include synchronous/asynchronous end-to-end communications and collective communications, while secondary functions based on the basic functions are also provided.

At node-level parallelization, multithreading is a common strategy to exploit the shared-memory property of multicore chip. Numerous multithreading frameworks are widely available in computer science communities. In HPC area, multithreading frameworks generally equipped with modern compilers include POSIX thread (Pthread) [45] and Open Multiprocessing (OpenMP) [46]. Pthread provides most of the low-level operations with low overhead to the users with the cost of complicated coding. On the other hand, OpenMP offers simpler framework for multithreading but many operations cost considerable overhead.

Variants of this message-passing/multithreading hybrid scheme have been proposed and studied broadly [37, 47-51]. Peng et al. [35] added one more step of parallelism at the data level through single-instruction multiple-data (SIMD) techniques, while recent studies [41, 52] have extended this scheme to cover GPGPU context. The hierarchical framework is expected to exploit data concurrency efficiently and continue to scale on future multicore platforms.

Since most of the legacy software operates efficiently on supercomputers using MPI, the great challenge for the hybrid message-passing and multithreading parallelization lies on the multithreading part. The most typical issues associated with multithreading are race conditions and load balancing. Race condition is one of the common problems that plague program developers since the earliest age of computing. Although numerous methods were proposed to deal with race conditions (e.g. semaphores, critical section, transactional memory), the performance-sensitive nature of high-performance applications usually renders these naïve
methods impractical. On the other hand, load imbalance at the thread level usually occurs at fine-grain scope, which requires that load balancers must be much more dynamic. Also, issues like context switching and false-sharing place tighter constrains on the design and usage of multithreading.

1.3.4 \textit{n-Tuple Molecular Dynamics}

MD simulation using a differentiable interatomic potential-energy function $\Phi$ was started by Rahman in 1964 [53] using a pair-wise potential energy, in which $\Phi$ is a sum of atomic-pair energies. Since then, scientists started performing many-body MD simulations that use \textit{n}-tuple ($n \geq 3$) energy functions for accurate description of a wider range of materials. In one type of \textit{n}-tuple computation (\textit{i.e.} static \textit{n}-tuple computation) used typically in biomolecular simulations [54], the list of atomic \textit{n}-tuples is fixed throughout the simulation. In another (\textit{i.e.} dynamic \textit{n}-tuple computation), \textit{n}-tuple lists within given interaction ranges are constructed at every simulation time step [55, 56]. Recent advancements in chemically reactive MD simulations [57] have renewed interests in efficient implementation of dynamic \textit{n}-tuple computation [58]. Reactive MD describes the formation and breakage of chemical bonds based on a reactive bond-order concept [57]. In the ReaxFF approach, for example, $n$ is 4 explicitly, and force computation involves up to $n = 6$ due to chain-rule differentiations through bond-order terms [59-61].

1.3.5 \textbf{Recent Development of Parallel MD Algorithms}

Scalable implementation of MD on massively parallel computers has been one of the major driving forces of supercomputing technologies [62-68]. Earlier parallel implementations of MD were based on spatial decomposition, in which the simulated physical volume is subdivided
into spatially localized sub-volumes that are assigned to different processors [69]. For long-range pair \((n = 2)\) computation, octree-based \(O(N)\) algorithms \((N\) is the number of atoms) [20] have highly scalable parallel implementations [70, 71]. For short-ranged (or range-limited) pair computation, Plimpton relaxed the conventional “owner-compute” rule \((i.e.,\) computation is performed by a processor that has data) to design a force-decomposition algorithm to increase the concurrency [72]. Since then, various hybrid spatial-force decomposition algorithms have been designed [73]. On distributed-memory parallel computers, atomic data needed for range-limited pair computations are copied from neighbor processors. The most primitive scheme for these atom-caching operations is full shell (FS), in which data from 26 (face-, edge-, and corner-sharing) neighbor sub-volumes are imported from other processors. In the half-shell (HS) scheme, Newton’s third law is utilized to halve the number of imported sub-volumes to 13 [69]. Relaxation of the owner-compute rule further reduces this number to 7 in the eighth-shell (ES) scheme [11]. In the case of special-purpose computers with low network latency, neutral-territory (NT) [74] and related [75] schemes achieve asymptotically smaller import volumes for fine granularities, \(N/P\) \((P\) is the number of processors). In addition to these parallel algorithms for dynamic pair computations, numerous schemes have been employed in biological MD codes to efficiently compute static \(n\)-tuple computations [12, 73].

1.4 Contributions

The main contributions of this dissertation research are two-fold: Metascalable hybrid message-passing and multithreading parallel algorithmic framework on emerging multicore parallel clusters, and a novel computation-pattern algebraic framework to design a scalable
algorithm for general $n$-tuple computation and prove its optimality in a mathematically rigorous manner.

To design a metascaleable algorithm, we have developed and thoroughly analyzed algorithms for hybrid MPI + OpenMP parallelization of $n$-tuple MD simulation, which are scalable on large multicore clusters. Two data-privatization thread scheduling algorithms via nucleation-growth allocation have been designed: (1) compact-volume allocation scheduling (CVAS); and (2) breadth-first allocation scheduling (BFAS). These two algorithms combine fine-grain dynamic load balancing and minimal memory-footprint threading. Theoretical study has revealed decent asymptotic memory efficiency for both algorithms, thereby reducing 75% memory consumption compared to a naïve-threading algorithm. Furthermore, performance benchmarks have confirmed higher performance of the hybrid MD algorithm over a traditional algorithm on large multicore clusters, where 2.58-fold speedup of the hybrid algorithm over the traditional algorithm was observed on 32,768 nodes of IBM BlueGene/P.

To aid the multithreading program-design decision, we have investigated the performance characteristics of HTM on the IBM BlueGene/Q computer in comparison with conventional concurrency control mechanisms, using an MD application as an example. Benchmark tests, along with overhead-cost and scalability analysis, have quantified 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 have proposed transaction fusion and spatially compact scheduling techniques to reduce the overhead of HTM with minimal programming. A strong scalability benchmark has shown that the fused HTM has the shortest runtime among various concurrency control mechanisms without extra memory. Based on the performance
characterization, we have derived a decision tree in the concurrency-control design space for general multithreading applications.

Regarding the second contribution, we have developed a computation-pattern algebraic framework to mathematically formulate general \( n \)-tuple computation. Based on translation/reflection-invariant properties of computation patterns within this framework, we have designed a shift-collapse (SC) algorithm for cell-based parallel MD. Theoretical analysis has quantified the compact \( n \)-tuple search space and small communication cost of SC-MD for arbitrary \( n \), which are reduced to those in best pair-computation approaches (e.g. eighth-shell method) for \( n = 2 \). Benchmark tests have shown that SC-MD outperforms our production MD code at the finest grain, with 9.7- and 5.1-fold speedups on Intel-Xeon and BlueGene/Q clusters. SC-MD has also exhibited excellent strong scalability.

In addition, we have analyzed the computational and data-access patterns of MD, which led to the development of a performance prediction model for short-range pair-wise force computations in MD simulations. The analysis and performance model provide fundamental understanding of computation patterns and optimality of certain parameters in MD simulations, thus allowing scientists to determine the optimal cell dimension in a linked-list cell method. The model has accurately estimated the number of operations during the simulations with the maximum error of 10.6% compared to actual measurements. Analysis and benchmark of the model have revealed that the optimal cell dimension minimizing the computation time is determined by a trade-off between decreasing search space and increasing linked-list cell access for smaller cells.

One difficulty about MD is that it is a dynamic irregular application, which often suffers considerable performance deterioration during execution. To address this problem, an optimal
A data-reordering schedule has been developed for runtime memory-access optimization of MD simulations on parallel computers. Analysis of the memory-access penalty during MD simulations has shown that the performance improvement from computation and data reordering degrades gradually as data translation lookaside buffer misses increase. We have also found correlations between the performance degradation with physical properties such as the simulated temperature, as well as with computational parameters such as the spatial-decomposition granularity. Based on a performance model and pre-profiling of data fragmentation behaviors, we have developed an optimal runtime data-reordering schedule, thereby archiving speedup of 1.35, 1.36 and 1.28, respectively, for MD simulations of silica at temperatures 300 K, 3,000 K and 6,000 K.

We expect that the hybrid algorithms and mathematical approaches developed in this dissertation will provide a generic framework to a broad range of applications on future extreme-scale computing platforms.

1.5 Structure of This Dissertation

The organization of this dissertation is as follows. Chapter 2 presents the proposed hybrid message-passing and multithreading algorithm along with detailed theoretical and statistical analyses. Chapter 3 characterizes and compares several concurrency control mechanisms for multithreading. Chapter 4 presents a hybrid performance model. Chapter 5 presents the formulation and analysis of a computation-pattern algebraic framework for $n$-tuple computation. A theoretical analysis and design of an MD application on a single node are discussed in chapter 6. Chapter 7 discusses analysis and optimization of irregular memory-access behaviors in MD. Finally, conclusions are drawn in chapter 8.
Chapter 2

Hybrid Message-Passing and Multithreading Parallelization

In this chapter, we propose threading algorithms for hybrid MPI/OpenMP parallelization of MD simulation, which are scalable on large multicore clusters [76]. Two data privatization thread scheduling via nucleation-growth allocation are presented: (1) compact-volume allocation scheduling (CVAS); and (2) breadth-first allocation algorithm (BFAS). This chapter is organized as follows. Section 2.1 provides a brief introduction to the problem and a hybrid parallel algorithm. Section 2.2 summarizes the hierarchy of parallel operations in a sample MD code, ddcMD, followed by the detail description of the proposed data-privatization algorithms in section 2.3. Theoretical analysis of data-privatization algorithms is given in section 2.4. Section 2.5 evaluates the performance of the hybrid parallelization algorithm, and conclusions are drawn in section 2.6.

2.1 Introduction

As multicore chip becomes the standard of modern supercomputers, two significant issues have emerged: (1) the performance of traditional parallel applications, which are solely
based on MPI, is expected to degrade substantially [34]; and (2) the memory available per core tends to decline, since the growth of the number of cores per chip is considerably faster than that of the available memory. To address the emerging complications, hierarchical parallelization frameworks, integrating several parallel methods to provide different levels of parallelism, have been proposed as a solution to solve the scalability problem on multicore platforms [6, 35-37].

On multicore architectures, hybrid parallelization based on MPI/threading schemes will likely replace MPI-only parallel MD. This is because the benefit from a shared-memory model of threading can be exploited within a multicore chip, while MPI handles the communication among compute nodes. In the next section, we describe detail algorithmic concept of ddcMD, our prototypical MD application representing broad-range of scientific codes.

2.2 Domain Decomposition Molecular Dynamics

In this chapter, we focus on the highly efficient particle-particle/particle-mesh (PPPM) method [17] to compute Coulomb potential. 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 split of work between the short-range and long-range part is controlled though a “screening parameter” $\alpha$. With the appropriate choice of $\alpha$, computational complexity for these methods can be reduced to $O(N\log N)$.

Because the long-range part of the Coulomb potential can be threaded easily (as a parallel loop over many individual 1D fast Fourier transforms), this study explores efficient parallelization of the more challenging short-range part of the Coulomb potential using OpenMP threading. The short-range part is a sum over pairs: $\phi = \sum_{i<j} q_i q_j \text{erfc}(ar_{ij})/r_{ij}$, 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. In addition to this intranode parallelization, the ddcMD code is already parallelized across nodes using a particle-based domain decomposition implemented using MPI. Combining the existing MPI-based decomposition with the new intranode parallelization yields a hybrid MPI/OpenMP parallel code. An extensive comparison of MPI-only ddcMD with other pure MPI codes can be found in [77].

In section 2.2.1, we provide node-level communication concepts of ddcMD together with the detail of a load-balancing algorithm used in ddcMD. Section 2.2.2 describes how computation is handled within each domain. Hybrid MPI/OpenMP approach is described in section 2.2.3.

### 2.2.1 Internode Operations

In typical parallel MD codes the first level of parallelism is obtained by decomposing the simulation volume into domains, each of which is assigned to a compute core (i.e., an MPI task). Because particles near domain boundaries interact with particles in nearby domains, internode communication is required to exchange particle data between domains. The surface-to-volume ratio of the domains and the choice of potential set the balance of communication to computation.

The domain-decomposition strategy in ddcMD allows arbitrarily shaped domains that may even overlap spatially. Also, remote particle communication between nonadjacent domains is possible when the interaction length exceeds the domain size. A domain is defined only by the position of its center and the collection of particles that it “owns.” Particles are initially assigned to the closest domain center, creating a set of domains that approximates a Voronoi tessellation.
The choice of the domain centers controls the shape of this tessellation and hence the surface-to-volume ratio for each domain. The commonly used rectilinear domain decomposition employed by many parallel codes is not optimal from this perspective. The best surface-to-volume ratio in a homogeneous system is achieved if domain centers form a bcc, fcc, or hcp lattice, which are common high-density packing of atomic crystals.

In addition to setting the communication cost, the domain decomposition also controls load imbalance. Because the domain centers in ddcMD are not required to form a lattice, simulations with a non-uniform spatial distribution of particles (e.g., voids or cracks) can be load balanced by an appropriate non-uniform arrangement of domain centers. The flexible domain strategy of ddcMD allows for the migration of the particles between domains by shifting the domain centers. As any change in their positions affects both load balance and the ratio of computation to communication, shifting domain centers is a convenient way to optimize the overall efficiency of the simulation. Given an appropriate metric (such as overall time spent in MPI barriers) the domains can be shifted “on-the-fly” in order to maximize efficiency [78].

2.2.2 Intranode Operations

Once particles are assigned to domains and remote particles are communicated, the force calculation begins. Figure 2.1(a) shows a schematic of the linked-list cell method used by ddcMD to compute pair interactions in $O(N)$ time. In this method, each simulation domain is divided into small cubic cells, and a linked-list data structure is used to organize particle data (e.g., coordinates, velocities, type, and charge) in each cell. By traversing the linked list, one retrieves the information of all particles belonging to a cell, and thereby computes interparticle
interactions. The dimension of the cells is determined by the cutoff length of the pair interaction, $r_c$.

Linked-list traversal introduces a highly irregular memory-access pattern, resulting in performance degradation. To alleviate this problem, we reorder the particles within each node at the beginning of every MD step, so that the particles within the same cell are arranged contiguously in memory when the computation kernel is called. At present we choose an ordering specifically tailored to take advantage of the BlueGene “double-Hummer” SIMD operations. However, we could just as easily reorder the data to account for NUMA or GPGPU architectural details. We consistently find that the benefit of the regular memory access far outweighs the cost of particle reordering. The threading techniques proposed here are specifically constructed to preserve these memory-ordering advantages.

The computation within each node is described as follows. Let $L$ be the total number of cells in the system, $\{C_k | 0 \leq k < L\}$ be the set of cells within each domain. The computation within each node is divided into a collection of small chunks of work called a computation unit $\lambda$. A single computation unit $\lambda_k = \{(r_i, r_j) | r_i \in C_k; r_j \in \text{nn}^+(C_k)\}$ for cell $C_k$ is defined as a collection of pair-wise computations (see Figure 2.1(b)), where $\text{nn}^+(C_k)$ is a set of half the nearest-neighbor cells of $C_k$. Newton’s third law allows us to halve the number of force evaluations and use $\text{nn}^+(C_k)$ instead of the full set of nearest-neighbor cells, $\text{nn}(C_k)$. The pairs in all computation units are unique, and thus the computation units are mutually exclusive. The set of all computation units on each node is denoted as $\Lambda = \{\lambda_k | 0 \leq k < L\}$. Since most of our analysis is performed at a node level, $n = N/P$ hereafter denotes the number of particles in each node ($P$ is the number of nodes), and $p$ is the number of threads in each node.
Figure 2.1: (a) 2D schematic of the linked-list cell method for pair computation with the cell dimension $r_c$. Only forces exerted by particles within the cutoff radius (represented by a two-headed arrow) are computed for particle $i$. The blue particles represent particles in the top-half neighbor cells where the force is computed directly, while the gray particles is a particle whose force is computed via reaction force. (b) 2D schematic of a single computation unit $\lambda_k$. The shaded cells $C_j$ pointed by the arrows constitute the half neighbor cells, $\text{nn}^+ (C_k)$.

### 2.2.3 Hybrid MPI/OpenMP Parallelization

The hybrid MPI/OpenMP parallelization of ddcMD is implemented by introducing a thread scheduler into the MPI-only ddcMD. Figure 2.2 shows the workflow of the hybrid MPI/OpenMP code using a master-slave approach. The program repeats the following computational phases: First, the master thread performs initialization and internode communications using MPI; the scheduler computes the scheduled work for each thread; and the worker threads execute the workloads in an OpenMP parallel section. Our algorithm utilizes a scheduler to distribute the workload before entering the pair computation, i.e., parallel section. In this approach, the scheduling cannot interfere with the worker threads, since the scheduling is already completed before the worker threads are started. Because the schedule is recomputed every MD step (or perhaps every few MD steps), there is adequate flexibility to adapt load balancing to the changing dynamics of the simulation. This approach also reduces the number of
synchronizations and minimizes context switching compared to a real-time dynamic scheduling approach.

We parallelize the explicit pair-force computation kernel of ddcMD, which is the most computationally intensive kernel, at the thread level using OpenMP. Two major problems commonly associated with threading are: (1) race condition among threads; and (2) thread-level load imbalance. The race condition occurs when multiple threads try to update the force of the same particle concurrently.

Figure 2.2: Schematic workflow of a hybrid MPI/OpenMP scheme. The master thread schedules works for the worker threads before entering the parallel threading section. The master thread also handles internode communication, which is explicitly performed outside the threading region.

We have designed algorithms that combine a mutually exclusive scheduler with a reduced memory data-privatization scheme to address a race condition and fine-grain load balancing issues. The algorithm will be discussed in detail in section 2.3. This research focuses on hybrid MPI/OpenMP parallelization on Sequoia, the third generation of BlueGene, which will be online in 2012 at LLNL. On this symmetric multiprocessing (SMP) platform, MPI-only programming will not be an option for full-scale runs with at least 3.2 million concurrent threads.
2.3 Data Privatization Scheduling Algorithm

Molecular-dynamics computation patterns often lead to serious fine-grain race conditions. If reaction force is exploited, every pair interaction involving an identical particle (e.g. interactions between particles $i-j$ and $j-k$) possibly causes a write conflict when two different threads computes on each interaction concurrently. The simplest solution is not to use reaction force but this approach immediately doubled the computation (or more than doubled in case of many-body potentials). Hence, this approach is not the best solution in most circumstances.

To address these issues, we use a data privatization algorithm, which allows the reaction forces to be exploited while eliminating race conditions from pair computations. However, a naïve implementation of this algorithm consumes considerably more memory. This will be described in section 2.3.1. In section 2.3.2, the nucleation-growth allocation algorithm, an enhanced data privatization with reduced memory consumption, will be discussed.

2.3.1 Naïve Data Privatization Threading Algorithm

A naïve data-privatization algorithm avoids write conflicts by replicating the entire write-shared data structure and allocating a private copy to each thread (Figure 2.3). The memory requirement for this redundant allocation scales as $O(np)$. Each thread computes forces for each of its computation units and stores the force values in its private array instead of the global array. This allows each thread to compute forces independently without a critical section. After the force computation for each MD step is completed, the private force arrays are reduced to obtain the global forces.
2.3.2 Nucleation-Growth Allocation Algorithm

As explained in the previous section, the memory requirement of the data-privatization algorithm is $O(np)$. However, it is not necessary to allocate a complete copy of the force array for each thread, since only a subset of all computation units $\Lambda$ is assigned to each thread. Therefore, we allocate only the necessary portion of the global force array corresponding to the computation units assigned to each thread as a private force array. This idea is embodied in a three-step algorithm (Figure 2.4): (1) the scheduler assigns computation units to threads and then determines which subset of the global data each thread requires; (2) each thread allocates its
private memory as determined by the scheduler; and (3) private force arrays from all threads are reduced into the global force array.

Figure 2.4: Memory layout and three-step algorithm for nucleation-growth allocation algorithm. Worker threads only allocate the essential portion of the force arrays, corresponding to their assigned workload.

To do this, we create a mapping table between the global force-array index of each particle and its thread-array index in a thread memory space. Since ddcMD sorts the particle data based on the cell they reside in, only the mapping from the first global particle index of each cell to the first local particle index is required. The local ordering within each cell is identical in both the global and private arrays.

It should be noted that assigning computation unit $\lambda_k$ to thread $T_i$ requires memory allocation more than the memory for the particles in $C_k$. Since each computation unit computes the pair forces of particles in cell $C_k$ and half of its neighbor cells $\text{n}n^+(C_k)$ as shown in figure 1(b), the force data of particles in $\text{n}n^+(C_k)$ need to be allocated as well. In order to minimize the memory requirement of each thread, the computation units assigned to it must be spatially proximate, so that the union of their neighbor-cell sets has a minimal size. This is achieved by minimizing the surface-to-volume ratio of $T_i$. To achieve this, we implement two algorithms
based on a nucleation-growth concept. This approach systematically divides the work as equally as possible with minimal memory overhead. These two algorithms are: (1) compact volume allocation scheduling (CVAS); and (2) breadth-first allocation scheduling (BFAS) algorithms. CVAS and BFAS employ a fine-grain load-balancing algorithm, which will be shown in subsection 2.3.2.1. In subsections 2.3.2.2 and 2.3.2.3, we will discuss CVAS and BFAS algorithms in more detail.

2.3.2.1 Thread-level load-balancing algorithm

We implement thread-level load balancing based on a greedy approach, i.e., iteratively assigning a computation unit to the least-loaded thread, until all computation units are assigned. Let $T_i \subseteq \Lambda$ denote a mutually exclusive subset of computation units assigned to the $i$-th thread, $\bigcap_{k=0,..,L-1} \lambda_k = \emptyset$. The computation time spent on $\lambda_k$ is denoted as $\tau(\lambda_k)$. Thus, the computation time of each thread $\tau(T_i)$ is a sum of all computation units assigned to thread $T_i$. The algorithm initializes $T_i$ to be empty, and loops over $\lambda_k$ in $\Lambda$. Each iteration selects the least-loaded thread $T_{\min} = \text{argmin}(\tau(T_i))$, and assigns $\lambda_k$ to it. This original approach is a 2-approximation algorithm [79] and its pseudocode is shown in Figure 2.5.

<table>
<thead>
<tr>
<th>Algorithm Fine-Grain Load Balancing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. for $0 \leq i &lt; p$</td>
</tr>
<tr>
<td>2. $T_i \leftarrow \emptyset$</td>
</tr>
<tr>
<td>3. end for</td>
</tr>
<tr>
<td>4. for each $\lambda_k$ in $\Lambda$</td>
</tr>
<tr>
<td>5. $T_{\min} \leftarrow \text{argmin}_i(\tau(T_i))$</td>
</tr>
<tr>
<td>6. $T_{\min} \leftarrow T_{\min} \cup \lambda_k$</td>
</tr>
<tr>
<td>7. end for</td>
</tr>
</tbody>
</table>
Figure 2.5: Thread load-balancing algorithm.

### 2.3.2.2 Compact-volume allocation scheduling (CVAS) algorithm

CVAS algorithm consists of the following steps. First, we randomly assign a root computation unit to each thread. Then, the iteration begins by selecting the least-loaded thread $T_{\text{min}}$. From the surrounding volume of $T_{\text{min}}$, we select a computation unit $\lambda_{j^*}$ that has the minimum distance to the centroid of $T_{\text{min}}$, and add it to $T_{\text{min}}$. The algorithm repeats until all computation units are assigned. If all of the surrounding computation units of $T_{\text{min}}$ are already assigned, $T_{\text{min}}$ randomly chooses a new unassigned computation unit as a new cluster’s root and continue to grow from that point. The pseudocode of the algorithm is shown in Figure 2.6.

**Algorithm** Compact volume allocation scheduling

1. Assign random root cell $\lambda_{\text{root}}$ to each thread
2. **while** not all of the cells are assigned
3. Find the least loaded thread $T_{\text{min}}$
4. **if** nearest neighbor cells of $T_{\text{min}}$ are already assigned to other threads
5. Randomly pick new root $\lambda_{\text{root}*}$ for thread $T_{\text{min}}$
6. Assign $\lambda_{\text{root}*}$ to $T_{\text{min}}$
7. Set $\lambda_{\text{root}*}$ to be a new seed of $T_{\text{min}}$
8. **else**
9. Find an unassigned cell $\lambda_{j^*}$ which has minimal distance to the centroid of $T_{\text{min}}$
10. Assign $\lambda_{j^*}$ to $T_{\text{min}}$
11. **end if**
12. **end while**

Figure 2.6: Pseudocode of compact-volume allocation scheduling algorithm.
2.3.2.3 Breadth-first allocation scheduling (BFAS) algorithm

One weakness of the CVAS algorithm is a fairly expensive scheduling cost. To improve this aspect, we propose a breadth-first allocation scheduling (BFAS) algorithm as an alternative. BFAS is designed to slightly reduce memory saving but significantly reduces scheduling cost. BFAS consists of the following steps. The initialization step begins by randomly assigning the root computation unit $\lambda_{\text{root}}$ to each thread. Then, an empty queue data structure $Q_i$ is defined as a search queue for thread $T_i$. After that, each thread $T_i$ adds all of the computation units surrounding $\lambda_{\text{root}}$ into $Q_i$. In the next step, the algorithm proceeds iteratively. First, find the least-loaded thread $T_{\text{min}}$. Then, from the top of the queue $Q_{T_{\text{min}}}$, find the computation unit $\lambda_j$ that has not assigned to any thread. Subsequently, assign $\lambda_j$ to $T_{\text{min}}$ and update workload sum of each thread and repeat to find $T_{\text{min}}$ again. The algorithm terminates when all of the computation units are assigned. Note that BFAS algorithm proceeds similarly to the breadth-first search (BFS) algorithm. The pseudocode of BFAS is shown in Figure 2.7.

