TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Rechnertechnik und Rechnerorganisation /
Parallelrechnerarchitektur der Technischen Universität München

Ensemble-based Programming for Scientific Applications

Haowei Huang

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Nassir Navab

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Hans Michael Gerndt

2. Univ.-Prof. Dr. Dieter Kranzlmüller

Ludwig-Maximilians-Universität München

Die Dissertation wurde am 27. März 2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25. Mai 2013 angenommen.

Abstract

Nowadays, HPC systems are widely used for accelerating different kinds of calculation-intensive irregular applications, e.g., molecular dynamics (MD) simulations, astrophysics applications, irregular grid applications, and so on. As the scalability and complexity of current HPC systems keeps growing, it is difficult to parallelize these applications in an efficient fashion due to irregular communication patterns, load imbalance issues, dynamic characteristics, and many more. Current parallel programming approaches start from the global computation and break it up into blocks mapped onto the hardware. Developers need to implement and optimize irregular applications for hierarchical structure the architectures including the data partitioning, local computations, exchanging information and reallocating data, which are tricky and time-consuming.

This dissertation presents an ensemble-based programming scheme, on which programmers are able to implement parallel scientific applications in a fine granular and SPMD (single program multiple data) fashion. Its main objective is to facilitate programmers' work in implementing and optimizing scientific applications. Different from current programming models starting from the global data structure, this programming scheme provides a high-level and object-oriented programming interface that supports writing applications by focusing on the finest granular elements and their interactions. The programming interface is implemented on different types of concrete machines namely sequential, shared memory, and distributed memory machines. Its implementation framework takes care of the implementation details e.g., the data partition, automatic EP aggregation, memory management, data communication, and so on.

This programming scheme can be applied to multi-body application, irregular grid application, and regular grid application areas as well. The experimental results show that the ensemble-based implementations of multi-body and irregular applications are a bit slower than the manual implementations using C++ with OpenMP or MPI. However, it improves the programming productivity in terms of the source code size, the coding method, and the implementation difficulty. Compared to current parallel programming models, the ensemble-based programming scheme manages the granularity of computation, data distribution, communication and load balance for irregular applications with reasonable overhead.

Keywords: High-Performance Computing, Parallel Programming Model, Irregular Applications, High-Level Programming, Ensemble-based Programming, MPI, OpenMP

Zusammenfassung

Die Arbeit stellt ein neues paralleles Programmierparadigma vor. Anwendungen werden in diesem Ensemble-basierten Paradigma als feingranulare parallele Objekte mit einer SPMD-artigen Verarbeitung spezifiziert. Das Ziel ist, eine höhere Produktivität durch eine abstrakte Darstellung der feingranularen Objekte und ihrer Interaktionen zu erreichen. Das Programmierparadigma wurde für sequentielle Rechner und parallele Rechner mit gemeinsamem und verteilten Speicher implementiert.

Acknowledgments

This dissertation would not have been possible without the help and support of my principal supervisor, Professor Dr. Michael Gerndt, not to mention his advice and unsurpassed knowledge of parallel programming. I would like to express my deepest gratitude to him for his excellent guidance, caring, patience, and providing me with an excellent atmosphere for doing research. The good support from my second supervisor, Professor Dr. Dieter Kranzmueller, has also been invaluable. I am very grateful for his help and useful advices to my thesis submission.

In addition, I should express my deepest respect to Professor Dr. Arndt Bode. He is always very kind of helping me with any administrative issues. He was patiently engaged in guided tours to the LRZ and academia exchange programs for many times even if he is always very busy. I am extremely grateful.

I would like to thank Dr. Kraja Fisnik, Dr. Houssam Haitof who let me know a lot of practical skills and writing technics beyond the textbooks. My thesis writing would not have been possible without their helps.

I would also like to thank my colleagues in our chair, Josef Weidendorfer, Ventsislav Petkov, Marcel Meyer, Andreas Hollmann, Alin Murarasu, Robert Mijakovic, Isaias A. Compres Urena for guiding my research for the past several years and helping me to develop the background of my research area.

I want to thank my parents for bring me into this world. They were always supporting me and encouraging me with their best wishes. Finally, I would like to thank my wife, Yaxian Zhou. She was always there cheering me up and stood by me through the good times and bad.

Contents

Abstract i

Zusammenfassung iii

Acknowledgments v

1. Introduction 1
- 1.1. Overview of Current HPC Systems . . . 1
- 1.2. Different Programming Approaches and Challenges . . . 1
- 1.3. Regular Applications and Irregular Applications . . . 3
- 1.4. Ensemble-based Programming Approach . . . 4
- 1.5. Implementation Framework of Ensemble-based Programming . . . 6
- 1.6. Main Contributions . . . 7
- 1.7. Dissertation Structure . . . 7

2. Parallel Architectures and Programming Models 9
- 2.1. Overview . . . 9
- 2.2. Parallel Architectures . . . 11
  - 2.2.1. Shared Memory Architectures . . . 11
  - 2.2.2. Distributed Memory Architectures . . . 12
  - 2.2.3. Hybrid Architectures . . . 12
  - 2.2.4. Heterogeneous Architectures with CPU and GPU . . . 14
- 2.3. Programming Different Parallel Architectures . . . 14
  - 2.3.1. Overview . . . 14
  - 2.3.2. Programming Shared Memory Architectures . . . 15
  - 2.3.3. Programming Distributed Memory Architectures . . . 17
  - 2.3.4. Programming Hybrid Architectures . . . 18
  - 2.3.5. Programming Heterogeneous Architectures with CPU and GPU . 19
- 2.4. Summary . . . 19

3. Parallel Application Areas 21
- 3.1. Overview . . . 21
- 3.2. Linear Algebra . . . 21
- 3.3. Regular Grid Applications . . . 22
  - 3.3.1. 2D Heat Distribution . . . 22

3.3.2. Cellular Automaton . . . 22
3.4. Irregular Grid Applications . . . 23
3.5. Adaptive Grid Applications . . . 23
3.6. Multi-body Applications . . . 24
3.6.1. Cosmological Simulation . . . 24
3.6.2. Molecular Dynamics Simulation . . . 25
3.7. Summary . . . 26

4. Related Work **27**
4.1. Overview . . . 27
4.2. Parallel Programming Languages and Libraries . . . 27
4.2.1. High Performance Fortran (HPF) . . . 27
4.2.2. Global Arrays (GA) . . . 28
4.2.3. TBB . . . 28
4.2.4. PARTI / CHAOS Library . . . 28
4.2.5. Charm++ . . . 29
4.2.6. UPC (Unified Parallel C) . . . 29
4.2.7. OpenCL (Open Computing Language) . . . 30
4.3. Domain-Specific Languages . . . 30
4.4. Summary . . . 31

5. Ensemble-based Programming **33**
5.1. Overview . . . 33
5.2. Machine Model . . . 33
5.2.1. Overview . . . 33
5.2.2. Fine Granular Processors (FGPs) . . . 34
5.2.3. Control Processor (CP) . . . 35
5.2.4. Interactions between the CP and FGPs . . . 36
5.3. Programming Paradigm . . . 37
5.3.1. Overview . . . 37
5.3.2. Elementary Points . . . 38
5.3.3. Ensemble . . . 39
5.3.4. Master Thread . . . 43
5.4. Programming Interface . . . 44
5.4.1. Overview . . . 44
5.4.2. An Object-Oriented Programming Approach . . . 44
5.4.3. Overview of a Running Example . . . 45
5.4.4. Template Hierarchy . . . 46
5.4.5. *ElementaryPoint* and its Derived Templates . . . 47
5.4.6. *Ensemble* and its Derived Templates . . . 49
5.4.7. *Topology* and its Derived Templates . . . 55
5.5. Example: An MD Simulation . . . 59

5.6. Summary 63

6. Implementation Framework **65**
6.1. Overview 65
6.2. Mapping to Sequential Machines 66
6.2.1. Overview 66
6.2.2. *Ensemble* Management 66
6.2.3. *Topology* Management 70
6.3. Mapping to Shared Memory Machines with OpenMP 72
6.3.1. Overview 72
6.3.2. *Ensemble* Management 73
6.3.3. OpenMP Support on NUMAs 73
6.3.4. *Topology* Management 80
6.4. Mapping to Distributed Memory Machines with MPI 81
6.4.1. Overview 81
6.4.2. Storage of Elementary Points 82
6.4.3. Implementations of *Ensemble* Operations 83
6.4.4. EP Distribution and Communication Management 85
6.4.5. *Topology* Management 92
6.4.6. Communication Optimization 97
6.5. Summary 98

7. Experimental Results **99**
7.1. Overview 99
7.2. Experimental Platform 99
7.3. Irregular Grid Applications 100
7.3.1. Overview 100
7.3.2. Data Sets 101
7.3.3. Sequential Comparison 101
7.3.4. OpenMP Comparison 102
7.3.5. MPI Comparison 105
7.3.6. Summary for Irregular Grid Applications 109
7.4. Molecular Dynamics Simulation 109
7.4.1. Overview 109
7.4.2. Sequential Comparison 109
7.4.3. OpenMP Comparison 110
7.4.4. MPI Comparison 112
7.5. Summary 113

8. Conclusion and Future Work **115**
8.1. Ensemble-based Programming Approach 115

8.2. Programming Scheme . . . 115
8.2.1. Machine Model . . . 115
8.2.2. Programming Paradigm . . . 116
8.2.3. Programming Interface . . . 116
8.3. Implementation Framework . . . 117
8.4. Evaluation . . . 118
8.5. Future Work . . . 118

A. Appendix **121**
A.1. Compiler Commands and Options . . . 121
A.2. Full Declaration of Template Hierarchy . . . 121
A.2.1. *ElementaryPoint* . . . 121
A.2.2. *Topology* . . . 123
A.2.3. *Ensemble* . . . 124
A.3. Ensemble-based Programs . . . 129
A.3.1. Irregular Grid Program . . . 129
A.3.2. Molecular Dynamics Program . . . 132
A.4. Acronyms . . . 138

Bibliography **141**

List of Figures

1.1. Irregular applications 4

2.1. Hierarchy of Flynn's taxonomy and parallel architectures 10
2.2. UMA architecture 12
2.3. Distributed memory architecture 13
2.4. Hybrid architecture 13
2.5. Heterogeneous architecture 14
2.6. OpenMP execution model 17
2.7. MPI collective operations 18
2.8. Hybrid MPI/OpenMP execution 19

5.1. Machine model 34
5.2. Fine Granular Processor 35
5.3. Control Processor 36
5.4. Programming paradigm 38
5.5. Shadow copies 39
5.6. Ensemble 40
5.7. Topology of four EPs 41
5.8. Shadow copy updates 42
5.9. Organization of the template hierarchy 46

6.1. Overview of the implementation framework 65
6.2. *Ensemble* implementation object 67
6.3. Shadow copies of four EPs 68
6.4. Optimized arrangement of the shadow copies 69
6.5. Organization of neighbor EP list 70
6.6. Id-based graph and neighbor list 72
6.7. EP partition on NUMAs 75
6.8. EP storage based on reallocation 76
6.9. EP re-indexing 77
6.10. The storage of EPs and their shadow copies 78
6.11. Overview of MPI mapping 81
6.12. MPI mapping of *parallel* operation 84
6.13. MPI mapping of collective operation 86

6.14. Cell classification . . . 87
6.15. Communication between sub domains . . . 88
6.16. Redundant communication (cutoff) . . . 89
6.17. Redundant communication (empty cells) . . . 89
6.18. Cell-based graph . . . 90
6.19. Cell partition . . . 91
6.20. Communication pattern based on irregular cell partitioning . . . 92
6.21. Movement of an EP within a subdomain . . . 94
6.22. Movement of an EP cross subdomains . . . 94
6.23. Local EPs and SC organization . . . 95
6.24. EP distribution based on graph partitioning . . . 96
6.25. Communication reduction . . . 97

7.1. Execution time of sequential programs . . . 102
7.2. Execution time of OpenMP programs (Grid64) . . . 103
7.3. Execution time of OpenMP programs (Grid128) . . . 104
7.4. OpenMP speedup curves on Grid64 and Grid128 . . . 104
7.5. Ensemble-based re-indexing . . . 105
7.6. MPI speedup curves (Block vs. METIS) . . . 106
7.7. MPI comparison using 1 process . . . 107
7.8. Execution time comparison with MPI . . . 108
7.9. Speedup curve comparison with MPI . . . 108
7.10. Execution time of sequential MD programs . . . 110
7.11. Execution time of OpenMP MD programs . . . 111
7.12. Speedup curves of OpenMP MD Programs . . . 111
7.13. Execution time of MPI MD programs . . . 112
7.14. Speedup curves of MPI MD programs . . . 113

1. Introduction

1.1. Overview of Current HPC Systems

Nowadays, high-performance computing (HPC) is currently experiencing very strong growth in all computing sectors. Many HPC systems are widely used for accelerating different kinds of calculation-intensive applications including quantum physics, weather forecasting, climate research, oil and gas exploration, molecular dynamics, and so on[1][2][3][4]. These systems typically consist of a large number of processors or accelerators as well as hundreds of terabytes of memory. The processors and memories are organized in nodes which can be small like in the IBM Blue Gene[5][6] with two embedded processors and little memory, or powerful with Intel Xeon[7] processors and GPGPUs as accelerators coupled by highly scalable and efficient inter-connections. The peak performance of current HPC systems is up to a few peta (10^{15}) FLOPS (floating point operations per second) according to the TOP500 list released in November 2012[8]. The next generation exascale (10^{18}) computing capability systems are scheduled to be deployed in around 2018. The major challenge for building exascale systems is power consumption, so new highly power efficient technologies both on hardware and software have to be exploited in achieving exascale computing[9][10].

Among current supercomputers in the TOP500 list, a series of Blue Gene systems by IBM have led for several years. A Blue Gene/Q system called Sequoia[11] was ranked as the world's fastest supercomputer in June 2012. It consists of 1.6 million processor cores and 1.6 PB of memory running at 20.1 PFLOPS peak, 16.32 PFLOPS sustained (Linpack[12][13]). In addition, as the price-performance of graphics processing units (GPUs) has improved, general purpose GPU (GPGPU) computing has become a hot research topic and a number of petaflops heterogeneous supercomputers such as Tianhe-1A[14]and Nebulae[15] have constructed relying on them. These systems are tuned to score well on specific benchmarks like Linpack, but the overall applicability is limited unless significant effort is spent to tune the applications towards it.

1.2. Different Programming Approaches and Challenges

As the scalability and complexity of current HPC systems keep growing, it is a nightmare for the developers to program these supercomputers in terms of balancing the

computational load among processors, manipulating the hierarchical memory architecture, managing communication and synchronization among processors, and so on. Multiple parallel programming interfaces, platforms, and libraries are designed to program such supercomputers and utilize their computing power efficiently. The major programming interfaces are OpenMP[16][17], MPI[18][19][20], and CUDA[21][22]. OpenMP is a portable, scalable model that provides programmers with a simple and flexible interface mainly for parallelizing loops by a number of threads on shared memory systems. MPI instead is a standard library for communication among processes. It allows programmers to specify processes accessing private data structures in order to collocate data and computation in compute nodes of distributed memory systems. CUDA is a computing engine and programming model that supports parallel fine-grained computation on Nvidia GPGPUs and interactions between CPUs and the GPGPUs.

These parallel programming approaches start from the global computation and break it up into blocks, which are mapped across different units of execution (UEs). The UEs can be threads, processes, or light weight CUDA threads depending on the description of the hardware. These programming approaches are based on domain decomposition, which divides the global computational domain of an application into multiple subdomains. Each UE is responsible for the computation of a subdomain. Ideally, the execution of the application can obtain a linear speedup if all the UEs execute computation on its own subdomain in parallel independently. However, in order to achieve good speedup, programmers have to take care of many challenging implementation aspects including data distribution and mapping, computational load balance, memory overhead, synchronization and communication among UEs, and so on.

Data distribution and mapping among UEs has a major impact on computational load balance and communication costs among different UEs. An optimal data distribution balances the computational load and minimizes the communication among UEs. For example, a block or cyclic distribution can be efficient for parallelizing regular grid applications. The computational load balance is a key aspect that influences the performance of parallel programs significantly, since the execution time of a program is determined by the slowest UE. In addition, for adaptive applications, the computational load varies throughout the evolution of the solution, which needs further data redistribution in order to keep load balancing.

Current parallel systems consist of a hierarchical memory architecture, which means that accessing a lower layer imposes typically higher latency and lower bandwidth. Therefore, it is important to have good locality in the computation and require memory optimizations. For example, blocking iterations is applied on the loop level to optimize cache behavior. Memory reallocation and array re-indexing are applied for irregular applications in order to reduce non-local accesses among sockets on a single computing node. In addition, communication optimization can improve the memory locality among different nodes with independent address space.

In most practical applications, UEs are not able to execute independently, there must be synchronization and communication among UEs. Synchronization refers to the idea that multiple UEs are to join up at a certain point to reach an agreement or commit to a certain sequence of actions. Communication happens when different UEs exchange data for their local computation. On shared memory systems, the communication among UEs is done by accessing shared variables, while on distributed memory systems, it is handled by explicit message passing.

Programming productivity is decreasing because of the growing complexity of supercomputers and parallel applications. For example, MPI is a portable and efficient library that can be ported on almost all types of supercomputers. However, the users have to manage all the implementation details including data distribution, balancing computational load, explicit communication among processes using MPI operations, synchronization among processes, and many more. It is usually time consuming and error-prone to implement and optimize MPI programs. Instead, OpenMP is much easier to program, but it is only portable on shared-memory systems, and its support for aggregating tasks is limited to the block-based loop scheduling strategies.

In addition, a large number of high-level programming models have been developed to improve the programming productivity and implementation efficiency as well, e.g., High Performance Fortran (HPF)[23][24], Charm++[25][26][27], and Threading Building Blocks (TBB)[28][29]. All these high-level programming approaches are designed to obtain better programming productivity using higher level abstraction or automatic parallelization. In HPF, programmers can express data distributions and the compiler is responsible for generating appropriate message passing programs with MPI automatically. Although programming in HPF was much easier for application developers, the performance of many applications, especially irregular grid applications, was not as good as expected. Charm++ is a parallel object-oriented programming language based on C++. Charm++ provides message-driven objects that encapsulate computation and communication in a proper granularity. However, it is still the programmers' task to manage the granularity of these objects in order to achieve excellent performance. TBB is a C++ template library developed for writing programs that take advantage of multi-core processors. It allows programmers to avoid some complications arising from manipulating threads. Instead TBB abstracts access to multiple processors by allowing the operations to be treated as "tasks", which are allocated to individual cores dynamically by its run-time engine.

1.3. Regular Applications and Irregular Applications

Applications with regular computational models are called regular applications, e.g., linear algebra computation, matrix to matrix calculations, matrix-based iterations, and so on. For example, a CFD (computational fluid dynamics) solver is a typical computational

kernel for simulating many scientific problems[30][31]. The parallel implementation of such applications is to decompose the whole matrix into multiple sub-matrices in a block-wise fashion. Each UE is responsible for the computation of a sub-matrix and requires boundary information from neighbor sub-matrices. The storage layout of the data in the memory is typically continuous and the memory access pattern is relatively regular. Thus, regular applications can be efficiently parallelized with current programming approaches.

On the other hand, irregular applications are relatively difficult to parallelize in an efficient and scalable fashion due to irregular communication patterns, load imbalance issues, and dynamic characteristics, and so on. Figure 1.1 presents the status of an N-body cosmological simulation[32] and an airplane simulation based on irregular grids[33]. N-body simulations[34] and irregular grid-based simulations are typical irregular applications. For example, a molecular dynamics (MD) simulation[35] is a form of N-body computer simulation in which molecules interact with other molecules within a certain domain for a period of time. The molecules may move in the domain according to the interactions with others, which changes their storage layout and communication pattern during execution.

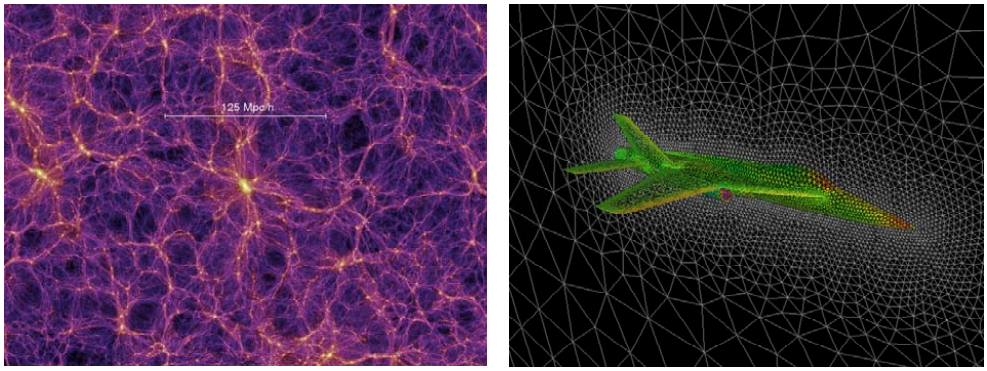


Figure 1.1.: Irregular applications

Irregular applications are difficult to implement and optimize on distributed memory systems in terms of decomposing the computational domain, managing irregular communication patterns among processes, and manipulating data migration among processes, maintaining computational load balance, and so on. In addition, the performance is not ideal because of non-continuous memory access patterns, array indirect accessing, and inefficient memory locality, and etc.

1.4. Ensemble-based Programming Approach

This dissertation presents an ensemble-based programming scheme, on which programmers are able to implement parallel scientific applications in a fine granular and SPMD

(single program multiple data) fashion. The major objective is to improve the programming productivity. Different from current programming models starting from the global data structure, this programming scheme provides a programming interface that supports writing applications by focusing on the finest granular elements and their interactions. These fine granular elements are organized as an ensemble, which manages the elements, their topologies, and high-level operations. By using the high-level operations explicitly, developers can control the actions of the elements including communication, synchronization, parallel operations, and so on. The ensemble-based programming scheme consists of an abstract machine model, programming interface, and implementation framework ported on sequential, shared memory, and distributed memory systems.