**Algorithm** Breadth-first allocation scheduling

1. for each thread $T_i$
2. Initialize queue $Q_i = \emptyset$ as a search queue
3. Assign random seed cell $\lambda_{\text{root}}$ to $T_i$
4. Enqueue all neighbor cells of $\lambda_{\text{root}}$ to $Q_i$
5. end for
6. while not all of the cells are assigned
7. Find the least loaded thread $T_{\text{min}}$
8. if $Q_{T_{\text{min}}}$ is empty
9. Randomly pick new seed $\lambda_{\text{root}^*}$ for thread $T_{\text{min}}$
10. Assign $\lambda_{\text{root}^*}$ to $T_{\text{min}}$
11. Add all neighbor cells of $\lambda_{\text{root}^*}$ to $Q_{T_{\text{min}}}$
2.4 Theoretical Analysis

In this section, we present an upper bound analysis for load imbalance (section 2.4.1), a lower bound analysis for memory consumption (section 2.4.2), and computational complexity analysis (section 2.4.3) of the proposed algorithms.

2.4.1 Load Imbalance Analysis

The fine-grain load balancing algorithm presented in the previous section is simple yet provides an excellent load-balancing capability. In this section, we show that this approach has a well-defined upper bound on the load imbalance. First, to quantify the load imbalance, we define a load-imbalance factor $\gamma$ as the difference between the runtime of the slowest thread and the average runtime,

$$\gamma = \frac{\max(\tau(T_i)) - \tau_{\text{average}}}{\tau_{\text{average}}}.$$  \hspace{1cm} (2.1)

By definition, $\gamma = 0$ when the loads are perfectly balanced. Since $\min(\tau(T_i)) \leq \tau_{\text{average}}$, 

Figure 2.7: Pseudocode of breadth-first allocation scheduling algorithm.
\[ \gamma \leq \frac{\max(\tau(T_i)) - \min(\tau(T_i))}{\tau_{\text{average}}}. \] (2.2)

In our load-balancing algorithm, the workload of \( T_{\text{min}} \) is increased at most by \( \max(\tau(\lambda)) \) at each iteration. This procedure guarantees that the variance of the workloads among all threads is limited by

\[ \max(\tau(T_i)) - \min(\tau(T_i)) \leq \max(\tau(\lambda_k)). \] (2.3)

Substituting Eq. (2.3) in Eq. (2.2) provides an upper limit for the load-imbalance factor,

\[ \gamma \leq \frac{\max(\tau(\lambda_k))}{\tau_{\text{average}}}. \] (2.4)

Performance of this load-balancing scheduling algorithm depends critically on the knowledge of time spent on each computation unit \( \tau(\lambda_k) \). Since the runtime of the computation units are unknown to the scheduler prior to the actual computation, the scheduler has to accurately estimate the workload of each computation unit. Fortunately, since particle positions change slowly \( \tau(\lambda_k) \) remains highly correlated between the consecutive MD steps. Therefore, we use \( \tau(\lambda_k) \) measured in the previous MD step as an estimator of \( \tau(\lambda_k) \). This automatically takes into account any local variations in the cost of the potential evaluation. For the first step as well as steps when the cell structure changes significantly (e.g., redistribution of the domain centers), the workload of cell \( \tau(\lambda_k) \) is estimated by counting the number of pairs in \( \lambda_k \).

### 2.4.2 Memory Consumption Analysis

Our nucleation growth scheduling algorithms tend to distribute workload equally among threads while minimizing the surface volume of the assigned workload in the physical space. Let
us assume that the number density of particles of the system, \( \rho \), is uniform. The memory required for storing particle forces for each thread \( m_t \) is proportional to the volume \( V_i \) occupied by the particles required for the force computation of each thread:

\[
m_t = \rho V_i. \tag{2.5}
\]

The particles within volume \( V_i \) consist of two groups: (1) main particles, \( i.e. \) the particles that are directly assigned to the thread; and (2) surface particles, \( i.e. \) the particles that interact with the main particles for the thread. We define main volume \( \Omega \) as a volume occupied by the main particles and surface volume \( \omega \) as a volume occupied by the surface particles. Then, \( V_i \) is written as

\[
V_i = \Omega + \omega, \tag{2.6}
\]

From Eqs. (2.5) and (2.6), we can calculate the memory consumption of data privatization if the main and surface volumes are known. Fortunately, both CVAS and BFAS algorithms tend to form a particular geometry, which can be estimated analytically.

CVAS and BFAS algorithms are different only during the workload assignment phase. CVAS choose the closest unassigned cells to its centroid because it minimizes the change of its centroid after assignment. This results in a spherical shape formed by CVAS (Figure 2.8(b)). On the other hand, BFAS chooses unassigned cells layer-by-layer starting with the closest to its root cell. Since the cell has cubic geometry, BFAS volume tends to form a cubic shape (Figure 2.8(a)). These considerations on the CVAS and BFAS volume geometry do not take account the following facts: (1) the other threads might prevent the algorithms to obtain the optimal cells; and (2) discretization error might cause sub-optimal geometry for the algorithms (see Figure 2.8(c) and (d)). Since these factors always increase the memory consumption from the optimal
shape, an assumption on perfect CVAS and BFAS geometry implies the best-scenario \((i.e.\) lower-bound) memory consumption for both algorithms. We perform this analysis in detail in the following subsections.

Figure 2.8: Geometric effects on surface-to-volume ratio. (a) and (b) show the surface volume (shell area) of cubic and spherical shapes with the same main volume. (c) and (d) show the discretized version, which indicates that the spherical shape consumes less volume than the cubic shape. The square grid represents cells in which particles reside. The blue area refers to the main volume while the red area shows the surface volume. Variable \(l\) is defined as the side length of main volume created by BFAS while \(r\) denotes the radius of CVAS. The surface thickness is \(r_c\).

**2.4.2.1 Lower-bound memory consumption of CVAS algorithm**

In the best case, \(\Omega\) of CVAS forms a spherical shape. In this situation, the main volume obtained by each thread is

\[
\Omega_{\text{CVAS}} = \frac{4}{3} \pi r^3, \tag{2.7}
\]
or

\[ r = \left( \frac{3}{4\pi} \Omega_{CVAS} \right)^{1/3}, \]  

(2.8)

where \( r \) is the radius of main volume’s sphere; see Figure 2.8(b). The surface volume can be calculated from half of the shell volume of the main volume (due to Newton’s third law) with the shell radius equal to \( r_c \):

\[
\omega_{CVAS} = \frac{2}{3} \pi ((r + r_c)^3 - r^3)
\]

\[
= \frac{2}{3} \pi (r^3 + 3r^2r_c + 3rr_c^2 + r_c^3 - r^3)
\]

\[
= 2\pi r^2r_c + 2\pi rr_c^2 + \frac{2}{3} \pi r_c^3.
\]  

(2.9)

Substitution of Eq. (2.9) with \( r \) from Eq. (2.8) yields

\[
\omega_{CVAS} = 2\pi \left( \frac{3}{4\pi} \Omega \right)^{2/3} r_c + 2\pi \left( \frac{3}{4\pi} \Omega \right)^{1/3} r_c^2 + \frac{2}{3} \pi r_c^3,
\]  

(2.10)

where \( \Omega = \Omega_{CVAS} \). Let dimensionless quantity \( \beta \) be

\[
\beta = \Omega^{1/3} \frac{1}{r_c},
\]  

(2.11)

or,

\[
r_c = \Omega^{1/3} \frac{1}{\beta}.
\]  

(2.12)

Substituting Eq. (2.10) with \( r_c \) from Eq. (2.12), we obtain
\[
\omega_{CVAS} = 2\pi \left( \frac{3}{4\pi} \Omega \right)^{2/3} \left( \Omega^{1/3} \beta^{-1} \right) + 2\pi \left( \frac{3}{4\pi} \right)^{1/3} \left( \Omega^{2/3} \beta^{-2} \right) + \frac{2}{3} \pi \left( \Omega \beta^{-3} \right) 
\]

\[
\omega_{CVAS} = 2\pi \left( \frac{3}{4\pi} \right)^{2/3} \Omega \beta^{-1} + 2\pi \left( \frac{3}{4\pi} \right)^{1/3} \Omega \beta^{-2} + \frac{2}{3} \pi \Omega \beta^{-3} 
\]

\[
\omega_{CVAS} = \Omega \left( 2\pi \left( \frac{3}{4\pi} \right)^{2/3} \beta^{-1} + 2\pi \left( \frac{3}{4\pi} \right)^{1/3} \beta^{-2} + \frac{2}{3} \pi \beta^{-3} \right) 
\]

\[
\omega_{CVAS} = \Omega \left( 2.4180 \beta^{-1} + 3.8978 \beta^{-2} + 2.0944 \beta^{-3} \right) 
\]

Therefore, the total volume required by each thread of CVAS is

\[
V_t^{CVAS} = \Omega_{CVAS} + \omega_{CVAS} 
= \Omega_{CVAS} \left( 1 + 2.4180 \beta^{-1} + 3.8978 \beta^{-2} + 2.0944 \beta^{-3} \right). 
\]

### 2.4.2.2 Lower-bound memory consumption of BFAS algorithm

In the best case, \( \Omega_{BFAS} \) forms a cubic shape. In this situation, the main volume obtained by each thread is

\[
\Omega_{BFAS} = l^3, 
\]

or

\[
l = \Omega_{BFAS}^{1/3}, 
\]

where \( l \) is the side length of the main volume; see Figure 2.8(a). The surface volume can be calculated from the cubic shell volume of the main volume with shell thickness equal to \( r_c \):
\[ \omega_{\text{BFAS}} = \frac{1}{2}((l + 2r_c)^3 - l^3) \]
\[ = \frac{1}{2}(l^3 + 6l^2r_c + 12lr_c^2 + 8r_c^3 - l^3) \]
\[ = 3l^2r_c + 6lr_c^2 + 4r_c^3 \quad (2.17) \]

Substitution of Eq. (2.17) with \( l \) from Eq. (2.16) yields

\[ \omega_{\text{BFAS}} = 3\Omega^{2/3}r_c + 6\Omega^{1/3}r_c^2 + 4r_c^3, \quad (2.18) \]

where \( \Omega = \Omega_{\text{BFAS}} \). Substituting \( r_c \) from Eq. (2.18) with \( \beta \) from Eq. (2.12), we obtain

\[ \omega_{\text{BFAS}} = 3\Omega^{2/3}(\Omega^{1/3}\beta^{-1}) + 6\Omega^{1/3}(\Omega^{2/3}\beta^{-2}) + 4\Omega\beta^{-3} \]
\[ = \Omega(3\beta^{-1} + 6\beta^{-2} + 4\beta^{-3}) \quad (2.19) \]

Therefore, the total volume requires by each thread of BFAS is

\[ V_t^{\text{BFAS}} = \Omega_{\text{BFAS}} + \omega_{\text{BFAS}} \]
\[ = \Omega_{\text{BFAS}} \left(1 + 3\beta^{-1} + 6\beta^{-2} + 4\beta^{-3}\right) \quad (2.20) \]

### 2.4.2.3 Memory scaling comparison between CVAS and BFAS

From Eqs. (2.14) and (2.20), we can compare how the memory consumption of CVAS and BFAS scales. First, let us assume that \( \Omega_{\text{CVAS}} = \Omega_{\text{BFAS}} \) for direct comparison. Then, substitute \( \beta \) from Eq. (2.12) into Eq. (2.14) to obtain

\[ \frac{V_t^{\text{CVAS}}}{\Omega} = 1 + 2.4180\Omega^{-1/3}r_c + 3.8978\Omega^{-2/3}r_c^2 + 2.0944\Omega^{-1}r_c^3 \quad (2.21) \]

Since each cell has volume of \( r_c^3 \), we express the main volume in terms of \( r_c^3 \) as

\[ \Omega^* = \frac{\Omega}{r_c} \quad (2.22) \]
Substitution of $\Omega^*$ from Eq. (2.22) into the RHS of Eq. (2.21) yields

$$\frac{V_{\text{CVAS}}^\prime}{\Omega} = 1 + 2.4180 \Omega^{1/3} + 3.8978 \Omega^{2/3} + 2.0944 \Omega^{1}$$ \hspace{1cm} (2.23)

Similarly, using the same analysis on BFAS volume in Eq. (2.20), we obtain

$$\frac{V_{\text{BFAS}}^\prime}{\Omega} = 1 + 3 \Omega^{1/3} + 6 \Omega^{2/3} + 4 \Omega^{1}.$$ \hspace{1cm} (2.24)

Figure 2.9 shows the scaling ratio of $V_{\text{t}}/\Omega$ as a function of $\Omega^*$ for both algorithms. The result indicates a decreasing function of the ratio for both CVAS and BFAS. The ratios decrease sharply when $\Omega^*$ is less than 6 for both algorithms. Subsequently, the ratios decrease more slowly because of the slow decrease of $\Omega^{1/3}$ term. We observe that CVAS algorithm consumes less memory compared to the BFAS algorithm due to the geometric effect on the surface-to-volume ratio of sphere and cubic as seen in Figure 2.8 (a) and (d). The memory consumption difference between BFAS and CVAS is large for small $\Omega^*$ and diminishes for larger $\Omega^*$. BFAS uses 18% more memory for $\Omega^*=50$, and less than 10% more memory when $\Omega^*>269$. This suggests that for a very large system, a relative memory saving between both algorithms is insignificant.
Figure 2.9: Illustrates a lower-bound memory consumption estimation of CVAS and BFAS algorithms. The graph indicates that CVAS consumes less memory than BFAS.

2.4.2.4 Asymptotic memory consumption for CVAS and BFAS

Equations (2.14) and (2.20) show that both algorithms have the analytical form of minimum memory requirement as

$$m_t = \rho V_t$$

$$= \rho \Omega \left(1 + a\beta^{-1} + b\beta^{-2} + c\beta^{-3}\right),$$

(2.25)

where $a$, $b$, $c$, $\in \mathbb{N}$ are pre-factors. Substitution of $\beta$ from Eq. (2.12) into Eq. (2.25) yields

$$m_t = \rho \Omega \left(1 + ar_\varepsilon\Omega^{-1/3} + br_\varepsilon^2\Omega^{-2/3} + cr_\varepsilon^3\Omega^{-1}\right)$$

$$= \rho \left(\Omega + a\Omega^{2/3} + b\Omega^{1/3} + c\right).$$

(2.26)

Let $n$ be the number of particles in a node and $p$ be the number of threads per node. On average, each thread has $n/p$ particles in the main volume $\Omega$. From the assumption that the system has uniform density, then
\[ \Omega = \frac{n}{p \rho}. \]  

(2.27)

Substitution of Eq. (2.27) into Eq. (2.26) yields

\[ m_t = \rho \left( \frac{n}{p \rho} + a \left( \frac{n}{p} \right)^{2/3} + b \left( \frac{n}{p} \right)^{1/3} + c \right) \]

\[ = \frac{n}{p} + a \left( \frac{n}{p} \right)^{2/3} + b \left( \frac{n}{p} \right)^{1/3} + c \]

(2.28)

\[ = \Omega \left( \frac{n}{p} + \left( \frac{n}{p} \right)^{2/3} \right), \]

as lower bound memory consumption per thread, which is identical to our average-case analysis [80] of \( \Theta(n/p + (n/p)^{2/3}) \) per node or \( \Theta(n + n^{2/3}p^{1/3}) \) per thread.

### 2.4.3 Computation Complexity Analysis

In this section, we analyze the computational complexity of CVAS and BFAS. CVAS algorithm starts by assigning one cell to each thread. The least loaded thread has to choose a cell that will minimize the change of its centroid after assigned, which is the unassigned cell that is the closest to the current centroid. To find that cell, the distances from the centroid to all the surrounding cells (i.e. surface cells of the thread’s volume cells) need to be calculated. Focusing on the cost of one thread \( T_i \), there is only one cell in the thread volume, which is the root cell, in the first step. The number of distance computations scales as the surface area of one cell. In the second assignment phase of \( T_i \), there are two cells in \( T_i \), thus the number of distance computations scales as the surface area of two cells. In the last assignment step of \( T_i \), \( T_i \) has \( L/p \) assigned cells, thus the number of distance computations scales as the surface area of \( L/p \) cells.
Since the surface area scales as the 2/3 powers, the number of distance computations of one thread is

\[ t_{\text{thread}} = 1 + (2)^{2/3} + (3)^{2/3} + \ldots + \left(\frac{L}{p}\right)^{2/3} = \sum_{k=1}^{L/p} k^{2/3}. \]  

(2.29)

For \( p \) threads in one node, the cost is

\[ t_{\text{node}} = pt_{\text{thread}} = p \sum_{k=1}^{L/p} k^{2/3}. \]  

(2.30)

For large \( L/p \), the summation can be approximated by integration. Therefore, the computation cost is approximated as

\[ p \sum_{k=1}^{L/p} k^{2/3} < p \int_{0}^{L/p} k^{2/3} \, dk = p \left[ k^{5/3} \right]_{k=0}^{L/p} = \frac{3}{2} p \left( \frac{L}{p} \right)^{5/3} = \Theta(L^{5/3} / p^{2/3}) \]  

(2.31)

Since particles are uniformly distributed, the number of particles for each node \( n \) scales as number of cells \( L \). Therefore,

\[ t_{\text{node}} = \Theta(\rho n^{5/3} / p^{2/3}) = \Theta(L^{5/3} / p^{2/3}) \]  

(2.32)

On the other hand, BFAS algorithm is identical to BFS on the assigning work phase. The worst-case complexity of a generic BFS is \( O(V+E) \), where \( V \) is the number of vertices and \( E \) is the number of edges. In BFAS algorithm, the number of vertices is equivalent to the number of cells \( L \) and the number of edges is equal to \( 27L \). Although multiple BFS searches run simultaneously in BFAS, the total number of vertex traversals is \( L \) steps. Each traversal consumes constant amount of neighbor-cell lookups. Thus, the running time of BFS in BFAS algorithm scales as \( \Theta(L) = \Theta(\rho n) = \Theta(n) \).
2.5 Performance Evaluation

In this section, we measure the performance of CVAS and BFAS algorithms described in section 2.3. Section 2.5.1 shows benchmarks on the scheduling cost of CVAS and BFAS algorithms. Section 2.5.2 measures the load-imbalance factor of our scheduler, and section 2.5.3 measures the memory-footprint reduction for CVAS, confirming its $\Theta(n + p^{1/3}n^{2/3})$ scaling on average. Section 2.5.4 demonstrates the reduction of the scheduling cost of CVAS without affecting the quality of load balancing, followed by strong-scaling comparison of the hybrid MPI/OpenMP and MPI-only schemes in section 2.5.5 for CVAS.

2.5.1 Scheduling Cost

We have measured average scheduling time of a 128,000-particles system over 1,000 MD steps on 64 BG/P nodes. Figure 2.10 shows the average scheduling cost when the scheduling is performed at every 15 steps. We observe that CVAS spends 8.1, 1.5, and 9.8% more time on scheduling than BFAS for the number of threads $p = 2$, 3, and 4, respectively. Since the scheduling itself takes almost no time, this measurement implies the overhead of a scheduling routine. Based on this assumption, Fig. 10 shows that the routine overhead accounts for 30-40% of CVAS and BFAS costs.
2.5.2 Thread-Level Load Balancing

We perform a load-balancing test for the CVAS on a dual six-core AMD Opteron 2.3 GHz with $n = 8,192$. In Figure 2.11, the actual measurement of the load-imbalance factor $\gamma$ is plotted as a function of $p$, along with its estimator introduced in section 2.3.2.1 and the theoretical bound, Eq. (2.4). The results show that $\gamma_{\text{estimated}}$ and $\gamma_{\text{actual}}$ are close, and are below the theoretical bound. $\gamma$ is an increasing function of $p$, which indicates the severity of the load imbalance for a highly multi-threaded environment and highlights the importance of the fine-grain load balancing.

We also observe that the performance fluctuates slightly depending on a selection of seed nodes in CVAS algorithm. While the random root selection tends to provide robust performance compared to deterministic selection, it is possible to use some optimization techniques to dynamically optimize the initial cell selection at runtime. For more irregular applications, it is
conceivable to combine the light-overhead thread-level load balancing in this chapter with a high quality node-level load balancer such as a hypergraph-based approach [81].

![Figure 2.11: Illustrates the load-imbalance factor $\gamma$ of CVAS as a function of $p$ from theoretical bound, scheduler estimation, and actual measurement.](image)

### 2.5.3 Memory Consumption

To test the memory efficiency of the CVAS algorithm, we perform simulations on a four quad-core AMD Opteron 2.3 GHz machine with a fixed number of particles $n = 8,192, 16,000, \text{ and } 31,250$. We measure the memory allocation size for 100 MD steps while varying the number of threads $p$ from 1 to 16. Figure 2.12 shows the average memory allocation size of the force array as a function of the number of threads for the proposed algorithm compared to that of a naïve data-privatization algorithm. The results show that the memory requirement for 16 threads is reduced by 65%, 72%, and 75%, respectively, for $n = 8,192, 16,000, \text{ and } 31,250$ compared with the naïve $O(np)$ memory per-node requirement. In Figure 2.12, the dashed curves show the reduction of memory requirement per thread estimated as
\[ m = ap^{-1} + bp^{-2/3} \]  

(2.33)

where the first term represents the memory scaling from actual assigned cells and the second term represents scaling from surface cells of each thread, see Eq. (2.28). The regression curves fit the measurements well, indicating that the average memory requirement is accurately modeled by \( \Theta(n^{1/3}n^{2/3}) \).

Figure 2.12: Presents average memory consumption for the private force arrays as a function of \( p \) using CVAS compared to the naïve method. Numbers in the legend denote \( n \).

2.5.4 Reduction-Sum Operation Cost

We also measure the computation time spent for the reduction sum of the private force arrays to obtain the global force array. Figure 2.13 shows the reduction-sum time as a function of the number of threads \( p \) for \( n = 8,192, 16,000, \) and \( 31,250 \) particles. Here, dashed curves represent the regression,

\[ t_{\text{reduction}} = ap^{1/3} + b \]  

(2.34)

which fit all cases well.
Figure 2.13: Shows average reduction-sum operation time of the CVAS as a function of \( p \). Numbers in the legend denote \( n \).

### 2.5.5 Strong-Scaling Benchmark

We measure the strong-scaling benchmarks of CVAS. Figure 2.14(a) shows the thread-level strong-scaling speedup on a four quad-core AMD Opteron 2.3 GHz. The algorithm achieves a speedup of 14.43 on 16 threads, i.e., the strong-scaling multi-threading parallel efficiency is 0.90. As shown in section 2.5.3, the combined algorithm reduces the memory consumption up to 65% for \( n = 8,192 \), while still maintaining excellent strong scalability.

Next, we compare the strong-scaling performance of the hybrid MPI/OpenMP and MPI-only schemes for large-scale problems on BlueGene/P at LLNL. One BlueGene/P node consists of four PowerPC 450 850 MHz processors. The MPI-only implementation treats each core as a separate task, while the hybrid MPI/OpenMP implementation has one MPI task per node, which spawns four worker threads for the force computation. The test is performed on \( P = 8,192 \) nodes, which is equivalent to 32,768 MPI tasks in the MPI-only case and 32,768 threads for hybrid MPI/OpenMP. Figure 2.14 (b) and (c) show the running time of 843,750 and 1,687,500 particles
systems for the total number of cores ranging from 1,024 to 32,768. The result indicates that the hybrid scheme performs better when the core count is larger than 8,192. On the other hand, the MPI-only scheme gradually stops gaining benefit from the increased number of cores and becomes slower. The MPI/OpenMP code shows 2.58× and 2.16× speedup over the MPI-only implementation for \( N = 0.84 \) and 1.68 million, respectively, when using 32,768 cores. Note that the crossover granularity of the two schemes is \( n/p \sim 100 \text{ particles/core} \) for both cases.
Figure 2.14: (a) Thread-level strong scalability of the parallel section on a four quad-core AMD Opteron 2.3 GHz with fixed problem size at $n = 8,192$ particles. (b) Total running time per MD steps of $1,024 – 32,768$ Power PC 450 850 MHz cores on BG/P for a fixed problem size at $N = 0.84$-million particles and (c) 1.68-million particles.
This result indicates that the Amdahl’s law limits the performance of the MPI/OpenMP code when using small number of nodes. Namely, only the pair kernel is parallelized, while the rest of the program is sequential in the thread level. This disadvantage of the MPI/OpenMP code diminishes as the number of cores increases. Eventually, the hybrid MPI/OpenMP code performs better than the MPI-only code after 8,192 cores. The main factors underlying this result are: (1) the surface-to-volume ratio of the MPI-only code is larger than that of the hybrid MPI/OpenMP code; and (2) the communication latency for each node of the MPI-only code is four times larger than that of the hybrid MPI/OpenMP code. This result confirms the assertion that the MPI/OpenMP model (or similar hybrid schemes) will be required to achieve better strong-scaling performance on large-scale multicore architectures.

### 2.6 Summary

We have designed two new data privatization algorithms CVAS and BFAS based on a nucleation-growth allocation, which have excellent memory consumption compared to a naïve data privatization. Both algorithms have been thoroughly analyzed theoretically and compared. We found that the geometric effect is the cause of more memory consumption in BFAS. The best-case memory consumption is found to scale as the same as the previously proposed average case, \( \Theta(n+p^{1/3}n^{2/3}) \). The analysis shows that the computational costs of CVAS and BFAS are bounded by \( \Theta(n^{5/3}p^{-2/3}) \) and \( \Theta(n) \), respectively. Also, benchmarks of massively parallel MD simulations suggest significant performance benefits of the hybrid MPI/OpenMP scheme for fine-grain large-scale applications. Using data privatization algorithm. Namely, 2.58× speedup has been observed for 0.8-million particle simulation on 32,768 cores of BlueGene/P.
Chapter 3

Performance Characteristics of Concurrency Control Mechanisms

In this chapter, we investigate the performance characteristics of hardware transactional memory (HTM) on the BlueGene/Q computer in comparison with conventional concurrency control mechanisms, using an MD application as an example. Benchmark tests, along with overhead-cost and scalability analyses, quantify relative performance advantages of HTM over other mechanisms. We have 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.

This chapter is organized as follows. Section 3.1 introduces concurrency control and race condition problem in general. Section 3.2 provides background information on MD and
concurrency controls. Section 3.3 characterizes each concurrency control mechanism based on a cost model. Section 3.4 analyzes the performance of concurrency controls including HTM for ddcMD code. Section 3.5 proposes a decision tree in the multithreading concurrency-control design space. Conclusions are drawn in section 3.6.

3.1 Introduction

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 MD [6, 12, 82] 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 multithreaded system, BlueGene/Q has become the first commercial platform with transactional memory (TM) implemented in hardware [83, 84]. TM is an opportunistic concurrency control [85], allowing multiple threads to execute through a critical section concurrently by simply defining a critical code block. Since software TM usually suffers considerable overhead [86], 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 chapter, we characterize the performance of HTM on BlueGene/Q, along with those of traditional concurrency control mechanisms using the MD code, ddcMD [22, 80], 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.

3.2 Background

3.2.1 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 \( \phi (r_1, r_2, \ldots, r_N) \), where \( r_i \) 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 [17-19]. We focus on the highly efficient particle-particle/particle-mesh (PPPM) method [17]. 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” $\alpha$, and computational complexity for the force calculation is reduced to $O(N\log N)$.

In this study, we use a hybrid parallel implementation of the ddcMD code [80], based on 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. 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: $\phi = \sum_{i<j} q_i q_j \text{erfc}(\alpha r_{ij})/r_{ij}$, 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.
3.2.2 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 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.

3.2.2.1 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.
3.2.2.2 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 guarantee 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.

3.2.2.3 Data Privatization

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.

3.2.3 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 [87-91]. 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 [83, 84].

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 3.1. 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 [92].

In this study, we use a hybrid parallel implementation of the ddcMD code [80], 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. 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: \( \phi = \sum_{i<j} q_i q_j \text{erfc}(\alpha r_{ij})/r_{ij}, \)
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.

Table 3.1: Specification of BlueGene/Q at LLNL.

<table>
<thead>
<tr>
<th>Processor</th>
<th>PowerPC A2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock speed</td>
<td>1.6 GHz</td>
</tr>
<tr>
<td>Physical cores per node</td>
<td>16</td>
</tr>
<tr>
<td>Hardware threads per core</td>
<td>4</td>
</tr>
<tr>
<td>Instruction unit</td>
<td>1 integer/load/store OP + 1 floating point OP per clock cycle</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>16 KB (per core)</td>
</tr>
<tr>
<td>L2 data cache</td>
<td>32 MB (shared by all cores)</td>
</tr>
<tr>
<td>Memory per node</td>
<td>16 GB DDR3</td>
</tr>
<tr>
<td>Networking</td>
<td>5D torus</td>
</tr>
<tr>
<td>Number of nodes/cores</td>
<td>98,304 nodes/1,572,864 cores</td>
</tr>
<tr>
<td>Compiler</td>
<td>IBM XLC/C++ version 12.1: mpixlc_r</td>
</tr>
</tbody>
</table>
3.3 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.

3.3.1 Concurrency-Control Cost Modeling

Consider a computational kernel, where a conflict region consists of several atomic updates placed inside a loop structure, see Fig. 3.1. The total overhead cost of a control mechanism is estimated from:

\[ t_\beta = t_{\text{control}} - t_{\text{serial}} \quad (\beta \in \{\text{critical,atomic,HTM}\}) \] (3.1)

where \( t_{\text{control}} \) is the running time of the kernel with concurrency control, \( t_{\text{serial}} \) 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, \( t_{\text{critical}} \) is independent of the size of calculation within the conflict region. Therefore, the cost model of OMP-critical reads

\[ t_{\text{critical}} = mc_{\text{lock+unlock}} \] (3.2)

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 [92]. 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

\[ t_{\text{atomic}} = m \mu c_{\text{atomic}} \]  \hspace{1cm} (3.3)

where \( \mu \) is the number of atomic updates enforced with OMP-atomic, and \( c_{\text{atomic}} \) 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 [83]) associated with every transaction. Furthermore, all speculative executions occur in L2 cache, which incurs considerable latency for L2 accesses. Therefore, the cost of each transaction also depends on the size of conflict region. Hence, the HTM cost model reads:

\[ t_{\text{HTM}} = m(c_{\text{HTM,overhead}} + \mu c_{\text{HTM,update}}) \]  \hspace{1cm} (3.4)

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 3.4.

3.3.2 Model Parameter Fitting Using Microbenchmark

Figure 3.1 shows the serial code as a reference. The serial kernel consists of a loop, in which each iteration contains 8 atomic updates. Figure 3.2 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. 3.2, HTM is applied to 4 atomic updates. The benchmark is executed using a single thread on a single core.

Figure 3.3 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 $c_{\text{atomic}}$ 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 ($\leq 4$), 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.
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);
    }
}

Figure 3.1: The benchmark kernel of a serial code without concurrency control.

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);
    }
}

Figure 3.2: The benchmark kernel with HTM control over 4 updates.
3.4 Molecular Dynamics Benchmark

Figure 3.1 shows the serial code as a reference. The serial kernel consists of a loop, in which each iteration contains 8 atomic updates. Figure 3.2 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.

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 3.4 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 $F_B[j]$ in line 28-33; and (2) the force update in
array $fA[0]$ in line 36-41 in Fig. 3.4. 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.
FOUR_VECTOR ftmp;
ftmp.v = ftmp.x = ftmp.y = ftmp.z = 0.0;
double rA0x, rA0y, rA0z;
double qA0 = qA[0];

rA0x = rA[0].x;
rA0y = rA[0].y;
rA0z = rA[0].z;

for (int k=0;k<nPList;k++)
{
double fs, vij, fxij, fyij, fzij;
int j=list[k];
double complex ff = fv[k];
vij= creal(ff);
fs = cimag(ff);
fxij=fs*(rA0x - rB[j].x);
fyij=fs*(rA0y - rB[j].y);
fzij=fs*(rA0z - rB[j].z);

double qBj = qB[j];
ftmp.v += qBj*vij;
ftmp.x += qBj*fxij;
ftmp.y += qBj*fyij;
ftmp.z += qBj*fzij;

//conflict region #1 in fB[j]
{
    fB[j].v += qA0*vij;
fB[j].x -= qA0*fxij;
fB[j].y -= qA0*fyij;
fB[j].z -= qA0*fzij;
}
}
//end nPList loop
//conflict region #2 in fA[0]
{
    fA[0].v += ftmp.v;
fA[0].x += ftmp.x;
fA[0].y += ftmp.y;
fA[0].z += ftmp.z;
}

Figure 3.4: Code fragment from the force update routine in ddcMD code, which has potential memory conflicts.
3.4.1 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 3.5(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 3.4.3. This technique significantly reduces the memory footprint,
i.e., memory reduction of a factor $64/5.04 \sim 12.7$ compared with naïve data privatization for 64 threads [80].

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.

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. (3.4), this will reduce the bookkeeping cost $mc_{HTM\text{-}overhead}$, though the update cost $m\mu c_{HTM\text{-}update}$ will be unchanged. To do this, the entire $\text{NPList}$ loop, along with the second conflict region (line 10-41 in Fig. 3.4) 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 3.5(b) shows the runtime of each mechanism normalized by that of the baseline for different numbers of 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.
Figure 3.5: (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.
3.4.2 Rollback Penalty

To quantify the cost penalty and frequency of rollback, we first define the rollback ratio, 
\[ f_{rb} = \frac{n_{rb}}{n_{\text{transaction}}}, \]
where \( n_{rb} \) and \( n_{\text{transaction}} \) 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 3.4.1) 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 3.6 shows the extra runtime as a function of rollback ratio. The total number of committed transactions in all 64 nodes is \( n_{\text{transaction}} = 9.85 \times 10^7 \) 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.

![Figure 3.6: Impact of rollback on runtime, indicating linear correlation between runtime overhead and rollback ratio.](image)

### 3.4.3 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. 3.7(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. 3.7(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 [80].
Figure 3.7: 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 3.8(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 3.8(b) shows the increase of the runtime as a function of \( p \), corresponding to the rollback profiles in Fig. 3.8(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. 3.8(b) and the rollback ratio in Fig. 3.8(a) is consistent with that in subsection 3.4.2. 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.

Figure 3.8: Rollback ratio (a) and the runtime overhead (b) as a function of the number of threads for the three scheduling algorithms.
3.5 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 3.9 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

$$f_{CR} = \frac{t_{CR}}{t_{TL}} \quad (3.5)$$

In addition to the pre-profiling parameters, we define the number of threads $p$ and the number of atomic updates inside the conflict region $\mu$. The user specifies $p$, while $\mu$ can be obtained easily from code inspection. $M_{avail}$ is the memory available for data privatization.
Figure 3.9: Typical program structure of loop-oriented applications

Figure 3.10 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 3.4, 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. $p \leq 8$ in Figs. 3.5(a) and 3.5(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 $f_{\text{CR}}$. 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 3.3 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 $f_{\text{rb}}$. 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. [84], the threshold for the usability of HTM is $f_{\text{rb}} \leq 1$. 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.
Figure 3.10: Decision tree for concurrency-control design space. We should note that OMP-critical at the bottom has very high performance penalty.

Once the decision of concurrency control mechanism is made, we can estimate the theoretical parallel efficiency for each mechanism. Table 3.2 summarizes the parallel efficiency estimate based on the profiling parameters. In Table 3.2, $t_{\text{total}}$ is the total runtime of the serial
code and $c_{\text{reduction}}$ denotes the reduction cost per particle per hypercube dimension. Figure 3.11 summarizes the advantages/disadvantages of the concurrency control mechanisms considered in this study.

Table 3.2: Parallel efficiency estimate resulting from the decision tree

<table>
<thead>
<tr>
<th>Concurrency control mechanism</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP-critical</td>
<td>$e = \min(\frac{1}{p f_{\text{cr}}}, 1)$</td>
</tr>
<tr>
<td>OMP-atomic</td>
<td>$e = \frac{t_{\text{total}}}{t_{\text{total}} + m u c_{\text{atomic}}}$</td>
</tr>
<tr>
<td>Data privatization</td>
<td>$e = \frac{t_{\text{total}}}{t_{\text{total}} + c_{\text{reduction}} n \log p}$</td>
</tr>
<tr>
<td>HTM</td>
<td>$e = \frac{t_{\text{total}}}{t_{\text{total}} + m (c_{\text{HTM, overhead}} + u c_{\text{HTM, update}})}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Scalability</th>
<th>Overhead</th>
<th>Memory</th>
<th>Applicability</th>
<th>Ease of Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical section</td>
<td>X</td>
<td>Δ</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Atomic</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
</tr>
<tr>
<td>Data Privatization</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>Δ</td>
<td>Δ</td>
</tr>
<tr>
<td>Transactional Memory</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
<td>Δ</td>
<td>O</td>
</tr>
</tbody>
</table>

Figure 3.11: Analysis of four concurrency control mechanisms. Circles and triangles denote strong and weak advantages, respectively, while crosses denote disadvantage.
3.6 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.
Chapter 4

Embedded Performance Prediction Model

We have shown that a hybrid message-passing and multithreading algorithm performs better than the MPI-only scheme when the number of executing cores is large and the problem size per core (i.e. granularity) is small. However, the number of cores is expected to be larger than millions and more in most of the emerging extreme-scale clusters, which causes the communication cost to be significantly increased compared with all of the previous studies. This renders previous performance models insufficient for the future generation of extreme-scale clusters.

In this chapter, we propose a performance prediction model for hybrid message-passing and multithreading parallelization for future massive-scale clusters based on the LogP model [93], which aims to provide performance trends of hybrid and MPI-only algorithms. As a demonstration, we will develop a model for a hybrid parallelization scheme of MD simulations, which represents a computationally sophisticated scientific problem. The model will be validated
on BlueGene/P clusters at Lawrence Livermore National Laboratory and Argonne National Laboratory.

4.1 Embedded performance prediction model

As the multicore era has emerged for extreme-scale clusters, it is uncertain which algorithms would be the best option at such a large scale. The longstanding MPI-only algorithm proved to be exceptionally efficient on current supercomputers [22, 77], but might not perform well in future generations. While the prospective hybrid algorithm has yet to prove efficient in all scenarios [80], a guiding model is needed in order to foresee the scalability of the current scientific software on the incoming clusters.

Shaw [54] proposed a novel parallel MD algorithm called Neutral Territories along with a performance prediction model for bio-MD at the extreme-scale strong scaling (i.e. small problem on a large number of cores). The model explained the circumstances where the number of processor approach infinity that Shaw’s algorithm is the optimal options against the other decomposition algorithms. The performance model also demonstrated a good analysis on a scaling of bandwidth cost but did not account for a latency cost because the program was run on a specialized low-latency hardware [23]. We should note that the latency cost is significant for most commodity clusters.

Chow and Hysom [61] presented an empirical performance model suggesting that the message-passing efficiency can be measure based on the ratio of the communication time of hybrid application and the communication time of the pure MPI version. Adhianto and Chapman [58] extended the Chow-Hysom model so as to address multiple communication threads. Although these models have only been tested on small-scale clusters, they provide good insight
to a hybrid scheme’s behavior within a node. Liebrock and Goudy [94] proposed a relaxed analytical model and a detailed methodology to design a performance model for a hybrid program, suggesting that a semi-empirical approach provides good balance between prediction accuracy and extrapolation capability. Wu and Taylor [95] proposed a performance prediction model and benchmarked it for hybrid memory-intensive applications, which confirmed that the reduced performance of such applications was due to bandwidth limitation.

In our hybrid performance model, we aim to provide a performance model that predicts performance trends toward large-scale applications using hybrid parallel algorithms. To achieve this, we need a model that balances between accuracy of empirical approach and adaptability of the analytical model. Based on our experiences, we make the following assumptions:

(i) Hybrid parallel algorithms are based on combinations of shared and distributed memory models.
(ii) In one program, a different code sections have different computation/communication patterns.
(iii) Scalability of domain decomposition via message passing is limited by the communication, while the parallel efficiency in the computation section is usually excellent.
(iv) Multithreading benefits from the shared memory, which has negligible communication cost among cores.

Assumption (i) is based on a hierarchical structure of modern platforms, which are: (1) Inter-node parallelization utilizes distributed-memory parallelism through MPI; (2) inter-core communication uses shared-memory paradigm via multithreading; (3) core-to accelerator interaction explicitly transfers data by message passing; and (4) intra-accelerator communication uses shared-memory paradigm though specialized multithreading scheme (e.g. single-instruction multiple-threads in GPGPU). The architectural hierarchy described above is shown in Figure 4.1.
Assumption (ii) holds true since a typical program usually comprises of many computation kernels that use different algorithms. Assumption (iii) is consistent with the model proposed by [58, 61, 95], which expresses the parallel efficiency of message passing via the communication cost ratio of hybrid and MPI-only implementations. Assumption (iv) was confirmed in our previous study [35], where the time spent on parallel section was overwhelmed by the sequential section as the number of core usage increased. This finding agrees with the result presented by Chow and Hysom [61].

Based on the aforementioned assumptions, we have designed an embedded performance prediction model (EPM) to predict the performance trend of applications that implement a combination of message passing and multithreading.
4.1.1 Grand Scheme of the Embedded Performance Prediction Model

The grand scheme of EPM expresses application runtime as a linear combination of computation and communication costs. Let $l = 1...n_{\text{arch}}-1$ denotes the architecture level, where $l = 1$ refers to the parallelism on the top-level architecture (e.g. message passing in the hybrid MPI/OpenMP scheme). $n_{\text{arch}}$ denotes the number of levels in the structure of hybrid architecture.

Runtime prediction $T(N)$ of the $l$-th level for the message-passing parallel scheme is:

$$T_{l}^{\text{MP}}(N) = \sum_{k \in S} a_{l}^{k} \Omega_{l}^{k} \left( \frac{N}{P_{l}} \right) + \sum_{k \in S} L_{l}^{k} + \sum_{k \in S} b_{l}^{k} W_{l}^{k} \omega_{l}^{k} \left( N, P_{l}^{k} \right), \quad (4.1)$$

while a runtime prediction of multithreading is:

$$T_{l}^{\text{MT}}(N) = \left( \sum_{k \in S} a_{l}^{k} + \sum_{k \in S} b_{l}^{k} \right) \Omega_{l}(N), \quad (4.2)$$

where

$N$ = Problem size

$S$ = Set of computation code sections

$S'$ = Set of communication code sections

$\Omega_{l}^{k}(N)$ = Computation cost function at $k$-th section of $l$-th level.

$\omega_{l}^{k}(N, P)$ = Communication cost function at $k'$-th section of $l$-th level.

$L_{l}^{k}$ = Latency cost function at $k$-th section of $l$-th level.

$W_{l}^{k}$ = Bandwidth cost function $k'$-th section of $l$-th level.
\( p_i^k \) = Number of parallel units (e.g. node, core) used at \( k \)-th section of \( l \)-th level.

\( a_i^k, b_i^k \) = Pre-factor cost of \( k \)-th section at level 1.

The model allows a program to have various sections with different computation/communication characteristics, following the assumption (ii) in section 4.1. The model uses the semi-empirical approach suggested by Liebrock and Goudy [94] to balance accuracy and extrapolatability. Architecture-and-program dependent parameters need to be fitted to provide accurate cost of key operations in the applications.

The grand scheme is a framework to describe common factors that affect performance of general scientific codes. To improve the accuracy of the model, specific analytical terms should be adopted. We illustrate this tailoring of the grand scheme for MD in subsection 4.1.1.2.

4.1.2 Tailored Scheme of the Embedded Performance Prediction Model For Molecular Dynamics

Molecular dynamics is a challenging application for performance prediction. The main complications are: (1) dynamic nature of the simulation causes rapidly fluctuating performance; and (2) irregular memory-access pattern incurs uneven performance for each run. Here, we assume a hybrid algorithm similar to the ddcMD code proposed in chapter 2. Hence, the tailored model of hybrid algorithm is expressed as

\[
T(N) = a\Omega_{\text{thread}} \left( \frac{N}{P} \right) + 12t_{\text{latency}} + bW \left( \frac{N}{P} \right)^{2/3} + t_{\text{latency}} \log P ,
\]

and
\[ \Omega_{\text{thread}} (N) = \left( \frac{c}{p} + d \right) \left( T_s \frac{13}{2} \rho^3 r_c^6 L^3 + T_c \frac{2}{3} \pi \rho^3 r_c^6 L^3 \right). \] (4.4)

On the other hand, the MPI-only model is expressed as

\[ T(N) = a \Omega_{\text{MPI}} \left( \frac{N}{Pp} \right) + 12 t_{\text{latency}} + b Wp \left( \frac{N}{Pp} \right)^{2/3} + t_{\text{latency}} \log Pp, \]
\[ = a \Omega_{\text{MPI}} \left( \frac{N}{Pp} \right) + 12 t_{\text{latency}} + b Wp^{1/3} N^{2/3} P^{2/3} + t_{\text{latency}} \log Pp, \] (4.5)

and

\[ \Omega_{\text{MPI}} (N) = \left( T_s \frac{13}{2} \rho^3 r_c^6 L^3 + T_c \frac{2}{3} \pi \rho^3 r_c^6 L^3 \right), \] (4.6)

where

\[ N \quad \text{= Number of atoms} \]
\[ P \quad \text{= Number of nodes} \]
\[ p \quad \text{= Number of core per nodes} \]
\[ \rho \quad \text{= Number density of atoms per cell} \]
\[ r_c \quad \text{= Cut-off radius of the potential} \]
\[ L \quad \text{= Total number of cells per node} \]
\[ T_s, T_c \quad \text{= Key operations cost} \]
\[ a, b, c, d \quad \text{= Fitting parameters} \]

The models in Eqs. (4.3)-(4.6) consider the following functions of the code: (1) force computation routines; (2) atom caching through halo exchange; (3) latency cost of global summations; (4) pair-search kernel in computation routines; and (5) actual force computation kernel. Since only small amount of data is communicated through global summation in this MD
code, the bandwidth cost for global communication is negligible. There are a total of six fitting parameters required in the tailored model. Fortunately, we can reduce the complexity of parameterization by fitting each parameter individually because runtime can be measured for each code sections separately.

### 4.1.3 Model Validation

In the first phase, the model will be tested on a single machine in order to validate the threading parts of the model. Subsequently, we will perform our small- and moderate-scale benchmarks at the HPCC facility of USC. In this facility, we have access to a collection of different architectures of Intel Xeon and AMD Opteron machines. In the final stage, the model will be validated on BlueGene/P and/or BlueGene/Q at Lawrence Livermore National Laboratory and Argonne National Laboratory. This gives an opportunity to validate the extrapolation capability of the model on large-scale runs, thus confirming the ability to predict the performance toward future architectures. The platform specifications are given in the Table 4.1.

**Table 4.1: Benchmark cluster specifications**

<table>
<thead>
<tr>
<th>Specification</th>
<th>USC-HPCC</th>
<th>LLNL-BlueGene/P</th>
<th>LLNL-BlueGene/Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of nodes</td>
<td>785 dual cores/dual processors + 1,990 quad- and hex-cores nodes</td>
<td>36,864 nodes</td>
<td>98,304 nodes</td>
</tr>
<tr>
<td>Number of cores per node</td>
<td>2/8/12</td>
<td>4</td>
<td>16 computation + 1 operating system + 1 reserved</td>
</tr>
<tr>
<td>Memory-access</td>
<td>Mixed NUMA and</td>
<td>SMP</td>
<td>SMP</td>
</tr>
<tr>
<td>Specification</td>
<td>USC-HPCC</td>
<td>LLNL-BlueGene/P</td>
<td>LLNL-BlueGene/Q</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------</td>
<td>----------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>model</td>
<td>SMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>Mostly 16-24 GB/node</td>
<td>4 GB/node, 147 TB total</td>
<td>16 GB/node, 1.6 PB total</td>
</tr>
<tr>
<td>Clock speed</td>
<td>2.3/2.5/2.66 GHz</td>
<td>850 MHz</td>
<td>1.6 GHz</td>
</tr>
<tr>
<td>Networking</td>
<td>Myrinet 10g</td>
<td>Proprietary</td>
<td>Proprietary</td>
</tr>
<tr>
<td>Network topology</td>
<td>Mixed</td>
<td>3D mesh + specialized for collective</td>
<td>3D mesh + specialized for collective</td>
</tr>
</tbody>
</table>

### 4.2 Optimal Parallelization Prediction via Performance Model

Based on the benchmark results comparing hybrid MPI/OpenMP and MPI-only parallelization of MD in [80], we expect that the crossover of optimality of between the two schemes is strongly correlated with granularity. Figure 4.2 shows a performance benchmark on BlueGene/P at LLNL, which implies that the optimal-region boundary is linear in terms of granularity $N/P$. In this section, we develop a performance models which explain those results and predicts the behaviors at the extremities.
Figure 4.2: Optimal region of MPI-only versus hybrid MPI/OpenMP implementations. The crossover line expected to remain constant at $N/P$.

### 4.2.1 Performance Model Design Principles

The observations from the benchmark results in [80] can be summarized as the following:

- There are at least 2 terms in the performance model – decreasing as a function of number of nodes $P$ and increasing as a function of $P$.
- The decreasing term is dominant when $P$ is small while the increasing term is dominant when $P$ is large.
- Since the MPI-only code uses the same computation kernel as the hybrid code, the model should be described by the same function with different parameters.
Based on the aforementioned observations, we design an embedded performance model (EPM) using on LogP paradigm [93] (i.e. the model described by latency, overhead, bandwidth, and number of processors). The following assumptions are made for the EPM model, which is closely to the environment/system in [80]:

- One thread per core is used to obtain maximum performance.
- Memory bandwidth is not described in this model.
- The multithreading model is based on SMP architecture.
- Network is not congested.
- Not all of the computation routine can be multithreaded. Only a fixed fraction of the routine. (i.e. Amdahl’s law).
- For a particular multicore architecture, the number of core per node is constant.
- Network topology of the cluster is 3D torus (e.g. on BG/L and BG/P) and the MPI task assignment is random.