The machine model is an abstract architecture consisting of a control processor (CP) and a large number of fine granular processors (FGPs). The execution model of the machine is similar to the SIMD model. The CP issues a "single instruction", which represents a specific parallel operation on FGPs. Multiple FGPs then perform coarse-grained computations specified by the programmers in parallel.

The programming interface supports an object-oriented programming (OOP) approach implemented in C++ to specify the software entities including elementary points (EPs), the ensemble, and topologies. It consists a template hierarchy formed of multiple predefined C++ class templates. The top level of the template hierarchy includes three base templates namely *ElementaryPoint*, *Ensemble*, and *Topology*. These templates have derived templates targeting to three application areas, i.e., multi-body, irregular grid, and regular grid applications. It can be extended to other application areas by adding new domain-specific templates into the template hierarchy. User-defined entities with local attributes and operations can be defined as C++ classes derived from the templates. The master thread is expressed in the main function of an ensemble-based program using C++ syntax. It controls the behavior of the EPs in the ensemble by calling high-level operations defined in the template hierarchy.

In order to implement scientific data parallel applications, ensemble-based programming primarily includes defining the EPs, specifying the EPs' topologies, inserting the EPs and the topologies into the ensemble, creating the master thread that manages the computation and communication of the EPs using predefined high-level operations, and so on. Take an MD simulation as an example. A programmer defines local attributes and computation on the level of molecules and specifies the topologies according to the interaction of the molecules. After the molecules and topologies are inserted into the ensemble, the programmer can apply high-level ensemble operations to manage communication and parallel operations of the molecules automatically.

1.5. Implementation Framework of Ensemble-based Programming

The basic idea of the ensemble-based implementation framework is to aggregate fine granular EPs into appropriate blocks that are bound to threads or processes based on the description of the hardware. The EP-to-EP communication is coarsened to the communication among blocks accordingly. The implementation framework consists of machine-specific libraries, i.e., a sequential library, an OpenMP-based library, and an MPI-based library. They are currently designed for both multi-body and irregular grid applications. A platform-independent and ensemble-based program can be translated into different executables by linking these libraries with different compiler commands and options. The executables run on different target machines including sequential, shared memory, and distributed memory machines.

The sequential library is a standard OOP implementation of the programming interface. Both the communication and parallel operations of the EPs in the ensemble are handled by a single-threaded process. It is designed to demonstrate the basic implementation of the programming interface on one process.

The OpenMP-based library implements the programming interface on top of OpenMP. It translates an ensemble-based program to an OpenMP program that is executed on NUMAs. The communication and parallel operations of EPs in the ensemble are done by multiple threads in parallel. The computation of a group of EPs is aggregated and bound to a single thread and the communication among EPs is handled by accessing the shared memory. In order to minimize non-local access, the OpenMP-based library employs an EP reallocation strategy combined with re-indexing for irregular grid applications.

The MPI-based library implements the programming interface in C++ with MPI. It employs optimized EP distribution strategies to distribute computational workload across multiple processes. The communication among processes is optimized by aggregating fine granular communication among EPs into coarser MPI messages. The communication pattern among EPs is determined by topologies, which are managed by the MPI-based library automatically. For multi-body applications, the MPI-based library employs both the domain decomposition and the parallel linked cells (PLC) algorithm for EP distribution and communication management; while for irregular grid applications, it applies graph partitioning algorithms to achieve optimal EP distribution and communication efficiency.

We ported the implementation framework on SuperMUC[36], the petascale supercomputer at LRZ (Leibniz Supercomputing Centre) in Germany, and tested it with irregular applications including an irregular grid application and an MD simulation. The experimental results show that the execution of these applications on the implementation framework is a bit slower than the manual implementations using C++ with OpenMP

or MPI. However, it improves the programming productivity in terms of the source code size, the coding method, and the implementation difficulty. Compared to current parallel programming models, the ensemble-based programming scheme manages the granularity of computation, data distribution, communication and load balance for irregular applications with reasonable overhead.

1.6. Main Contributions

The main contributions of this dissertation are presented as follows:

1. This dissertation introduces a straightforward and efficient programming scheme. It is a fine granular and ensemble-based programming scheme, which is different from standard programming approaches starting from the global data structures. Based on this ensemble-based programming interface, programmers only focus on specifying EPs and their topologies without worrying about data distribution or communication.
2. The ensemble-based programming interface is implemented in an OOP fashion. It consists of the template hierarchy having domain-specific templates for irregular grid application and multi-body application areas. It is feasible to extend the programming interface to support other application areas by adding new domain-specific templates into the template hierarchy.
3. The ensemble-based program is platform-independent. It can be translated to the codes run on different types of architectures by linking to different domain-specific template libraries. The aggregation of the EPs' communication and computation is done by the programming scheme automatically according to the descriptions of different architectures.
4. An implementation framework of the ensemble-based programming is ported on SuperMUC, and irregular grid and multi-body applications have been tested on the framework. The experimental results show that the ensemble-based programming approach can be implemented on standard systems including sequential, shared memory, and distributed memory systems. The implementation overhead originating from ensemble-based programming is reasonable and acceptable compared to manual implementations using MPI or OpenMP.

1.7. Dissertation Structure

The rest of this dissertation is organized as follows: chapter 2 introduces different types of parallel architectures, i.e., shared memory, distributed memory, hybrid, and heterogeneous architectures. Additionally, multiple standard programming interfaces and libraries are designed to utilize the computing power of such architectures. Pthreads,

OpenMP, MPI, and CUDA are discussed in this chapter; chapter 3 mainly introduces parallel application areas, which are roughly classified into regular and irregular applications. Regular applications include linear algebra calculations and regular grid computations. Irregular applications mainly include irregular and adaptive grids applications, N-body simulations, MD simulations, and so on; chapter 4 introduces related work on other state-of-art parallel programming models. For example, programming models like HPF, TBB, Charm++ are designed to provide a higher level programming environment to the developers; chapter 5 presents the specification of the ensemble-based programming scheme. It introduces an abstract machine model, the programming paradigm, the programming interface, and the template hierarchy. At the end of this chapter, a pseudo ensemble-based code of a simple MD simulation is demonstrated in order to show the productivity of the ensemble-based programming; chapter 6 presents the implementation framework of the ensemble-based programming. It demonstrates this programming approach can be implemented on different systems including a sequential, shared memory, and distributed memory system; in chapter 7, we choose an irregular grid application and an MD simulation to evaluate on the implementation framework on SuperMUC. The experimental results show that the overhead originating from the ensemble-based programming is acceptable compared to the manual implementations; chapter 8 gives the conclusion of this dissertation and future work.

2. Parallel Architectures and Programming Models

2.1. Overview

Parallel architectures are now ubiquitous in different kinds of mainstream parallel systems, e.g., desktops, servers, supercomputers, and so on. The computer architectures are typically classified by Flynn's taxonomy proposed by Michael J. Flynn[37] based on the number of concurrent instructions and data streams available in the architectures. These classes are Single Instruction Single Data stream (SISD), Single Instruction Multiple Data streams (SIMD), Multiple Instruction Single Data stream (MISD) and Multiple Instruction Multiple Data streams (MIMD). Their definitions are showed as follows:

1. SISD: It is an architecture that exploits no parallelism in either instructions or data streams. A uni-processor system like an old PC is a typical SISD architecture, which is not quite often used nowadays.
2. SIMD: It is an architecture that exploits multiple data streams against a single instruction stream to perform operations that are parallelized. For instance, vector machines or GPGPUs are typical representatives of SIMD architectures.
3. MISD: It currently doesn't exist.
4. MIMD: It is an architecture that consists of multiple processors simultaneously execute different instructions on different data is a MIMD architecture. Almost all current parallel systems or supercomputers are MIMD architectures.

In this chapter, four different types of parallel architectures belonging to SIMD and MIMD are described based on the classification system. These parallel architectures are:

1. Shared memory architecture: It is an architecture with a global shared memory, physically distributed or not, where all processors have full access.
2. Distributed memory architecture: It is an architecture where there is no global shared memory. Each node in distributed memory architectures has its own private memory and use explicit message passing for communication among nodes.
3. Hybrid architecture: It consists of clusters of nodes with separate address spaces. Each node is a shared memory architecture with a number of processors connected with a global shared memory.

4. Heterogeneous architecture: It is a parallel computing platform with different types of computational units including CPUs, GPGPUs, or other accelerators.

The hierarchy of Flynn's taxonomy and different kinds of architectures is shown in Figure 2.1.

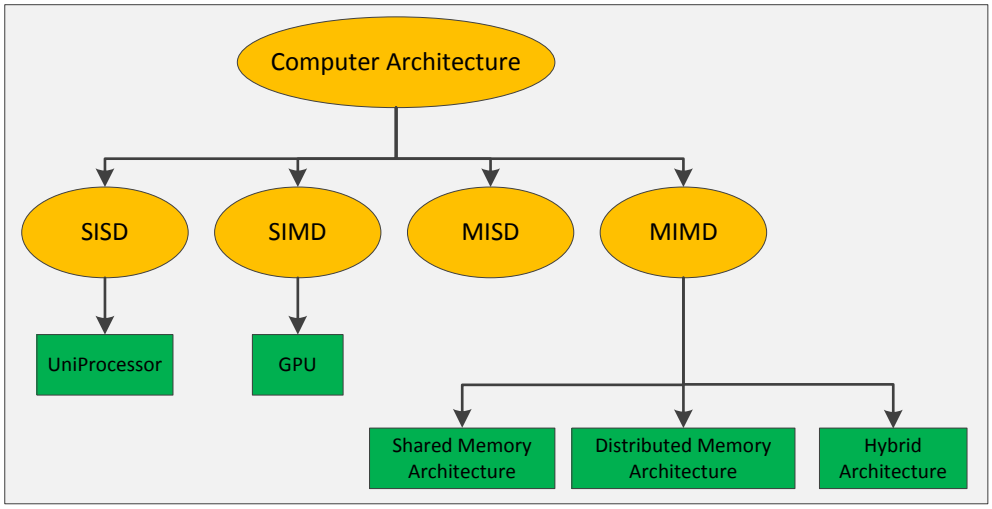


Figure 2.1.: Hierarchy of Flynn's taxonomy and parallel architectures

With respect to programming these parallel architectures, there are many programming platforms and libraries designed to efficiently utilize their computing power. In this chapter, four basic programming platforms are introduced, i.e., Pthreads, OpenMP, MPI, and CUDA. Pthreads and OpenMP are parallel programming libraries, which represent thread-based and directive-based programming approaches. Pthreads is a POSIX standard for threads and an API for creating and manipulating threads. OpenMP is an API that supports shared memory multi-threaded programming in C, C++, and Fortran. MPI is a parallel library designed for explicit communication among processes. It is implemented on both distributed memory architectures and shared memory architectures. CUDA is a parallel computing architecture developed by Nvidia for graphics processing and scientific computing on GPUs. It can be applied as a programming platform on heterogeneous architectures with CPUs and GPUs.

The rest of this chapter is organized as follows: Section 2.2 introduces four different parallel architectures namely the shared memory architecture, distributed memory architecture, hybrid architecture, and heterogeneous architecture with CPUs and GPUs; Section 2.3 introduces programming such architectures using four basic parallell programming platforms, which are Pthread, OpenMP, MPI, and CUDA. Section 2.4 summarize these architectures and parallel programming platforms introduced in the chapter.

2.2. Parallel Architectures

2.2.1. Shared Memory Architectures

MIMD architectures based on Flynn's taxonomy consist of two classes according to their memory organization. One class of MIMD architectures is called shared memory architecture. The shared memory architecture has a global shared memory, which can be accessed by all processors in the architecture. The processors are connected using buses or point-to-point networks. The communication among processors is established by writing to and reading from shared locations in the memory.

The global shared memory is either physically shared or distributed among processors. If the global shared memory is physically shared, the processors in the architecture have the same access time to the memory. If the global shared memory is physically distributed, the memory access time depends on the memory location relative to processors. According to the access time, shared memory architectures can be divided into two categories, which are Uniform Memory Access (UMA) architectures and Non-Uniform Memory Access (NUMA) architectures.

- UMA: All processors in the UMA architecture physically share the global shared memory uniformly.
- NUMA: All processors in the NUMA architecture share the global shared memory, but the access time to the memory is non-uniform.

2.2.1.1. UMA

One class of shared memory architectures is the UMA, which is also called a Symmetric Multiprocessor (SMP). The UMA has global shared memory among processors, which have the same access time to the shared memory. Based on the UMA architecture, programmers do not need to distribute data structures among processors. Multiple processors symmetrically access data in the global shared memory and execute on the data in parallel, which accelerates the execution of programs.

However, increasing the number of processors increases contention for the memory, and the processor to memory bandwidth becomes a limiting bottleneck. Thus, traditional shared memory architectures do not scale well to a large number of processors, which may typically have saturation problems beyond 8 or 16 processors. Current multi-core processors are based on the UMA architecture. Its basic structure is shown in Figure 2.2.

2.2.1.2. NUMA

The other class of shared memory architectures is called the NUMA architecture. The NUMA architecture has a global shared memory among processors like UMA does, but

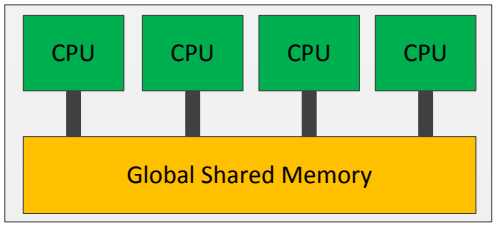


Figure 2.2.: UMA architecture

the memory is physically distributed among the processors in the architecture. The access time to the global shared memory depends on the memory location relative to processors. A processor can access its own local memory faster than non-local memory, which is local to another processor.

The global shared memory is uniformly addressable from all processors, but some blocks of memory are physically more closely associated with some processors than others. This reduces the memory bandwidth bottleneck and allows systems with more processors. However, the access time from a processor to a memory location can be significantly different depending on how "close" the memory location is to the processor.

To mitigate the effects of non-uniform accesses, each processor has a cache, along with cache coherency protocols to keep cache entries coherent. Hence, another name for these architectures is a cache coherent Non Uniform Memory Access architecture (cc-NUMA). Logically, programming a ccNUMA architecture is the same as programming an UMA/SMP, but to obtain good performance, programmers need to be more careful about locality issues and cache effects.

2.2.2. Distributed Memory Architectures

The distributed memory architecture is a message passing architecture that uses scalable point-to-point networks to exchange data. There is no global shared memory among all processors, since each processor has its private memory address space and can only access its own local memory. It means that communication and synchronization among processors is performed by message passing. As a result, communication is not transparent, programmers must explicitly program all the communication between processors according to certain data distribution. This is because all system resources, like memories, disks, and so on, are distributed and only private to the local processor. Distributed memory architectures can scale to large numbers of processors using scalable networks.

2.2.3. Hybrid Architectures

The hybrid architecture is a cluster of nodes with separate address spaces, in which each node is a shared memory architecture with a number of processors. The distributed

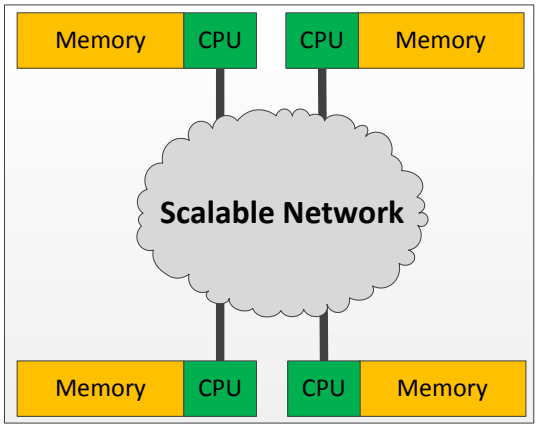


Figure 2.3.: Distributed memory architecture

nodes in the hybrid architecture are connected by scalable networks. Each node has its private memory and can only access its own local memory. The communication among processors within a node is created by writing to and reading from shared memory locations, while the communication between nodes is established by explicit message passing. Programmers have to use communication operations to exchange data between nodes. Figure 2.3 shows the structure of a hybrid architecture. For example, SuperMUC[36] is a hybrid architecture supercomputer, which is based on the Intel Xeon processor nodes and consists of more than 150,000 cores with a peak performance of about 3 Peta FLOPS.

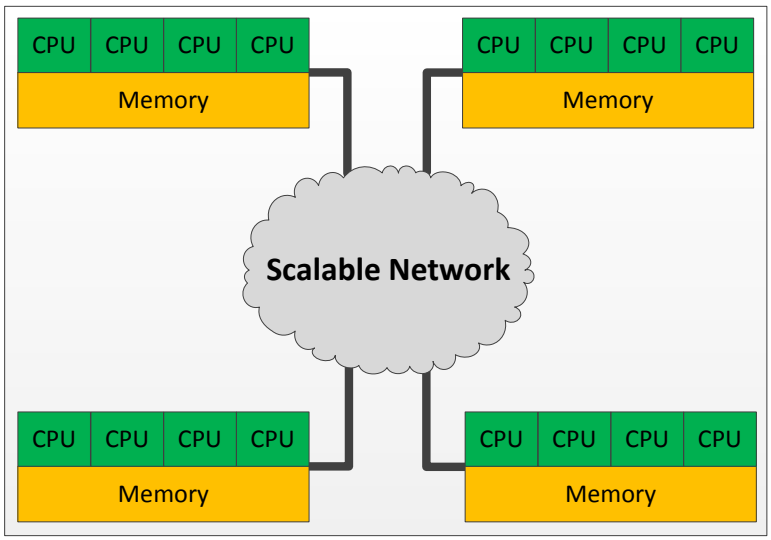


Figure 2.4.: Hybrid architecture

2.2.4. Heterogeneous Architectures with CPU and GPU

The heterogeneous architecture is a parallel computing platform with different types of computational units. A computational unit can be a general-purpose processor, a special-purpose processor, or custom accelerators, e.g., a Digital Signal Processor (DSP), an Application Specific Integrated Circuit (ASIC), a Field Programmable Gate Array (FPGA), or a GPGPU.

In this chapter, the heterogeneous architecture with CPUs and GPUs is introduced. A CPU usually consists of two to eight thread-based cores, while a GPU consists of hundreds of multi-threading, in-order and single-instruction issued streaming cores that share the control and instruction cache with other cores. The CPU is responsible for basic executional functionalities, memory management, and triggering computational workload on the GPUs. The GPU is an accelerator that handles computing tasks assigned by the CPU, which would take longer time to perform on the CPU. With specialized hardware and software support, GPUs can handle computations not only for graphics, but also general computations that are typically done by CPUs. These types of GPUs are called GPGPUs. The heterogeneous architecture obtains better performance for some applications, since the GPGPUs can accelerate calculation-intensive and time consuming applications running on the CPU. The heterogeneous architecture with a CPU and a GPU is shown in Figure 2.5.

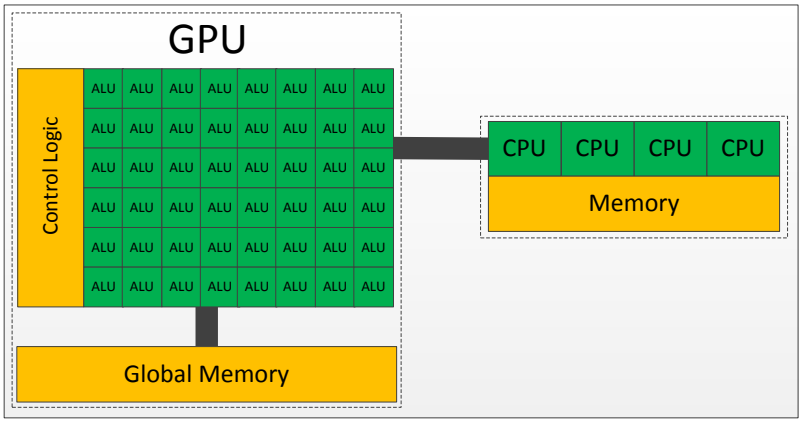


Figure 2.5.: Heterogeneous architecture

2.3. Programming Different Parallel Architectures

2.3.1. Overview

This section describes how to program the parallel architectures introduced in Section 2.2 in order to utilize their computing power efficiently. Multiple programming models,

platforms, and libraries are designed to support specifying threads, processes, and their interactions, e.g., sharing data among threads within a process, communication among processes, and so on. In this section, we introduce some basic parallel programming models and libraries extensively applied on shared memory architectures, distributed memory architectures, hybrid architectures, and heterogeneous architectures.

2.3.2. Programming Shared Memory Architectures

On a shared memory architecture, the parallel UEs are threads, which share the global memory. In order to accelerate parallel applications on such architecture, multiple threads within one process are created and spawned on different cores automatically. Programmers are responsible for specifying threads and their interactions using thread-based programming interfaces like Pthreads or OpenMP.

On the other hand, the parallel UEs on shared memory architectures can also be processes. The communication among the processes is specified by MPI, which is a language-independent protocol as well as an implementation for inter-process communication. MPI is briefly discussed in this subsection and details of MPI will be described in Subsection 2.3.3 based on the scenario of distributed memory architectures.

2.3.2.1. Pthreads

Pthreads[38] are defined as a set of C/C++ language programming types and procedure calls implemented with the pthread.h header file and a thread library. There are around 100 standard Pthreads procedures, which are responsible for thread management, creating and joining threads, mutex condition and synchronization between threads, and so on. It allows programmers to specify multiple threads of execution scheduled across different cores in order to gain speedups. Typically, creating threads requires less overhead than forking new processes on a shared memory system.

As it is shown in Listing 2.1, this simple code creates 4 threads with the pthread_create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit(). These threads are executed in parallel by multiple cores, which shorten the execution time compared to a sequential execution.

Listing 2.1: A sample Pthreads program

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 4

void *PrintHello(void *threadid) {
   //Specify what a thread does
```

```
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
    //Create a thread array
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        //Run all the threads
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t)
            ;
        if (rc){
           printf("ERROR; return code from pthread_create() is %d\n",
                rc);
           exit(-1);
        }
    }
    //Exit thread library
    pthread_exit(NULL);
}
```

2.3.2.2. OpenMP

OpenMP[39] is a standard API that supports the development of parallel applications on shared memory architectures ranging from standard desktops to supercomputers. It uses a portable and scalable model that provides programmers with flexible directive-based interfaces. OpenMP is combined with C, C++, and Fortran to create a multi-threading programming language. It is the goal of OpenMP's creators to make OpenMP easy for application developers to handle.

OpenMP is based on the assumption that the UEs are threads that share a global address space. The execution model of OpenMP is based on the fork/join programming paradigm. As can be seen from Figure 2.6, an OpenMP program starts with a single master thread and forks additional threads to form a team of threads at certain points where parallel execution is desired. A section of code that execute by the team of threads is called a parallel region. At the end of a parallel region, the threads wait until the full team arrives and joins back together. At that point, the original master thread continues until the next parallel region or the end of the program.

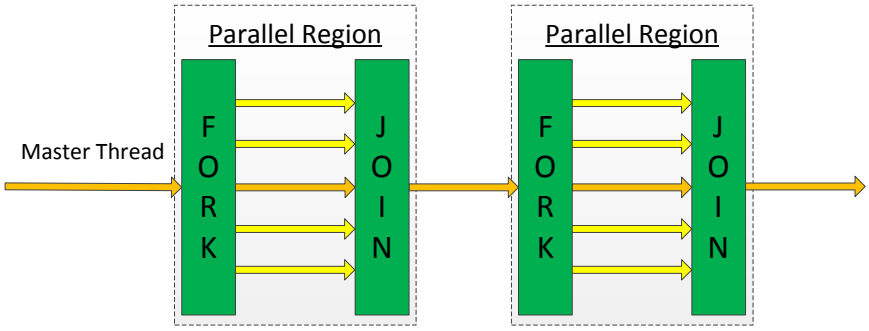


Figure 2.6.: OpenMP execution model

The parallelism is exploited by parallel loops using OpenMP compiler directives. The compiler generates code to execute the iterations of the loops in parallel according to the directives. Each thread is responsible for the computation of a subset of iterations. The thread creation and management are done by the OpenMP runtime system automatically.

OpenMP programs tend to work efficiently on SMPs because of their uniform memory access pattern. However, it is less efficient on ccNUMA without special optimizations, since its underlying programming model does not include a notion of non-uniform memory access time.

2.3.2.3. MPI

MPI is typically designed for distributed memory architectures, it can also be applied as a programming interface on shared memory architectures. The basic UEs are MPI processes. The global shared memory is logically partitioned among MPI processes, each of which keeps its own logical memory. The communication between MPI processes can be done by copying data in the global shared memory rather than transferring messages between processes. Although forking new processes usually requires more overhead than creating threads on shared memory architectures, multi-process program can achieve good performance, since it avoids expensive locks and synchronization among threads and maintains the data locality.

2.3.3. Programming Distributed Memory Architectures

The UEs on distributed memory architectures are processes. Each process keeps its independent memory space on a single processor and the communication among processes is handled by the underlying parallel architectures. MPI is a standard programming interface for distributed memory architectures. MPI is a set of library routines provided for

process management, message passing, and some collective communication operations. It is a standard programming library used in Fortran, C and C++.

The central construct in MPI is point-to-point communication. Its semantics is that one process packages information into a message and sends this message to another process. The other process receives the message and puts it into the allocated local memory for local computations. Except for this simple point-to-point message passing, MPI includes routines to synchronize processes, collective operations among a group of processes, e.g., scattering and gathering data across a group of processes, and many more. Figure 2.7 shows the semantics of different MPI collective operations including broadcast, reduction, scatter and gather operations.

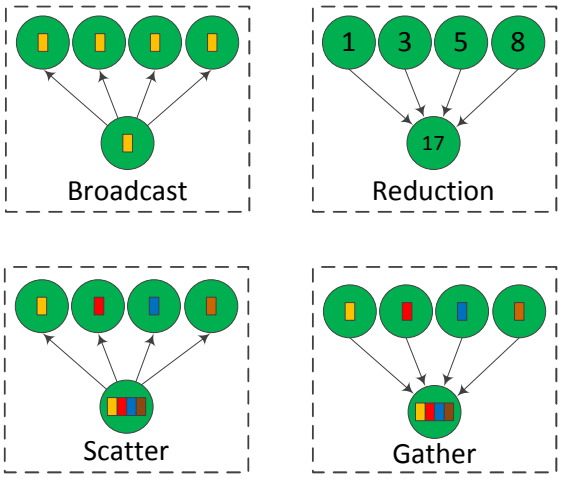


Figure 2.7.: MPI collective operations

The major drawback of MPI is the programming productivity, since MPI programs are usually difficult to implement and optimize. Experienced programmers have to take care of all the implementation details including data distribution, explicit inter process communication using MPI operations, synchronization among processes, and so on.

2.3.4. Programming Hybrid Architectures

As described in Subsection 2.2.3, a hybrid architecture consists of nodes with separate address spaces, each node is a shared memory architecture with several powerful CPUs. Neither OpenMP nor MPI is an ideal programming paradigm for hybrid architectures. OpenMP is only portable on a single node and doesn't support inter-node communication. Additionally, a single node might not be able to run as many processes as cores because of memory size restriction on the node. The solution is hybrid programming in which OpenMP is used on each shared memory node and MPI is applied among the nodes. Communication only happens among nodes, which reduces the number of mes-

sages and improves the communication efficiency. This works well, but it requires the programmer to work with two different programming models within a single program. The execution model of hybrid OpenMP/MPI programming is shown in Figure 2.8.

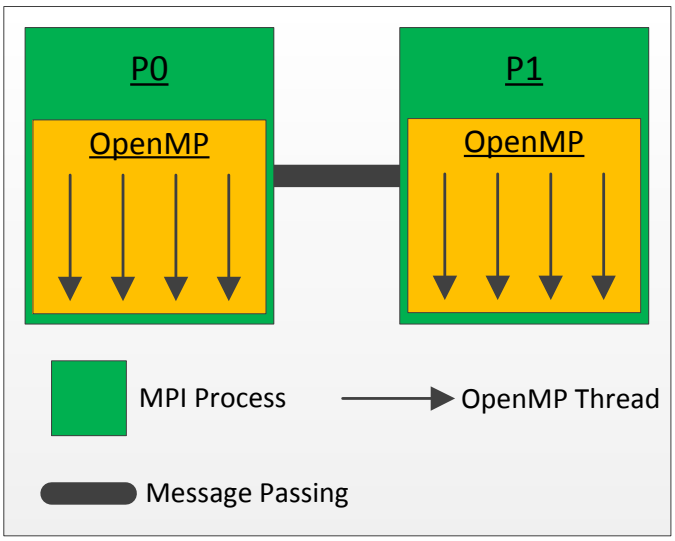


Figure 2.8.: Hybrid MPI/OpenMP execution

2.3.5. Programming Heterogeneous Architectures with CPU and GPU

In modern scientific computing applications, programs often exhibit a rich amount of data parallelism allowing many arithmetic operations performed in a simultaneous manner. OpenMP can only exploit data parallelism with tens of threads due to thread management overheads and cache coherence hardware requirements. Therefore, CUDA[40] with the simple thread management mode is designed to harvest a large amount of data parallelism on Nvidia GPGPUs. CUDA achieves much higher scalability with simple, low-overhead thread management and no cache coherence hardware requirements. CUDA is easy to learn, so programmers who have MPI and OpenMP background can handle it quickly. Especially, many of the performance optimization techniques are common among these models. Programmers can use C with CUDA extensions and certain restrictions (C CUDA) to code scientific applications. Many highly scalable applications[41][42][43] fit well into the simple thread management model of CUDA and thus enjoy the scalability and performance.

2.4. Summary

This chapter gives an overview of some mainstream parallel architectures, i.e., the shared memory, distributed memory, hybrid, and heterogeneous architecture. These architec-

tures are ubiquitous and most of current supercomputers are built based on them. Four major programming interfaces and libraries are designed in order to program such parallel architectures namely Pthreads, OpenMP, MPI, and CUDA. These programming interfaces are widely used on almost all kinds of parallel architectures. In addition, some specialized and high-level programming approaches will be discussed in Chapter 4.

3. Parallel Application Areas

3.1. Overview

This chapter introduces parallel application areas including linear algebra, regular grid, irregular grid, adaptive grid applications, and multi-body applications. Linear algebra includes calculations between matrices and vectors, which are typical regular computational models. In order to solve ordinary differential equations (ODEs) and partial differential equations (PDEs), the finite element method (FEM) discretizes a computational domain into grids of elements, which are roughly classified into irregular and adaptive grids. In addition, the other mainstream application area is N-body applications, e.g., astrophysics, cosmological simulations and MD simulations.

The rest of this chapter is organized as follows: Section 3.2 introduces the overview of linear algebra calculations; Section 3.3 describes regular grid applications including the 2D heat distribution based on Jacobi kernel and cellular automaton; Section 3.4 introduces irregular grid applications based on the FEM; Section 3.5 introduces adaptive grid applications; Section 3.6 describes the basic theory of N-body applications and some typical parallelization algorithms for cosmological and MD simulations.

3.2. Linear Algebra

Linear algebra plays an important role in many scientific domains and application areas. It constitutes kernel operations for computations not only in computer science, but also in natural science, engineering, and computer graphics[44][45][46]. In the supercomputing area, a well-known benchmark called Linpack[47] has been used for evaluating high-performance systems since 1970s. It includes all the typical linear algebra problems, e.g., vector-vector operations called DAXPY, matrix multiplications, matrix eigenvalue problems, least-squares solutions of linear systems of equations, and so on.

Take the matrix multiplication as an example, it is a classical linear algebra calculation in many scientific areas. A parallel matrix multiplication takes advantage of the computing power of parallel computers to improve its performance. The fundamental idea of a parallel implementation is to distribute the computational load across available UEs and maintain the communication among the UEs. It is assumed that the number of blocks distributed among the UEs determines the computational load of the UEs.

Therefore, the matrices have to be divided into blocks of almost equal size in different ways, which are row-wise, column-wise and square block-wise decomposition. Based on these decomposition methods, the communication pattern among processes is regular and relatively easy to specify by programmers. There are a lot of publications about parallelizing matrix multiplication like [48][49][50].

3.3. Regular Grid Applications

A regular grid is an N-dimensional Euclidean domain consisting of elements with regular shape. Each element is addressed by its coordinate in the domain, e.g., (i, j) in a 2D domain and (i, j, k) in a 3D domain. Regular grid applications are based on regular grids. They usually update the state of the elements in the domain iteratively according to a deterministic neighborhood. Each element in a regular grid is able to access its neighbor elements by referencing their coordinates.

In the computational science community, there are a lot of regular grid applications including an ocean circulation model[51], computational electromagnetics simulations[52], image-processing applications[53], 3D heat distribution based on the Jacobi iteration[54], and so on. This section briefly describes a 2D heat distribution and cellular automaton[55].

3.3.1. 2D Heat Distribution

The 2D heat distribution is solved with the Jacobi kernel. It updates the values of the points on a regular grid according to a certain neighbor relation. The update function computes the arithmetic mean of a point's neighbors. The neighbors of point (i, j) are the points $(i, j-1)$, $(i, j+1)$, $(i-1, j)$, $(i+1, j)$, which are called Von Neumann neighborhood[56]. The value of a_{ij} at iteration step $N+1$ is calculated according to the values of its "neighbors" at iteration step N, it is described as follows:

$$a_{ij}^{N+1} = 0.25 * (a_{i(j-1)}^{N} + a_{i(j+1)}^{N} + a_{(i-1)j}^{N} + a_{(i+1)j}^{N})$$

The iterative computation stops only when either the number of iterations has reached the maximum number or the residual of a_{ij} between two time steps is smaller than a certain threshold.

3.3.2. Cellular Automaton

A cellular automaton consists of a number of cells based on a regular grid. Each cell has its own attributes, e.g., the coordinates, the local states, and so on. The local attributes of a cell are updated according to its neighbor cells kept at certain locations of the regular grid. Take a 2D grid as an example. The neighbors of cell (i, j) can be cells

$(i-1, j-1)$, $(i-1, j)$, $(i-1, j+1)$, $(i, j-1)$, $(i, j+1)$, $(i+1, j-1)$, $(i+1, j)$, $(i+1, j+1)$, which are called Moore neighborhood[57]. Similar to the Jacobi method, all the cells update their local attributes iteratively from an initial state. The local attributes of a cell at time step $N + 1$ are determined by the attributes of its neighbors at time step N according to user-defined computational rules. Typically, the computational rules are the same for the entire cells and do not change during their evolution.

3.4. Irregular Grid Applications

Irregular grids are widely used to represent complicated domains. Irregular grids are different from regular grids, where the connectivity between elements must be explicitly defined. It is more flexible to define complex shapes using irregular grids because they have no constraints on their arrangement. However, the interconnection of the elements has to be explicitely defined.

In the scientific computing area, PDEs are used to describe the underlying dynamics of a wide variety of applications including heat dissipation, electro-dynamics, fluid dynamics, and so on[58][59]. The FEM is an important numerical technique for finding approximate solutions to PDEs. In simple terms, FEM is a method for dividing up a complicated problem into small ones that can be solved. Its central idea is to discretize a complicated domain into an irregular grid of many individual elements. The local values of the elements are updated iteratively according to certain computational rules and the topology. Different from regular grids, the neighbor elements cannot be referenced by their indices but through one level of indirect accessing the topology. This characteristic prevents automatic compiler strategies and makes irregular grid applications relatively difficult to parallelize.

3.5. Adaptive Grid Applications

An adaptive grid can be regular or irregular grids with adaptiveness. The granularity of the elements in an adaptive grid usually changes during the evolution of a solution. It may start with a base coarse grid. As the solution proceeds, if the regions require more resolution by the parameters characterizing the solution, finer and finer sub grids are added in a certain fashion to increase the accuracy of the solution. Take a car crash simulation as an example, the areas near the crash require more computation than other parts. Thus, the granularity of elements in such key areas could be much finer than those in minor areas in order to get more accurate results.

The SAMRAI (Structured Adaptive Mesh Refinement Application Infrastructure) library [60] is a framework for exploring application, numerical, parallel computing, and software issues based on regular structured adaptive grids. Its major objective

is to support computational physics algorithm research for adaptive methods on high-performance computing platforms.

Richard I. Klein has used adaptive mesh refinement (AMR) to model a collapsing giant molecular cloud core[61]. This model manages the AMR of 3-D self-gravitational hydrodynamics problems by employing multiple irregular grids at multiple levels of resolution. These grids can be automatically and dynamically added and removed as necessary to maintain the adequate resolution.

3.6. Multi-body Applications

Multi-body applications (N-body) include numerous application areas, such as astrophysics, molecular dynamics, plasma physics, and so on[62][63]. A classical N-body kernel simulates the evolution of a system of many bodies according to their force interactions. The simulation is an iterative method that proceeds over a large number of time steps. Within each time step, the force on the bodies is computed and all the bodies move in the simulation domain according to the equations of motion. From the motion of the bodies, a variety of useful microscopic and macroscopic information can be extracted. This method is deterministic, since the state of a molecule system is determined at any time if the initial state of the bodies is specified. A typical N-body simulation method is based on the Newton's second law and the equations of motion:

$$\overrightarrow{F} = m\overrightarrow{a}$$

$$\triangle S = \triangle v_0 t + \overrightarrow{a}t^2$$

where $\overrightarrow{F}$ is the force exerted on a body, m is its mass and $\overrightarrow{a}$ is its acceleration, $\triangle S$ is its displacement within a time step, v_0 is its initial velocity and t is the time length of a time step. Integration of the equations of motion determine the accelerations, velocities and displacements of the bodies as they vary in each time step. If all pairwise forces are computed directly, this requires $O(N^2)$ operations at each time step.

3.6.1. Cosmological Simulation

Cosmological simulations are based on the N-body kernel with all pairwise interactions between the bodies. It is very expensive to compute such simulations if the number of bodies is enormous. Thus, a lot of approximation methods are used to improve the computational efficiency[64]. Josh Barnes and Piet Hut published a paper and presented a classical approximation using a tree-structured hierarchical algorithm[64]. It scales the computational complexity from $O(N^2)$ to $O(N\,logN)$. Many researchers

have applied tree-based algorithms to cosmological simulations. For example, the fast multipole method (FMM)[65] introduced by Greengard and Rokhlinis a mathematical technique that was developed to speed up the calculation of long-ranged forces. The multipole tree algorithm[66] introduced by John Board and William Elliott is a multipole-accelearted algorithm. Parallel multipole tree algorithm developed by the scientific computing group at Duke University is a hybrid of the Barnes-Hut and multipole method[67]. GADGET[68][69] is a freely available framework for cosmological N-body simulations on distributed memory systems. It uses an explicit communication model implemented with the standardized MPI communication interfaces. The code can be run on almost all current supercomputers.

3.6.2. Molecular Dynamics Simulation

The MD simulation is commonly used for simulating the properties of liquids, solids, and molecules[70]. It is a special case of N-body simulations with an approximation based on distance between molecules. Each of the N atoms or molecules in the simulation is treated as a mass point and Newton's equations are integrated to compute their motion. The molecule-to-molecule force mainly consists of the bonded force and non-bonded force. For simplicity, we discuss the non-bonded force only, since the cost of computing the non-bonded force dominates the computational load. Among different models of describing non-bonded force, the Lennard-Jones potential[71] is a simple mathematical model that approximates the non-bonded interaction between a pair of molecules. It is often used to describe the properties of gases. A form of the potential was proposed in 1924 by John Lennard-Jones. The expression of the L-J potential is mathematically described as follows:

$$V_{LJ} = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right].$$

where ε is the depth of the potential well, σ is the finite distance at which the molecule potential is zero and r is the distance between molecules. Then the force can be calculated by the following expression:

$$F_{LJ}(r) = -\frac{\partial}{\partial r} V_{LJ}(r) = 4\varepsilon \left(12 \frac{\sigma^{12}}{r^{13}} - 6 \frac{\sigma^{6}}{r^{7}} \right).$$

The LJ-potential is used extensively in MD simulations even though more accurate potentials exist. To save computational time, the LJ-potential is often truncated at a cut-off radius distance of r_c, where i.e., at r_c the LJ-potential, $V_{LJ_{trunc}}$, is almost equal to 0. Beyond r_c, the truncated potential is set to zero. The truncated LJ-potential is defined as follows:

$$V_{LJ_{trunc}}(r) = \begin{cases} V_{LJ}(r) - V_{LJ}(r_c) & for\ r < r_c \\ 0 & for\ r \geqslant r_c \end{cases}$$

Then the truncated LJ force based on the cut-off radius is described as follows:

$$F_{LJ_{trunc}}(r) = \begin{cases} F_{LJ}(r_c) - F_{LJ}(r) & for\ r < r_c \\ 0 & for\ r \geqslant r_c \end{cases}$$

Based on the truncated LJ force equation, the non-bonded interactions of molecules beyond a certain cut-off radius can be neglected and the computational time scales as $O(N)$ for$N$ molecules better than $O(N^2)$ with pairwise distance calculations.

There are three typical ways to parallelize MD simulations on distributed memory systems[72], which are the atom decomposition[73], force decomposition[74], domain decomposition [75][76][77], and hybrid methods[78]. The simplest way of parallelizing MD simulations is to assign subsets of molecules to different processes. This method is called the atom decomposition method. Each process is responsible for the computation of the local molecules. This method can guarantee that all the processes keep almost the same computational load but the communication complexity is $O(N)$. The force decomposition is based on decomposing the force matrix, which keeps the interactions of all the molecules. The computational load among processes is almost equal, but the communication overhead is still not as good as expected. The domain decomposition is to decompose the spatial simulation domain into sub domains. Each processor is responsible for the computation and communication of the molecules within a subdomain. The communication complexity is $O(N/P)$ for cut-off radius based MD simulations, which is the best among all the decomposition methods. However, the computational load among processes may not be balanced, since it is determined by the distribution of the molecules and their movement patterns.

3.7. Summary

In this chapter, we roughly classify parallel applications into regular applications and irregular applications. Regular applications include linear algebra calculations and regular grid applications. These applications have relatively regular computational models and communication patterns among UEs on distributed memory systems. Additionally, there are irregular applications including FEM based on irregular grids and adaptive grids, N-body simulations, and MD simulations. These applications are difficult to implement and optimize in terms of defining computation, communication, data management, and so on. This dissertation mainly focuses on the iterative method on regular and irregular grids, as well as MD simulations with a cut-off radius.

4. Related Work

4.1. Overview

There are a variety of parallel programming approaches, ranging from standard parallel programming languages to high-level programming environments, e.g., automatic loop parallelization, task-based generic OOP approaches, cross-platform interfaces for heterogeneous architectures, domain specific languages (DSLs), and so on. This chapter briefly introduces high-level parallel programming interfaces namely HPF, GA, TBB, PARTI / CHAOS library, Charm++, UPC, and OpenCL. In addition, we also describe DSLs, which are designed and implemented for specific application areas, e.g., MD simulations, CFD solvers, stencil codes, and so on.

The rest of this chapter is organized as follows: Section 4.2 introduces the overview of some high-level parallel programming languages and libraries; Section 4.3 introduces DSLs designed and optimized for specific application areas;

4.2. Parallel Programming Languages and Libraries

4.2.1. High Performance Fortran (HPF)

HPF[24] is a high-level and data parallel programming model based on Fortran. The main objective of HPF is to provide convenient programming support for scalable parallel systems with an emphasis on data parallelism. In HPF, programmers can express data distributions with a single memory view, and the HPF compiler is responsible for the actual distribution of data and communication among processors. The most important construct in HPF is FORALL, which supports parallel loops. The semantics of the FORALL construct is that multiple processes execute a certain operation on different subsets of arrays in parallel.

Although programming in HPF is much easier for application developers, its support for irregular applications is not very efficient. These applications usually need at least one level of array indirection, whose information can only be obtained at runtime rather than compile time. This prevents the HPF compiler from generating efficient code. Some techniques were developed trying to determine data distributions automatically[79][80][81], but these techniques are still not as effective as expected due to limitations in the information available at compiler time.

4.2.2. Global Arrays (GA)

The GA toolkit[82] provides a shared memory programming environment in the context of distributed memory systems. It provides high-level array data structures called global arrays to support parallel operations on these arrays. Developers can directly program global arrays as if they are stored in shared memory. The details of data distribution and mapping, data migration and accessing are encapsulated in the data structures. GA can be ported both on massively-parallel distributed memory and scalable shared memory systems as well. Similar to HPF, the major drawback of GA is the support for irregular applications.

4.2.3. TBB

TBB[28][83] offers a task-based and object-oriented programming environment for parallel programming of multi-core systems. It is a template library that helps programmers to take advantage of multi-core processor performance without having to be thread experts. Different from data parallel programming, TBB supports task parallel programming that allows developers to encapsulate computational loads into tasks. Based on TBB, programmers need to specify tasks in an application, and the TBB task scheduler is responsible for mapping these tasks onto concrete threads and managing thread operations including the synchronization, load balancing, cache optimization, and so on. In addition, The TBB library automatically creates and manages a thread pool, and it does dynamic load balancing of parallel work.

The major restriction is that TBB cannot be ported on distributed memory architectures. Based on standard C++ without extensions, the template-based approach of TBB is flexible and easy to extend. However, programmers need to have strong background on STL[84][85][86] generic programming and task parallelism, which is quite different from data parallel programming.

4.2.4. PARTI / CHAOS Library

It is difficult to parallelize sparse or irregular problems on distributed memory systems, since the patterns of data access and communication of such problems can only be obtained at runtime. The PARTI/CHAOS library[87][88] is designed to support efficient execution of irregular problems on distributed memory systems. It adds a pre-compute step, called an inspector, before the actual computation happens. The inspector step is responsible for arranging local and remote data, generating communication patterns among processes, translating remote indices to local ones, and so on. The ensemble-based implementation framework reuses this method for the irregular data distribution, the memory arrangement for local and non-local data, and management of communication

among processes on distributed memory systems. Details of the implementation will be discussed in Chapter 6.

In addition to the PARTI / CHAOS library, other runtime compilation methods were developed for a variety of irregular applications including irregular CFD solvers, adaptive grid applications, direct simulation Monte Carlo (DSMC) codes, MD simulations, and so on[89][90][91].

4.2.5. Charm++

Charm++ is a parallel object-oriented programming language based on C++. It is designed to enhance programming productivity by providing a high-level abstraction of a parallel program while delivering good performance on a variety of underlying platforms, e.g., shared memory architectures, distributed memory architectures, even GPGPUs. A scalable and portable parallel MD application called NAMD[92][93] is implemented with Charm++. It can scale to hundreds of cores for typical simulations and beyond 200,000 cores for highly scalable simulations.

A Charm++ program is formed of a number of cooperating message-driven objects called chares. A chare is a C++ object with special language extensions. It has local attributes, member functions, and special functions called entry methods, which are used for communication among chares. Chares that are organized into indexed data structures like arrays or groups are mapped onto threads or processes. The communication among chares is specified by remote access to entry methods. The mapping of chares onto threads or processes is transparent to programmers, and the runtime system is able to change the mapping dynamically due to measurement-based load balancing.

Charm++ is easy to program, and portable to different architectures. It applies an automatic aggregation strategy to bind active objects to a single thread or process. The performance improvement mainly originates from efficient asynchronous communication and dynamic load balancing by the runtime system. However, the granularity of the chares is still managed by the users, since the communication among chares has great impact on the performance of applications.

4.2.6. UPC (Unified Parallel C)

In computer science, a partitioned global address space (PGAS) combines the performance and data locality features of distributed memory programming with the programmability and simplicity features of shared-memory programming. In order to improve the data locality, the shared memory address space is logically partitioned into different portions, which are local to different processes. There are some PGAS languages like UPC[94], Co-array Fortran[95], Titanium[96], Fortress[97][98], Chapel[99], and X10[100].

UPC[94] is an extension of C designed for large-scale parallel systems, including shared memory systems as well as distributed memory systems. The execution model of UPC is based on SPMD. It provides standard library functions to move data between shared and private spaces in the partitioned global address space.

4.2.7. OpenCL (Open Computing Language)

OpenCL[101][102] is an unified programming model for heterogeneous platforms consisting of CPUs, GPGPUs, or other accelerators. There is a remarkable similarity between the key features of OpenCL and CUDA except that OpenCL is more generic for different types of platforms, while CUDA is only for Nvidia GPUs. OpenCL is a cross-platform programming language, which means that programmers can write OpenCL applications once and run them on any OpenCL-compliant hardware. An OpenCL program consists of a host program and kernels. The host program manages a command queue for each device, while the kernels are code executed on OpenCL devices. Similar to CUDA, OpenCL provides APIs for general programming for the host and interfaces for writing kernels. The shared memory between the host and OpenCL devices is managed by the runtime system of an OpenCL implementation.

OpenCL is based on the fine granular programming approach. The kernel of OpenCL is an atomic function that performs a computation on each element of an application domain. The kernels are mapped onto light-weighted threads of devices, which are executed in parallel. For example, a kernel can be an arithmetic operation between two elements of a matrix, or a value update of a single point in a regular grid, and so on. This fine granular approach is also the basic idea of the ensemble-based programming scheme.

In addition, it is worth designing platform-independent or cross-platform programming models, since they can improve the programming productivity and utilize the computing power of different parallel systems. However, it is still a challenge to design and implement cross-platform models due to the increasingly complicated and specialized hardware and software of parallel architectures.

4.3. Domain-Specific Languages

A domain-specific language (DSL) is a type of programming language designed for specific application domains. Compared to general programming languages, DSLs are efficient and easy to use for the application developers for solving a particular type of problems on different computing platforms, e.g., general purpose computing systems, and specialized systems like GPGPUs and FPGAs.

MDL[103], developed by Trevor Cickovski, Chris Sweet, etc., is a DSL for MD simulations. It provides an abstraction of MD-specific entities and algorithms in a simple