The non-congest computation-communication cost model is defined as:

\[ T = T_{\text{comp}} + T_{\text{comm}}, \]

where

\[ T_{\text{comm}} = T_{\text{bandwidth}} + T_{\text{latency}}. \]

Thus, the EPM model is

\[ T(N, P, \eta, \lambda, \alpha) = a \left( \frac{\alpha}{\lambda} + (1 - \alpha) \right) \frac{N}{P \eta} + b \eta \left( \frac{N}{P} \right)^{\frac{2}{3}} + c \eta P^{\frac{1}{3}}, \quad (4.7) \]

where

- \( T \) = Predicted runtime
- \( N \) = Number of atoms
- \( P \) = Number of nodes
\[ \eta \quad = \text{Number of MPI tasks per node} \]

\[ \lambda \quad = \text{Number of threads per MPI task} \]

\[ \alpha \quad = \text{Thread parallelizable fraction, } 0 \leq \alpha \leq 1 \]

\[ a, b, c \quad = \text{Fitting parameters} \]

Note that the number of cores per node \( q = \eta \lambda \) is constant while total number of cores in the system is \( Pq \). In this study, we compare the performance of the traditional MPI-only and a hybrid MPI/OpenMP implementations. We will derive the tailored models for MPI-only implementation and a full hybrid MPI/OpenMP as special cases in the following section.

### 4.2.1.1 Traditional MPI-only performance model

In the traditional MPI-only implementation, we use one MPI task per core. In this case, we assign \( \eta = q \) and \( \lambda = 1 \) in the EPM model. Hence, the MPI model reads

\[
T_{\text{MPI}}(N, P, q) = a \frac{N}{Pq} + bq^{\frac{1}{3}} \left( \frac{N}{P} \right)^{\frac{2}{3}} + cqP^{\frac{1}{3}}.
\] (4.8)

### 4.2.1.2 Hybrid MPI/OpenMP performance model

Hybrid MPI/OpenMP describes hybrid implementation, which handles all of the compute cores in each node using multithreading. In particular, we assign \( \eta = 1 \) and \( \lambda = q \) in the EPM model. Therefore, the hybrid performance model can be expressed as

\[
T_{\text{hybrid}}(N, P, q, \alpha) = a \left( \frac{\alpha}{q} + (1 - \alpha) \right) \frac{N}{P} + b \left( \frac{N}{P} \right)^{\frac{2}{3}} + cP^{\frac{1}{3}}.
\] (4.9)
4.2.2 Model Fitting and Validation

In this section, we fit the MPI-only and hybrid performance models using the runtime data of ddcMD in [80]. The system size is varying from 1.6, 0.8, 0.4, and 0.2 million-particle on up to 256-8,196 BG/P nodes ($2^{10}$-$2^{15}$ cores) at Lawrence Livermore National Laboratory. We first need to find the prefactors (i.e. $a$, $b$, $c$ in Eqs. (4.7)-(4.9)). We begins by using linear regression to fit parameters. However, the result cannot be obtained (i.e. at least one of $a,b,c$ becomes negative) due to the 2 magnitudes different of the residues from the data values. This is the same situation that we observed in our previous study [97], thus we use bi-square robust regression instead. The resulting parameters are shown in Table 4.2.

Table 4.2: Parameter-fitting result of MPI-only and hybrid models using bi-square regression

<table>
<thead>
<tr>
<th>Model</th>
<th>$a$ (µs)</th>
<th>$b$ (µs)</th>
<th>$c$ (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI-only</td>
<td>52.42</td>
<td>68.14</td>
<td>182.8</td>
</tr>
<tr>
<td>Hybrid $\alpha = 0.25$</td>
<td>40.79</td>
<td>228.8</td>
<td>297.6</td>
</tr>
<tr>
<td>Hybrid $\alpha = 0.50$</td>
<td>53.02</td>
<td>228.8</td>
<td>297.6</td>
</tr>
<tr>
<td>Hybrid $\alpha = 0.75$</td>
<td>75.75</td>
<td>228.8</td>
<td>297.6</td>
</tr>
</tbody>
</table>

Using parameters from Table 4.2, we compare the predicted runtime of a 1.6 million-particle system with the actual measurement for MPI-only implementation (Fig. 4.2(a)) and hybrid implementation (Fig. 4.2(b)). The plots indicate that most of the prediction results agree with the on both systems. For MPI-only model, only at the smallest granularities that the model predicts slowdown but the actual result is not. However, when we compare the two predicted
results (Fig. 4.3), the crossover of optimality is predicted to be at ~100 particles/core. This agrees well with the actual measurement.

To see the results of performance model further, Figure 4.5 shows an extrapolation of the prediction model using 54.8 million particles on up to 1 million cores cluster. The number of cores per node remains at 4. The result suggested that the crossover is shifted to 200 – 400 particles/core instead of ~100 particles per core in the smaller cluster. The performance result predicts that the optimal region of hybrid implementation is increase compared to the traditional model.

Figure 4.3: Validation of (a) MPI-only and (b) hybrid MPI/OpenMP performance models. Blue lines denote the model results while the red dots denote the actual measurement on BlueGene/P.
Figure 4.4: The performance model predictions of 1.6 million-particle system of MPI-only versus hybrid MPI/OpenMP implementations, indicating the same result as in terms of optimality region as the actual measurements.

Figure 4.5: Extrapolation of the prediction models, fixing granularity per core constant. The predicted model suggested that the crossover is moved up to ~ 200 – 400 particles/core.

4.3 Summary

We have designed performance prediction models for MPI and MPI/OpenMP schemes based on $LogP$. The crossover of optimality obtained from the model between the two schemes is strongly correlated with granularity, which agrees well with the experiment results. An extrapolation of the prediction model using 54.8 million particles on up to 1 million cores cluster indicates that the optimal region of hybrid implementation increase at such extremity compared
to the traditional model. This suggests the likely benefit of hybrid model on the extreme-scale
run of small system, which aims to minimize time-to-solution of the problem.
Chapter 5

*n*-Tuple Computation Formalism and Solution for Ranged-Limited Many-Body Molecular Dynamics

In this chapter, we discuss a development of a computation-pattern algebraic framework to mathematically formulate general *n*-tuple computation used in reactive MD. We also discuss translation/reflection-invariant properties of computation patterns within this framework, which is the basis of our shift-collapse (SC) algorithm for cell-based parallel MD.

This chapter is organized as follows. Section 5.1 discusses the problem regarding *n*-tuple computation. Section 5.2 provides background information on *n*-tuple MD. Section 5.3 presents the computation-pattern algebraic framework for *n*-tuple computation and the SC algorithm along with a proof of its correctness. Section 5.4 analyzes the search-space size and communication cost of the SC algorithm. Section 5.5 presents performance benchmark results on BlueGene/Q and Intel Xeon clusters. Conclusions are drawn in section 5.6.
5.1 Problem Statement

In contrast to these remarkable progresses in parallel algorithms for dynamic pair \( (n = 2) \) and static \( n \)-tuple computations, parallel dynamic \( n \)-tuple computation is still in its beginning stage. Fundamental questions include: How can we generalize the computation-redundancy removal in the HS scheme and the import-volume reduction in the ES scheme developed for pair computation into arbitrary dynamic \( n \)-tuple computations?

To answer these questions in a mathematically rigorous and systematic manner, we here develop a novel computation-pattern algebraic framework. Based on this framework and translation- and reflection-invariant properties of \( n \)-tuple computations, we then design a shift-collapse (SC) algorithm. Our algebraic framework allows not only a formal proof of force-computation completeness but also a quantitative analysis of \( n \)-tuple search cost and import volume. We will show, for general dynamic \( n \)-tuple computations, that: (1) the pair HS scheme can be generalized to a reflective-collapse (RC) scheme to tighten the \( n \)-tuple search space; and (2) the pair ES scheme can be generalized to an octant-compression (OC) shift scheme to minimize the import volume.

5.2 Background

In this section, we describe many-body MD, followed by the introduction of dynamic range-limited \( n \)-tuple computation in it.

5.2.1 Many-Body Molecular Dynamics

MD simulation follows the time evolution of an \( N \)-atom system by numerically integrating Newton’s equations of motion:
\[ m_i \frac{d^2 x_i}{dt^2} = f_i = -\frac{\partial \Phi(X)}{\partial x_i}, \]

(5.1)

where \( X = \{x_0, \ldots, x_{N-1}\} \) denotes a set of atomic positions, \( m_i \) and \( f_i \) are the mass of and force acting on atom \( i \), and \( t \) is time. In Eq. (5.1), the many-body interatomic potential-energy function \( \Phi(X) \) is a sum of \( n \)-body potential terms \( \Phi_n \):

\[
\Phi(X) = \Phi_2 + \Phi_3 + \cdots + \Phi_{n_{\text{max}}} = \sum_{n=2}^{n_{\text{max}}} \Phi_n,
\]

(5.2)

where \( n_{\text{max}} \) is the maximum \( n \) and \( \Phi_n \) is a function of \( n \)-tuples of atomic positions \((x_0, \ldots, x_{n-1})\).

Force computation is the most time-consuming step in MD simulation. For each simulation time step, we need to map the current set of atomic positions, \( R = \{r_0, \ldots, r_{N-1}\} \), to \( n \)-body force terms for all \( n \in [2, n_{\text{max}}] \) and all atoms \( i \in [0, N-1] \) as

\[
f_i^{(n)} = -\sum_{\forall (x_0, \ldots, x_{n-1}) \in \Gamma^{(n)}} \frac{\partial}{\partial x_i} \Phi_n(x_0, \ldots, x_{n-1})_{(x_0, \ldots, x_{n-1})=(r_0, \ldots, r_{n-1})},
\]

(5.3)

where \( \Gamma^{(n)} \) denotes the set of all \( n \)-tuples of atom positions in \( R \). For a given \( n \)-tuple, \( \chi = (r_0, r_1, \ldots, r_{n-1}) \) (see Figures 5.1 (a)-(c) for \( n = 2-4 \)), forces exerted on all atoms in \( \chi \) can be calculated simultaneously in one computational step:

\[
\forall r_i \in \chi : f_i^{(n)} \leftarrow f_i^{(n)} - \frac{\partial}{\partial x_i} \Phi_n(x_0, \ldots, x_{n-1})_{(x_0, \ldots, x_{n-1})=(r_0, \ldots, r_{n-1})}. \]

(5.4)

Namely, the set of \( n \)-body forces on all atoms can be decomposed into partial contributions from different \( n \)-tuples \( \chi \).
\[
\left\{ \mathbf{F}^{(n)}_i | i \in [0, N-1] \right\} = \mathbf{F}^{(n)} = \bigcup_{\chi \in \Gamma^{(n)}} \mathbf{F}^{(n)}_\chi.
\] (5.5)

In all problems we consider, \( \chi \) in Eq. (5.5) is undirectional, reflecting the Newton’s 3\textsuperscript{rd} law [61, 98]. Namely, \((r_0, r_1, ..., r_{n-1})\) produces the same forces as \((r_{n-1}, r_{n-2}, ..., r_0)\), thus they are not counted as separate entities. We call this reflective equivalence.

Assume that force calculation for each \( \mathbf{F}^{(n)}_\chi \) can be executed in constant time. The computational complexity of \( n \)-body force computation is then proportional to the size of \( \Gamma^{(n)} \), \textit{i.e.}, \( N!/[(2(N-n)!)] = O(N^n) \) for a system with \( N >> n \) (which is the case in most MD simulations). Thus, the computational complexity is dictated by that of the largest \( n \)-body term, \( O(N^{n_{\text{mx}}}) \).

Figure 5.1: Schematic of range-limited \( n \)-tuples: (a) pair \((n = 2)\), (b) triplet \((n = 3)\), and (c) quadruplet \((n = 4)\).

5.2.2 Dynamic Range-Limited \( n \)-Tuple Computation

In Atomic interaction is often range-limited, \textit{i.e.}, only atoms within short distances contribute to forces. Force computation for a range-limited \( n \)-body interatomic potential is defined as Eq. (5.3), where \( \Gamma^{(n)} \) is replaced by its subset \( \Gamma^{*(n)} \subseteq \Gamma^{(n)} \):

\[
\Gamma^{*(n)} = \left\{ (r_0, ..., r_{n-1}) \right\} \mid r_{k,k+1} < r_{\text{cut}-n} \text{ for all } k \in \{0, ..., n-2\}, \tag{5.6}
\]

where \( r_{k,k+1} \) is the interatomic distance between \( r_k \) and \( r_{k+1} \) and \( r_{\text{cut}-n} \) is the cutoff distance for \( n \)-body interaction, see Figures 5.1(a)-(c).
Formal complexity of pruning $\Gamma^{(n)}$ to obtain $\Gamma^*(n)$ is exponential in $n$, which is not practical. However, it is possible to efficiently find a set of $n$-tuples $S^{(n)} \subseteq \Gamma^{(n)}$ such that $|S^{(n)}| \ll |\Gamma^{(n)}|$ and $\Gamma^*(n) \subseteq S^{(n)}$ (Figure 5.2). After finding $S^{(n)}$, elements in $\Gamma^*(n)$ can be obtained simply by exhaustive “filtering” from $S^{(n)}$. Since $S^{(n)}$ is used to compute forces, hereafter we denote $S^{(n)}$ as a force set, and $S^{(n)}$ that satisfies $\Gamma^*(n) \subseteq S^{(n)}$ as a bounding force set. Thus, MD force computation can be restated as a problem to find a bounding force set $S^{(n)}$ for all $n \in \{2, \ldots, n_{\text{max}}\}$. To solve this problem, an efficient pruning algorithm $A: R \Rightarrow S^{(n)}$ is required. A widely used approach to achieve this is a cell method. It employs a cell data structure to prune $\Gamma^{(n)}$ in $O(N)$ time to obtain $S^{(n)}$ that tightly bounds $\Gamma^*(n)$.

![Figure 5.2: Set diagram showing the relationship of a bounding force set $S^{(n)}$ with $\Gamma^{(n)}$ and $\Gamma^*(n)$](image)

Cell-based methods are powerful yet simple. Its procedure is straightforward to implement for the case of pair computation. However, cell methods for general $n$-tuple computation have not been formalized mathematically. To address this problem systematically, we formulate a cell-based $n$-tuple MD problem using an algebraic framework in the next section.

### 5.3 Computation-Pattern Algebraic Framework

In this section, dynamic range-limited $n$-tuple MD is formalized using a computation-pattern algebraic framework. Subsection 5.3.1 introduces a uniform-cell pattern (UCP) formalism, which is an algebraic generalization of conventional cell methods to solve $n$-tuple
MD problems for arbitrary $n$. In subsection 5.3.2, we propose a shift-collapse (SC) algorithm to efficiently solve $n$-tuple MD on the basis of UCP. Correctness of the SC algorithm is proved in subsection 5.3.3.

5.3.1 Algebraic Formulation of Cell-Based MD

5.3.1.1 Cell data-structure

Cell-based MD divides a simulation volume into a non-overlapping set of small cells with side lengths equal or slightly larger than $r_{\text{cut}-n}$. Each cell volume $\zeta$ contains a subset of atoms in $\mathbf{R}$ that fall within its volume:

$$ c = \{ \mathbf{r}_i | \forall \mathbf{r}_i \in \mathbf{R} \text{ and in volume } \zeta \}, \quad (5.7) $$

see Figure 5.3(a). For simplicity, we consider a simple lattice of cubic cells. Then, each cell $c(q)$ can be indexed using a 3-element vector $q = (q_x, q_y, q_z)$, which specifies the Cartesian coordinate of the cell position in the lattice. Let $L_x, L_y,$ and $L_z$ be the number of cells in the $x$, $y$, and $z$ directions, respectively. Then, the set of all cell indices constitutes a vector space $\mathbf{L}$:

$$ \mathbf{L} = \left\{ q = (q_x, q_y, q_z) \left| \begin{array}{c} \forall q_x \in \{0,\ldots,L_x-1\} \\ \forall q_y \in \{0,\ldots,L_y-1\} \\ \forall q_z \in \{0,\ldots,L_z-1\} \end{array} \right. \right\}. $$

An essential ingredient of our algebraic formulation is a cell domain $\Omega$, which is defined as a set of all cells:

$$ \Omega = \{ c(q) | \forall q \in \mathbf{L} \}. \quad (5.8) $$
In this chapter, we assume periodic boundary conditions in all Cartesian directions. Then, for cell \( c(q) \), we define a cell-offset operation \( c(q + \Delta) \rightarrow c(q') \) \((\Delta \in \mathbb{Z}^3)\) as

\[
q' = (q + \Delta) \% L \quad (\alpha \in \{x, y, z\})
\]

where \( \% \) is a modulo operation. Note that \( \Omega \) needs to be dynamically constructed every MD step because the positions of atoms are changing as the computation proceeds.

![Diagram](image)

Figure 5.3: (a) Cell data-structure, where circles are atoms and small squares are cells. Dashed circle denotes the cutoff radius centered at atom \( i \). (b) Uniform-cell pattern for pair computation, which generates all pairs between each cell and its neighbor cells.

### 5.3.1.2 Uniform-cell pattern MD

UCP formalism is a generalization of a cell-based MD method using an algebraic approach (Figure 5.3(b)). To compute forces for each \( n \)-body potential term, UCP generates a force set \( S^{(n)} \) using information from cell domain \( \Omega \) along with a computation pattern \( \Psi^{(n)} \), which is a set of pre-define rules regarding interaction among cells. UCP applies \( \Psi^{(n)} \) to each cell \( c \) and generates a cell search-space \( S_{\text{cell}} \), a set of \( n \)-tuples that forms a force subset \( S_{\text{cell}}(c, \Psi^{(n)}) \subseteq S^{(n)} \). By looping over all cells, it generates the entire \( S^{(n)} \). This is analogous to stencil computation for solving partial differential equations, in which a stencil (equivalent to computation pattern in our
framework) defines local computation rules for each grid point and its neighbor grid points and the stencil is applied to all grid points in a lattice to solve the problem [27, 28].

To describe UCP, we first define a computation path \( p^{(n)} \) for \( n \)-tuple computation as a list of \( n \) vectors in \( \mathbb{L} \) (see each arrow in Figure 5.3(b) as an example of pair-computation path):

\[
p^{(n)} = (v_0, \ldots, v_{n-1}) \in \mathbb{L}^n.
\]

The inverse of \( p \) is defined as \( p^{-1} = (v_{n-1}, \ldots, v_0) \). In addition, we define a differential representation, \( \sigma(p) \in \mathbb{L}^{n-1} \), of each path \( p \) as

\[
\sigma(p) = (v_1 - v_0, \ldots, v_{n-1} - v_{n-2}).
\]

Then, a computation pattern is defined as a set of computation paths \( \Psi^{(n)} = \{p^{(n)}\} \) (i.e. set of all arrows in Figure 5.3(b)). Given a cell domain \( \Omega \) and a computation pattern \( \Psi^{(n)} \), a force set \( S^{(n)} \) is obtained as a union of cell search-spaces \( S_{\text{cell}} \):

\[
S^{(n)} = \text{UCP}(\Omega, \Psi^{(n)}) = \bigcup_{\forall c(q) \in \Omega} S_{\text{cell}}(c(q), \Psi^{(n)}), \tag{5.9}
\]

where cell search-space \( S_{\text{cell}} \) for each cell \( c(q) \) is defined as

\[
S_{\text{cell}}(c(q), \Psi^{(n)}) = \left\{ (r_0, \ldots, r_{n-1}) \right\} \quad \forall p = (v_0, \ldots, v_{n-1}) \in \Psi^{(n)} \quad \forall k \in \{0, \ldots, n-1\} : \forall r_k \in c(q + v_k). \tag{5.10}
\]

Namely, \( S_{\text{cell}} \) is a set of \( n \)-tuples contained in all paths \( p \in \Psi^{(n)} \), which is associated with cell \( c(q) \), see Figure 5.4. An algorithm that performs operations in Eqs. (5.9) and (5.10) is given in Figure 5.5. \( S^{(n)} \) in Eq. (5.9) is obtained by looping over all \( c \in \Omega \) (line 2), where each iteration
adds $S_{\text{cell}}(c)$ to $S^{(n)}$ (line 9). In order for UCP to be a valid MD computation, its computation pattern $\Psi^{(n)}$ must satisfy the $n$-body completeness condition by generating a bounding force set:

$$\Gamma^{(n)}(\Omega, \Psi^{(n)}) \subseteq \text{UCP}(\Omega, \Psi^{(n)}).$$ (5.11)

In this way, MD problem amounts to finding a computation pattern that satisfies Eq. (5.11). Such a computation pattern is regarded as $n$-complete. Existing cell-based methods can be expressed as particular computation patterns, and our algebraic formalism can be used to prove their completeness (e.g. HS and ES for pair-completeness). This will be discussed in section 5.4.3.

Satisfying Eq. (5.11) only guarantees that $\Psi^{(n)}$ is sufficient to generate a bounding force set using the UCP algorithm in Figure 5.5. The generated force set could still contain duplicated or reflectively equivalent tuples as mentioned in section 5.2.1. In a practical implementation, these redundant tuples must be removed. Hence, the cost of the UCP algorithm, $T_{\text{UCP}}$, is a cost for filtering out the unnecessary tuples from cell search-space $S_{\text{cell}}$ of every cell. This is proportional to a sum of all search-space sizes $|S_{\text{cell}}|$:

$$T_{\text{UCP}} \propto \sum_{\forall c \in \Omega} |S_{\text{cell}}(c, \Psi^{(n)})|. \quad (5.12)$$
Figure 5.4: \( n \)-tuple \((n = 3)\) generation for cell search-space \( S_{\text{cell}} \) of cell \( c(q) \) (colored in magenta) and computation path \((v_0, v_1, v_2)\). \( S_{\text{cell}} \) generates all triplets where the 1\(^{st}\), 2\(^{nd}\), and 3\(^{rd}\) atoms in the triplet are in cells \( c(q+v_0) \), \( c(q+v_1) \), and \( c(q+v_2) \), respectively. Dashed line shows one of the triplets \((r_i, r_j, r_k)\).

**Algorithm UCP**

**Input:**

\( \Omega \) : a cell domain

\( \Psi^{(n)} \subseteq L^n \) : a computation pattern

**Output:**

\( S^{(n)} \subseteq R^n \) : a force set

**Steps:**

1. \( S^{(n)} = \emptyset \)
2. for \( \forall c(q) \in \Omega \)
3. \( S_{\text{cell}}(c(q)) = \emptyset \)
4. for \( \forall p = (v_0, ..., v_{n-1}) \in \Psi^{(n)} \)
5. for \( \forall r_0 \in c(q + v_0) \)
6. \( \vdots \)
7. for \( \forall r_{n-1} \in c(q + v_{n-1}) \)
8. \( S_{\text{cell}}(c(q)) \leftarrow S_{\text{cell}}(c(q)) \cup \{(r_0,\ldots,r_{n-1})\} \)

9. \( S^{(n)} \leftarrow S^{(n)} \cup S_{\text{cell}}(c(q)) \)

Figure 5.5: Uniform-cell pattern MD algorithm.

**5.3.1.3 Parallel UCP**

We consider parallel MD algorithms, where different sets of atoms \( c \) along with associated computation \( S_{\text{cell}}(c, \Psi^{(n)}) \) are assigned to different processors. Since \( S_{\text{cell}}(c, \Psi^{(n)}) \) requires data from other cells, these data must be imported if they reside in other processors. Depending on computation paths in \( \Psi^{(n)} \), different cells need be imported. This leads to different import volumes (i.e. the number of imported cells), and hence different communication costs.

To quantify the import volume, we define the cell coverage \( \Pi \) of a computation pattern \( \Psi^{(n)} \) as a set of cells that are needed in order to compute \( S_{\text{cell}}(c(q), \Psi^{(n)}) \):

\[
\Pi(c(q), \Psi^{(n)}) = \left\{ c(q + v_k) \mid \forall p = (v_0,\ldots,v_{n-1}) \in \Psi^{(n)} \land \forall k : v_k \in p \right\},
\]

for an arbitrary cell \( c(q) = c \). We also define the cell footprint of a computation pattern to be the cardinality of \( \Pi(c, \Psi^{(n)}) \). The cell footprint is independent of cell, thus we denote it as \( |\Pi(\Psi^{(n)})| \).