fashion, which helps domain experts to prototype MD simulations without knowing the details of computer languages. Its programming interface is based on Python[104], which supports better error-checking and debugging compared to general purpose and other scripting languages.

Liszt[105] is a DSL for solving PDEs on irregular (unstructured) grids. It adds domain knowledge into the high-level language, and the compiler is responsible to map Liszt codes to run on a range of parallel architectures including shared memory architectures, and heterogeneous architectures with CPUs and GPGPUs. For example, it introduces language extensions for specifying mesh elements, sets of elements, their topological relationships, and parallel loops to express parallelism.

Wang Luzhou and Kentaro Sano[106] presented a DSL for stencil computation (DSLSC). Programmers can implement stencil computations by specifying mathematical interactions between elements instead of writing explicit operations based on general languages. Its compiler is able to generate parallelized stencil codes mapped on processing elements of FPGA-based systolic computational-memory arrays (SCMAs).

The Berkeley autotuner[107],focuses on optimizing the performance of stencil kernels by automatically selecting tuning parameters. In order to minimize runtime overhead on current shared memory architectures, it examines a wide variety of optimizations including NUMA-aware allocation, multilevel blocking, loop unrolling and reordering, prefetching, and so on.

The Pochoir[108] stencil compiler allows a programmer to write a simple specification of a stencil in a domain-specific stencil language embedded in C++. The Pochoir compiler then translates the stencil code into high-performing Cilk[109][110] code that employs an efficient parallel cache-oblivious algorithm. It automatically produces a highly optimized, cache-efficient, parallel implementation according to simple functional specification written by the application developers. Pochoir achieves a substantial performance improvement over a straightforward loop parallelization for typical stencil applications, such as the 2D heat equations and Conway's game of Life application[111].

4.4. Summary

This chapter introduces some high-level parallel programming languages for current parallel architectures and some runtime libraries for irregular problems. The major characteristics of these programming models are to enhance the programming productivity and combine the advantages of various programming models. According to the description of these programming models, we can see that it is still challenging to build a parallel programming model or interface that is high-level, platform-independent, and efficient as well.

The ensemble-based programming tries to combine the major advantages of the high-level programming models and runtime strategies described in this chapter. Similar to HPF and GA, the ensemble-based programming approach provides a shared memory programming view, but can be implemented on distributed memory platform. In order for the extensibility, it applies the template-based OOP approach like TBB and Charm++ do. The ensemble-based programming starts from the basic points, which fits the idea of OpenCL and CUDA proposed. However, OpenCL and CUDA support programming concrete light weight threads in a fine granular fashion, while our programming model aggregates fine granular entities and map them to CPU threads or processes automatically. The irregular support of the ensemble-based programming reuses the idea of the PARTI/CHAOS library in order to manage the data storage and communication among processes efficiently on distributed memory systems.

5. Ensemble-based Programming

5.1. Overview

This chapter introduces the ensemble-based programming scheme, which mainly includes the machine model, the programming paradigm, and the programming interface. In addition, an MD simulation as a running example is presented to explain ensemble-based programming. The machine model is an abstract SPMD architecture that consists of a control processor and multiple distributed fine granular processors. In order to program this machine, we introduce the programming paradigm, which consists of software entities including elementary points (EPs), topologies, and the ensemble. Based on the programming paradigm, the object-oriented programming interface is introduced. It maintains a template hierarchy that supports specifying the software entities described in the programming paradigm. Different application areas including multi-body, irregular grid and regular grid applications are supported by the programming interface.

The rest of this chapter is organized as follows: Section 5.2 introduces the ensemble-based abstract machine model; Section 5.3 introduces the programming paradigm defined on top of the machine model; Section 5.4 describes the object-oriented programming interface; Section 5.5 introduces the ensemble-based implementation of a simple MD simulation.

5.2. Machine Model

5.2.1. Overview

The machine model of the ensemble-based programming is an abstract architecture that is composed of Fine Granular Processors (FGPs) and a Control Processor (CP). Figure 5.1 gives an overview of the abstract machine model. The machine model is different from a concrete architecture with CPUs, main memory, and I/O devices. It is a simple and fine-granular hardware platform, which hides optimizations such as cache optimizations, data locality, and communication overlapping, and so on. These optimizations are taken into account in the implementation of the mapping from the machine model to concrete architectures, e.g., sequential architectures, shared memory architectures, and distributed memory architectures as well. The objective of designing the

machine model is to provide an easier and more straightforward parallel programming platform compared to current programming interfaces

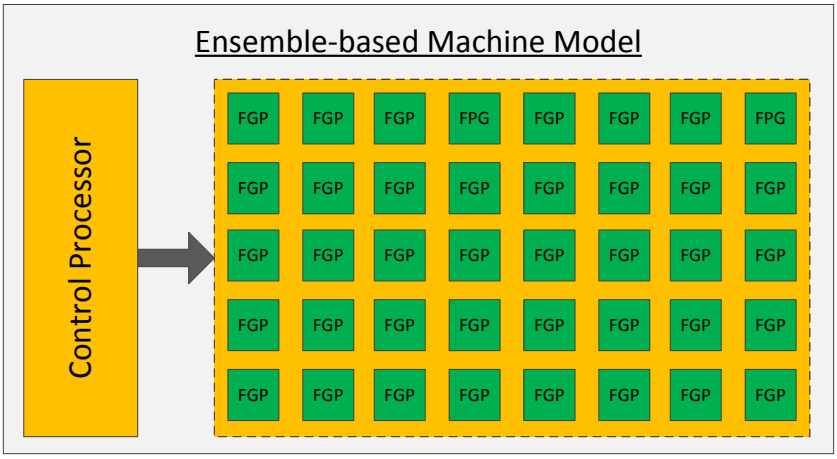


Figure 5.1.: Machine model

The FGPs are distributed on the machine and each FGP has its own local memory and a Local Processing Unit (LPU). The local memory in a FGP keeps local information of the FGP. The LPU in the FGP can perform coarse-grained application-specific computations on the data in the local memory. The CP consists of main memory, a Local Processing Unit (LPU) and a Control Unit (CU). The main memory keeps local data of the CP. The LPU in the CP is responsible for managing a single-threaded global control flow. The CU is an interface that triggers parallel operations of the FGPs in the machine. The execution model of the machine is similar to SIMD. It means that the CP issues a "single instruction" representing a specific parallel operation on FGPs; multiple FGPs then perform coarse-grained computations in parallel. This section mainly introduces main components of a FGP, details of the CP, and interactions between the CP and FGPs.

5.2.2. Fine Granular Processors (FGPs)

Most of the computational workload of an application is accomplished by FGPs in the machine model. The FGPs are distributed on the machine and each FGP is an abstract computational processor that consists of local memory and a LPU. The local memory stores the local data of the FGP. The LPU performs coarse-grained computations on the data in the local memory. It is not supported that a FGP accesses data of another FGP directly, since the FGPs are logically distributed. The communication between FGPs is established by point-to-point communication, which is controlled by the CP globally. In addition, coarse-grained computations executed on the LPU are triggered by the CP. The basic structure of a FGP is showed in Figure 5.2 and details of the components in

the FGP are described as follows:

1. Local memory: The local memory primarily stores application-specific data for local computation. It also stores information from remote FGPs, which is obtained by communication among FGPs. The local data can be directly accessed by the LPU.
2. LPU: The LPU is responsible for local computations on the data in the local memory. Application-specific coarse-grained computations can be executed on the LPU locally. The computations executed on the LPU of the participating FGPs asynchronously. The LPU is typically different from a computational unit in SIMD machines, which only performs fine grained computations, e.g., add, subtract, multiply and other simple arithmetic computations. In addition, input and output (I/O) operations can be processed by the LPU if I/O is implemented in a distributed fashion.

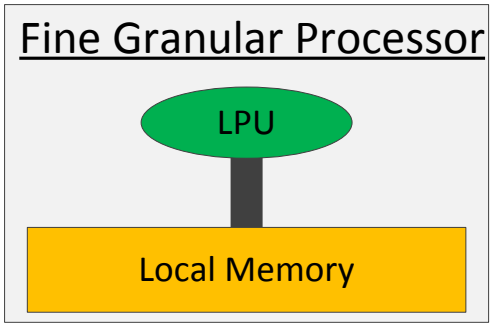


Figure 5.2.: Fine Granular Processor

5.2.3. Control Processor (CP)

The CP consists of main memory, a LPU and a CU. The main memory stores some local data and global information related to FGPs in the machine. The LPU is responsible for managing the global control flow, which is sequentially executed and in a single-threaded fashion. The CU can perform parallel operations triggered by the global control flow. These parallel operations control communication and computation of FGPs on the machine. Basic components of the CP are shown in Figure 5.3. Details of the components are described as follows:

1. Main memory: The main memory stores local data and some global information about FGPs in the machine. For example, input data and results obtained from the computation of FGPs are local data, while some application-specific global data and data dependence information are global information.
2. LPU: The LPU is responsible for managing a global control flow, which mainly includes input and output (I/O) operations, local computations, and parallel oper-

ations that control the FGPs on the machine. The global control flow is sequentially executed in a single-threaded fashion. For example, reading input data into the main memory and writing results to devices are typically I/O operations that are done by the LPU; the parallel operations that executed on the CU are triggered by the global control flow.

3. CU: Parallel operations that control behaviors of the FGPs on the machine are instructed by the global control flow and run on the CU. The parallel operations execute on the CU in a synchronous fashion. The semantics of synchronous parallel operations is that the next control operation on FPGs starts only when all the FGPs involved in the previous operation are accomplished.

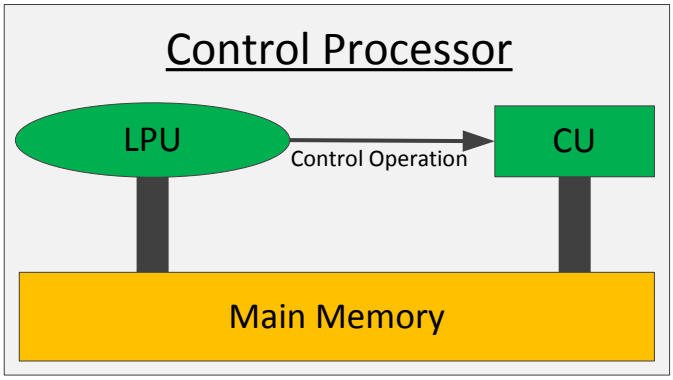


Figure 5.3.: Control Processor

5.2.4. Interactions between the CP and FGPs

There are several interactions between the CP and the FGPs on the machine. The CP controls the FGPs and the FGPs perform certain operations based on parallel operations executed on the CU in the CP. For example, the CU can issue a communication operation, which leads to communication among all the FGPs or a subset of FGPs in the machine following their dependences; the CP can also issue a parallel computation or collective operation on FGPs, which leads to a parallel computation or collective operation of the participating FGPs. The details of these interactions and some other interactions between the CP and the FGPs are described as follows:

1. Explicit communication among FGPs: The communication among FGPs arises when some FGPs require information from other FGPs for their local computations. The communication among FGPs is done by sending and receiving that are triggered by the CP explicitly, and the participating FGPs can exchange their local information. After the communication is accomplished, the remote data is assumed to be available in the local memory of the FGPs.

2. Parallel computation of FGPs: The global computation operation that is triggered by the CP can start the computation of FGPs in the form of parallel operations. The LPU in each FGP performs the computation on its local memory in parallel. Thus the participating FGPs update their local data concurrently and asynchronously.
3. Collective operation of among FGPs: The collective operation that is executed on the CU triggers a collective operation on a set of FGPs in the machine model. All the participating FGPs start the operation cooperatively to get collective results. The results can be either returned to the main memory of the CP or kept in the local memory of the FGPs.
4. Collective operation between CP and FGPs: It is not supported that FGPs can access the main memory of the CP, but the CP can access the local memory of FGPs in the machine by an explicit collective operation between CP and FGPs.

5.3. Programming Paradigm

5.3.1. Overview

The machine model consisting of a CP and multiple FGPs provides an ensemble-based programming platform. In order to program the platform, the programming paradigm on top of the machine model is introduced in this section. The programming paradigm is an abstraction of the machine model. It mainly consists of software entities, relations among the entities and a mapping from the entities to the machine model. In this section, three major software entities and their relations are introduced, which are Elementary Points (EPs), the ensemble, and the master thread.

The programming paradigm is based on a fork-join model. As it is shown in Figure 5.4, the execution starts with the master thread that triggers parallel operations on a set of EPs in the ensemble. An EP represents a basic computational object in the domain of an application, which is mapped to a single FGP in the machine model. It is primarily composed of local attributes and local operations. The ensemble is a software container that keeps all the EPs and their topologies. It is mainly responsible for data parallel workloads. The behaviors of the ensemble and the EPs in the ensemble are controlled by the master thread, which is stored in the main memory of the CP in the machine model. The master thread is sequentially executed on the LPU in the CP. It triggers parallel operations, update operations, and collective operations on the EPs in the ensemble. All these operations are blocking operations. It means that they terminate after all the constituent parts finished. In addition, the major tasks of the master thread include I/O operations, initialization work, local computations, and so on. Details of EPs, the ensemble, and the master thread are described in this section.

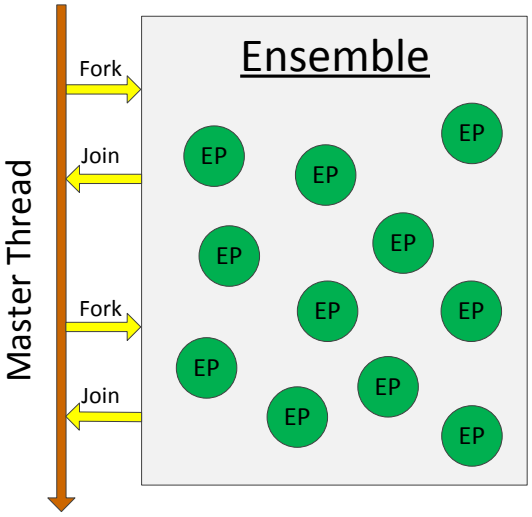


Figure 5.4.: Programming paradigm

5.3.2. Elementary Points

An EP is a software entity that represents the finest granular computational object in the domain of an application. Each EP is mapped to a single FGP in the machine model. For example, a particle or molecule in molecular dynamics simulations, a grid point in heat dissipation simulations and a celestial body in cosmological simulations can be represented as an EP.

An EP has local attributes and local operations. Local attributes represent application-specific characteristics of the EP. Local operations can read and write local attributes as well as read attributes of other EPs. If an EP requires data of other EPs for local operations, it has to access the data in shadow copies of the other EPs locally. Explicit communication operations can copy local attributes of EPs into their shadow copies located on the FGP of the receiving EP. These communication operations are triggered by update operations, whose communication patterns are specified in form of a topology. Details of local attributes, local operations, and shadow copies are described in the following subsections.

Local Attributes

Local attributes of an EP keeps application-specific information that is specified by programmers. The local attributes are stored in the local memory of the FGP that the EP is located on. An EP usually has local attributes including basic types of data, data arrays or complicated data structures as to describe characteristics of the single EP. For example, the mass information, the position information, the velocity informa-

tion, or domain-specific properties are typical local attributes of an EP in a multi-body simulation.

Shadow Copies

A shadow copy is a remote copy of an EP. It is located on remote EPs that require data of the EP. As it is shown in Figure 5.5, two EPs (EP_1 and EP_2) that are located on two FGPs. EP_1 has a shadow copy of EP_2. Thus, EP_1 can access EP_2's data locally if the data are copied into the shadow copy of EP_2 by explicit communication operations. The shadow copy of EP_2 is read only to EP_1.

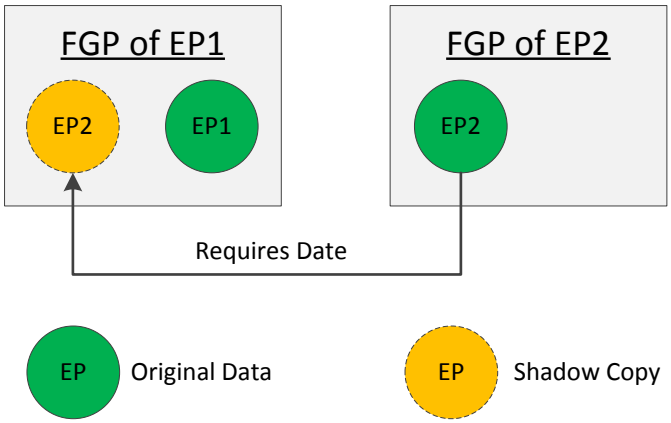


Figure 5.5.: Shadow copies

Local Operations

Local operations are usually coarse-grained and application-specific. They are used to update local attributes of a single EP according to certain algorithms. They are specified by programmers and triggered by the master thread in form of parallel operations. All the participating EPs perform local operations asynchronously after the required data are already available in the EPs. The input of local operations is local attributes and the data in shadow copies of remote EPs, while the output of the local operation is written back into the EP's local attributes. The data in shadow copies of remote EPs cannot be modified by local operations.

5.3.3. Ensemble

The ensemble is a software container that stores a set of EPs and manages their information, communication, computation, and so on. The main constituents in the ensemble

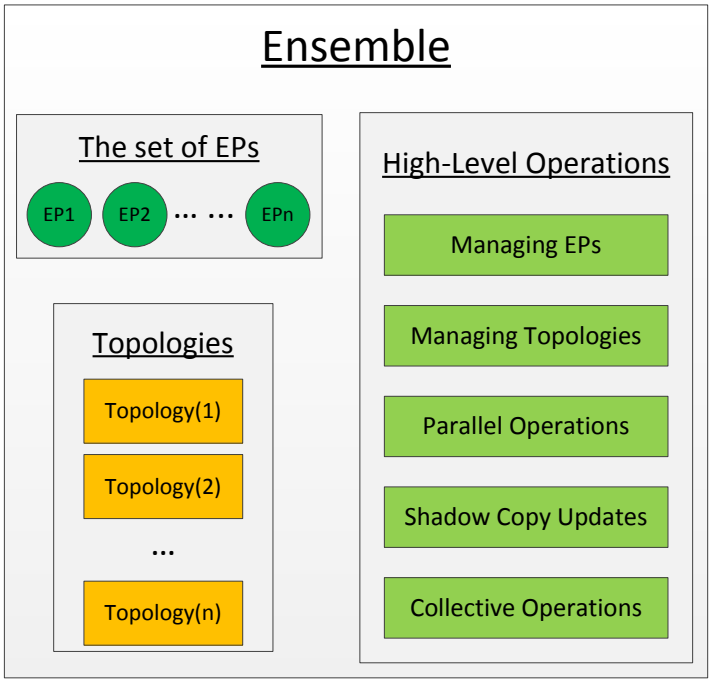


Figure 5.6.: Ensemble

are a set of EPs, topologies, high-level operations for managing EPs and topologies, parallel, shadow copy update, and collective operations. The basic structure of the ensemble is shown in Figure 5.6.

The ensemble keeps the information of the set of EPs and manages the EPs based on the information. A topology describes communication patterns of all the EPs or a subset of the EPs in the ensemble. Multiple topologies can be kept in the ensemble in order to maintain communication patterns of different subsets of EPs. The operations for managing EPs and topologies are used for inserting, removing, and organizing the EPs and the topologies in the ensemble. The parallel, shadow copy update and collective operations are triggered by the master thread and performed on the EPs.

Operations for Managing EPs and Topologies

EPs can be inserted into and removed from the ensemble by using an operation for managing EPs. After the EPs have been inserted into the ensemble, topologies can be created based on the EPs. An operation for managing topologies is primarily responsible for inserting and removing topologies.

Topologies

A topology defines communication patterns resulting from the need for the information of a set of EPs in the ensemble. The EPs can exchange their status based on certain topologies. A topology of four EPs is shown in Figure 5.7. The topology in the figure determines the EPs (EP_1, EP_2, EP_3, and EP_4) and their relations, which describe dependency information among them. For example, EP_1 requires data from EP_2, EP_3, and EP_4 for its local computation, and EP_2 requires data from EP_1 and EP_4 for its local computation.

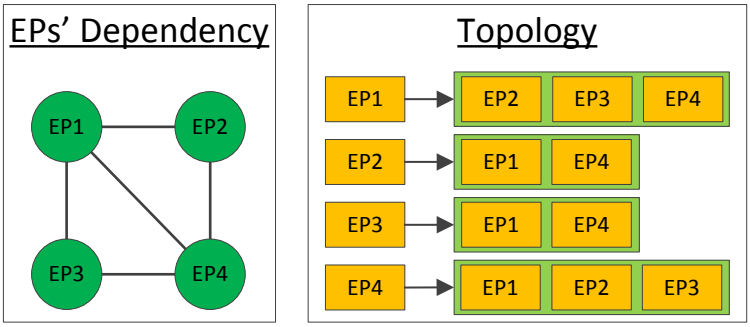


Figure 5.7.: Topology of four EPs

The ensemble is able to manage multiple topologies, which keep communication patterns of different sets of EPs in the ensemble. The topologies are used to guide shadow copy update and parallel operations on the EPs. For example, the ensemble can update the shadow copies of a set of EPs based on a certain topology, and then the local data of each EP are copied into its shadow copies on remote EPs depending on the topology. According to the topology shown in Figure 5.7, the local data of EP_2, EP_3, and EP_4 need to be copied into their shadow copies located on the FGP of EP1. In addition, topologies can be updated during the entire execution of irregular applications, e.g., molecular dynamics simulations and irregular grid applications.

The programming paradigm supports different types of topologies, i.e., *rootTopology*, *subTopology*, and *baseTopology*. The definitions and relations of these topologies are described below.

1. *rootTopology*: A root topology is a topology that manages the communication pattern of all the EPs in the ensemble.
2. *subTopology*: *Top1* is a sub topology of *Top2* if and only if *Top1* keeps a sub set of *Top2*'s EPs and maintains the same communication pattern in *Top2*. A topology can have multiple sub topologies.
3. *baseTopology*: *Top1* is a base topology of *Top2* if and only if *Top2* is a sub topology of *Top1*. Any topology can only have a single base topology. The communication pattern of a base topology can be reused by its sub topologies.

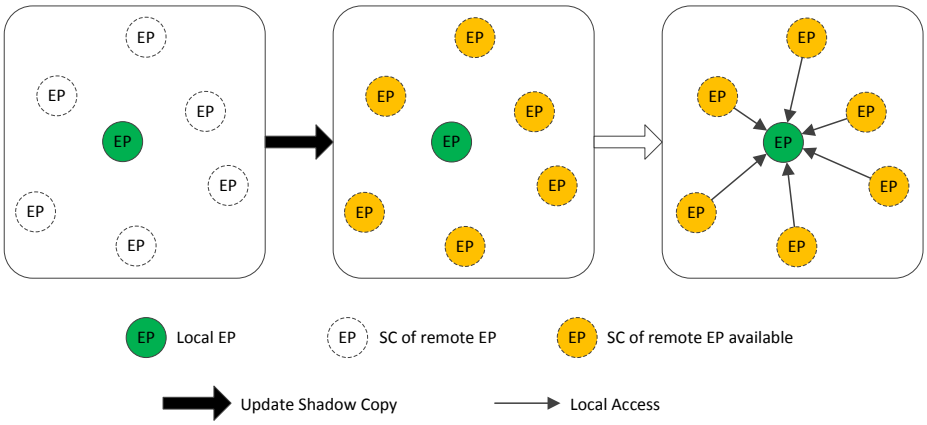


Figure 5.8.: Shadow copy updates

Shadow Copy Update (SC-update) Operations

A SC-update operation is executed on a set of EPs in the ensemble. It can be performed on different subsets of EPs in the ensemble by specifying different topologies. In addition, a SC-update operation can update not only full status of EPs but also partial status of EPs. As it is shown in Figure 5.5, an EP has multiple SCs of remote EPs. At the beginning, there are no data available in these shadow copies before the SC-update operation is executed. Once a SC-update operation is performed, the data of remote EPs are copied into the EP's shadow copies. Then, the SCs are available and can be accessed by the EP locally.

In order to improve the efficiency of the SC-update operation, all the EPs involved in the operation update their shadow copies in parallel. Thus, after the SC-update operation is accomplished, all the shadow copies of the participating EPs are available and all the EPs can access data in these shadow copies for their local operations.

Parallel Operations

Parallel operations are controlled by the master thread to trigger parallel local operations of EPs. The participating EPs in a parallel operation perform a certain local operation in parallel. This is similar to a SIMD execution fashion. It means that multiple coarse-grained operations execute in parallel when a single instruction is issued by using the interfaces.

Collective Operations

The EPs involved in a collective operation perform a certain communication collectively. Different types of collective operations are supported in the ensemble. Results of the

collective operation can be returned either to the EP or to the master thread. There are two major types of collective operations in terms of the ways of returning results. These collective operations are described as follows:

1. Local returning: It supports that results of a collective operation are written back to the participating EPs.
2. Global returning: It supports that results are written back to the master thread that is executed on the CP in the machine model.

5.3.4. Master Thread

The master thread is executed on the CP in a fork-join fashion. It is responsible for globally managing the ensemble-based program written to implement a scientific application. The major tasks of the master thread include performing I/O, creating the ensemble and topologies, creating and initializing EPs, managing topologies in the ensemble, local computations, and triggering parallel operations, and so on. These tasks are described as follows.

1. The master thread is responsible for I/O operations including getting data from input files and writing results to output files. Usually, the input data that are obtained by the master thread can be used for creating EPs or topologies. In addition, the I/O can be accomplished by EPs in a fine granular and distributed fashion, but it is not discussed in this section.
2. The master thread can create the ensemble by using different specifications, e.g., the application type, the number of EPs, the number of topologies, and so on. The ensemble is kept during the entire execution of the program after it is created by the master thread. In this work, we restrict the paradigm to code with just a single ensemble for an application. The extension to the programming paradigm that supports multiple ensembles is discussed in future work.
3. A number of EPs are created depending on the description of EPs and input data. The master thread is responsible for creating EPs and inserting them into the ensemble by using the operations for managing EPs.
4. As it is described in Subsection 5.3.3, the ensemble can keep multiple topologies. The master thread takes care of creating topologies and inserting topologies into the ensemble.
5. One of the major tasks of the master thread is to trigger parallel operations on the ensemble's EPs. Once a parallel operation is issued, the participating EPs can perform a certain operation in parallel asynchronously. Parallel operations are executed in a blocking fashion, which means that the next parallel operation starts only when the previous operation is done by the participating EPs.

5.4. Programming Interface

5.4.1. Overview

This section introduces the programming interface of the machine model described in Section 5.2. The programming interface supports an object-oriented programming (OOP) approach implemented in C++ to specify the software entities described in Section 5.3, including elementary points, the ensemble, and topologies. The programming interface is easy to use because its host language is standard C++ without any extensions. In addition, it consists of a template hierarchy that supports specifying the software entities in an OOP fashion. The template hierarchy current supports multi-body, irregular grid, and regular grid applications, and other application areas can be supported by extending the template hierarchy. For clarity, the template hierarchy is explained with a concrete MD simulation as an running example. Details of the ensemble-based implementation of the MD simulation are described in this section.

5.4.2. An Object-Oriented Programming Approach

The object-oriented programming (OOP) approach is based on objects, which are data structures consisting of data fields and methods. Developers create objects together with their interactions to design applications. For example, objects can inherit characteristics from other objects, or access data of other objects for local computation. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance. The entities described in Section 5.3 can entirely be defined as objects based on the OOP approach. For example, an elementary point consisting of local data and local operations can be specified by objects with attributes and member functions. The interactions between objects are defined as member function calls.

Many modern programming languages now support OOP, at least as an option. For example, C++, Java[112] and Smalltalk[113] are three of the widely used OOP languages. C++ is one of the most popular programming languages that perform OOP. It is implemented on a wide variety of hardware and operating system platforms. It introduces OOP features to C by adding object oriented features, such as classes, templates, virtual functions, inheritance, and so on. Therefore, the host language of the ensemble-based programming interface is C++, which is a popular OOP language that most programmers are currently working with. Programmers can write ensemble-based programs with the support of the programming interface by using standard C++ syntax. This characteristic improves the programmability, since there is no need for programmers spending additional time on new programming features beyond C++.

One of the advanced OOP features in C++ is the template[114]. It is a powerful tool that can be used for generic programming, template meta-programming, code optimization,

and so on. C++ supports both function and class templates, which can be parameterized by types, compile-time constants, or other templates. C++ templates are implemented by instantiation at compile-time. To instantiate a template, compilers substitute specific arguments for a template's parameters to generate a concrete function or class. The template-based programming approach can widely use for designing C++ libraries. For example, the STL is a C++ software template-based library. It applies compile-time polymorphism that is more efficient than traditional run-time polymorphism. In order to improve the programming productivity and reusability, the programming interface consists of a template hierarchy for specifying the software entities described in Section 5.3 including elementary points, their topologies, and the ensemble. Concrete classes derive from those class templates to implement application-specific functionalities. The template hierarchy is easy to reuse and extend. It helps library developers to add new class templates into the template hierarchy to support new application areas.

5.4.3. Overview of a Running Example

In this section, a molecular dynamics (MD) simulation is presented as a running example, which helps to explain how to program with the template hierarchy. An ensemble-based MD program primarily consists of a main function and three active entities, i.e., a set of molecules, a topology, and an ensemble of molecules. A molecule is a fine granular element in the MD simulation domain. It includes attributes and member functions that perform computations according to the equations of movement described in Chapter 2. The ensemble is a container of the set of molecules. It consists of operations that can be triggered by the main function to perform certain operations on the molecules in the ensemble. The topology describes relationships and communication patterns of the molecules in the ensemble. The main function implements a master thread that controls the behavior of these active entities. Its major steps are described as follows:

1. Creating the ensemble of molecules;
2. Creating multiple molecules and inserting them into the molecule ensemble;
3. Creating topologies of the molecules and inserting the topologies into the molecule ensemble;
4. Triggering parallel operations on the molecules in the ensemble, e.g., exchanging data based on topologies, parallel operations, reducing certain attributes, and so on.

The behavior of a molecule primarily consists of these steps, e.g., *initialization*, *forceCalculation*, *integration*, and *updateLocation*. All the molecules in the simulation domain behave in this way. The steps from the 2nd to the 4th usually execute iteratively for a large number of time steps. The pseudo code of the implementation details will be presented in the following subsections respectively.

1. *initialization*: The molecule is created and initialized with input data.
2. *forceCalculation*: The molecule can get its neighbor molecules and calculate forces or energies exerted between them based on certain algorithms.
3. *integration*: The attributes of the molecule are calculated and integrated according to the equations of motion.
4. *updateLocation*: The molecule moves to a new location with the updated attributes, which can be used for computations of the next iteration.

5.4.4. Template Hierarchy

The template hierarchy supports specifying the software entities described in the programming paradigm. It starts from three top-level base templates, which are *ElementaryPoint*, *Ensemble*, and *Topology*. These base templates have derived templates called application-specific templates, which support multi-body, irregular grid, and regular grid applications. User-defined entities with local attributes and operations can be defined as C++ classes derived from the application-specific templates. The support for other application areas can be extended by adding new application-specific templates into the template hierarchy.

The organization of the template hierarchy is shown in Figure 5.9. Details of *ElementaryPoint*, *Ensemble*, *Topology*, and the application-specific templates are described in following subsections. To make the consistency of the terminologies in this subsection, we use lowercased words to represent the objects instantiated by classes. For example, "elementary point" means the object instantiated from *ElementaryPoint,* and "ensemble" means the object instantiated from *Ensemble.*

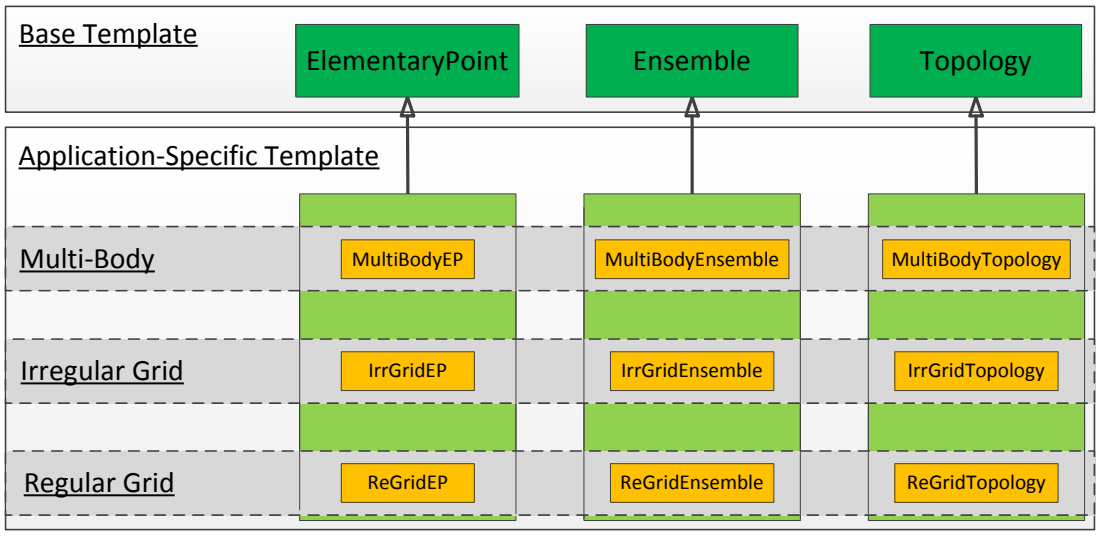


Figure 5.9.: Organization of the template hierarchy

5.4.5. *ElementaryPoint* and its Derived Templates

In the template hierarchy, the base templates for creating elementary points are *ElementaryPoint* and its derived templates namely *MultiBodyEP*, *IrrGridEP*, and *ReGridEP*. These templates have multiple predefined attributes and functions. They are base templates for creating user-defined classes in different application areas. The header files with full declaration of these templates are attached in the Appendix.

ElementaryPoint has two attributes, *id* and *ensemblePointer*. It has getter and setter functions for reading and writing these attributes. In addition, get address functions defined by programmers are presented in this section. An outline of the template's definition is shown in Listing 5.1. Details of the attributes and functions are described as follows:

Listing 5.1: Elementary Point

```
template<class Ensemble>
class ElementaryPoint {
protected:
    //The identifier
    unsigned long id;

    //The Ensemble pointer
    Ensemble*ensemblePointer;
public:
    //Default Constructor
    ElementaryPoint();

    //Default Destructor
    virtual ~ElementaryPoint();

    //Get the identifier
    inline unsigned long getId();

    //Set the identifier
    inline void setId(unsigned long id);

    //Get the Ensemble pointer
    inline Ensemble*getEnsemble();

    //Set the Ensemble pointer
    inline void setEnsemble(Ensemble*ensemblePointer);
};
```

1. *id*: It is an unsigned long integer, which is used to distinguish elementary points. It is read and written by programmers using predefined setter and getter functions. Programmers have to guarantee that different elementary points have different ids.

2. *ensemblePointer*: the *Ensemble* pointer can be used by elementary points to call high-level operations defined in *Ensemble*. For example, a molecule can get its neighbor list from the ensemble for local computations by specifying *ensemblePointer* ->getNeighbors().
3. Getter and setter functions: These functions are used to get and set predefined attributes in *ElementaryPoint*.
4. Get address functions: These functions are public member functions in application-specific classes derived from *ElementaryPoint* in order to return addresses of attributes in the classes. They can be used as parameters of the operations defined in *Ensemble*. A get address function is defined in such a format in Listing 5.2. The way of using this function will be discussed in the next subsections.

Listing 5.2: Get address function

```
//A1 is an attribute of an elementary point
//The DataType is the data type of attribute A
DataType*GetAddressA1{
    return &(this->A1);
}
```

MultiBodyEP and ReGridEP

The derived templates, *MultiBodyEP* and *ReGridEP*, provide additional attributes used for the elementary points in multi-body and regular grid applications. *MultiBodyEP* provides *position* and *ReGridEP* has *coordinate* as their attributes. These attributes are described below.

1. *position*: It keeps the spatial location of a multi-body EP in the domain of an application. The data type and the dimension of *position* are template parameters to create a *MultiBodyEP* class. For example, it can keep two-dimensional or three-dimensional floating point or double precision floating point values. The value of *position* changes if multi-body EPs move in the simulation domain during execution.
2. *coordinate*: It stores the coordinates of *ReGridEP* objects in a multi-dimensional regular domain. The data type is integer. The dimension of *coordinate* is a template parameter to create *ReGridEP* classes. Usually, the coordinates of *ReGridEP* objects don't change during the entire execution of regular grid applications.

The Running Example

Our running example, the MD simulation, belongs to the multi-body application area. *Molecule* is derived from *MultiBodyEP*. The declaration of *Molecule* and the way of creating molecules are shown in Listing 5.3. The attribute list in the figure typically consists of

basic data types or compound data structures, e.g., integer values, floating point values, arrays, vectors, and so on. The member functions primarily perform application-specific computations and update values of the attributes in molecules, e.g., *initialization*, *forceCalculation*, *integration*, and *update*, which are described in Subsection 5.4.3. In addition, programmers can create molecules or molecule pointers by using constructors defined in *Molecule* with a parameter list.

Listing 5.3: Declaration of Molecule and molecule creation