Also, we define the cell-domain coverage as a union of the cell coverage of all cells in the domain:

\[
\Pi(\Omega, \Psi^{(n)}) = \bigcup_{c \in \Omega} \Pi(c, \Psi^{(n)}),
\]

which is the set of cells that are needed to do computation of all cells in \( \Omega \). Here and in subsequent discussions of parallel MD, \( \Omega \) denotes a cell domain for each processor. Then, the set
of cells $\omega$ that needs to be imported from other processors are the cells that are in the cell-domain coverage but not in the cell domain:

$$\omega(\Omega, \Psi^{(n)}) = \Pi(\Omega, \Psi^{(n)}) - \Omega.$$  

The import volume of $n$-tuple computation is defined as the size of all cells that need to be imported (i.e. the cardinality of $\omega$):

$$V_n(\Omega, \Psi^{(n)}) = \left| \omega(\Omega, \Psi^{(n)}) \right|. \quad (5.14)$$

The overall import volume $V_{\text{import}}$ is determined by the largest import volume among all $n$:

$$V_{\text{import}} = \max_n(V_n(\Omega, \Psi^{(n)})).$$

### 5.3.1.4 Optimal UCP-MD Problem

Based on the computational and communication cost functions of UCP in Eqs. (5.12) and (5.14), the parallel MD problem is reduced to the following optimization problem:

**Problem (Optimal UCP-MD).** Given a cell domain $\Omega$, find a set of $n$-complete computation patterns $\{\Psi^{(n)}\}$ for all $n \in \{2, \ldots, n_{\text{max}}\}$ such that each $\Psi^{(n)}$ satisfies the following:

1. $\Psi^{(n)} = \arg\min_{\Psi^{(n)}} \left( \sum_{c \in \Omega} \left| S_{\text{cell}}(c, \Psi^{(n)}) \right| \right)$
2. $\Psi^{(n)} = \arg\min_{\Psi^{(n)}} \left( \left| \bigcup_{c \in \Omega} \Pi(c, \Psi^{(n)}) - \Omega \right| \right)$

The first condition minimizes the search cost (i.e. filtering out the unnecessary tuples), while the second condition minimizes the import volume in parallel MD. In the next subsection, we present an SC algorithm to solve the optimal UCP-MD problem. We will show in section 5.4 that the SC algorithm in the case of pair computation is reduced to the best cell methods.
5.3.2 Shift-Collapse Algorithm

We propose an SC algorithm, which makes a complete computation-pattern while addressing the two optimality conditions presented in subsection 5.3.1.4. The SC algorithm consists of three main phases (see Figure 5.7). The first phase, full-shell generation subroutine (GENERATE-FS in Figure 5.8) performs \((n-1)\)-fold nested loops (lines 2-5) to enumerate all possible paths to generate an \(n\)-complete pattern (e.g. for \(n = 3\) in Figure 5.6(a)). This creates a computation pattern centered at the original cell (i.e. \(c(q)\) in Eq. (5.9)) surrounded by \((n-1)\)-layers of nearest-neighbor cells (yellow region in Figure 5.6(a)). In the following, we refer to this computation pattern as *full shell* (FS).

In the second phase, octant-compression shift subroutine (OC-SHIFT) shifts all paths in the FS pattern to the first octant in order to compact the cell footprint \(|\Pi(\Psi^{(0)})|\). To achieve this, all of the paths in FS are shifted toward the upper corner (the highest Cartesian coordinates in all 3 directions, see Figure 5.6(b)) of the FS cell coverage (see lines 3-4 in Figure 5.9).

In the last phase, reflective collapse subroutine (R-COLLAPSE) finds and removes redundant paths that generate the same force set (Figure 5.10). This is achieved by doubly nested loops (lines 2-3) over all pairs of paths to compare their equivalence (line 4). If a path-pair is found to be equivalent, one of the paths is removed from the pattern (line 5). The equivalence between two arbitrary paths \(p\) and \(p'\) is tested as \(\sigma(p') = \sigma(p^{-1})\), which will be proven in Lemma 3 in section 5.3.3.

The most important property for the SC algorithm is that a path-shift operation does not alter the resulting force set, *i.e.*, translational invariance. Let \(p = (v_0,\ldots,v_{n-1})\) be an \(n\)-tuple
computation path, then path shifting is defined as a translation of the origin of the computation path:

\[ p + \Delta = (v_0 + \Delta, \ldots, v_{n-1} + \Delta), \]

where \( \Delta \in \mathbb{Z}^n \) denotes a shifting vector. The following theorem proves the path-shift invariance of force computation.

**THEOREM 1 (PATH-SHIFT INVARIANCE).** Let \( \Omega \) be a cell domain and \( p \) be an \( n \)-tuple computation path. For an arbitrary shift vector \( \Delta \in \mathbb{Z}^n \), \( \text{UCP}(\Omega, \{p\}) = \text{UCP}(\Omega, \{p + \Delta\}). \)

**PROOF.** For a particular cell \( c(q) \in \Omega \) and \( p = \{v_0, \ldots, v_{n-1}\} \), \( S_{\text{cell}}(c(q), \{p + \Delta\}) \) reads

\[
S_{\text{cell}}(c(q), \{p + \Delta\}) = \{(r_0, \ldots, r_{n-1}) | \forall r_k \in c(q + v_k + \Delta)\}
\]

\[
= \{(r_0, \ldots, r_{n-1}) | \forall r_k \in c((q + \Delta) + v_k)\}.
\]

\[
= S_{\text{cell}}(c(q + \Delta), \{p\}). \tag{5.15}
\]

The union of Eq. (5.15) over all cells in the domain yields

\[
\bigcup_{c(q + \Delta) \in \Omega} S_{\text{cell}}(c(q + \Delta), \{p\}) = \bigcup_{c(q') \in \Omega} S_{\text{cell}}(c(q'), \{p\})
\]

\[
= \text{UCP}(\Omega, \{p\}).
\]

The last equality comes from the definition of UCP in Eq. (5.9). Therefore, \( \text{UCP}(\Omega, \{p + \Delta\}) = \text{UCP}(\Omega, \{p\}) \). □
Figure 5.6: 2D schematic of a shift-collapse (SC) computation pattern. (a) FS computation pattern for \( n = 3 \), containing all possible paths of length 3 originated from the center cell (magenta). (b) SC computation pattern after OC-shift and R-collapse subroutines. Yellow cells denote the cell coverage of the center cell, which for SC is much smaller than that for FS.

**Algorithm** Shift-Collapse (SC)

**Input:**

\( n \in \{2,3,\ldots\} \), the length of a tuple

**Output:**

\( \Psi_{SC}^{(n)} \subseteq L^n \): Shift-collapse pattern

**Steps:**

1. \( \Psi_{FS} \leftarrow \text{GENERATE-FS}(n) \)
2. \( \Psi_{OC} \leftarrow \text{OC-SHIFT}(\Psi_{FS}) \)
3. \( \Psi_{SC} \leftarrow \text{COLLAPSE}(\Psi_{OC}) \)

Figure 5.7: Shift-Collapse algorithm.

**Algorithm** Generate full-shell (GENERATE-FS)

**Input:**

\( n \in \{2,3,\ldots\} \), the length of a tuple

**Output:**

\( \Psi_{FS}^{(n)} \subseteq L^n \): Full-shell computation pattern

**Steps:**
1. $\Psi_{FS}^{(n)} = \emptyset$, $v_0 = (0,0,0)$

2. for $\forall v_1 = (\alpha_1^x, \alpha_1^y, \alpha_1^z)$ where $-1 \leq \alpha_1^{x,y,z} \leq 1$

3. for $\forall v_2 = (\alpha_2^x, \alpha_2^y, \alpha_2^z)$ where $\alpha_1^{x,y,z} - 1 \leq \alpha_2^{x,y,z} \leq \alpha_1^{x,y,z} + 1$

4. for $\forall v_{n-1} = (\alpha_{n-1}^x, \alpha_{n-1}^y, \alpha_{n-1}^z)$ where $\alpha_{n-2}^{x,y,z} - 1 \leq \alpha_{n-1}^{x,y,z} \leq \alpha_{n-2}^{x,y,z} + 1$

5. $\Psi_{FS}^{(n)} \leftarrow \Psi_{FS}^{(n)} \cup (v_0, v_{n-1})$

Figure 5.8: FS generation algorithm.

---

**Algorithm** Octant-compression shift (OC-Shift)

**Input:**

$\Psi^{(n)} \subseteq L^n$: computation pattern

**Output:**

$\Psi_{OC}^{(n)} \subseteq L^n$: OC computation pattern

**Steps:**

1. $\Psi_{OC}^{(n)} = \emptyset$

2. for $\forall p = (v_0, ..., v_{n-1}) \in \Psi^{(n)}$

3. $\Delta_{\text{min}} = \min_{0 \leq i < n} (v_i^\beta)$ for all $\beta \in \{x,y,z\}$

4. $p' \leftarrow p - \Delta_{\text{min}}$

5. $\Psi_{OC}^{(n)} \leftarrow \Psi_{OC}^{(n)} \cup \{p'\}$

Figure 5.9: OC-shift algorithm.

---

**Algorithm** Reflective-collapse (R-COLLAPSE)

**Input:**

$\Psi^{(n)} \subseteq L^n$: computation pattern

**Output:**

$\Psi_{RC}^{(n)} \subseteq L^n$: non-redundant computation pattern
Steps:
1. $\Psi^{(n)}_{RC} = \Psi^{(n)}$
2. for $\forall p \in \Psi^{(n)}$
3. for $\forall p' \in \Psi^{(n)}$ where $p' \neq p$
4. if $\sigma(p') = \sigma(p^{-1})$
5. $\Psi^{(n)}_{RC} \leftarrow \Psi^{(n)}_{RC} - \{p\}$

Figure 5.10: Reflective-collapse algorithm.

5.3.3 Correctness of SC Algorithm

To prove the correctness of the SC algorithm, we will show that the computation pattern generated by it is complete as defined in Eq. (5.11). The SC algorithm consists of three phases. We will show that the first phase generates a complete pattern, while the second and third phases do not alter the resulting force set.

First, we prove that the GENERATE-FS routine generates a pattern that constitutes a bounding force set.

**Lemma 1.** Given a computation pattern $\Psi^{(n)}_{FS} = \text{GENERATE-FS}(n)$, the force set $S^{(n)} = UCP(\Omega, \Psi^{(n)}_{FS})$ is a bounding force set for an arbitrary cell domain $\Omega$.

**Proof.** GENERATE-FS(n) in Figure 5.8 constructs $\Psi^{(n)}_{FS}$ as a union of $(v_0, \ldots, v_{n-1})$ such that $v_{k+1} = (v^x_k \pm 1, v^y_k \pm 1, v^z_k \pm 1)$ for all $0 \leq k < n-1$. Thus $c(q+v_k)$ is a nearest neighbor cell of $c(q+v_{k+1})$ for an arbitrary $q \in L$. Consider an arbitrary $n$-tuple $\chi = (r_0, \ldots, r_{n-1}) \in \Gamma^{(n)}$, where $r_{k+1} < r_{\text{cut}-n}$ for all $0 \leq k < n-1$. We will prove that $\chi$ is included in a cell set $(c(q+v_0), c(q+v_1), \ldots, c(q+v_{n-1}))$ using mathematical induction:
Base: $r_0 \in c(q + v_0)$.

Induction: if $r_k \in c(q + v_k)$, then $\exists v_{k+1} : r_{k+1} \in c(q + v_{k+1})$.

Base step is trivial because we choose $q$ such that $r_0 \in c(q+v_0)$. For the induction step, atomic pair $(r_k, r_{k+1})$ with distance $r_{k,k+1} < r_{\text{cut}}$ is guaranteed to reside in cell $c(q+v_k)$ and $c(q+v_{k+1})$ because the cell size is larger than the cutoff $r_{\text{cut}}$. This proves the induction step.

Hence, $\exists v_k : r_k \in c(q + v_k)$ for all $0 \leq k < n-1$ and thus, the entire $n$-tuple $\chi$ is contained in a cell set $(c(q + v_0), c(q + v_1), ..., c(q + v_{n-1}))$. From the definition of $S_{\text{cell}}$ in Eq. (5.10), for an arbitrary $n$-tuple in $\Gamma^{(n)}$ there exists $S_{\text{cell}}(c(q), \Psi^{(n)}_{FS})$. Therefore from the definition of UCP in Eq. (5.9), $\Gamma^{(n)} \subseteq \text{UCP}(\Omega, \Psi^{(n)})$ is satisfied and UCP$(\Omega, \Psi^{(n)})$ is complete. □

Next, we prove that OC-SHIFT operation does not alter the resulting force set.

**Lemma 2.** Consider an arbitrary computation pattern $\Psi^{(n)}$ and a cell domain $\Omega$. For $\Psi^{(n)}_{OC} = \text{OC-SHIFT}(\Psi^{(n)})$, $\text{UCP}(\Omega, \Psi^{(n)}_{OC}) = \text{UCP}(\Omega, \Psi^{(n)})$.

**Proof.** The OC-SHIFT subroutine performs a series of shifting operations to all paths in $\Psi^{(n)}$. From Theorem 1, each shifting operation keeps the force set unchanged. Therefore, $\text{UCP}(\Omega, \Psi^{(n)}_{OC}) = \text{UCP}(\Omega, \Psi^{(n)})$. □

To prove that R-COLLAPSE keeps the force set unchanged, we first prove the equivalence between paths (i.e. line 4 in Figure 5.10) used in R-COLLAPSE.

**Lemma 3 (Reflective Invariance).** Consider a cell domain $\Omega$ and paths $p, p'$ of size $n$. If $\sigma(p^{-1}) = \sigma(p')$, then $\text{UCP}(\Omega, \{p\}) = \text{UCP}(\Omega, \{p'\})$. 

108
Let $p = (v_0, \ldots, v_{n-1})$, $p' = (u_0, \ldots, u_{n-1})$ and the inverse path of $p$ be $p^{-1} = (v_{n-1}, \ldots, v_0)$. From the assumption $\sigma(p^{-1}) = \sigma(p')$, we have

$$u_{k+1} - u_k = v_{n-2-k} - v_{n-1-k}.$$  \hspace{1cm} (5.16)

for all $k \in \{0, \ldots, n-2\}$. From the shift-invariant property in Theorem 1,

$$\bigcup_{\forall c(q) \in \Omega} S_{cell}(c(q), \{p\}') = \bigcup_{\forall c(q) \in \Omega} S_{cell}(c(q), \{p' + \Delta\}).$$

Let $\Delta = v_{n-1} - u_0$, then $S_{cell}(c(q), \{p' + \Delta\})$ reads

$$S_{cell}(c(q), \{p\}') = \{ (r_0, \ldots, r_{n-1}) | \forall r_k \in c(q + u_k + \Delta) \}$$
$$= \{ (r_0, \ldots, r_{n-1}) | \forall r_k \in c(q + v_{n-1} - u_0 + u_k) \}.$$  \hspace{1cm} (5.17)

Using telescoping, $q + v_{n-1} - u_0 + u_k$ in Eq. (5.17) can be rewritten as

$$q + v_{n-1} - u_0 + u_k = q + v_{n-1} + \sum_{i=0}^{k-1} (u_{i+1} - u_i).$$  \hspace{1cm} (5.18)

$u_{i+1} - u_i$ in the R.H.S. of Eq. (5.18) can be replaced by $v_{n-2-i} - v_{n-1-i}$ according to Eq. (5.16), which yields

$$q + v_{n-1} + \sum_{i=0}^{k-1} (u_{i+1} - u_i) = q + v_{n-1} + \sum_{i=0}^{k-1} (v_{n-2-i} - v_{n-1-i})$$
$$= q + v_{n-1-k}$$

Therefore,

$$q + v_{n-1} - u_0 + u_k = q + v_{n-1-k}.$$  \hspace{1cm} (5.19)

Substituting Eq. (5.19) back into Eq. (5.17) yields,
\[ \mathbf{S}_{\text{cell}}(c(q), \{p\}) = \{(r_0, \ldots, r_{n-1}) | \forall r_k \in c(q + v_{n-1-k})\} \]
\[ = \{(r_0, \ldots, r_{n-1}) | \forall r_{n-1-k} \in c(q + v_{k})\}. \]  
\[ = \{(r_{n-1}, \ldots, r_0) | \forall r_k \in c(q + v_{k})\} \]  
\[ = \{(r_{n-1}, \ldots, r_0) | \forall r_{n-1-k} \in c(q + v_{k})\}. \]

Based on the undirectionality of tuples described in section 5.2.1, \((r_0, \ldots, r_{n-1}) = (r_{n-1}, \ldots, r_0)\).

Therefore Eq. (5.20) reads

\[ \mathbf{S}_{\text{cell}}(c(q), \{p\}) = \{(r_0, \ldots, r_{n-1}) | \forall r_k \in c(q + v_{k})\} \]
\[ = \{S_{\text{cell}}(c(q), \{p\})\}. \]

Based on the definition of UCP in Eq. (5.9), this proves \(UCP(\Omega, \{p\}) = UCP(\Omega, \{p\})\).

Using Lemma 3, we can now prove that the R-COLLAPSE subroutine preserves the completeness of a pattern.

**Lemma 4.** Let \(\Omega\) be an arbitrary cell domain and \(\Psi^{(n)}\) be a computation pattern. For \(\Psi^{(n)}_{\text{RC}} = R\text{-COLLAPSE}(\Psi^{(n)})\), \(UCP(\Omega, \Psi^{(n)}_{\text{RC}}) = UCP(\Omega, \Psi^{(n)})\).

**Proof.** In R-COLLAPSE subroutine (Figure 5.10), lines 2-3 loop over all pairs of computation paths \(\{p, p'\}\) in \(\Psi^{(n)}\). For each pair of paths, we remove one of them (i.e. collapsing) only when \(\sigma(p') = \sigma(p^{-1})\) (lines 4-5). In such a case, \(\Psi^{(n)}_{\text{RC}} = \Psi^{(n)} - \{p\}\). Based on Lemma 3, \(p\) and \(p'\) produce the same force set, and thus

\[ UCP(\Omega, \Psi^{(n)}_{\text{RC}}) = UCP(\Omega, \Psi^{(n)} - \{p\}). \]  
\[ = UCP(\Omega, \Psi^{(n)} - \{p\}) \]  
\[ = UCP(\Omega, \Psi^{(n)} - \{p\}). \]  

Substituting \(\Psi^{(n)}_{\text{RC}} = \Psi^{(n)} - \{p\}\) into Eq. (5.21) yields \(UCP(\Omega, \Psi^{(n)}) = UCP(\Omega, \Psi^{(n)}_{\text{RC}})\), which proves the Lemma.  

\(\Box\)
Using Lemmas 1, 2, and 4, we can now prove the correctness of the SC algorithm by proving the completeness of SC pattern.

**Theorem 2.** Given an arbitrary cell domain $\Omega$, a computation pattern $\Psi^{(n)}_{SC}$ generated from the shift-collapse algorithm is $n$-complete.

**Proof.** Lemma 1 states that $\text{GENERATE-FS}$ produces $\Psi^{(n)}_{FS}$, which is an $n$-complete pattern. Lemma 2 guarantees that for $\Psi^{(n)}_{OC} = \text{OC-SHIFT}(\Psi^{(n)}_{FS})$, $\text{UCP}(\Omega, \Psi^{(n)}_{OC}) = \text{UCP}(\Omega, \Psi^{(n)}_{FS})$ and thus $\Psi^{(n)}_{OC}$ is also complete. According to Lemma 4, for $\Psi^{(n)}_{SC} = \text{R-COLLAPSE}(\Psi^{(n)}_{OC})$, $\text{UCP}(\Omega, \Psi^{(n)}_{SC}) = \text{UCP}(\Omega, \Psi^{(n)}_{OC})$. Therefore $\Psi^{(n)}_{SC}$ is complete. □

### 5.4 Theoretical Analysis

In this section, we perform theoretical analysis of the SC algorithm. First, we quantify the cost of $n$-tuple searches of the SC pattern. We then analyze and estimate the import volume of the SC the pattern. Finally, we discuss the relation of SC to previous works in the case of pair computation.

#### 5.4.1 Search-Cost Analysis of SC Pattern

In this subsection, we show that the search cost for the SC pattern is approximately half that of FS. This size reduction arises from $\text{R-COLLAPSE}$, which removes all redundant computation paths using their reflective-invariant property in Lemma 3. First, we show that the size of a search space can be estimated by the cardinality of a computation pattern.

**Lemma 5.** Assume that the atom distribution is uniform. For an arbitrary cell domain $\Omega$ and a computation pattern $\Psi^{(n)}$, the search cost $TUCP$ is proportional to $|\Psi^{(n)}|$. 
P \text{ROOF. Let } \langle \rho_{\text{cell}} \rangle \text{ be the average number of atoms per cell. For all paths } p = (v_0,\ldots,v_{n-1}) \text{ in } \Psi(n) \text{ and an arbitrary cell } c(q) \in \Omega, \text{ the average number of atomic pairs in a cell-pair } c(q+v_k) \text{ and } c(q+v_{k+1}) \text{ is } \langle \rho_{\text{cell}} \rangle^2. \text{ Thus, for an } n-1 \text{ consecutive cell-pairs, the number of tuples in a cell search-space } S_{\text{cell}} \text{ centered at an arbitrary cell } c(q) \text{ is}

\left| S_{\text{cell}}(c(q),\{p\}) \right| = \langle \rho_{\text{cell}} \rangle^{n-1}. \quad (5.22)

By summing Eq. (5.22) over all paths in } \Psi^{(n)}, \text{ we obtain the average number of } n\text{-tuples associated with cell } c(q) \text{ as}

\left| S_{\text{cell}}(c(q),\Psi^{(n)}) \right| = \langle \rho_{\text{cell}} \rangle^{n-1} |\Psi^{(n)}|. \quad (5.23)

Summation of Eq. (5.23) over all cells in } \Omega \text{ yields the search cost in Eq. (5.12):

\sum_{c \in \Omega} \left| S_{\text{cell}}(c,\Psi^{(n)}) \right| = T_{\text{UCP}} = L \langle \rho_{\text{cell}} \rangle^{n-1} |\Psi^{(n)}|. \quad (5.24)

Hence, } T_{\text{UCP}} \propto |\Psi^{(n)}|. \quad \Box

Based on Lemma 5, we can estimate the costs of UCP(\Omega,\Psi^{(n)}_{FS}) \text{ and UCP}(\Omega,\Psi^{(n)}_{SC}) \text{ in terms of } |\Psi^{(n)}_{FS}| \text{ as follows. For each path } p \text{ in } \Psi^{(n)}_{FS}, \text{ there are } n-1 \text{ consecutive cell-vector pairs } (v_k, v_{k+1}) \text{ for } 0 \leq k < n-1. \text{ For a particular cell } q+v_k, \text{ there are 27 nearest-neighbor cells } q+v_{k+1}. \text{ Thus for } n-1 \text{ neighbor-search steps, we have the number of paths as}

\left| \Psi^{(n)}_{FS} \right| = 27^{n-1}. \quad (5.25)

Substituting Eq. (5.25) in Eq. (5.24), the search cost for UCP(\Omega,\Psi^{(n)}_{FS}) \text{ is obtained as}
Next, we analyze the search cost of \( \text{UCP}(\Omega, \Psi_{\text{SC}}^{(n)}) \). Paths in \( \Psi_{\text{SC}}^{(n)} \) are only removed in the R-COLLAPSE routine, where redundant paths are collapsed. To estimate \( |\Psi_{\text{SC}}^{(n)}| \), we need to find how many paths are collapsed. Thus, we first categorize paths in \( \Psi_{\text{FS}}^{(n)} \) into collapsible and non-collapsible paths as

\[
\Psi_{\text{FS}}^{(n)} = \Psi_{\text{collapsible}}^{\text{FS}} \cup \Psi_{\text{non-collapsible}}^{\text{FS}},
\]

where \( \Psi_{\text{collapsible}} \) and \( \Psi_{\text{non-collapsible}} \) are disjoint sets of collapsible and non-collapsible paths, respectively. In the following, we prove that for each path in \( \Psi_{\text{FS}}^{(n)} \), there exists a unique path in \( \Psi_{\text{FS}}^{(n)} \) that produces the same force set.

**Lemma 6 (Reflective Path-twin (RPT)).** For an arbitrary cell domain \( \Omega \) and a path \( p =(v_0, ..., v_{n-1}) \in \Psi_{\text{FS}}^{(n)} \), there exists a unique reflective path-twin \( \text{RPT}(p) = p^{-1} - v_{n-1} \in \Psi_{\text{FS}}^{(n)} \), such that \( \sigma(p^{-1}) = \sigma(\text{RPT}(p)) \).

**Proof.** For \( p = (v_0, ..., v_{n-1}) \), we have \( p^{-1} = (v_{n-1}, ..., v_0) \). From the definition of UCP (Eq. (5.9)) for the case of a single-path pattern \( \{p\} \) and the undirectionality of n-tuple, it is straightforward to prove that

\[
\text{UCP}(\Omega, \{p\}) = \text{UCP}(\Omega, \{p^{-1}\}).
\]

For a particular path \( p' = p^{-1} - v_{n-1} \), we have
\[
p' = p^{-1} - v_{n-1}
= (v_{n-1} - v_{n-1}, v_{n-2} - v_{n-1},..., v_0 - v_{n-1})
= (0, v_{n-2} - v_{n-1},..., v_0 - v_{n-1})
\]

Since the first cell offset of \(p\) is \((0,0,0)\), we have \(p' \in \Psi^{(a)}_{FS}\) from the generation of \(\Psi^{(a)}_{FS}\) (line 1 in Figure 5.9). Thus, for every path \(p \in \Psi^{(a)}_{FS}\), there exists a path-twin \(RPT(p) = p' = p^{-1} - v_{n-1} \in \Psi^{(a)}_{FS}\).

The a differential representation of \(p'\) is
\[
\sigma(p') = (v_{n-2} - v_{n-1} - 0,...,(v_0 - v_{n-1}) - (v_1 - v_{n-1}))
= (v_{n-2} - v_{n-1},..., v_0 - v_1),
\]
while that of \(p^{-1}\) is
\[
\sigma(p^{-1}) = (v_{n-2} - v_{n-1},..., v_0 - v_1).
\]

Hence \(\sigma(p') = \sigma(p^{-1})\). From Lemma 3, this proves the equivalence of \(p\) and its twin \(RPT(p)\). Next we consider the uniqueness of \(RPT(p)\). Since all paths in a computation pattern begin with the same cell, they are all inequivalent if the path is directional. The only source of equivalence therefore arises from the undirectionality of paths, which is a direct consequence of the undirectionality of \(n\)-tuples stated in section 5.2.1. The reflective equivalence is two-fold by definition, and it can produce at most one redundant path. □

From Lemmas 3 and 6, there exists a unique path twin that produces the identical force set for every path \(p\). However, if \(p = p^{-1}\), then \(RPT(p)\) is \(p\) itself. Thus, it is not collapsible.

**Corollary 1 (Self-reflection).** For an arbitrary cell domain \(\Omega\) and a path \(p \in \Psi^{(a)}_{FS}\), if \(p = p^{-1}\), then \(RPT(p) = p\).
PROOF. Let \( p \) be \((v_0, \ldots, v_{n-1})\). From Lemma 6, \( \text{RPT}(p) = p^{-1} - v_{n-1} \). From the assumption, \( p = p^{-1} \), thus \( \text{RPT}(p) = p - v_{n-1} \). Also, \( p = p^{-1} \) implies \( v_0 = v_{n-1} \). Since \( p \in \Psi^{(n)}_{\text{FS}}, v_0 = 0 \). Hence, 
\[ \text{RPT}(p) = p - v_{n-1} = p - v_0 = p. \]
\( \square \)

According to the self-reflection in Corollary 1, \( \psi_{\text{non-collapsible}} = \{ p \mid \forall p \in \Psi^{(n)}_{\text{FS}}: p = p^{-1} \} \).

To estimate \( |\psi_{\text{non-collapsible}}| \), we need to count the number of paths that are self-reflective. Note that
\[
p = p^{-1} = \begin{cases} (v_0, \ldots, v_{(n-1)/2}, v_{(n-1)/2}, v_{(n-1)/2+1}, \ldots, v_0) & ; n \text{ is odd} \\ (v_0, \ldots, v_{n/2}, v_{n/2}, \ldots, v_0) & ; n \text{ is even} \end{cases}
\]
thus, the number of self-reflective (i.e. non-collapsible) paths is
\[
|\psi_{\text{non-collapsible}}| = 27^{\left\lfloor \frac{n+1}{2} \right\rfloor - 1}. \tag{5.27}
\]

Substituting Eq. (5.27) in Eq. (5.26) yields
\[
|\psi_{\text{collapsible}}| = |\Psi^{(n)}_{\text{FS}}| - |\psi_{\text{non-collapsible}}| \\
= 27^{n-1} - 27^{\left\lfloor \frac{n+1}{2} \right\rfloor - 1} \tag{5.28}
\]

Lemma 6 states that there is a unique reflective path-twin of \( p \) for each collapsible path \( p \) such that one of them can be removed. Thus, the number of collapsed paths in SC pattern is half of the collapsible paths in Eq. (5.28):
\[
|\Psi^{(n)}_{\text{SC}}| = \frac{1}{2} |\psi_{\text{collapsible}}| + |\psi_{\text{non-collapsible}}| \\
= \frac{1}{2} \left( 27^{n-1} - 27^{\left\lfloor \frac{n+1}{2} \right\rfloor - 1} \right) + 27^{\left\lfloor \frac{n+1}{2} \right\rfloor - 1} \tag{5.29}
\]
Hence, the search cost of SC is roughly half that of FS for large \( n \).


5.4.2 Communication Analysis of SC Pattern

In this subsection, we analyze the communication cost of the parallel SC-MD. The communication cost is defined as

\[ T_{\text{comm}} = T_{\text{bandwidth}} + T_{\text{latency}}. \quad (5.30) \]

Here, we assume that the bandwidth cost \( T_{\text{bandwidth}} \) is proportional to the amount of imported data. The imported data in parallel MD is proportional to the imported volume \( V_{\text{import}} \) in subsection 5.3.1.3 in the average case. On the other hand, the latency cost \( T_{\text{latency}} \) is proportional to the number of nodes \( n_{\text{comm\_node}} \) from which data needs to be imported. Hence, Eq. (5.30) can be rewritten as

\[ T_{\text{comm}} = c_{\text{bandwidth}} V_{\text{import}} + c_{\text{latency}} n_{\text{comm\_nodes}}, \quad (5.31) \]

where \( c_{\text{bandwidth}} \) and \( c_{\text{latency}} \) are prefactors. To quantify \( T_{\text{comm}} \) for SC-MD, we first determine the import volume of the SC pattern. For simplicity, we assume that the given cell domain has a cubic shape such that \( L_x = L_y = L_z = l \).

From the definition of import volume in Eq. (5.14), we need to find the union of cell coverage of all cells in the domain. To analyze the cell coverage, we define a layered cell coverage in the range \([a, b]\) (where \( a \) and \( b \) are nonnegative integers such that \( a \leq b \)) as

\[ c[a, b]_q = \{ c(q + \delta) | \forall \delta \in \{a, a+1, ..., b\} \text{ for all } \beta \in \{x, y, z\} \}. \]

In the SC algorithm, the key step that affects cell coverage is the OC-SHIFT subroutine. OC-SHIFT performs 1st-octant compression, in which all computation paths in the pattern are shifted to the first octant of cell coverage (i.e. non-negative cell indices along \( x, y, \) and \( z \))
directions relative to the center cell). Thus, the cell coverage \( \Pi(c(q), \Psi_{SC}^{(n)}) \) for an arbitrary cell \( c(q) \) is

\[
\Pi(c(q), \Psi_{SC}^{(n)}) = c[0, n-1]_q .
\]

From Eq. (5.13), the cell-domain coverage of the SC pattern is

\[
\Pi(\Omega, \Psi_{SC}^{(n)}) = \bigcup_{c(q) \in \Omega} c[0, n-1]_q .
\]

Hence, the import volume of the SC pattern reads

\[
V_{\omega}(\Omega, \Psi_{SC}^{(n)}) = \left| \bigcup_{c(q) \in \Omega} c[0, n-1]_q - \Omega \right| . \tag{5.32}
\]

Since \( \Omega \subseteq \Pi(\Omega, \Psi_{SC}^{(n)}) \), the cardinality in the R.H.S. of Eq. (5.32) can be taken individually. Hence the import volume of \( \Psi_{SC}^{(n)} \) is

\[
V_{\omega}(\Omega, \Psi_{SC}^{(n)}) = \left| \bigcup_{c(q) \in \Omega} c[0, n-1]_q - \Omega \right| = (l + n - 1)^3 - l^3 \tag{5.33}
\]

Regarding the latency cost, unlike some methods such as NT [74] that minimize the bandwidth cost at the expense of latency cost, all cell-based methods have small constant latency. In SC-MD, we only need to import atom data from 7 nearest processors using only 3 communication steps via forwarded atom-data routing.

### 5.4.3 Relation to Previous Works

Numerous algorithms have been designed to utilize the cell data structure for efficient range-limited \( n \)-tuple search. For the case of pair computation \( (n = 2) \), in particular, shell-based
methods—such as full-shell (FS), half-shell (HS), and eighth-shell (ES)—are used extensively for efficient search of pairs in $\Gamma^{(2)}$. These methods can be described systematically in terms of UCP. In the following subsections, we provide their unified descriptions and compare them with SC.

## 5.4.3.1 Full-shell method

FS is the simplest among shell methods and it produces a bounding force set. FS searches all atoms within the nearest-neighbor cells (i.e. cells of indices within $\pm 1$ in the $x$, $y$, $z$ directions) surrounding the center cell [69]. In our algebraic notation, FS is equivalent to $\Psi_{FS}^{(2)}$, and thus is 2-complete according to Lemma 1. However, FS is not optimal in the light of the search cost: $|\Psi_{FS}^{(2)}| = 27$ (see Figure 5.11(a)). Since FS has reflective redundancy, the extra computational cost is involved for filtering the collapsible pairs.

## 5.4.3.2 Half-shell method

HS reduces the search cost of FS using the symmetric property of reflected pairs to eliminate redundant search (see Figure 5.11(b)) [69]. This amounts to $\Psi_{HS} = R$-COLLAPSE($\Psi_{FS}^{(2)}$). Thus, the search cost and import volume are reduced by nearly half compared to that of FS, $i.e.$, $|\Psi_{HS}| = 14$.

## 5.4.3.3 Eighth-shell method

ES improves over HS by relaxing the owner-compute rule, thereby interacting only with the neighbor cells in the upper-corner octant [11]. This amounts to $\Psi_{ES} = OC$-SHIFT($\Psi_{HS}$) = $\Psi_{SC}^{(2)}$. Hence, ES is a special case of the SC algorithm for $n = 2$. OC-shift reduces the cell
footprint of ES to $|\Pi(\Psi_{ES})| = 7$, see Figure 5.11 (c). This in turn reduces the import volume for parallel computation, which is Eq. (5.33) for $n = 2$.

Figure 5.11: Schematic of shell methods for pair computation: (a) full-shell, (b) half-shell, and (c) eighth-shell patterns.

### 5.5 Performance Benchmark

In this section, we benchmark the performance of SC-MD and two existing $n$-tuple computation codes—FS-MD and Hybrid-MD—for a real many-body MD application. Specifically, we consider MD simulation of silica (SiO$_2$), which involves dynamic pair and triplet ($n = 2$ and 3) computations [56]. In this application, $r_{\text{cut-3}}$ is smaller than $r_{\text{cut-2}}$, i.e., $r_{\text{cut-3}}/r_{\text{cut-2}} \sim 0.47$. FS-MD uses a computation pattern generated from GENERATE-FS in Figure 5.8 without further performing OC-shift and R-collapse. Hybrid-MD is a production code presented in Ref. [12]. In this code, the pair computation is done by constructing a dynamic pair list (called Verlet neighbor list) within UCP for $\Psi_{FS}^{(2)}$. Then, Hybrid-MD exploits the shorter cutoff of the triplet computation by pruning the triplet search directly from the pair list without using the cell data structure with $r_{\text{cut-3}}$. Although this hybrid cell/Verlet-neighbor-list approach reduces the triplet search cost in this particular situation, the import volume is not reduced from that of FS-MD.

Performance evaluations in this section are performed on two platforms: BlueGene/Q at Argonne National Laboratory and an Intel-Xeon cluster at the Center for High Performance
Computing and Communication of the University of Southern California (USC-HPCC) [99]. The BlueGene/Q consists of 49,152 compute nodes, where each node has 16 PowerPC A2 cores. Each core supports 4 hardware threads, which is clocked at 1.6 GHz. The network topology of BlueGene/Q is a 5D torus. Four MPI tasks are spawn on each core of BlueGene/Q to fully take advantage of integer pipeline and cache-latency hiding in the BlueGene/Q architecture. Details of the BlueGene/Q architecture can be found in Ref. [29]. Tests on the USC-HPCC cluster are performed on dual 6-core processor nodes with 2.33 GHz Intel Xeon X5650 processors and 48 GB memory per node (see Table 5.1).

Table 5.1: Specifications of testing platforms.

<table>
<thead>
<tr>
<th></th>
<th>ANL-Vesta</th>
<th>USC-HPCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>IBM PowerPC A2</td>
<td>Intel Xeon X5650</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>1.6 GHz</td>
<td>2.66 GHz</td>
</tr>
<tr>
<td>Number of nodes</td>
<td>36,864</td>
<td>256</td>
</tr>
<tr>
<td>Cores per node</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Memory per node</td>
<td>32 GB</td>
<td>24 GB</td>
</tr>
<tr>
<td>Network</td>
<td>3D torus</td>
<td>Myrinet 10G</td>
</tr>
</tbody>
</table>

5.5.1 Search Cost of SC and FS Algorithms

According to the analysis in section 5.4.1, the search cost of SC is much smaller than that of FS (asymptotically half for large $n$). To confirm this assertion, we measure the actual number
of \( n \)-tuples in the force set for SC-MD and FS-MD. Figure 5.12 shows the measured number of triplets per MD step (averaged over 10,000 time steps) as a function of the number of cells, where the average cell density \( \langle \rho_{\text{cell}} \rangle \) is fixed for each measurement. The plot shows that the triplet count of FS-MD is \( \sim 2.13 \) times that of SC-MD.

![Figure 5.12](image.png)

Figure 5.12: Average number of triplets as a function of domain size. Error bars are too small to be visible in the plot.

### 5.5.2 Fine-Grain Parallelism

We compare the runtime of SC-MD with those of FS-MD and Hybrid-MD to characterize the performance as a function of granularity (\( i.e. \) the number atoms per core, \( N/P \)) on 48-64 nodes for small grains (\( N/P = 24 – 3,000 \)).

Figure 5.13(a) shows an average runtime over 10,000 MD steps of the three codes on 48 nodes of the Intel-Xeon cluster. The average grain size is varied from 24 to 3,000 atoms per core. At the smallest grain (\( N/P = 24 \)), the runtime per MD step of SC-MD is much shorter than those
of FS-MD and Hybrid-MD—by factors of 10.5 and 9.7, respectively. This is mainly due to the small import volume of SC-MD. Accordingly, SC-MD is faster than FS-MD for all granularities. However, the Hybrid-MD performance relative to that of SC-MD improves gradually as the granularity increases. This can be understood as follow. While SC-MD reduces the import volume as compared to Hybrid MD, Hybrid-MD reduces the triplet search cost by taking advantage of the special cutoff condition. For larger granularities (or larger computation/communication ratios), the advantage of smaller import-volume for SC-MD becomes overshadowed by its larger triplet search cost, and hence Hybrid MD becomes more advantageous. The crossover of performance advantage from SC-MD to Hybrid-MD occurs at the granularity of 2,095 atoms per core.

Figure 5.13(b) shows an average runtime over 10,000 MD steps of the three codes on 64 nodes of BlueGene/Q. In accordance with the result on the Intel Xeon platform, the finest-grain result of SC-MD shows 5.7- and 5.1-fold speedups over FS-MD and Hybrid-MD, respectively. In this test, the crossover of performance advantage from SC-MD to Hybrid MD on BlueGene/Q is found to be at $N/P = 425$, which is considerably smaller than that on Intel Xeon. This is likely due to the lower computational power per core of BlueGene/Q compared with that of Intel Xeon. Consequently, the benefit of smaller triplet search space for Hybrid-MD is emphasized to shift down the trade-off point between the search cost and import-volume size.
Figure 5.13: Runtime of SC-MD (red), FS-MD (green), and Hybrid-MD (blue) as a function of the granularity on (a) 48 Intel Xeon nodes and (b) 64 BlueGene/Q nodes. The plot shows that SC-MD is the fastest for $N/P < 2,095$ and 425 on Intel Xeon and BlueGene/Q platforms, respectively.
5.5.3 Strong-Scaling Benchmark

In this subsection, we perform a strong-scaling benchmark of SC-MD, FS-MD, and Hybrid-MD on 1–512 BlueGene/Q nodes (i.e. 16–8,192 cores) and on 1–64 Intel Xeon nodes (i.e. 12–768 cores). Strong-scaling speedup is defined as

\[ S_{\text{strong}} = \frac{T_{\text{reference}}}{T_{\text{parallel}}}, \]

where \( T_{\text{reference}} \) denotes the time spent on a reference test (e.g. sequential or small parallel runs) and \( T_{\text{parallel}} \) is the time spent on a larger parallel run to solve the same problem. The corresponding parallel efficiency is \( \eta_{\text{strong}} = S_{\text{strong}} \left( \frac{P_{\text{parallel}}}{P_{\text{reference}}} \right) \), where \( P_{\text{parallel}} \) and \( P_{\text{reference}} \) are the numbers of cores used in the parallel and reference runs, respectively. The total number of atoms used in the benchmark is fixed at 0.88 and 0.79 millions atoms on Intel-Xeon and BlueGene/Q platforms, respectively. Atoms in both systems are uniformly distributed.

Figure 5.14(a) shows the strong-scaling speedup of SC-MD, FS-MD, and Hybrid-MD as a function of the number of cores on the Intel-Xeon cluster. The plot shows that SC-MD achieves excellent scalability from 1–64 nodes with a 59.3-fold speedup (or 92.6% parallel efficiency) on 768 Intel Xeon cores. On the other hand, the scalability of FS-MD and Hybrid-MD decline after the number of cores exceeds 96, so that \( S_{\text{strong}} \) becomes 24.5 and 17.1 on 768 cores, respectively. The corresponding parallel efficiencies are 38.3% and 26.8%. The number of cores and system size used in this benchmark are in an affordable range for general scientists using commodity clusters.

The second benchmark measures strong scalabilities of the three codes on BlueGene/Q. Similarly to the previous benchmark, the results in Figure 5.14(b) indicate that SC-MD maintains
excellent strong scalability with $S_{\text{strong}} = 465.6$ (or $\eta_{\text{strong}} = 90.9\%$) on 8,192 cores. This demonstrates an excellent strong scalability of SC-MD on larger platforms. On the other hand, FS-MD and Hybrid-MD maintain decent parallel efficiency only on small numbers of nodes, where 7.1- and 7.0-fold speedup is observed on 8 nodes. After that, their scalabilities decrease gradually, and only 55.1- and 95.2-fold speedups (10.8\% and 18.6\% efficiencies) are observed on 8,192 cores ($N/P \sim 100$ or $\sim 26$ atoms per MPI task).

![Figure 5.14: Strong scaling speedup of SC-MD, Hybrid-MD, and FS-MD on (a) Intel Xeon cluster and (b) BlueGene/Q.](image)

5.6 Conclusion

We have developed a computation-pattern algebraic framework to formalize dynamic $n$-tuple computation in many-body MD. This new formalism allows us to perform systematically and mathematically proven analysis of dynamic range-limited $n$-tuple MD, which to the best of our knowledge, has not been done before. The new formulation and analysis have led to the development of the SC algorithm, which generalizes some of the best pair ($n = 2$) computation algorithms to an arbitrary $n$. Benchmark tests have shown that SC-MD outperforms our
production Hybrid-MD code for fine granularities. Excellent strong scalability has also been observed for SC-MD. The results demonstrate the advantage of SC-MD over Hybrid-MD when the time-to-solution (rather than simulating the largest possible system size) is the major goal.

There is an additional benefit of SC-MD compared with Hybrid-MD that combines cell and Verlet-neighbor-list methods. Namely, SC exposes maximal concurrency on heterogeneous architectures such as GPU-accelerated and many-core clusters. Since SC executes different \( n \)-tuple computations independently, they can be assigned to different hardware (e.g. multiples GPUs). In contrast, Hybrid-MD has a sequential dependence, i.e., the Verlet-neighbor list must be constructed within the pair computation before any \( n > 2 \) computation can be performed. Another issue is the cell size. Though we have restricted ourselves to the cell size larger than \( r_{\text{cut-}}^n \) for simplicity, it is straightforward to generalize the SC algorithm to a cell size less than \( r_{\text{cut-}}^n \) as was done, e.g., in the midpoint method [100]. In this case, the SC algorithm improves the midpoint method by further eliminating redundant searches. Relative advantages between ES and midpoint methods have been thoroughly discussed by Hess et al. [12].
In this chapter, we present an analysis of a computation cost model for MD simulations within a single node. The analysis and the developed model pave a way to provide understanding about computational behavior of MD, thereby benefiting the development of prediction models for hybrid algorithms.

We develop a performance prediction model for short-range interaction computations in MD simulations, thereby predicting the optimal cell dimension in a linked-list cell method. The model expresses computation time in terms of the number and unit computation time of key operations. The model accurately estimates the number of operations during the simulations with the maximum standard error of 10.6% compared with actual measurements. Then, the unit computation times of the operations are obtained by bisquare regression. Analysis of this model reveals that the optimal cell dimension to minimize the computation time is determined by a trade-off between decreasing search space and increasing linked-list cell access for smaller cells.
The model predicts the optimal cell dimension correctly for 80% of all tested cases, resulting in an average speedup of 10% and 52% for the cutoff radius of interaction, 6.6 and 10.0 Å, respectively.

This chapter is organized as follows. Section 6.1 describes the problem, approach, and opportunity for optimization. Section 6.2 presents the linked-list cell MD and reduced linked-list cell method, to which the proposed performance model is applied. Section 6.3 describes the theoretical analysis of simulation parameters related to the cell dimension selection. Section 6.4 presents the verification of the performance model including the fitting procedure of each estimated parameters. Section 6.5 evaluates the model against measured computation times. Conclusions are drawn in section 6.6.

6.1 Introduction

Large-scale MD simulations involving multibillion atoms are beginning to address broad problems. One of the widely used method for improving the performance and scalability of such large-scale MD simulations is the linked-list cell method, which employs a divide-and-conquer strategy to reduce the search space for atomic pairs within the cutoff radius $R_c$ of the interatomic interaction.

The conventional cell decomposition method uses cubic cell geometry with the dimension $\sim R_c$. However, this incurs considerable amount of unnecessary pair evaluations due to the excessive cell dimension. A commonly used method to decrease the redundant calculations is reducing the cell dimension, see e.g. Mattson et al. [101]. In particular, Yao et al. stated that the cell dimension of $R_c/2$ usually gives the best performance [102]. Our own experience, however, indicates that the optimal cell dimension varies considerably for different parameters
such as $R_c$ and the number of atoms. It is thus of great interest to systematically study and identify the factors that determine the optimal cell dimension.

To address this problem, we carry out a theoretical analysis to estimate the amount of non-bonded interaction computations, which dominate the computational time. Then, the platform-dependent computational cost associated with each key operation is determined by regression. Finally, we use the model to predict the performance as a function of the cell dimension, thus obtain the optimal cell dimension to minimize the computation time.

### 6.2 Cell-Base Molecular Dynamics Simulations

In this chapter, the dominant computation of MD simulation is the evaluation of $E(r^N)$, which consists of two-body $E_2(r_i,r_j)$ and three-body $E_3(r_i,r_j,r_k)$ terms in the program considered here [6].

![2D schematic of the linked-list cell method](image)

Figure 6.1: 2D schematic of the linked-list cell method. (a) Illustration of conventional linked-list cell method with cell dimension $R_s \geq R_c$. (b) 2D schematic of the reduced linked-list cell method. Atoms in the shaded region will be excluded from the force computation of center atom $i$. Only forces exerted by atoms within the cutoff radius (represented by a two-headed arrow) are computed.
Figure 6.1(a) shows a schematic of the computational kernel of MD, which employs a linked-list cell method to compute interatomic interactions in $O(N)$ time. 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. By traversing the linked list, one retrieves the information of all atoms belonging to a cell, and thereby computes interatomic interactions. Thus, the cutoff radius $R_c$ of the interatomic interaction usually determines the dimensions of the cells.

In order to explain the computational procedures in the linked-list cell method, let us define $S$, which is a set of all atomic pairs, $(i,j)$, within the nearest-neighbor cells. The first phase of the computation, i.e. pair search, constructs a subset, $S_0$, of pairs $(i, j)$ whose distance $r_{ij}$ satisfies $r_{ij} \leq R_c$. Subsequently, second phase computes the forces between all atom pairs within $S_0$. Figure 6.2 shows the pseudocode for pair search and force computation.

<table>
<thead>
<tr>
<th>Phase 1: Pair search</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0 = \emptyset$</td>
</tr>
<tr>
<td>for $\forall (i,j) \in S$ do</td>
</tr>
<tr>
<td>if $r_{ij} \leq R_c$ do</td>
</tr>
<tr>
<td>$S_0 = S_0 \cup (i,j)$</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Phase 2: Force computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $\forall (i,j) \in S_0$ do</td>
</tr>
<tr>
<td>compute interaction force $\vec{f}<em>{ij}(\vec{r}</em>{ij})$</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

Figure 6.2: Force computation algorithm.
The reduced linked-list cell method attempts to reduce the search space $S$ by decreasing the dimension $R_s$ of the linked-list cell. To illustrate this, Fig. 6.1(b) shows the situation where $R_s$ is reduced by half. The atoms in the shaded region are at least one cutoff distance away from the center atom $i$ such that it can be safely excluded. Hence, less number of pairs needs to be evaluated, potentially resulting in the performance increase. However, this performance gain could be offset by the increased number of linked-list cells that need to be scanned (e.g., from 9 to 25 cells in the 2D example) and the associated computational overhead for processing the increased number of linked lists.

The MD program used in this chapter is parallelized based on spatial decomposition and message passing [6]. However, the focus of this chapter is performance optimization within each compute node, and thus the number of atoms $N$ hereafter denotes the number of atoms per node.