```
//The declaration of class Molecule
class Molecule : public MultiBodyEP {
    //Attribute List
public:
    //Default Constructor
    Molecule();

    //Constructor with parameter list
    Molecule(Attribute List);

    //Initializing parameters of the molecule
    void initialization(Parameter List);

    //calculate the forces exerted on the molecule
    void forceCalculation(Parameter List);

    //update locations by integrating equations of movement
    void integration(Parameter List);
};

//create a molecule with a attribute list
Molecule molecule(Attribute List);

//create a molecule pointer with a attribute list
Molecule*moleculePointer = new Molecule(Attribute List);
```

5.4.6. *Ensemble* and its Derived Templates

Ensemble and its derived templates are used to create the ensemble of an application. *Ensemble* has two template parameters, *EP* and *Top*. *EP* is user-defined class derived from the application-specific templates of *ElementaryPoint*. *Top* can be *MultiBodyTopolgy*, *IrrGridTopology*, or *ReGridTopology*. The full declaration of *Ensemble* and its derived templates is attached in the Appendix. An outline of *Ensemble*'s definition is shown in Listing 5.4. In this subsection, we use the term "EP" to represent an object of a user-defined class derived from one of the derived templates of *ElementaryPoint*.

The term "ensemble" represents an object instantiated from *MultiBodyEnsemble*, *IrrGridEnsemble*, or *ReGridEnsemble*.

Listing 5.4: An outline of Ensemble

```
template<class EP, class Topology>
class Ensemble
{
protected:
    vector<EP> EP_Set;
    vector<Topology*> Topology_Set;
    unsigned long numOfEP;
    unsigned short numOfTop;
public:
    Ensemble(int *argc, char ***argv);

    Ensemble(int *argc, char ***argv, unsigned long numOfEPs);

    ~Ensemble();

    unsigned long getNumOfEPs();

    unsigned short getNumOfTops();

    void insertEP(EP*ep);

    void removeEP(EP*ep);

    void insertTopology(Topology*topology);

    void removeTopology(Topology*topology);

    void removeTopology(unsigned short topId);

    Topology* getTopology(unsigned short topId);

    vector<EP*>& getNghbs(EP*current, Topology*topology);

    void update(Topology*topology);

    template<typename GetOperation>
    void update(GetOperation getOp, Topology*topology);

    template<typename Operation>
    void parallel(Operation op);

    template<typename GetOperation>
    void allReduceOp(GetOperation getOp, int reduceOp);
```

```
    template<typename GetOperation>
    void reduceOp(GetOperation getOp, in reduceOp, void* result);

    virtual void finalize();
};
```

The attributes in *Ensemble* are shown as follows:

1. *EP_Set*: It stores all the EPs in the ensemble. Once an EP is created, it can be inserted into *EP_Set* by using *insertEP*.
2. *numOfEP*s: It is an unsigned long integer, which keeps the number of EPs stored in the ensemble.
3. *Topology_Set*: It stores multiple topology pointers. After a topology was created, it can be inserted into *Topology_Set*.
4. *numOfTop*s: It is an unsigned short integer, which keeps the number of topology pointers stored in *Topology_Set*.

Ensemble predefines a group of operations that can be called in the main function in order to control the behaviors of the EPs in the ensemble. Details of the operations are described as follows:

1. Constructor: It is mainly responsible for constructing the ensemble by input parameters argc, argv, and other application-specific parameters. For example, if the high-level operations of *Ensemble* are implemented on distributed memory systems with MPI, the constructor has to call MPI_Init() internally passing argc and argv. In addition, programmers should provide the overall number of EPs to the constructor.
2. *getNumOfEP*s: It is used to get the number of EPs stored in the ensemble.
3. *getNumOfTop*s: It is used to get the number of topologies kept in the ensemble.
4. *getTopology*: It is used to return a topology pointer stored in *Topology_Set*.
5. *insertEP*: Programmers can use it to insert EPs into the ensemble. The parameter of *insertEP* is a pointer to an EP.
6. *removeEP*: It is used to remove EPs from the ensemble. Programmers can either provide a pointer or an identifier of an EP to remove it from the ensemble.
7. *insertTopology*: It inserts a topology into the ensemble by providing its pointer.
8. *removeTopology*: It is used to remove a topology from the ensemble by providing the address or identifier of a topology.
9. *update*: As described in Subsection 5.3.3, *update* triggers SC-update operations, which supports exchanging the complete information as well as partial information

of EPs in the ensemble. The *update* with a topology pointer as its parameter exchanges the complete information of EPs based on the topology. The *update* with a get address function and a topology pointer as its parameters only exchanges partial information of EPs according to the topology. The get address function can be passed as a parameter of *update* in such a way, which is shown in Listing 5.5.

Listing 5.5: update with GetAddress Function

```
//Provide a member function GetAddressA1
update(mem_fun_ref(&Molecule::GetAddressA1), &topology);
```

The function *GetAddressA1* is passed as a parameter of *update* by using the function adapter *mem_fun_ref* defined in C++ STL[115]. *mem_fun_ref* converts a member function to a standard function object, which can be used as a parameter of this operation.

10. *parallel*: It triggers member functions of EPs to execute in parallel. The template parameter is a function object adapted from a member function of *ElementaryPoint*. For example, a *Molecule* class has a member function called *forceCalculation*, which calculates the forces exerted on a molecule by other molecules. As it is shown in Listing 5.4, programmers can execute *forceCalculation* of the molecules in parallel by using this operation.

11. *allReduceOp*: It provides collective reduction operations that return the result in all the involved EPs. There are different types of calculations supported by this operation. The type of calculations is a integer value, which is a parameter of the interface. For example, the calculation type is accumulation means that an attribute of the involved EPs is accumulated and the result is written back to the attribute of the EPs respectively. *GetOperation* is a get address function, which is used to specify a specific attribute of EPs. The basic format of these operation is shown in the program listing in Listing 5.4.

12. *reduceOp*: It supports collective operations to return a result to the main function. The type of collective calculations and an address of the result have to be specified as parameters of this operation. Its basic format is shown in the programming list in Listing 5.4. All the other formattings of this operations are presented in the Appendix.

13. *getNghbList*: This operation is called by an individual EP as to get a list of its neighboring EPs from the ensemble based on a topology. After it is accomplished, the EP can access data in the neighbor list for local computations. The full declaration of these forms is presented in the Appendix.

14. *finalize*: It is responsible for terminating the work of the ensemble. This operation has to be called at the end of the main function.

The derived templates of *Ensemble* inherit the attributes of *Ensemble* and implement its operations. They are used to create application-specific ensembles directly by specifying the same application type of topologies. For example, *MultiBodyEnsemble<MultiBodyEP, MultiBodyTopology>* can only be created by specifying *MultiBodyTopology* and *MultiBodyEP* as its template parameters. It is used to create the ensemble for multi-body applications.

MultiBodyEnsemble

MultiBodyEnsemble has *ElementaryPoint*, *Topology*, *Dimension*, and *DataType* as its template parameters. *ElementaryPoint* must be a class derived from *MultiBodyEP*. *Topology* must be specified by *MultiBodyTopology*. *Dimension* is a unsigned short integer, typically two or three, that depicts the dimension the spatial domain of *MultiBodyEnsemble*. *DataType* is the data type of lower and upper bounds of the simulation domain. It can be a C++ supported data type, i.e., a double floating point, a single floating point, or a long double floating point. *MultiBodyEnsemble* has *domainLower* and *domainUpper* as its attributes, which have to be provided to the constructor of *MultiBodyEnsemble*.

1. *domainLower*: It keeps lower bounds of the simulation domain in multiple dimensions using a specified data type. The lower bounds of the domain are usually 0.0.
2. *domainUpper:* It keeps upper bounds of the simulation domain in multiple dimensions using a specified data type.

IrrGridEnsemble

IrrGridEnsemble is a derived template of *Ensemble*. It has *ElementaryPoint* and *Topology* as its template parameters. *ElementaryPoint* must be a class that is derived from *IrrGridEP*, and *Topology* is specified by *IrrGridTopology*.

ReGridEnsemble

ReGridEnsemble is a derived template of *Ensemble*; it has *ElementaryPoint*, *Topology*, and *Dimension* as its template parameters. *ElementaryPoint* must be a class that is derived from *IrrGridEP*, and *Topology* must be specified by *IrrGridTopology*. *Dimension* is a unsigned short integer that depicts the dimension of the coordinates of EPs stored in the *ReGridEnsemble* object.

The Running Example

As it is shown in Subsection 5.4.3, the main function of the MD simulation is responsible for creating *moleculeEnsemble,* initializing *moleculeEnsemble* with input parameters, inserting molecules into *moleculeEnsemble,* updating shadow copies of the molecules in *moleculeEnsemble,* triggering *forceCalculation* function of the molecules in forms of parallel operations, and so on. All these basic steps are coded in the following way, which is shown in Listing 5.6.

Listing 5.6: An outline of the main function

```
#include "Ensemble.h"
#include "Molecule.h"
int main(int argc, char**argv){
    //create a moleculeEnsemble, the formatting of MultiBodyTopology
    //as a template parameter will be discussed in the following
        subsections
    MultiBodyEnsemble<Molecule, MultiBodyTopology<...> >
        moleculeEnsemble(argc,argv,NUM_OF_MOLECULE);

    //create multiple molecules and insert them into the
        moleculeEnsemble
    for(Number of molecules)
    {
       Molecule*molecule = new Molecule(Parameter List);
       moleculeEnsemble.insertEP(molecule);
    }

    //create a Topology
    Topology*topology = new Topology(cut_off);

    //shadowCopyUpdate and parallelExecution execute many iterations
    for(Number of Iterations){
        //update shadow copies of Molecules
        moleculeEnsemble.update(topology);

        //parallel execution on Molecules
        moleculeEnsemble.parallelExecution(mem_fun_ref(&Molecule::
            forceCalculation), topology);
    }

    //finializing the moleculeEnsemble
    moleculeEnsemble.finalize();

    return 0;
}
```

5.4.7. *Topology* and its Derived Templates

Topology is used to create topologies, which keep the communication patterns of EPs in the ensemble. It has three application-specific templates, *MultiBodyTopology*, *IrrGridTopology*, and *ReGridTopology*. The topologies are created by using predefined constructors in the template hierarchy. The declaration of *Topology* is shown in Listing 5.7. *Topology* has predefined attributes and operations. Details of the attributes are described below.

1. *id*: It is a short integer value used to distinguish topologies stored in the ensemble. The id of the root topology is always 0, while the ids of sub topologies are from 1 to N.
2. *numofEPs*: It stores the number of EPs managed by a topology.
3. *ensemblePointer*: It can be used to reference the ensemble, since a topology needs some information from the ensemble for local operations.
4. *isRootTop*: It is a bool value that depicts whether the current topology is a root topology defined in Section 5.3.3.
5. *baseTopPtr*: It is a *Topology* pointer. If a topology is a sub topology of another one, the pointer references to its base topology.

The predefined operations are shown as follows:

1. *initialization*: It initializes the internal data structures of *Topology* for the next operations.
2. *createNeighborList*: If the topology is the root topology, it creates the neighbor EP list for the EPs, which can be reused by its sub topologies.
3. *updateTopology*: It rebuilds a new topology according to the runtime information or the information specified by the users.

Listing 5.7: Declaration of Topology

```
#include "Ensemble.h"

template <class EP>
class Topology<EP> {
protected:
   unsigned long id;
   unsigned long numOfEP;
   bool isRootTop;
   Topology<EP>*baseTopPtr;
   Ensemble<EP, Topology<EP> >* ensemblePointer;
public:
   //Default constructor
   Topology();
```

```
    //Initialize the current topology
    void initialization();

    //Create the neighbor list for the EPs
    void createNeighborList();

    //Initialize the current topology
    void updateTopology();
};
```

Programmers can create topologies by using the predefined constructors of the application-specific templates directly. The application area of a topology and the ensemble should be the same. For example, a *MultiBodyTopology*<*Molecule*, 3, *double*> class is a template parameter of *MultiBodyEnsemble* to create a *MultiBodyEnsemble*<*Molecule*, *MultiBody-Topology*<*Molecule*, 3, *double*> > class. The full declaration of *Topology* and its derived templates is presented in the Appendix. Details of the *Topology*'s derived templates are described in the following subsections.

MultiBodyTopology

MultiBodyTopology is used to create topologies for multi-body applications. It has *ElementaryPoint*, *Dimension*, and *DataType* as its template parameters. *ElementaryPoint* can be one of the three derived templates of *ElementaryPoint* defined in Subsection 5.4.5. *Dimension* is an unsigned short integer. It specifies the dimension of the simulation domain. *DataType* is the data type of *cutOffRadius*. It can be a C++ supported data type, i.e., a double floating point, a single floating point, or a long double floating point. *MultiBodyTopology* has *cutOffRadius* as its attribute, which can be specified by programmers. *cutOffRadius* is a threshold of spatial distance of two EPs. Each of the EPs requires local data of the other one if their distance is smaller than *cutOffRadius*. If the value of *cutOffRadius* equals to -1, each EP requires data of all the other EPs in the ensemble. In addition, *MultiBodyTopology* has an *updateTopology* function, since topologies of a multi-body application change during execution. The communication pattern of the EPs is recalculated by calling *updateTopology* explicitly. Multi-body topologies can be created by calling predefined constructors, which are shown in Listing 5.8.

The constructors and some functions of *IrrGridTopology* are presented in Listing 5.9. Details of these functions are described as follows:

1. Constructor of the root topology with *cutOffRadius*: It constructs the root topology based on a cut off radius. In addition, the topology id, number of EPs, ensemble pointers have to be specified.
2. Constructor of a sub topology with a group of EP indices: It is used to create a sub topology by mainly specifying a vector of EP indices and its base topology.

3. *updateTopology*: It rebuilds a new topology according to the updated locations of all the EPs in the ensemble.

Listing 5.8: Declaration of MultiBodyTopology

```
template<class EP, unsigned short Dimension, typename DataType>
class MultiBodyTopology : public Topology<EP>{
    //Constructor with cutOffRadius, dLower, dUpper
    MultiBodyTopology(unsigned topId, DataType cutOffRadius, unsigned
        long numOfEPs, MultiBodyEnsemble<EP, MultiBodyTopology<EP,
       Dimension,DataType> , Dimension, DataType>*
       multiBodyEnsemblePtr);

    //Constructor with cutOffRadius, dLower, dUpper, subset of EPs
    MultiBodyTopology(unsigned topId, vector<unsigned long>
       subIndices, unsigned long numOfEPs, MultiBodyTopology<EP,
       Dimension,DataType>*baseTopPtr, MultiBodyEnsemble<EP,
       MultiBodyTopology<EP,Dimension,DataType> , Dimension, DataType
       >*multiBodyEnsemblePtr);

    //update the topology with a cutOffRadius
    void updateTopology();
}
```

IrrGridTopology

IrrGridTopology keeps the communication pattern of a set of EPs in the ensemble by using id-based undirected graphs. An undirected graph typically consists of vertices and edges. The vertices represent ids of EPs in the ensemble and the edges represent their interactions. There is an edge between two vertices if and only if both EPs require information of the other one. Usually the graphs are sparse graphs, which mean that the number of EPs is much larger than their interactions.

The Compressed Sparse Row (CSR) format is a widely used scheme for storing and representing sparse graphs. In this format the adjacency structure of a graph with n vertices and m edges is represented using two arrays *xadj* and *adjncy*. The *xadj* array is of size n + 1 whereas the *adjncy* array is of size 2m (this is because for each edge between vertices v and u we actually store both (v, u) and (u, v)). The adjacency structure of the graph is stored as follows. Assuming that vertex numbering starts from 0 (C++ style), then the adjacency list of vertex i is stored in *adjncy* starting at index *xadj*[i] and ending at (but not including) index *xadj*[i+1] (i.e., *adjncy*[*xadj*[i]] through and including *adjncy*[*xadj*[i+1]-1]). That is, for each vertex i, its adjacency list is stored in consecutive locations in *adjncy*, and *xadj* is used to point to where it begins and where it ends.

The adjacency list format is another representation of id-based sparse graphs. In an adjacency list representation, each node in the graph keeps a list of all other nodes

which it has an edge to. For example, EP*0* requires data of EP*1* if and only if EP*1* is kept in the EP*0*'s node list, which is called a neighbor list. The advantage of this format is that it is flexible to operate on a graph, e.g., inserting nodes, inserting directed or undirected edges, removing nodes, removing edges, and so on.

The constructors and some functions of *IrrGridTopology* are presented in Listing 5.9. Details of these functions are described as follows:

1. Constructor with CSR: It supports constructing the root topology with a CSR format graph. In addition, the topology id, number of EPs, ensemble pointers have to be specified.
2. Constructor with an adjacency list: It can be used to create a root topology by specifying ids of EPs and their neighbor list in an adjacency list format rather than the CSR format.
3. Constructor of a sub topology with a group of EP indices: It is used to create a sub topology by mainly specifying a vector of EP indices and its base topology.
4. *addEP*: It adds an EP into the topology by specifying the id of the EP.
5. *removeEP*: It removes an EP from the topology by specifying the id of the EP.
6. *addLink*: It adds a link between two EPs by providing the ids of the starting EP and the ending EP.
7. *removeLink*: It removes an link between two EPs by providing the ids of the starting EP and the ending EP.
8. *updateTopology*: It rebuilds a new topology according to the updated relations between edges and points. For example, the user may add EPs or links to the topology, and when the *updateTopology* is called, the updated topology will be available.

Listing 5.9: Declaration of IrrGridTopology

```
template <class EP>
class IrrGridTopology : public Topology<EP>{

    //Constructor with CSR format to create a root Topology
    IrrGridTopology(unsigned topId, unsigned long* xadj, unsigned
        long* adjncy, unsigned long numOfEPs, unsigned long numOfEdges
        , IrrGridEnsemble<EP, IrrGridTopology<EP,Dimension,DataType>,
        Dimension, DataType>* irrGridEnsemblePtr);

    //Constructor with a neighbor list indices
    IrrGridTopology(unsigned topId, vector<vector<unsigned long> >
        adjListIndices, unsigned long numOfEPs, IrrGridEnsemble<EP,
        IrrGridTopology<EP,Dimension,DataType>, Dimension, DataType>*
        irrGridEnsemblePtr);
```

```
    //Constructor with a sub set of neighbor lists and a sub set of
        indices
    IrrGridTopology(unsigned topId, vector<unsigned long> subIndices,
         unsigned long numOfEPs, IrrGridTopology<EP,Dimension,DataType
        >*baseTopPtr, IrrGridEnsemble<EP,IrrGridTopology<EP,Dimension,
        DataType>, Dimension, DataType>*irrGridEnsemblePtr);

    //Add an EP reference into the topology
    void addEP();

    //Remove an EP reference from the topology
    void removeEP();

    //Add a link between two EPs into the topology
    void addLink();

    //Remove a link between two EPs into the topology
    void removeLink();

    //update the topology according to the revised information
    void updateTopology();
};
```

ReGridTopology

ReGridTopology is used to create topologies for regular grid applications, which usually do not update during execution. It has *neighborStencil* as an attribute. The constructors both with the CSR format and with an adjacency list are supported by *ReGridTopology*. In addition, *ReGridTopology* objects are created based on a stencil, which is specified by users. For example, if *neighborStencil* is a set of {0, -1, 1} and {-1, 0, 1}, the coordinates of a regular grid point {a, b, c}'s neighboring points are {a, b-1, c+1} and {a-1, b, c+1}. Details of *ReGridTopology* are not discussed in this chapter.

5.5. Example: An MD Simulation

To implement the MD simulation based on the programming interface described in Section 5.4, programmers can write a program that mainly consists of two parts, the definition of molecules and the main function. The definition of molecules is specified by the class *Molecule* with local attributes and member functions. *Molecule* is used to generate a large number of molecules, which are defined as EPs. The main function

is executed as a master thread that controls these EPs by using operations defined in *Ensemble*.

Parts of the declaration of *Molecule* are shown in Listing 5.10. The code from line 9 to line 12 specifies the individual attributes of a molecule. From line 15 to line 16, the constructor is defined to create molecules by providing a list of input parameters related to the attributes. The *forceCalculation* is a member function, which requires data from other molecules. Therefore, the programmer defines a vector of molecule pointers, which stores the neighbor molecules returned from the ensemble by using *getNghbList* defined in *Ensemble*. The input of *getNghbList* is the current molecule pointer and the root topology defined based on the cut-off radius. After the neighbor list is returned, all the data in the neighbor molecules can be accessed for local computation. In line 26, an iterator of the molecule vector is defined to traverse the molecules in the vector. The *integration* function is used to update local attributes in *Molecule*. It just requires local information of the current molecule without calling functions defined in *Ensemble*. Programmers are free to define these member functions according to certain algorithms.

Listing 5.10: Declaration of Molecule

```
1  #include <vector>
2  #include <iterator>
3  #include "Ensemble.h"
4  #include "Topology.h"
5
6  class Molecule :
7          public MultiBodyEP<MultiBodyEnsemble<Molecule,
8          MultiBodyTopology<Molecule,3,double>>,3,double> {
9       double mass;
10      double initialVelocity[3];
11      double velocity[3];
12      double accelaration[3];
13 public:
14     //Constructor with local attributes
15     Molecule(unsigned long id, double mass,
16              double initialVelocity[3], position[3]);
17
18     //Initializing parameters of the molecule
19     void init(...);
20
21     //calculate the forces exerted on the molecule
22     void forceCalculation(...){
23        //The code below is written by programmers to
24        //implement specific algorithms
25        std::vector<Molecule*> nghbMolecules;
26        std::vector<Molecule*>::iterator nghbMoleculesIter;
27        nghbMolecules = this->ensemblePtr->getNghbList
```