### 6.3 Performance Analysis

Computing non-bonded interactions usually consumes the largest portion of time in MD simulation [23]. In our performance model, we assume that the computational cost consists of three terms: (1) pair search where the number of search operations $P_s = |S|$, i.e. the cardinality of set $S$ (phase 1 in ); (2) force computation for which the number of force computation operations $P_c = |S_0|$ (phase 2 in ); and (3) linked-list cells access overhead—$P_l$. Therefore, the non-bonded computation time in one MD step—$T$, is

$$T = T_s P_s + T_c P_c + T_m P_l,$$

(6.1)

where $T_s$, $T_c$, and $T_m$ are the operation cost per $P_s$, $P_c$, and $P_l$, respectively.
Generally, the memory access cost (the third term in Eq. (6.1)) in the conventional cell decomposition method is ignored when compared to the pair-search and force-computation costs. However, in the reduced linked-list cell method, this overhead often becomes dominant.

In section 6.3.1, we derive $P_s$, $P_c$, and $P_l$ for the conventional cell-decomposition method, while the extension of the model to the reduced cell method is derived in section 6.3.2. The operation costs $T_s$, $T_c$, and $T_m$ are obtained from regression described in section 6.3.3.

### 6.3.1 Short-Range Interaction Cost Model of the Conventional Linked-List Cell Method

#### 6.3.1.1 Pair search cost model

The search space for short-range force computations is defined as the number of pairs that need to be evaluated in phase 1 of . For $N$ atoms, the search space of an all-pair method is $O(N^2)$. In this section, we derive the approximation of the $O(N)$ search space in the linked-list cell method.

For the uniform cell decomposition method with cubic cell dimension for all cells, the volume of each cell is

$$V_c = R_s^3,$$  \hspace{1cm} (6.2)

where $R_s$ is the dimension of the linked-list cell such that $R_s \geq R_c$. If the number density of atoms in the system is $\rho$ atom/$\text{Å}^3$, the average number of atoms in one linked-list cell is

$$N_c = \rho R_s^3.$$  \hspace{1cm} (6.3)

The number of possible short-range pairs between atoms in two cells is then
\[ P_{cc} = \frac{1}{2} N_c (N_c - 1) = \frac{1}{2} N_c^2 = \frac{1}{2} \rho^2 R_s^6. \]  \hfill (6.4)

Let \( C_{ijk} \) be a linked-list cell with cell indices \( i, j, \) and \( k \) in the \( x, y, \) and \( z \) directions, respectively. To compute forces exerted to all atoms in \( C_{ijk} \), all atom-pair combinations between \( C_{ijk} \) and its nearest neighbor cells \( C'_{ijk} \) need to be evaluated, where

\[ S_{neighbor}(C_{ijk}) = \{C'_{ijk} | (i-1 \leq i' \leq i+1) \land (j-1 \leq j' \leq j+1) \land (k-1 \leq k' \leq k+1) \}, \hfill (6.5) \]

is the set of neighbor cells. Since the conventional cell decomposition method evaluates all atomic pairs between \( C_{ijk} \) and its 26+1 neighbor cells (including itself), the number of pairs to be evaluated per one linked-list cell is

\[ P_s(C) = \frac{1}{2} \rho^2 R_s^6 N_{\text{neighbor}}, \hfill (6.6) \]

where \( N_{\text{neighbor}} = |S_{\text{neighbor}}| = 27. \) Let \( L_x, L_y, \) and \( L_z \) be the number of cells in the \( x, y, \) and \( z \) directions, respectively, then the total number of atom pairs to be evaluated in one MD step is

\[ P_s = \sum_{i=1}^{L_x} \sum_{j=1}^{L_y} \sum_{k=1}^{L_z} P_s(C_{ijk}) = \frac{1}{2} \rho^2 R_s^6 L_x L_y L_z N_{\text{neighbor}}. \hfill (6.7) \]

### 6.3.1.2 Force computation cost model

After the interaction pairs have been evaluated, only the pairs that satisfy the evaluation criterion (\( i.e., r_{ij} \leq R_c \)) undergo force computation. The number of force computations can be estimated as follows. First, we count the number of atoms that are located inside the sphere with radius \( R_c \) around each atom. The volume of such sphere is
\[ V_{fc} = \frac{4}{3} \pi R_c^3, \quad (6.8) \]

and the number of atoms that reside within \( V_{fc} \) is

\[ N_{fc} = \frac{4}{3} \pi \rho R_c^3. \quad (6.9) \]

Let \( N \) denote the number of atoms in the system. From Eq. (6.3), \( N \) can be expressed as

\[ N = \rho R_s^3 L_x L_y L_z. \quad (6.10) \]

Combining Eqs. (6.9) and (6.10), the number of force computation operations—\( P_c \) is

\[ P_c = N_{fc} N = \frac{4}{3} \pi \rho^3 R_s^3 R_c^3 L_x L_y L_z. \quad (6.11) \]

The number of pair-force computations in Eq. (6.11) can be halved by applying Newton’s 3rd law (i.e., the forces acting on a pair of atoms are equal and in the opposite directions) such that

\[ P_c = \frac{2}{3} \pi \rho^3 R_s^3 R_c^3 L_x L_y L_z. \quad (6.12) \]

### 6.3.1.3 Linked-list cell access cost model

Linked-list cell access cost refers to the overhead introduced by the cell decomposition scheme. To compute the forces on atoms in cell \( C \), \( N_{\text{neighbor}} \) cells need to be traversed. Therefore, the number of linked-list cell accesses is

\[ P_l = L_x L_y L_z N_{\text{neighbor}}. \quad (6.13) \]
Combining Eqs. (6.7), (6.12) and (6.13), the prediction model for short-range pair computation is given by

\[
T = T_s \left( \frac{1}{2} \rho^2 R^6_s L_x L_y L_z N_{\text{neighbor}} \right) \\
+ T_c \left( \frac{2}{3} \pi \rho^2 R^3_s R^3_s L_x L_y L_z \right) \\
+ T_m \left( L_x L_y L_z N_{\text{neighbor}} \right)
\]  
(6.14)

In the next subsection, we derive a corresponding model for the reduced cell method to be used to obtain the optimal cell dimension.

6.3.2 Short-Range Interaction Cost Model of the Reduced Linked-list Cell Method

6.3.2.1 Reduced-cell cost model

In this subsection, we derive the number of searched pairs \( P'_s \), that of pair-force computations \( P'_c \), and that of linked-list accesses \( P'_l \), when the cell dimension \( R_s \) is reduced to

\[
R'_s = \frac{R_s}{\mu},
\]  
(6.15)

where \( \mu \in \mathbb{N} \) is a cell dimension reduction factor. The cell dimension reduction also modifies the number of linked-list cells such that

\[
L'_x = \mu L_x \\
L'_y = \mu L_y \\
L'_z = \mu L_z
\]  
(6.16)
In the reduced cell method, the 1\textsuperscript{st}-nearest neighbor cells are not sufficient for interaction evaluations, and instead up to the $\mu$\textsuperscript{th}-nearest neighbor cells need to be evaluated. Therefore, $N_{\text{neighbor}}$ can be calculated as

$$N_{\text{neighbor}} = (2\mu + 1)^3.$$ \hfill (6.17)

Substituting $R_s'$, $L_x'$, $L_y'$, $L_z'$, and $N_{\text{neighbor}}$ to Eqs. (6.7), (6.12), and (6.13), respectively, yield

$$P_s' = \frac{1}{2} \rho^2 R_s'^6 L_x'L_y'L_z' N_{\text{neighbor}}$$

$$= \frac{1}{2} \rho^2 \left(\frac{R_s}{\mu}\right)^6 \mu^3 L_x L_y L_z (2\mu + 1)^3,$$ \hfill (6.18)

$$= \frac{(2\mu + 1)^3}{2\mu^3} \rho^2 R_s'^6 L_x'L_y'L_z'$$

$$P_c' = \frac{2}{3} \pi \rho^2 R_s'^3 R_c'^3 L_x'L_y'L_z'$$

$$= \frac{2}{3} \pi \rho^2 \left(\frac{R_s}{\mu}\right)^3 R_c'^3 \mu^3 L_x L_y L_z,$$ \hfill (6.19)

$$= \frac{2}{3} \pi \rho^2 R_s'^3 R_c'^3 L_x L_y L_z = P_c$$

$$P_l' = L_x'L_y'L_z' N_{\text{neighbor}}$$

$$= \mu^3 (2\mu + 1)^3 L_x L_y L_z.$$ \hfill (6.20)

Equation (6.19) indicates that $P_c'$ is independent of the reduction factor $\mu$. This is consistent with the requirement that the choice of cell dimension should not affect the simulation results as noted by Heinz \textit{et al.} \cite{103}. 

136
6.3.2.2 Asymptotic analysis of the number of key operations

In this section, we analyze the relation of the reduction factor $\mu$ to $P_s'$, $P_c'$, and $P_l'$. This can be done by the expansion of Eqs. (6.18) and (6.20) in the power of $\mu$:

$$P_s' = \left(\frac{1}{2} \mu^{-3} + 3 \mu^{-2} + 6 \mu^{-1} + 4\right) \rho^2 R_s L_s L_y L_z,$$

(6.21)

$$P_l' = \left(8 \mu^6 + 12 \mu^5 + 6 \mu^4 + \mu^3\right) L_x L_y L_z,$$

(6.22)

while $P_c'$ is independent of $\mu$. Equations (6.21) and (6.22) provide asymptotic behaviors of $P_s'$ and $P_l'$ with respect to $\mu$ as summarized in . (The constant term of $P_s'$, which is irrelevant for the determination of optimal $\mu$, is omitted in ) Since $P_s'$ and $P_l'$ are descending and ascending functions of $\mu$, respectively, the optimal $\mu$ value that minimizes the computation time is determined by their trade-off. The asymptotic analysis reveals that the increase of the number of linked-list cell accesses $P_l'$ with $\mu$ is significantly larger than the reduction of search space $P_s'$, while $P_c'$ is not affected by $\mu$. This implies that asymptotically, the overhead cost from linked-list access overwhelms the benefit due to the reduced the search space for large $\mu$. This agrees with finding by Mattson et al. [101] and Yao et al. [102] such that the computation time is minimal when $\mu$ is small.

From Eq. (6.1), the costs of pair search $T_s$ and that of linked-list access $T_m$ are essential parameters to quantify the trade-off between $P_s$ and $P_l$, so that the optimal value of $\mu$ can be determined as described in section 6.4.
Table 6.1: Asymptotic complexity of parameter $P'_s$, $P'_c$, and $P'_l$ over $\mu$.

<table>
<thead>
<tr>
<th>Operation count</th>
<th>Scaling with reduction factor $\mu$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P'_s$</td>
<td>$O(\mu^3)$</td>
</tr>
<tr>
<td>$P'_c$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$P'_l$</td>
<td>$O(\mu^6)$</td>
</tr>
</tbody>
</table>

6.4 Model Validation

As derived in section 6.3, the computation time for short-range interactions in the reduced linked-list cell method is given by

$$T = T'_s P'_s + T'_c P'_c + T'_m P'_l$$

$$= T'_s \left( \frac{(2\mu+1)^3}{2\mu^3} \rho^2 R_s^6 L_x L_y L_z \right) + T'_c \left( \frac{2}{3} \pi \rho^2 R_s^3 R_c^3 L_x L_y L_z \right) + T'_m \left( \mu^3 (2\mu + 1)^3 L_x L_y L_z \right).$$ (6.23)

In section 6.4.1, we validate the accuracy of the operation-counts models—$P'_s$, $P'_c$, and $P'_l$—by comparing them with measured values. In section 6.4.2, we fit the $P'_s$, $P'_c$, and $P'_l$ derived from the estimation, Eq. (6.23), to obtain the operation costs—$T'_s$, $T'_c$, and $T'_m$.

6.4.1 Validation of the Operation Counts Model

MD simulations of silica at temperature 300 K involving up to 192,000 atoms are performed on an Intel Core-i7 2.66 GHz machine, using the cell reduction factor $\mu$ ranging from 1 to 4. The cutoff radius $R_c$ of this dataset is set to 5.5 Å with atom density $\rho = 0.0654$ atom/Å$^3$, and the linked-list cell size $R_s = 5.728$ Å.
Under this condition, the actual values of $P_s'$, $P_c'$, and $P_l'$ measured during the simulations are compared with those obtained from the model; see Table 6.2. The relative error of $P_s'$ and $P_l'$ is less than 4% in all cases, confirming the applicability of our model for these parameters. For $P_c'$, the relative error is $\sim 10\%$ for the largest number of atoms ($N = 192,000$), while it is $\sim 8\%$ for the smallest system ($N = 1,536$). Nevertheless, the maximum error of 10% is acceptable for the subsequent analysis, especially in the light of the highly dynamic nature of MD simulations.

Table 6.2: Accuracy of the operation counts model.

<table>
<thead>
<tr>
<th>Number of atoms</th>
<th>Reduction factor, $\mu$</th>
<th>Relative error (%)</th>
<th>$P_s'$</th>
<th>$P_c'$</th>
<th>$P_l'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,536</td>
<td>1</td>
<td>1.67</td>
<td>8.01</td>
<td>3.60</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.31</td>
<td>8.01</td>
<td>0.70</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.96</td>
<td>8.01</td>
<td>0.19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.95</td>
<td>8.01</td>
<td>0.04</td>
<td></td>
</tr>
<tr>
<td>12,288</td>
<td>1</td>
<td>0.79</td>
<td>-0.51</td>
<td>3.50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.76</td>
<td>-0.51</td>
<td>0.60</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.49</td>
<td>-0.51</td>
<td>0.09</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.39</td>
<td>-0.51</td>
<td>-0.06</td>
<td></td>
</tr>
<tr>
<td>41,472</td>
<td>1</td>
<td>-0.24</td>
<td>-2.28</td>
<td>2.68</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>-0.21</td>
<td>-2.28</td>
<td>-0.20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>-0.40</td>
<td>-2.28</td>
<td>-0.70</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>-0.31</td>
<td>-2.28</td>
<td>-0.85</td>
<td></td>
</tr>
<tr>
<td>98,304</td>
<td>1</td>
<td>-1.43</td>
<td>-3.29</td>
<td>1.67</td>
<td></td>
</tr>
</tbody>
</table>
### 6.4.2 Parameter Fitting

In section 6.4.1, we confirm that Eq. (6.23) accurately approximates the operation counts. In this section, we use the prediction from the model and the running time from the execution of actual MD simulation to find the cost parameters $T_s$, $T_c$, and $T_m$. Here, we need to use more datasets in order to get better statistics for the fitting. Since $R_c$ is one of the most important variables, we here concentrate on the effect of $R_c$ on the performance.

We run two additional sets of MD simulations using the same setting as in section 6.4.1 except for $R_c$, which is set here to 6.6 and 10 Å. Using the measured timings, we perform least square regression to obtain $T_s$, $T_c$, and $T_m$ from Eq. (6.23). However, the results obtained from the least square regression show very large error up to 93% for estimated $T_c$; see . This may be explained by the two orders-of-magnitude difference between the largest and smallest dataset sizes (1,536 and 192,000 atoms), which causes the fitting residuals of larger datasets to dominate the significance of smaller datasets. To alleviate this artifact, a more robust regression algorithms, Huber regression and bisquare-weighted regression, are used instead of the least square regression [104, 105]. Table 6.3 shows that the estimated operation costs from Huber

<table>
<thead>
<tr>
<th>$R_c$</th>
<th>$T_s$</th>
<th>$T_c$</th>
<th>$T_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>192,000</td>
<td>2</td>
<td>-1.35</td>
<td>-3.29</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>-1.42</td>
<td>-3.29</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>-1.59</td>
<td>-3.29</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>-3.50</td>
<td>-10.59</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>-3.29</td>
<td>-10.59</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>-3.39</td>
<td>-10.59</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>-3.38</td>
<td>-10.59</td>
</tr>
</tbody>
</table>
regression have smaller standard errors (at most 40.8%), while those from bisquare-weighted regression have the least errors (at most 22.5%). Therefore, we use the estimated costs from bisquare-weighted regression to predict the performance and thereby obtain the optimal reduction factor $\mu$ in the next section.

Table 6.3: Estimated operation costs from various regression algorithms.

<table>
<thead>
<tr>
<th>Regression algorithm</th>
<th>Operation cost</th>
<th>Estimated cost (ns)</th>
<th>Standard error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least square</td>
<td>$T_s$</td>
<td>9.76</td>
<td>40.2</td>
</tr>
<tr>
<td></td>
<td>$T_c$</td>
<td>34.83</td>
<td>93.1</td>
</tr>
<tr>
<td></td>
<td>$T_m$</td>
<td>10.44</td>
<td>7.6</td>
</tr>
<tr>
<td>Bisquare</td>
<td>$T_s$</td>
<td>6.48</td>
<td>22.2</td>
</tr>
<tr>
<td>$c = 1.865$</td>
<td>$T_c$</td>
<td>52.89</td>
<td>22.5</td>
</tr>
<tr>
<td></td>
<td>$T_m$</td>
<td>11.80</td>
<td>2.5</td>
</tr>
<tr>
<td>Huber</td>
<td>$T_s$</td>
<td>10.77</td>
<td>14.9</td>
</tr>
<tr>
<td>$c = 1.345$</td>
<td>$T_c$</td>
<td>32.56</td>
<td>40.8</td>
</tr>
<tr>
<td></td>
<td>$T_m$</td>
<td>10.81</td>
<td>3.0</td>
</tr>
</tbody>
</table>

6.5 Performance Results

Here, we use the estimated operations counts and costs in the previous section to predict the performance of the simulation when the cell size is reduced. The optimal reduction factor $\mu^*$ is then determined as the one that minimizes the predicted computation time:

$$\mu^* = \arg \min \{T(\mu)\}.$$  \hspace{1cm} (6.24)
To study the computation time as a function of $\mu$ from the prediction, we compare the predicted and measured performances of the system with $R_c = 5.5, 6.6, 10.0$ Å, with the number of atoms ranging from 1,536, 12,288, 41,472, 98,304, to 192,000. Figure 6.3 shows that the prediction captures the essential behavior of the short-range computation time in the actual simulations as a function of $\mu$. This result agrees with the asymptotic characteristics of $\mu$ in Table 1, where the linked-list cell access cost dominates for larger $\mu$.

Figure 6.4 compares $\mu^*$ obtained from the prediction, Eq. (6.23), with those obtained from actual measurements. The $\mu^*$ for $R_c = 5.5$ Å from actual simulation dataset are all 1, similar to the predicted values. Namely, reducing the cell dimension does not improve the performance in this case. On the other hand, the predicted $\mu^*$ of the system with $R_c = 6.6$ and 10.0 Å is 2 in almost all cases with the average speedup of 1.10 and 1.52, respectively. This result indicates that the reduction of the cell dimension becomes more beneficial for larger $R_c$. In such cases, the pair search space is larger and the associated search cost $P_s'$ becomes more important relative to the list-processing overhead $P_l'$. Table 6.4 shows the speedup achieved from our optimization compared to the actual optimal speedup.
Figure 6.3: The running time from the prediction and the measurement for $R_c = 10 \text{ Å}$. The numerals are the number of atoms.

Figure 6.4: Comparison of speedup gained from predicted and actual $\mu^*$ for $R_c = 6.6$ and 10 Å.
Table 6.4: The predicted versus measured values of the optimal $\mu$ with their corresponding speedup compared to the case of no cell dimension reduction.

<table>
<thead>
<tr>
<th>$R_c$ (Å)</th>
<th>$N$</th>
<th>$\mu^*$</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Prediction</td>
<td>Actual</td>
</tr>
<tr>
<td>1,536</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>12,288</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>41,472</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>98,304</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>192,000</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>5.5</td>
<td>1,536</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>12,288</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>41,472</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>98,304</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>192,000</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>6.6</td>
<td>1,536</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>12,288</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>41,472</td>
<td>2</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>98,304</td>
<td>2</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td>192,000</td>
<td>2</td>
<td>1.02</td>
</tr>
<tr>
<td>10.0</td>
<td>1,536</td>
<td>3</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>12,288</td>
<td>3</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>41,472</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>98,304</td>
<td>3</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>192,000</td>
<td>2</td>
<td>2.00</td>
</tr>
</tbody>
</table>
6.6 Summary

We have developed a performance prediction model of short-range interaction computations of molecular dynamics simulations, which can be used to obtain the optimal cell dimension in the reduced linked-list cell algorithm. The model has been validated against measurement results. The model accurately predicts the optimal reduction factor, and average speedup of 1.10 and 1.52 are obtained for $R_c = 6.6$ and 10.0 Å datasets, respectively. Our analysis reveals that the optimal reduction factor is determined by a trade-off between decreasing search space and increasing linked-list cell access for smaller cells.
Chapter 7

Memory-Access Analysis and Optimization via Dynamic Data Reordering

One of the major problems on improving performance of sophisticated scientific applications such as MD simulation is to maintain data locality. Since the memory-access pattern in MD simulation is highly non-uniform and unpredictable, the locality optimization problem becomes even more challenging. The impact of this issue leads to a fluctuation of MD simulation performance. The unstable performance is intolerable for the design of performance model, thus need to be addressed. In this chapter, we first discuss the causes of performance fluctuation from poor memory-access pattern. We then propose a solution by dynamic data reordering.

This chapter is organized as follows. Section 7.1 presents the parallel MD program, to which the proposed memory optimization scheme is implemented. Section 7.2 describes the analysis of simulation parameters related to data fragmentation. Section 7.3 presents the performance measurements and discusses the correlation between fragmentation and
performance. Section 7.4 proposes our adaptive memory-access optimization scheme based on data fragmentation analysis and performance measurements. Summary is shown in section 7.5.

7.1 Introduction

A commonly used method to reduce the performance fluctuation from poor data locality issue is data reordering. Data reordering organizes data of irregular memory-access patterns in memory according to a certain locality metric [102, 106-108]. However, a further challenge arises from the dynamic, irregular nature of MD computations. Mellor-Crummey et al. [109, 110] showed that data reordering and computation restructuring enhance data locality, resulting in the reduction of cache and translation-lookaside buffer (TLB) misses and accordingly considerable performance improvement of MD simulations. The study also suggested the necessity of repeated runtime reordering, for which the remaining problem is how often the reordering should be performed in order to achieve the optimal overall performance. In addition, as the data reordering and computation restructuring incur computational overhead, such reordering cost should be considered to find the optimal reordering frequency.

Runtime data behaviors of MD simulations are closely related to both physical properties of the system being simulated (e.g. temperature, diffusion rate, and atomic configuration) and computational parameters (e.g. spatial decomposition granularity in parallel MD). Therefore, understanding how these quantities affect the deterioration of data locality is a prerequisite for designing the optimal runtime data-reordering schedule.

To address these challenges, we first introduce a data fragmentation ratio metric that quantifies the locality of atom data arrays during MD simulation. Then, we perform an analysis on how MD simulations with different physical/computational characteristics (e.g. temperature,
diffusion rate, and granularity) impact the data fragmentation ratio and performance deterioration of the system. Based on the data fragmentation analysis, we finally design and evaluate an adaptive memory optimization scheme that features an optimal runtime data-reordering schedule during MD simulation.

7.2 Data Fragmentation in Parallel Molecular Dynamics

High memory-access penalty is a major obstacle of irregular memory accessing applications, which needs to be mitigated. Our MD application performs data and computation orderings by organizing atom data contiguously in memory in the same order as the linked lists of the cells access them. Though the spatial locality after data reordering is retained for a while, data get disordered as simulation progresses due to atom movement among cells. How the benefit from data reordering deteriorates as simulation evolves at runtime essentially affects the performance of MD simulations. To quantify the level of fragmentation, we first define a fragmentation measurement ratio metric.

Let $C(i, j, k)$ be a linked-list cell with cell indices $i, j,$ and $k$ in the $x, y,$ and $z$ directions, respectively. Each linked-list cell contains indices of atoms whose coordinates are within their cell dimensions. Before data reordering, atom data are likely scattered in memory as illustrated in Figure 7.1(a), and data reordering improves the data locality as shown in Figure 7.1(b), where the atom data of the same physical cell volume reside continuously. Fragmentation in the atom data array could occur as follows. Suppose that the $a$-th atom in linked-list cell $C$ moves to another cell $C'$. Consequently, the memory block of $C'$ becomes partially fragmented because the $a$-th atom is not contiguous in memory space with other atoms in $C'$ such that their distances in the memory space exceed the page size. Therefore, any atom moving out of its original linked-
list cell introduces fragmentation in memory, which likely causes data translation lookaside buffer (DTLB) misses. To quantify the degree of fragmentation, we thus define a data fragmentation ratio as

\[
f = \frac{N_{\text{fragment}}}{N},
\]

where \(N_{\text{fragment}}\) is the number of atoms whose positions have moved out of the originally ordered cells and \(N\) is the total number of atoms. The data fragmentation ratio is between 0 (i.e., all atoms in all cells reside continuously in memory—fully ordered state, see Figure 7.1(b)) and 1 (i.e., no atom resides in the original cell—fully disordered state, see Figure 7.1(a)). The data fragmentation ratio will be used to quantify fragmentation extensively throughout this chapter. Note that the page size used in all experiments is 4 KB.