```
28                                  (this, this->ensemblePointer->
                                        getTopology(0));
29          for(nghbMoleculesIter = nghbMolecules.begin();
30              nghbMoleculesIter != nghbMolecules.end();
                    nghbMoleculesIter++){
31              //Computation between local attributes and data in
                    neighbor Molecules;
32          }
33      };
34
35      //update local attributes
36      void integration(...);
37  };
```

The main function is written to implement the master thread. It primarily consists of instantiating molecules, creating the ensemble of the molecules, inserting the molecules into the ensemble, creating a topology of the molecules, triggering parallel operations on the molecules, and so on. The main function is shown in Listing 5.11, and its major parts are explained as follows:

1. Lines 13 and 14, the main function creates an *mdEnsemble* by specifying argc and argv, the lower and upper bounds of the domain, and the overall number of EPs.
2. Lines 20 to 27, it instantiates a large number of molecules by calling the *Molecule*'s constructor with the data obtained from input files. After the molecules are created, they are inserted into the ensemble by specifying their pointers. These tasks are iteratively executed in a *for* loop until all the molecules were successfully inserted.
3. Lines 30 to 33, the main function creates the root topology by specifying the cutoff radius and inserts it into the ensemble.
4. In line 36, a parallel operation is called. The parameter of the parallel operation is the member function *init* defined in *Molecule*. It triggers all the molecules managed by *mdTopology* to execute the function *init* in parallel.
5. Lines 39 to 51, it specifies a single simulation step, including communication, parallel computation, and an update operation. This simulation step is executed for many time steps. The main function triggers *update* to exchange data among the molecules based on the cut-off radius topology. Then, the molecules execute *forceCalculation* on local attributes and data obtained from other molecules as well. After *forceCalculation* is done, each molecule executes *integration* in parallel to update, for example, velocity, acceleration, position, and so on according to local information. The molecules move to new locations in the context of the MD simulation, which may change the communication pattern of the molecules.

Therefore, the current topology needs to be updated by calling *updateTopology* explicitly.

Listing 5.11: The main function of an MD simulation

```
1  #include "MultiBodyEnsemble.h"
2  #include "MultiBodyTopology.h"
3  #include "Molecule.h"
4
5  //The macro definition of the number of molecules
6  #define NUM_OF_MOLECULE 1024
7
8  //The macro definition of the number of iterations
9  #define NUM_OF_ITER 128
10
11 int main(int argc, char**argv){
12     //create the ensemble with Molecule, MultiBodyTopology, domain
           bounds
13     MultiBodyEnsemble<Molecule, MultiBodyTopology<Molecule, 3, double
           > >
14                         mdEnsemble(argc, argv, domainLower, domainUpper
                               , NUM_OF_MOLECULE);
15
16     //Read data from input files and initialize
17     //variables e.g. i, mass,initialVelocity,position
18
19     //create molecules and insert them into the ensemble
20     Molecule*molecule;
21     for(int i=0;i<NUM_OF_MOLECULE){
22         //mass,initialVelocity,position can be obtained from input
               files
23         molecule = new Molecule(i, mass,initialVelocity,position);
24
25         //molecules are inserted into the mlEnsemble
26         mdEnsemble.insertEP(molecule);
27     }
28
29     //create a topology with Molecule, cutoff radius is assumed to be
           2.0
30     MultiBodyTopology<Molecule, 3, double> mdTopology(2.0);
31
32     //insert the topology into the ensemble
33     mdEnsemble.insertTop(&mdTopology);
34
35     //Parallel execute initialization of all molecules in the
           mlEnsemble
36     mdEnsemble.parallel(mem_fun_ref(Molecule::init), &mdTopology );
37
```

```
38      //Communication and computation happens itertively
39      for(int i=0;i<NUM_OF_ITER;i++){
40          //Communication of all molecules in the mlEnsemble
41          mdEnsemble.update(&mdTopology);
42
43          //Parallel execute forceCalculation of all molecules in the
                mdEnsemble
44          mdEnsemble.parallel(mem_fun_ref(Molecule::forceCalculation), &
                mdTopology);
45
46          //Parallel execute integration of all molecules in the
                mdEnsemble
47          mdEnsemble.parallel(mem_fun_ref(Molecule::integration), &
                mdTopology);
48
49          //Update the topology for next iterations
50          mdTopology->updateTopology();
51      }
52
53      return 0;
54  }
```

The ensemble-based program is straightforward and easy to write based on the programming interface. Programmers only focus on defining EPs (molecules in the MD simulation) and the master thread that controls them. The definition of EPs is specified as a C++ class, which typically consists of application-specific attributes and member functions. The communication pattern of the EPs is specified as a topology, which is managed by *Topology* in the template hierarchy automatically. The master thread is responsible for manipulating communication and computation of EPs by using the high-level interfaces defined in *Ensemble*. The ensemble-based program is platform-independent. It can be run on different target machines based on different implementations of the template hierarchy. The topic of implementations will be discussed in the next chapter.

5.6. Summary

In this chapter, main components of the ensemble-based programming are introduced including an abstract SPMD architecture, the programming paradigm, and the object-oriented programming interface. The machine model is a fine granular architecture consisting of a CP and multiple FGPs, which are responsible for parallel computations. Based on the machine model, the programming paradigm is described. The programming paradigm includes the main software entities, the elementary point, topology, and ensemble, which are key components that support fine-granular programming. According to the programming paradigm, we build the programming interface consisting of a

template hierarchy applied to implement these software entities in an OOP approach. The template hierarchy has some base class templates and application-specific templates for different application areas including multi-body, irregular grid, and regular grid applications. For clarity, an MD simulation, as a running example, is used to help explaining how to program with the template hierarchy. To sum up, the specification of ensemble-based programming can be mapped on the concrete C++ programming platform. Thus, it can be implemented on concrete systems, including sequential, shared memory, and distributed memory systems. The implementation framework of the ensemble-based programming will be described in Chapter 6.

6. Implementation Framework

6.1. Overview

In this chapter, we present the implementation framework that implements the programming interface on different types of machines, i.e., sequential, shared memory, and distributed memory machines. The framework is currently designed for both multi-body and irregular grid applications. The support for regular grid applications and other application areas can be extended by adding new implementation libraries into the framework.

As it is shown in Figure 6.1, the framework consists of machine-specific libraries including a sequential library, an OpenMP-based library, and an MPI-based library. An ensemble-based program can be compiled and linked to executables by these libraries with different compiler commands and options. The compiler commands and options are presented in the Appendix.

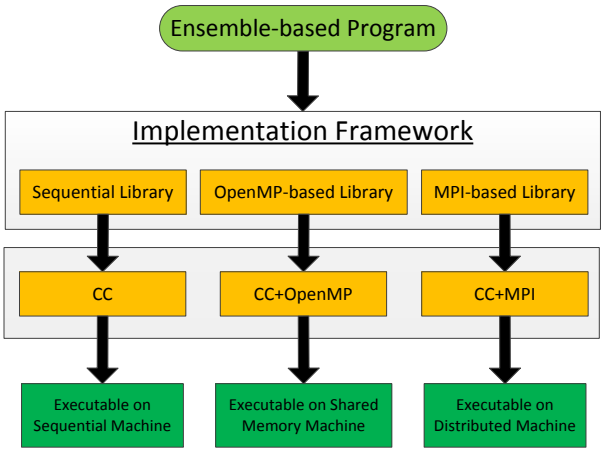


Figure 6.1.: Overview of the implementation framework

The sequential-based library provides a standard OOP implementation of the programming interface on sequential machines. Both the communication and parallel operations of the EPs in the ensemble are handled by a single-threaded process. It demonstrates the basic implementation of programming interface on one process.

The OpenMP-based library implements the programming interface on top of OpenMP. It translates an ensemble-based program to an OpenMP program executed on shared memory machines. The OpenMP library aggregates the computation of a group of EPs and binds it to a single thread. The parallel operations of the EPs in the ensemble are done by multiple threads in parallel. In addition, the topology can lead to optimized storage of the EPs and their shadow copies to keep the threads access local data on NUMAs more frequently.

The MPI-based library implements the programming interface in C++ with MPI. It employs both the domain decomposition and efficient graph partitioning algorithms to achieve optimal EP distribution and communication for multi-body and irregular grid applications. The communication between the processes is done by aggregating fine granular communication among EPs into coarser MPI messages to improve the efficiency of communication. The communication pattern among EPs is managed by the MPI-based library automatically.

The rest of this chapter is organized as follows: Section 6.2 introduces the implementation strategy of the sequential library; Section 6.3 describes the implementation and optimization of the OpenMP-based library on UMA and NUMA machines; Section 6.4 introduces the MPI-based library in terms of the EP distribution, communication optimization, topology management, and so on.

6.2. Mapping to Sequential Machines

6.2.1. Overview

This section presents the implementation of the sequential library to demonstrate how the ensemble-based programming interface is mapped on sequential machines. An ensemble-based program can be executed on sequential machines by linking to the sequential library. The entities defined in the programming interface are implemented in standard C++. The implementation of the sequential library includes the management of the ensemble, storage of EPs and their shadow copies, protection of "Fetch before Store" semantics, implementation of high-level *Ensemble* operations, management of topologies, and so on. In this section, the term "ensemble" and "topology" are used to represent the implementation objects of *Ensemble* and *Topology* defined in the programming interface.

6.2.2. *Ensemble* Management

There is only a single ensemble generated to implement the programming entity *Ensemble* defined in the programming interface. The ensemble is globally controlled by the master thread specified in the main function of an ensemble-based program. The overview of

the *Ensemble* implementation object is shown in Figure 6.2. It not only keeps memory space for storing EPs and their shadow copies (SCs) but also the references to the topologies, which maintain the access patterns between the EPs and their SCs. All the high-level *Ensemble* operations are implemented in C++, i.e., updating EPs' shadow copies, getting neighbor EPs, parallel execution of EPs' member functions, collective operations of EPs, and so on.

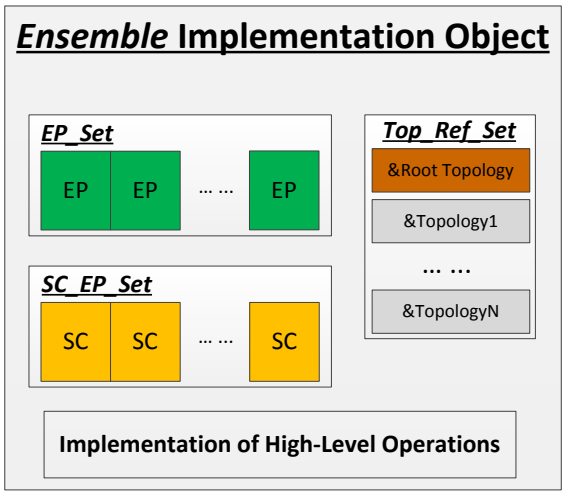


Figure 6.2.: *Ensemble* implementation object

Storage of Elementary Points

The ensemble keeps a C++ *vector* named *EP_Set* for storing the EPs. The *vector* is a generic container that supports dynamic insertion and removal of EPs efficiently. The type of the elements in the vector is EP, which is an user-defined class derived from the application-specific templates in the template hierarchy, *MultiBodyEP* or *IrrGridEP*. Based on the C++ *vector*, the EPs are stored in *EP_Set* following a linear sequence so that the ensemble is able to directly reference the EPs in *EP_Set* by their indices.

"Fetch before Store" Semantic Protection

In many scientific applications, the iterative computation of an EP is dependant on its neighbor EPs. Meanwhile, the EP's local information is required by its neighbor EPs as well. If the EP updates its local attributes before its neighbor EPs fetch them, the neighbor EPs will get "updated" information rather than the "original" information that they are supposed to get. The "Fetch before Store" problem happens in such a scenario. To efficiently avoid this problem, the ensemble stores EPs and their SCs separately in *EP_Set* and *SC_EP_Set*. Parallel operations can only modify the EPs in *EP_Set*, and

the SCs in *SC_EP_Set* are updated by *SC-Update* operations and read-only afterwards. Additionally, the *getNghbList* operations return the references to the SCs in *SC_EP_Set* rather than EP entities, which avoid redundant EPs-to-EPs memory copying as well.

Storage of EPs' Shadow Copies

As it is shown in Figure 6.3, each EP has a number of shadow copies based on the simple topology specified by the undirected graph. For example, EP1 has three SCs, i.e., EP0, EP2 and EP3. It means that if a *SC-Update* operation is called, data in EP0, EP2 and EP3 need to be copied into the memory space of EP0's shadow copies. However, there is some storage duplication if the *SC-Update* operation is called, since all the EPs copy their data into SCs of other EPs. As we can see from Figure 6.3, EP2 as a SC of EP0, EP1, and EP3 are stored three times, which is definitely not efficient in terms of memory utilization.

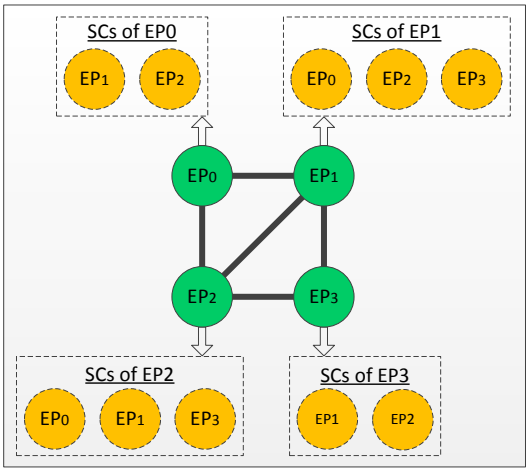


Figure 6.3.: Shadow copies of four EPs

In fact, all the SCs are read-only copies, which are only stored once but accessed for multiple times. Therefore, in the sequential implementation, the ensemble globally stores the SCs of the EPs in a C++ *vector* called *SC_EP_Set*. Similar to *EP_Set*, *SC_EP_Set* is a generic C++ *vector* with the type of EP. As it is shown in Figure 6.4, EP1 as a SC of EP0, EP2, and EP3 is only stored once in the ensemble rather than three times. This strategy can save a lot of memory space for storing the SCs if there are a large number of EPs stored in the ensemble.

According to the separate storage strategy, the size of *SC_EP_Set* equals to the size of *EP_Set*. Both the complete and partial information of the EPs can be copied into *SC_EP_Set* when a *SC-Update* operation is called, since the memory space for storing the SCs is fully allocated in *SC_EP_Set*.

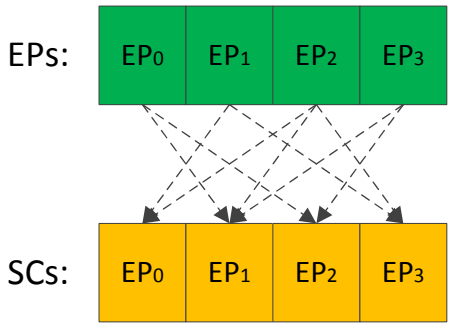


Figure 6.4.: Optimized arrangement of the shadow copies

getNghbList Operations

After an *SC-Update* operation was called, each EP can get its neighbor EPs by calling the *getNghbList* operation, which returns the pointers to the EPs in *SC_EP_Set* based on the topology specified in the *getNghbList* operation. For example, the neighbor EPs of EP1 are EP0, EP2 and EP3, which means that EP1 can get the addresses of *SC_EP_Set*[0], *SC_EP_Set*[2], and *SC_EP_Set*[3] as its neighbor EPs by calling the *getNghbList* operation. Then, EP1 can go through the neighbor list and get neighbor EPs' data for its local computation.

SC-Update Operations

The *SC-Update* operations are based on the topologies specified by the programmers. A *SC-Update* operation according to the root topology copies all the EPs in *EP_Set* into *SC_EP_Set* by a serial *for* loop. Each *EP_Set*[i] is copied into *SC_EP_Set*[i] respectively. A *SC-Update* operation according to a sub topology only updates the shadow copies of the EPs maintained by the sub topology. It tracks its base topology and gets the access pattern. It triggers a memory-to-memory copy from a subset of EPs in *EP_Set* to *EP_SC_Set*.

Parallel Operations

In the ensemble, *parallel* operations are implemented by *for* loops that go through all the EPs or a subset of EPs in *EP_Set* and trigger execution of their member functions. These operations are executed sequentially by a single-threaded process. The pseudo code of the implementation of a *parallel* operation is shown in Algorithm 6.1.

Algorithm 6.1 Pseudo code of parallel operation