Figure 7.1: Schematic of memory layout for atom data in (a) a fully disordered state and (b) a fully ordered state, where \(C(i,j,k)\) is the linked list for the cell with indices \(i, j,\) and \(k\) in the \(x, y,\) and \(z\) directions, respectively, \(r[a]\) is the data associated with the \(a\)-th atom.
Data fragmentation in memory is dictated by the dynamics of atoms, and thus understanding the factors that control the atom dynamics would provide an insight on how to prevent the fragmentation. One of the physical factors that are directly related to the dynamics is the temperature, since high temperature drives atoms to move faster and more freely. Among the computational factors, migration of atom data from one node to another across a subsystem boundary in parallel MD also causes fragmentation. The granularity of spatial decomposition \textit{(i.e.} the number of atoms per compute node\textit{)} is related to the subsystem surface/volume ratio, and hence is likely to control the degree of fragmentation via the amount of atom migrations. In the following subsections, we measure and analyze the effects of temperature and granularity on the data fragmentation: Sections 7.2.1 and 7.2.2 deal with fragmentation behavior influenced by temperature and granularity, respectively.

\textbf{7.2.1 Temperature Induced Fragmentation}

In this subsection, we study the effect of temperature on the data fragmentation during MD simulations on two systems: Silica material \cite{111} and combustion of aluminum nanoparticles \cite{112}. The silica simulations involving 98,304 atoms and 8,000 linked-list cells (20\times20\times20 in $x$, $y$, and $z$ directions, respectively) are performed for 3,000 MD time steps using a time discretization unit of 2 femtoseconds. The simulations are performed on a dual quadcore Intel Xeon E5410 2.33 GHz (Harpertown) at the High Performance Computing and Communication (USC-HPCC) facility of the University of Southern California. Temperature is a major physical factor that enhances the diffusion of atoms and hence the disordering of data arrays. To examine the effect of atomic diffusions on the data fragmentation ratio, we thus perform a set of simulations with three different initial temperatures, 300 K, 3,000 K, and 6,000 K, starting from the same initial atomic configuration—an amorphous structure with a uniform
density across the system. Each dataset represents a distinct phase of silica—solid (300 K), highly viscous liquid (3,000 K) [113], and low-viscosity liquid (6,000 K) (note that the melting temperature of silica is 1,800 K). Here, data ordering is performed only once at the beginning of the simulation, so that the atom data are fully ordered (Figure 7.1(b)) initially, and subsequently we measure the data fragmentation ratio as a function of MD steps without further data reordering.

In order to study the influence of temperature on data fragmentation, Figure 7.2 plots the data fragmentation ratio as a function of MD steps. We observe that simulations at higher temperatures exhibit larger fragmentation ratios. The fragmentation of 300 K dataset fluctuates in the first 500 steps, then rises to 0.15 after 1,000 steps, and remains nearly constant throughout the simulation thereafter. At 3,000 K, the fragmentation ratio quickly increases to 0.24 in the first 200 steps, and then increases linearly with time (0.033 per 1,000 steps), reaching 0.36 after 3,000 MD steps. In contrast, the fragmentation ratio at 6,000 K rapidly rises to 0.73 in the first 500 steps, then continues to increase with a slightly slower rate until the data gets almost fully disordered at 0.93 after 3,000 MD steps.
Figure 7.2: Time variation of data fragmentation ratio over 3,000 MD steps. Three datasets of different initial temperatures (300 K, 3,000 K, and 6,000 K) are plotted.

In order to understand the physical origin of the correlation between the fragmentation ratio and temperature in Figure 7.2, we measure the mean squared displacement (MSD) of atoms in each dataset as a function of time. MSD is used to identify the phase (e.g. solid vs. liquid) of the material and to calculate the diffusion rate of atoms in the system, which may be used to quantify the dynamics of atoms during the simulation. Figure 7.3 shows the MSD of 300 K, 3,000 K, and 6,000 K datasets over 6 picoseconds (3,000 MD steps). The result shows that at 300 K, silica remains in solid phase (MSD remains constant). In contrast, 3,000 K silica is melted and becomes a highly viscous liquid with a small diffusion coefficient of $2.31 \times 10^{-5}$ cm$^2$/s (the diffusion coefficient is obtained from the linear slope of the MSD curve). Similarly, MSD of 6,000 K dataset shows that the system is completely melted with a much larger diffusion coefficient of $4.17 \times 10^{-4}$ cm$^2$/s. Because atoms are not diffusing in 300 K dataset, 83% of the atoms ($f = 0.17$) remain in the same linked-list cell throughout the simulation. Only atoms close to the cell boundaries move back and forth between the original and the neighbor cells due to
thermal vibration. However, atoms in 3,000 K dataset are melted, and their slow diffusion from their original cells causes the gradually increasing fragmentation ratio. For 6,000 K system, atoms diffuse approximately 18 times faster than 3,000 K system, resulting in the rapid rise of fragmentation ratio. Only 3.2% of the atoms remain in the same cell after 3,000 steps. These results clearly show that diffusion is a major physical factor that contributes to data fragmentation in MD simulations.

![Graph of Mean Squared Displacement (MSD) and Diffusion Coefficients](image)

Figure 7.3: Mean squared displacement (MSD) of 300 K, 3,000 K, and 6,000 K datasets. The MSD of 300 K dataset remains constant at 0.23 Å² (too small to be seen in the figure). The inset shows the diffusion coefficients at the three temperatures.

Although the silica experiment shows that the temperature significantly affects data fragmentation, we also found that its effect on the fragmentation ratio is sensitive to the material and phenomenon being simulated. To demonstrate this point, the second experiment simulates flash heating of an aluminum nanoparticle surrounded by oxygen environment using a reactive interatomic potential [112], which involves 15,101,533 atoms on 1,024 processors of dual core AMD Opteron 270 2.0 GHz (Italy) at the USC-HPCC facility. Similar to the first simulation, the
atom data arrays are ordered only once at the beginning of the simulation. Then, the fragmentation ratio of 3,000 K, 6,000 K and 9,000 K datasets are measured at each step. Figure 7.4 shows the data fragmentation ratio as a function of MD steps. We see that the fragmentation ratios of all datasets rapidly rise to above 90% after 1,000 MD steps, and continue to increase to over 98% after 2,000 MD steps. Since the aluminum nanoparticles are at considerably high temperatures (far above the melting temperature of aluminum ~ 930 K) and are surrounded by oxygen atoms in the gas phase, the atoms are highly reactive and move very rapidly. This accelerates the fragmentation to proceed very quickly regardless of the temperature. In such applications, data reordering is indispensable and is required to be performed more often in order to maintain good spatial locality. These results indicate that fragmentation is highly system dependent, so that data reordering needs to be performed dynamically at runtime, adapting to the system’s behaviors.

Figure 7.4: Data fragmentation ratio during aluminum combustion simulation over 2,000 MD steps. The dataset involves 15 million atoms on 1,024 processors.
7.2.2 Granularity Induced Fragmentation

In addition to fragmentation induced by physical properties, parallel MD using spatial decomposition introduces computational artifacts such as spatial subsystem (or domain) boundaries, which also contribute to the fragmentation of atom data arrays. In this subsection, we study the influence of granularity (i.e. the number of atoms per compute node) on data fragmentation. Atoms near a domain-boundary surface tend to migrate among domains regardless of physical simulation conditions. This inter-domain atomic migration triggers rearrangements of data arrays, including the deletion of the migrated atom data from the original array, compression of data array after the removal of the migrated atoms, and their appending to the array in the destination domain. These newly migrated atoms in the destination domain cause data fragmentation, since they do not reside continuously in memory.

To confirm the expected high fragmentation ratio at the domain boundaries as explained above, the data fragmentation ratios of the surface cells (i.e. the outer-most layer cells that share at least one facets with the inter-domain surfaces) and core cells (i.e. non-surface cells deep inside each domain) are measured separately. Here, we consider a 12,288-atom silica dataset initially at 3,000 K temperature similar to the dataset used in the first experiment of section 7.2.1 but with a reduced domain size (their dimensions are reduced by half in all three directions, so that the domain volume is one-eighth of that in section 7.2.1). This dataset consists of 1,000 cells in total (10×10×10), of which 488 cells (48.8%) are surface cells. Figure 7.5(a) clearly shows that the data fragmentation ratio of the surface cells is larger than that of the core cells. In the first 100 MD steps, the fragmentation ratios of the two groups are almost identical. Then, the fragmentation ratio of the surface cells begins to increase at higher rate reaching 0.53 after 3,000 MD steps, whereas the fragmentation ratio of the core cells is only 0.39. Thus, the atom data of
the surface cells is 14% more fragmented than that in the core cells, yielding a total fragmentation ratio of 0.42 for the entire system. This result confirms that computational artifacts such as domain boundary indeed induce additional fragmentation.

Figure 7.5: (a) Time variation of the data fragmentation ratio of 12,288-atom silica at 3,000 K over 3,000 MD steps, for surface cells, core cells, and both combined. (b) Time variation of the data fragmentation ratio of 3,000 K silica with varying granularities over 3,000 MD steps.
The domain-boundary induced fragmentation also implies that systems with smaller granularities will have more fragmentation due to their larger surface/volume ratios (i.e., larger portions of linked-list cells are domain-boundary cells). To test this hypothesis, we perform MD simulations for 3,000 K silica system with three different granularities—98,304, 12,288, and 1,536 atoms, over 3,000 MD steps. The dimensions of the three systems are summarized in Table 7.1, which shows that the surface-to-volume ratio ranges from 27.1% for \( N = 98,304 \) to 78.4% for \( N = 1,536 \). Asymptotic analysis of surface-to-volume ratio shows that the ratio scales as \( O(N^{1/3}) \).

Figure 7.5(b) shows that datasets with smaller granularities indeed have larger fragmentation ratios. After 3,000 MD steps, the data array of the smallest granularity (1,536 atoms) dataset is 9.7% more fragmented than that in the largest granularity (98,304 atoms) dataset. Also, the 12,288 atoms dataset is 5.2% more fragmented compare to the largest dataset. The figure also shows large fluctuation for the fragmentation ratio for \( N = 1,536 \) dataset, due to less statistics inherent for the small system size.

Table 7.1: 3,000 K silica datasets with varying granularities: The number of linked-list cells, the portion of surface cells, and the surface-to-volume ratio of the spatial subsystem are listed for each dataset.

<table>
<thead>
<tr>
<th>Number of atoms</th>
<th>Number of cells</th>
<th>System dimensions (cells)</th>
<th>Portion of surface cells (%)</th>
<th>Surface-to-volume ratio (Å⁻¹)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,536</td>
<td>125</td>
<td>5×5×5</td>
<td>78.4</td>
<td>0.111</td>
</tr>
<tr>
<td>12,288</td>
<td>1,000</td>
<td>10×10×10</td>
<td>48.8</td>
<td>0.055</td>
</tr>
<tr>
<td>98,304</td>
<td>8,000</td>
<td>20×20×20</td>
<td>27.1</td>
<td>0.027</td>
</tr>
</tbody>
</table>
7.3 Performance Measurement

In the previous section, we described how physical and computational parameters of MD simulations affect the data fragmentation behavior in the memory space, utilizing the data fragmentation ratio as a metric. In this section, we first establish a correlation between the data fragmentation ratio and the performance of the program. In a fragmented dataset, atom data are likely to reside in different memory pages, which causes a large number of DTLB misses, when they are fetched. The reordering algorithm explained in section 7.2 clusters atom data that need to be fetched and computed in relatively proximate times (e.g. atoms in the same cells) in the same page, thereby reducing DTLB misses tremendously. However, as the simulation progresses, atoms that are once ordered possibly move out of their original cells. It is thus expected that the increase of DTLB misses caused by data fragmentation is a major source of performance degradation in MD.

To confirm the correlation between the number of DTLB misses and the data fragmentation ratio, we perform a test using the same datasets from the first experiment of section 7.2.1 (98,304 silica atoms with initial temperatures of 300 K, 3,000 K, and 6,000 K). In addition to the fragmentation ratio measured in section 7.2.1, we here measure the DTLB miss rate as a function of MD steps. We use the Intel VTune Performance Analyzer to monitor DTLB miss events during MD simulation on Intel Core i7 920 2.67 GHz (Nehalem) processor. We measure the number of DTLB misses after data ordering is performed, then normalize it by the original number of DTLB misses without data ordering. We find that the initial DTLB miss rates at all temperatures are approximately 0.07 (<< 1) right after the reordering is performed (namely, the reordering reduces DTLB misses by 93%). The great improvement highlights a significant role of data reordering in reducing DTLB misses for irregular memory-access applications. We
also monitor the number of DTLB misses as a function of MD time steps and observe distinct profiles at different temperatures (Figure 7.6). The DTLB misses ratio at 300 K and 3,000 K saturates at 0.08 and 0.13, respectively. In contrast, the 6,000 K simulation exhibits a continuous increase of the number of DTLB misses. The DTLB miss rate reaches 0.62 at 3,000 MD steps after the initial data reordering.

![Figure 7.6](image)

Figure 7.6: Time variation of the DTLB miss rate in 300 K, 3,000 K and 6,000 K dataset over 3000 MD steps.

Figure 7.7 shows the relation between the data fragmentation ratio and the DTLB misses rate at the three temperatures. The DTLB misses rate remains relatively small when the fragmentation ratio is small, while it rapidly increases when the fragmentation ratio increases above 0.8. This result clearly demonstrates a strong correlation between the DTLB miss rate and the data fragmentation ration. Namely, in MD simulations where atoms are moving extensively, the DTLB miss rate increases rapidly as its fragmentation ratio does.
Figure 7.7: Relation between the DTLB miss rate and the data fragmentation ratio in 300 K, 3,000 K and 6,000 K datasets.

To show that DTLB misses are the source of performance deterioration during simulation, we measure the running time of the same datasets as in the DTLB miss measurement. Figure 7.8 shows the running time of each MD step for 3,000 MD steps after ordering data at the first step. At 6,000 K, the running time gradually increases over time, and after 3,000 MD steps, the average running time per step increases by 8%. To study how much more performance degradation occurs after the initial data ordering, we extend the execution of 6,000 K dataset to 20,000 MD steps. The result shows an increase of the running time per step by 21%. This result indicates that without data reordering, the performance continues to degrade, and the running time per step continues to increase. On the other hand, the running time rapidly increases at the first 100 MD steps but then remains almost constant at 300 K and 3,000 K. These performance behaviors are akin to the fragmentation ratio and DTLB misses profiles in Fig. 7.2 and 7.6, respectively. These results thus confirm that data fragmentation in memory is indeed the source of performance deterioration during runtime through the increase of DTLB misses.
In Fig. 7.8, we also observe that 6,000 K simulation initially executes fastest compared to the other simulations (3.1% faster than 300 K and 2.5% faster than 3,000 K), which cannot be explained by the difference in the DTLB miss rate. One possible reason for this discrepancy is that higher temperature systems have sparser atomic distributions (i.e. lower atomic number densities), such that there are less number of atom interactions within the interaction range \( r_c \), and hence less number of floating-point operations. To test this hypothesis, we measure the atom-distance distribution for all datasets. The results show that the number of atom pairs within distance \( r_c \) of 6,000 K system is approximately 3.0% and 1.4% less than those of 300 K and 3,000 K systems, respectively. As a result, 6,000 K simulation has the least computational load resulting in the fastest running time at the beginning of the simulation, which however is rapidly offset by the increase in DTLB misses.

Figure 7.8: Running time per MD step as a function of MD steps at temperatures 300K, 3,000K, and 6,000K for 98,304 silica atoms.

In addition to the atom ordering discussed above, the cell ordering in the memory space also affects the locality and performance of the simulations. To address this issue, we measure
the performance of three different cell ordering methods: (1) sequential ordering; (2) Hilbert-curve ordering; and (3) Morton-curve ordering (Figures 7.9(a)-(c), respectively). The performance measurements are performed with 300 K temperature silica systems with grain size ranging from 12,288 - 331,776 atoms with reordering period of 20 MD steps on Intel Core i7 920 2.67 GHz (Nehalem) processor. The performance comparisons of all ordering methods are shown in Table 7.2. The results do not exhibit significant improvement due to different ordering methods. For example, the running time of Hilbert curve ordering is at most 1.76% less than that of the sequential ordering at largest granularity (331,776 atoms), while the running time of Morton curve ordering is less than 1% different for all granularities. One of the possible reasons of the insensitivity of the performance on the cell-ordering method is the small number of references across intra-node cell boundaries. For simplicity, we employ the sequential ordering in the following.

Table 7.2: Runtime result with different ordering methods for silica systems at temperature 300K.

<table>
<thead>
<tr>
<th>Number of atoms</th>
<th>System dimensions (cells)</th>
<th>Ordering method</th>
<th>Average running time per MD step (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12,288</td>
<td>10×10×10</td>
<td>Sequential</td>
<td>78.48</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Morton</td>
<td>78.57</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hilbert</td>
<td>78.45</td>
</tr>
<tr>
<td></td>
<td>98,304</td>
<td>Sequential</td>
<td>639.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Morton</td>
<td>640.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hilbert</td>
<td>637.1</td>
</tr>
<tr>
<td>331,776</td>
<td>30x30x30</td>
<td>Sequential</td>
<td>2125.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Morton</td>
<td>2117.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hilbert</td>
<td>2088.8</td>
</tr>
</tbody>
</table>
7.4 Reordering Frequency Optimization

The previous section showed a serious performance degradation of MD simulation during runtime. To minimize the performance degradation due to data fragmentation, we propose to repeat data reordering periodically during MD simulation. Specifically, we reorder atom arrays after every $N_{rp}$ MD steps (i.e., the reordering period $N_{rp}$ is the number of MD steps between successive reordering operations). Though such data reordering is beneficial and often improves overall application performance, ordering arrays itself introduces an additional computational cost. Therefore, we here develop a dynamic data-reordering schedule based on a performance model and the runtime measurement in section 7.3. To do so, we first introduce a model that accounts for the reordering overhead. Let $t_{cost}$ be the data reordering cost (i.e. the time to reorder arrays), $N_{total}$ the total simulation steps, and $t(n)$ the running time at step $n$ after ordering. The total running time as a function of reordering period is then written as

$$
\tau_{total}(N_{rp}) = \frac{N_{total}}{N_{rp}} \left( \sum_{n=1}^{N_{total}/N_{rp}} t(n) + t_{cost} \right) + \sum_{n=1}^{N_{total} \mod(N_{rp})} t(n). \quad (7.2)
$$
The optimal reordering period is then determined as the one that minimizes the total running time:

\[ N_{rp}^* = \text{argmin}(\tau_{\text{total}}(N_{rp})) \]  \hspace{1cm} (7.3)

To find the ordering cost parameters in Eq. (7.2), we measure the reordering cost with different grain sizes. The results are summarized in Table 7.3.

**Table 7.3: 3,000 K silica datasets varying granularities**

<table>
<thead>
<tr>
<th>Number of atoms</th>
<th>Number of cells</th>
<th>System dimensions (cells)</th>
<th>Reordering cost (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12,288</td>
<td>1,000</td>
<td>10x10x10</td>
<td>0.66±0.02</td>
</tr>
<tr>
<td>98,304</td>
<td>8,000</td>
<td>20x20x20</td>
<td>8.41±0.07</td>
</tr>
<tr>
<td>786,432</td>
<td>64,000</td>
<td>40x40x40</td>
<td>65.8±0.10</td>
</tr>
</tbody>
</table>

Figure 7.10 shows the total running times for 3,000 MD steps as a function of the reordering period estimated from Eq. (7.2) with the measured reordering costs in Table 7.3. We obtain the optimal reordering period (which minimizes the total running time) as 69, 5, and 3 steps for 300 K, 3,000 K and 6,000 K datasets, respectively (see the arrows in Fig. 7.10). With the optimally scheduled reordering thus determined, the overall performance is estimated to be improved by a factor of 1.35, 1.36 and 1.28 at 300 K, 3,000 K and 6,000 K, respectively.

To verify this model prediction, we measure the running time of MD simulations that implement data reordering with the optimal reordering schedule. Here, MD simulations of silica containing 98,304 atoms at temperatures 300 K, 3,000 K, and 6,000 K are executed using periodic reordering at every 69, 5, and 3 steps, respectively. The results show that the measured speedups in actual executions are 1.27, 1.25, and 1.17, respectively, with 6.44%, 7.59%, and
9.95% error from the estimated values. The running time of the system without ordering, the optimized running time estimated from the model, and the measured running time with optimally scheduled reordering are compared in Figure 7.11. The figure shows a substantial effect of runtime data reordering on the performance of MD simulations. It should be noted that the state-of-the-art MD simulations are run up to $10^{12}$ time steps [114], for which the $10^3$-step pre-profiling run for constructing the performance model, Eq. (7.2), is negligible.

Figure 7.10: Total running time after 3,000 MD steps as a function of reordering periods. The optimal period for 300 K, 3,000 K and 6,000 K datasets are 69, 5 and 3 steps, respectively.
Figure 7.11: Comparison of total running time of silica MD over 3,000 steps achieved by model prediction and actual measurement of periodic reordering compared with that of the original code without ordering. (All tests are conducted with 98,304 atoms per core.)

7.5 Summary

In summary, we have developed an adaptive data-reordering schedule for molecular dynamics simulations on parallel computers. Our quantitative analysis has identified physical/computational conditions such as a high initial temperature and a small granularity, which considerably accelerate data fragmentation in the memory space, thereby causing continuous performance degradation throughout the simulations at runtime. Our profiling results have revealed that the degree of data fragmentation correlates with the number of DTLB misses and have identified the former as a major cause of performance decrease. Based on the data fragmentation analysis and a simple performance model, we have developed an optimal data-reordering schedule, thereby archiving a speedup of 1.36× for 3,000 K silica simulation. This
study has thus proposed a practical solution to the dynamic data fragmentation problem that plagues many scientific and engineering applications.
Chapter 8

Conclusions

This dissertation research has developed a metascaleable hybrid message-passing and multithreading algorithm for \( n \)-tuple MD simulation, thereby achieving: (1) metascaleability of hybrid parallelization framework on multicore architectures; and (2) computation-pattern algebraic framework to design scalable algorithms for general \( n \)-tuple computation and proof of its optimality in a systematic and mathematically rigorous manner.

We have developed and thoroughly analyzed algorithms for hybrid MPI/OpenMP parallelization of \( n \)-tuple MD simulation, which are scalable on large multicore clusters. CVAS and BFAS algorithms were proposed, combining the benefit of fine-grain dynamic load balancing and minimal memory-footprint threading. Theoretical study revealed decent asymptotic memory efficiency for both algorithms, thereby reducing 75% memory consumption compared to a naïve-threading algorithm. Furthermore, performance benchmarks have confirmed higher performance of the hybrid MD algorithm over a traditional algorithm on large multicore
clusters, where 2.58× speedup of the hybrid algorithm over the traditional algorithm is observed on 32,768 nodes of BlueGene/P.

We have addressed issues of concurrency controls, by investigating the performance characteristics of HTM on the BlueGene/Q computer in comparison with conventional concurrency control mechanisms. Benchmark tests, along with overhead-cost and scalability analysis, quantified 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 have proposed transaction fusion and spatially compact scheduling techniques to reduce the overhead of HTM with minimal programming. A strong scalability benchmark showed that the fused HTM has the shortest runtime among various concurrency control mechanisms without extra memory. Based on the performance characterization, we derived a decision tree in the concurrency-control design space for multithreading application.

We have developed a computation-pattern algebraic framework to mathematically formulate general $n$-tuple computation for reactive MD. Based on translation/reflection-invariant properties of computation patterns within this framework, we designed an SC algorithm for cell-based parallel MD. Theoretical analysis quantified the compact $n$-tuple search space and small communication cost of SC-MD for arbitrary $n$, which are reduced to those in best pair-computation approaches (e.g. eighth-shell method) for $n = 2$. Benchmark tests showed that SC-MD outperforms our production MD code at the finest grain, with 9.7- and 5.1-fold speedups on Intel-Xeon and BlueGene/Q clusters. SC-MD also exhibited excellent strong scalability.

We have also analyzed the computational and data-access patterns of MD, which led to the development of a performance prediction model for short-range pair-wise force computations in MD simulations. The analysis and performance model provide fundamental understanding of
computation patterns and optimality of certain parameters in MD simulations, thus allowing scientists to determine the optimal cell dimension in a linked-list cell method. The model accurately estimated the number of operations during the simulations with the maximum error of 10.6% compared to actual measurements. Analysis and benchmark of the model revealed that the optimal cell dimension minimizing the computation time is determined by a trade-off between decreasing search space and increasing linked-list cell access for smaller cells.

We have developed an optimal data-reordering schedule of MD for runtime memory-access optimization to solve performance deterioration during execution of MD. Analysis of the memory-access penalty during MD simulations showed that the performance improvement from computation and data reordering degrades gradually as data translation lookaside buffer misses increase. We have also found correlations between the performance degradation with physical properties such as the simulated temperature, as well as with computational parameters such as the spatial-decomposition granularity. Based on a performance model and pre-profiling of data fragmentation behaviors, we have developed an optimal runtime data-reordering schedule, thereby archiving speedup of 1.35, 1.36 and 1.28, respectively, for MD simulations of silica at temperatures 300 K, 3,000 K and 6,000 K.

The contributions of this dissertation research—metascalable hybrid message-passing and multithreading parallel framework and a computation-pattern algebraic framework for general $n$-tuple computation—are expected to provide a generic framework to a broad-range of scientific applications on future extreme-scale clusters.
References


GPUs with OpenMP and OpenCL Languages and Compilers for Parallel Computing.