```
template <typename Operation>
template <class Topology>
void parallel(Operation op, Topology*topology){
    for(the EPs in topology)
        op(EPs);
}
```

6.2.3. *Topology* Management

There are two types of topologies for multi-body and irregular grid applications namely multi-body topologies and irregular grid topologies. A multi-body topology is created by specifying a cut off radius, while an irregular grid topology is created by defining an id-based graph of EPs. According to the specified information, the sequential library manages the root topology, which keeps the neighbor EP list for all the EPs in the ensemble. The neighbor EP list is organized in two dimensional dynamic array implemented by C++ *vector*s. As can be seen from Figure 6.5, each row of the array is a *vector* of EP pointers, which are the addresses of the neighbor EPs of the EPs in *EP_Set*. Each EP in the ensemble is able to get its neighbor EPs from the topology directly.

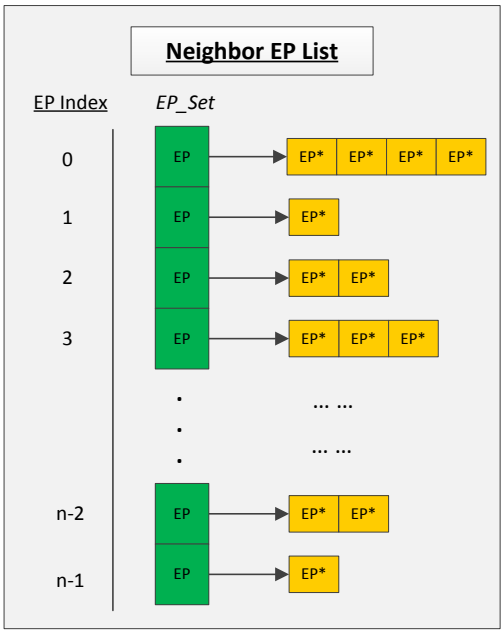


Figure 6.5.: Organization of neighbor EP list

MultiBodyTopology

The implementation of the multi-body topology is based on the linked cells (LC) algorithm, which works by subdividing the domain of a multi-body application into regular cells with an edge length equals to the cut-off radius specified in the root topology. The cut-off radius is a distance threshold that determines the interaction of two EPs. The root multi-body topology maintains a neighbor list for all the EPs in *EP_Set* according to the linked cells list. Based on the neighbor list, each EP can access its neighbor EPs in *SC_EP_Set* by referencing their indices.

Based on the LC algorithm, each cell represents a regular spatial domain with coordinates, e.g., [x, y] or [x, y, z] depending on the dimension of the simulation domain. It has neighbor cells that are closely located to itself with coordinates like [x, y-1], [x, y, z-1], [x+1, y, z-1], and so on. Each cell keeps addresses of the EPs that are spatially located in its spatial domain. The creation of the neighbor list in the topology is described in Algorithm 6.2.

Algorithm 6.2 Linked Cells (LC) algorithm

//an *EP* is kept in *Cell_ep*
1. Get the coordinate of *Cell_ep*
2. Calculate coordinatess of *Cell_ep*'s neighbor cells
   if the id of *Cell_ep* is [x,y,z], it has 26 neighbor cells in a 3D domain
   Cell_ep's neighbor cells are called *nghbCells*
3. Get neighbor cells *nghbCells*
   Their coordinates are [x±1, y±1, z±1]
4. Return *EP* pointers kept in the *nghbCells*
5. Determine the distances between the EPs in *nghbCells,* and *EP*
6. If the distance < cut-off Radius: Keep the *EP* pointers in *EP_neighbors*
   else ignore
7. Build neighbor EP list using these *EP* pointers

A multi-body topology can be updated by calling the *updateTopology* operation, since the EPs in the domain may move during execution. Therefore, the linked cells list as well as the neighbor list has to be recomputed according to the updated EPs' spatial information. The access patterns of the EPs are then changed accordingly.

IrrGridTopology

An irregular grid topology maintains the id-based graph of EPs, which typically consists of vertices and undirected edges. The vertices represent identifiers of the EPs in *EP_Set* and the edges represent EP-to-EP interactions. Two EPs require data from each other if and only if there is an edge connecting them. According to the id-based graph, the irregular grid topology generates a neighbor list of identifiers of EPs, which presents

the identifiers of the EPs' neighbor EPs. The neighbor EP list in the root topology can be built according to the neighbor list of identifiers. As it is shown in Figure 6.6, the irregular grid topology is specified by a CSR format representation. Based on the representation, an id-based undirect graph is created in the irregular grid topology. For example, the fourth line of the neighbor list in Figure 6.6 represents that the neighbor EPs of EP_3 are EP_1, EP_2, EP_4, and EP_5, which are stored in the memory location $SC\_EP\_Set[1]$, $SC\_EP\_Set[2]$, $SC\_EP\_Set[4]$, and $SC\_EP\_Set[5]$.

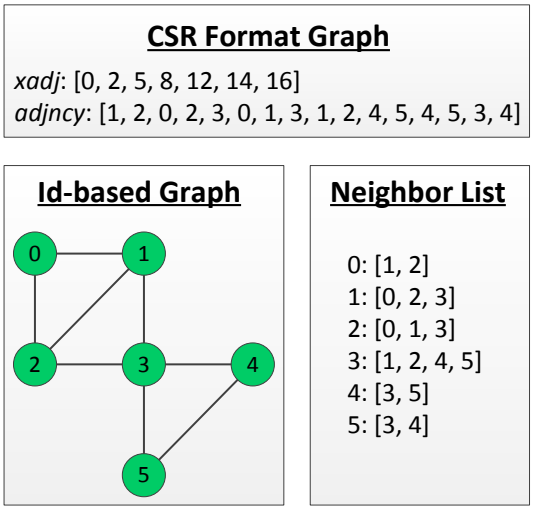


Figure 6.6.: Id-based graph and neighbor list

6.3. Mapping to Shared Memory Machines with OpenMP

6.3.1. Overview

This section presents the OpenMP-based library, which implements the programming interface on shared memory machines in C++ with OpenMP. By linking to the OpenMP-based library, an ensemble-based program is translated into an executable run by multiple threads. Similar to the implementation of the sequential library, all the entities in the programming interface are stored in the shared memory space and implemented in C++. In this section, the term "ensemble" and "topology" are used to represent the implementation objects of *Ensemble* and *Topology* on shared memory machines.

OpenMP uses a fork-join model of parallel execution, which meets the design approach of the ensemble-based machine model. The master thread defined in the machine model can be directly mapped to the OpenMP master thread. When the master thread in the machine model encounters high-level *Ensemble* operations that need to be executed in parallel, the master thread creates a thread team composed of itself and some additional threads to implement these operations. The OpenMP-based library manages the

parallel execution of multiple threads, and programmers don't need to take care of the multi-threading programming environment explicitly. After the *Ensemble* operations are accomplished, the master thread continues controlling the EPs in the ensemble sequentially. All the other threads in the team are shut down and wait to be summoned to join other teams for next *Ensemble* operations.

This section presents the management of the entities in the programming interface, i.e., *ElementaryPoint*, *Ensemble*, and *Topology*. In addition, this subsection mainly highlights the implementation and optimization of the OpenMP-based library on NUMA machines in terms of exploiting parallelism, synchronization of the thread team, machine-specific memory optimization, and so on.

6.3.2. *Ensemble* Management

Similar to the sequential library, there is only a single *Ensemble* implementation object during execution of an ensemble-based program. The ensemble is globally controlled by the master thread specified in the main function of the ensemble-based program. It maintains shared memory space for storing the EPs, their shadow copies, and references to the topologies. On shared memory machines, the communication among EPs is done by memory to memory copying or remote memory accesses from sockets to sockets. The high-level *Ensemble* operations are implemented in C++ with OpenMP.

6.3.3. OpenMP Support on NUMAs

As described in previous chapters, shared memory machines are typically classified into UMA and NUMA machines. On UMAs, the processors that share a connection to the global shared memory have the same access time to the shared memory. The implementation of the OpenMP-based library on UMAs is similar to the implementation of the sequential library except that it applies OpenMP to parallelize the *for* loops using OpenMP directives. This is not discussed in this section.

The other type of shared memory machines is NUMA, which has a global shared memory among processors like UMA does, but the memory is physically distributed among different sockets in the machine. Although all the CPUs on the sockets share the memory within a node, the accesses to CPUs' local physical memory are much faster than the non-local accesses to remote physical memory of other CPUs. This means that the memory overhead has dramatic effects on the performance of the OpenMP-based implementation on NUMAs. Since most modern powerful nodes are based on NUMA, this subsection will discuss details of the OpenMP-based library on NUMAs in terms of the storage of EPs, management of EP's shadow copies, implementation of high-level *Ensemble* operations, and so on.

Reallocation and Re-indexing to Manage EPs and their Shadow Copies

The ensemble stores all the EPs in a dynamic array named *EP_Set*, which is a generic array that supports dynamic insertion and removal of EPs. It is implemented as a dynamic EP array rather than a C++ *vector*, since the dynamic array is more flexible and efficient than the *vector*-based data structure in terms of manipulating the threads on NUMAs.

Based on the semantics of the machine model, EPs are inserted into the ensemble by using the *insertEP* operation executed sequentially. On NUMAs, the OpenMP-based library implements the *insertEP* operation by the master thread. This means that all the EPs are initially stored in the physical memory of the socket that is running the master thread. It is not efficient that the threads residing on other sockets have to access the EPs by non-local memory accesses when the ensemble-based program is executed by multiple threads. Therefore, the OpenMP-based library applies a reallocation strategy to distribute EPs across different physical memory of the sockets to reduce non-local EP accesses. The objective of the EP reallocation strategy is to guarantee that all the threads are able to mostly reference the EPs in their local physical memory.

The OpenMP-based library employs METIS[116] to distribute the EPs, since typical block-wise or round-robin decomposition methods are not efficient for irregular grid applications. METIS is a fundamental library, which consists of a number of executable programs and APIs to handle irregular data partitionning, e.g., the graph partitioning, mesh partitioning, matrix reordering tasks, and so on. To simplify the discussion, we present an irregular grid example on a two sockets NUMA. As it is shown in Figure 6.7, the id-based graph represents the topology of eight EPs, which need to be distributed on two different sockets by EP reallocation according to the partition array $[0, 1, 0, 1, 1, 0, 0, 1]$ generated from METIS. As can be seen from the figure, EP_0, EP_2, EP_5, and EP_6 are stored in the memory of Socket#0, the rest EPs are stored in the physical memory of Socket#1. It is assumed that this distribution generated by METIS guarantees that an equal number of EPs are stored in the memory of the two sockets and the number of non-local accesses is minimized.

In general, the OpenMP-based library applies an array called *indirection* to determine different threads to touch different sections of *EP_Set* in parallel. The *indirection* array is generated from the output of METIS. It keeps the identifiers of the EPs in *EP_Set*. The size of an array is $numEP/numSocket$ while the number of sockets is $numSocket$ and the number of EPs in *EP_Set* is $numEP$ (if $numEP$ is divisible by $numSocket$). The *indirection* array is organized in such a way. The the first $numEP/numSocket$ elements of the *indirection* array (indices from 0 to $numEP/numSocket-1$) stores the identifiers of EPs stored in the physically memory of socket#0. The second $numEP/numSocket$ elements (indices from $numEP/numSocket$ to $2 * numEP/numSocket - 1$) stores the identifiers of EPs stored in the physical memory of socket#1, and so on and so forth.

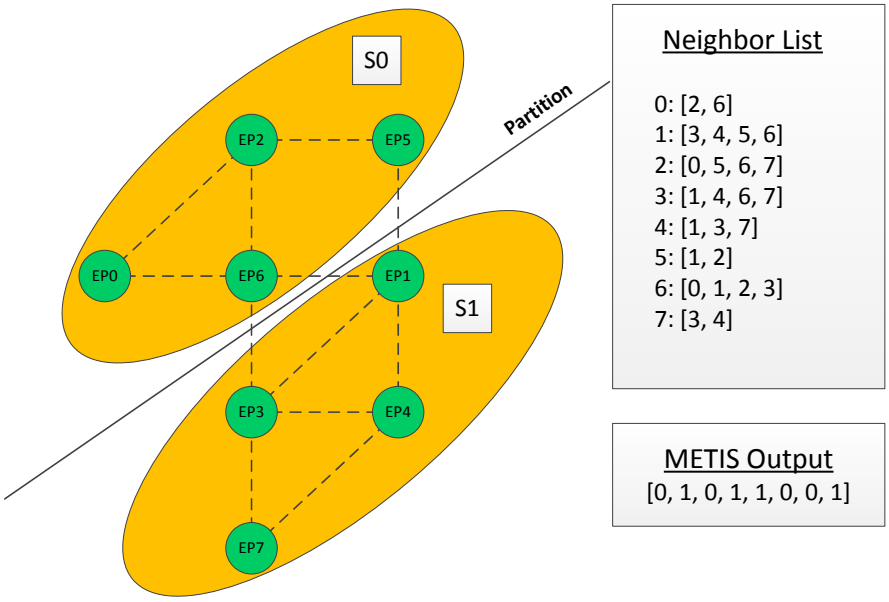


Figure 6.7.: EP partition on NUMAs

To simplify the discussion, we assume that there is only one thread is running on a single socket (multiple threads on a socket can be implemented by the OpenMP thread affinity). A simple example of EP reallocation is shown in Figure 6.8. The EPs are initially stored in *Buf_EP*, which is located in the physical memory of socket#0 after the *insertEP* operations are accomplished. During the EP reallocation, thread 0 is reponsible for the loop iteration space form 0 to 3, while thread 1 takes care of the iteraction index from 4 to 7. According to the *indirection* array, thread 0 touches elements $[0, 2, 5, 6]$, while thread 1 touches elements $[1, 3, 4, 7]$. Then, both threads residing on the two sockets touch different sections of the reserved *EP_Set* and initialize it by the EPs stored in the *Buf_EP*. *EP_Set* is distributed across the physical memory of the two sockets based on the "first touch" policy of memory. Additionally, once *EP_Set* was initialized by multiple threads, the memory space of *Buf_EP* can be freed.

The pseudo code of the EP reallocation is shown in Algorithm 6.3. As it is shown in the algorithm, the EP reallocation is implemented by a OpenMP parallel construct by multiple threads according to the array *indirection* generated from the output of METIS.

The major disadvantage of the EP reallocation according to the *indirection* array is the problem of the indirect accessing to EPs. As can be seen from Algorithm 6.4, a *parallel* operation is triggered on the EPs in *EP_Set* (the type of OpenMP loop scheduling is based on chunk). The *indirection* array is referenced to access the EPs in order to guarantee that all the threads work on the data in their local physical memory. Each thread needs to access the *indirection* array to reference *EP_Set*, since the EPs are already reallocated according to the *indirection* array. The indirect accessing happens

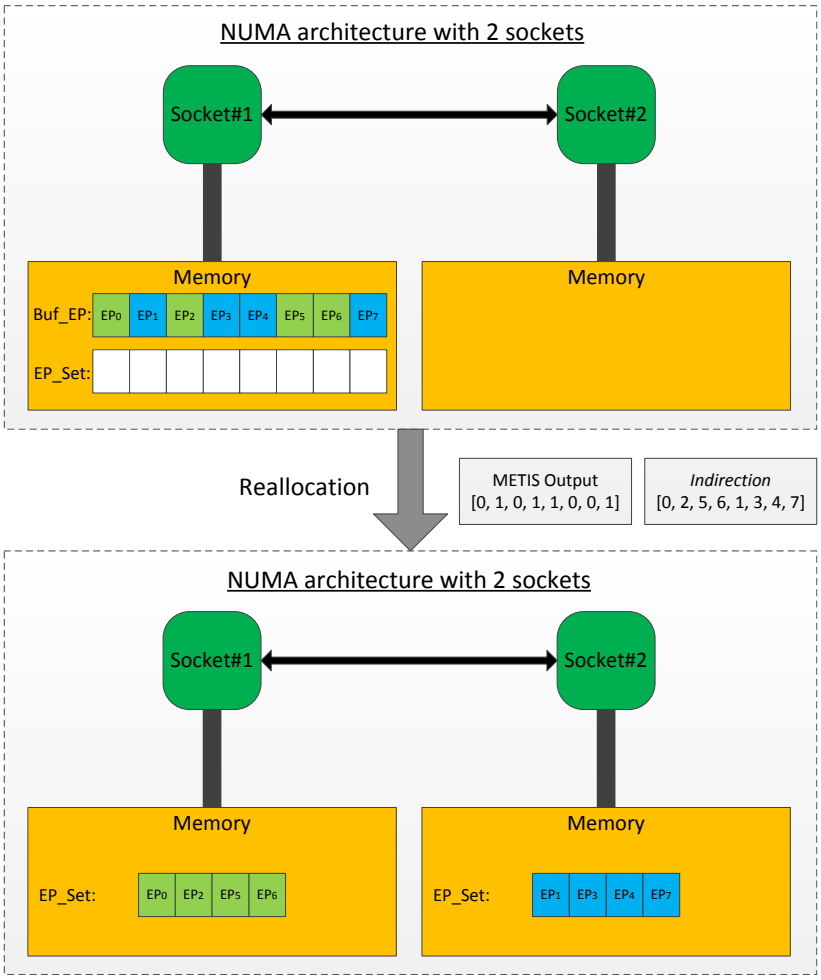


Figure 6.8.: EP storage based on reallocation

Algorithm 6.3 The EP reallocation

```
void reallocation(){
    EP *EP_Set = (EP*) malloc(numOfEPs * sizeof(EP));
#pragma omp parallel for
    for(i=0;i<numOfEPs;i++)
        EP_Set[i] = Buf_EP[indirection[i]];
    free(Buf_EP);
}
```

everytime when a *parallel* operation is called, which is really expensive and affects the execution performance significantly.

In order to avoid frequent accesses to the *indirection* array, the OpenMP-based library applies an EP re-indexing method. This method renumbers the indices of the EPs from 0 to n-1 after the EP reallocation is done. According to the partitionning out-

Algorithm 6.4 *parallel* operation based on *indirection*

```
template <typename Operation>
template <class Topology>
void parallel(Operation op, Topology*topology){
#pragma omp parallel for
      for(i=0;i<numOfEPs;i++)
           op(EP_Set[indirection[i]]);
      free(Buf_EP);
}
```

put of METIS, we create *indexOrigin2New* and *indexNew2Origin* arrays to manage the re-indexing translation. The *indexOrigin2New* array is used for the translation from orignial indices to new indices, while *indexNew2Origin* is used for the translation from the new indices back to orignial indices. Both *indexOrigin2New* and *indexNew2Origin* are organized in such a way. The *indexNew2Origin* array and the *indirection* array described above have the same organization. The *indexOrigin2New* array is generated from *indexNew2Origin* according to the rule:

$$indexOrigin2New[indexNew2Origin[i]] = i;$$

As it is shown in Figure 6.9, the original EPs' indices $[0, 2, 5, 6, 1, 3, 4, 7]$ are translated to their new indices $[0, 1, 2, 3, 4, 5, 6, 7]$. According to the rull above, the *indexNew2Origin* array is: $[0, 2, 5, 6, 1, 3, 4, 7]$, while the *indexOrigin2New* array is $[0, 4, 1, 5, 6, 2, 3, 7]$. After the EP reallocation and the re-indexing are accomplished, the *Ensemble* operations can directly reference EPs in *EP_Set* by their new indices without referencing *indirection*.

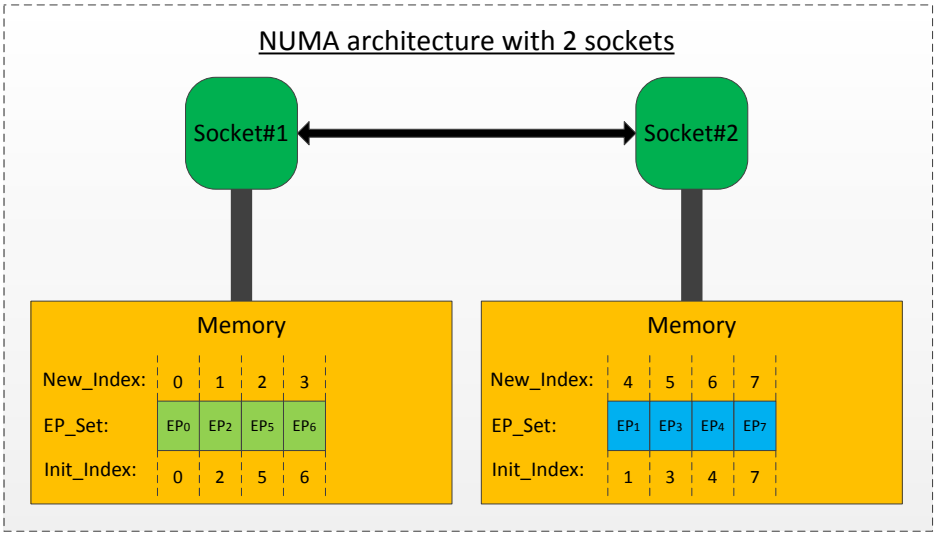


Figure 6.9.: EP re-indexing

Additionally, the distribution of *EP_Set* is the same as the distribution of in *SC_Set*, which stores the SCs of the EPs. *SC_Set* is stored across different physical memory

following the EP reallocation and re-indexing strategy. Based on this SC allocation strategy, most EPs in *EP_Set* can get their shadow copies in *SC_Set* locally. Remote memory accesses only happen when a few local EPs require data from the EPs stored in remote physical memory. The overview of the storage of the EPs and their shadow copies is shown in Figure 6.10. According to the "new" neighbor list, each EP can properly access its neighbor EPs *SC_Set* in mostly by local memory accesses.

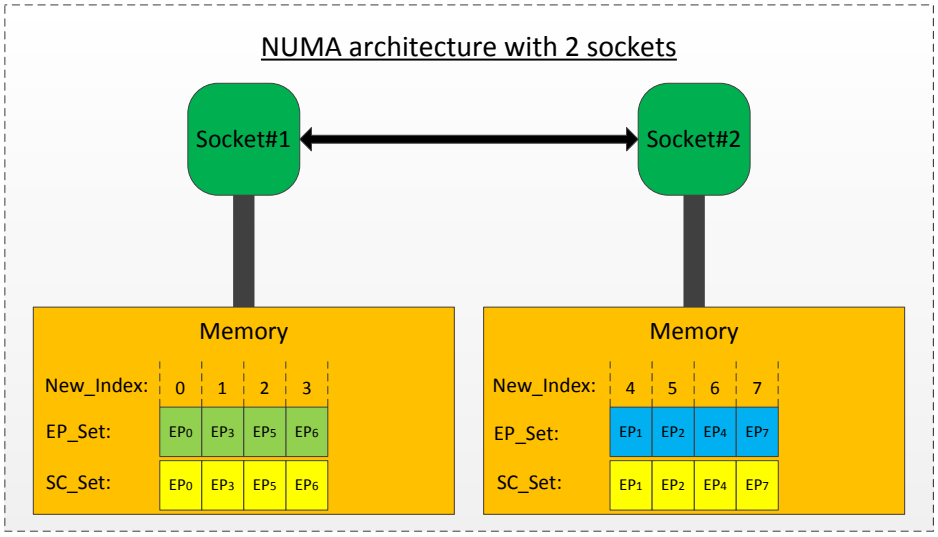


Figure 6.10.: The storage of EPs and their shadow copies

Both *EP_Set* and *SC_EP_Set* are stored in different physical memory according to the EP reallocation and re-indexing strategy described above. *SC-Update* operations are implemented by multiple threads using parallel constructs in OpenMP. The first *SC-Update* operation according the root topology will trigger all the threads to initialize *EP_SC_Set* respectively and store it into different local physical memory of the sockets. After the *SC-Update* operation is accomplished, the EPs in *EP_Set* can get their shadows copies mostly by local memory accesses. The pseudo code of a SC-Update operation according to the root topology is shown Algorithm 6.5.

Algorithm 6.5 *EP_SC_Set* initialization

```
template <typename Operation>
template <class Topology>
void update(Operation op, Topology*topology){
if(topology==root){
#pragma omp parallel for
    for(i=0;i<topology->numOfEPs;i++)
        SC_EP_Set[i] = EP_Set[i];
}
}
```

Although the memory-to-memory copying from a physical memory to other physical

memory is expensive and increase the memory overhead, the OpenMP-based library employs the EP reallocation and re-indexing strategy in order to reduces the number of non-local memory accesses during execution. The EP reallocation and re-indexing strategy is handled only once, but the information generated from the strategy can be reused for a long period of time. It is assumed to achieve better performance especially for scientific irregular applications.

Parallel Operations

The *parallel* operations of EPs' member functions are implemented by parallel constructs of OpenMP. A number of threads are created by OpenMP directives when a *parallel* operation is triggered. Each thread is responsible for the execution of a subset of EPs' member functions. As it is shown Algorithm 6.6, the OpenMP *parallel for* goes through the EPs in *EP_Set* and trigger parallel execution of their member functions specified by the programmers.

Algorithm 6.6 Pseudo code of parallel operation

```
template <typename Operation>
template <class Topology>
void parallel(Operation op, Topology*topology){
#pragma omp parallel for
    for(i=0;i<topology->numOfEPs;i++)
        op(EP_Set[i]);
}
```

Getting Neighbor EPs

The ensemble implements the *getNghbList* operation, which is called to return neighbor EPs of an individual EP. As described in previous sections, the *getNghbList* operation returns a *vector* of pointers to the neighbor EPs in *SC_Set* according to the topology specified in the operation. The major steps of the implementation are presented as follows:

1. Get the EP's "original" index and translate it to its "new" index;
2. Find the "new" indices of its neighbor EPs in the "new" neighbor list generated from the root topology;
3. Return a vector of pointers to the neighbor EPs in *SC_Set* according to their "new" indices.

Usually, the *getNghbList* operation is called by a *parallel* operation, and implemented by multiple threads. Each thread is responsible for the *getNghbList* operation of a subset of EPs in the ensemble. On NUMAs, the implementation of the *getNghbList* operation is combined with the EP reallocation and re-indexing scheme, which guarantees that

most of the EPs can get their neighbor EPs in the local physical memory where they are stored.

Collective Operations

Collective operations are implemented similar to the implementation of parallel operations by parallel constructs in OpenMP. Take a reduction operation as an example, it is implemented by an OpenMP reduction construct. The implementation of a collective operation is shown in Algorithm 6.7.

Algorithm 6.7 Pseudo code of collective operation

```
template <typename GetOperation>
template <class Topology>
void reduceOp(GetOperation getOp, Topology*topology, int reduceType, double*result){
#pragma omp parallel for reduction(reduceType:var)
    for(i=0;i<topology->numOfEPs;i++)
        var = var [reduceType] (*getOp(EP_Set[i]));
    *result = var;
}
```

6.3.4. *Topology* Management

Similar to the sequential library, a multi-body topology is created by specifying a cut off radius for multi-body applications, while a multi-body topology is created by specifying an id-based graph for irregular grid applications. The topologies include the root topology and its sub topologies, which are shared among all the OpenMP threads on a shared memory machine. The topology management on shared memory machines is not discussed in details in this sub section.

MultiBodyTopology

Similar to the sequential library, a multi-body topology is based on the linked cells (LC) algorithm as well. The neighbor EP list in the root multi-body topology is created based on the LC algorithm.

IrrGridTopology

An irregular grid topology manages the communication pattern of the EPs in *EP_Set* by keeping an id-based graph. The irregular grid topology generates the *indexOrigin2New* and *indexNew2Origin* by using METIS. The neighbor EP list in the root irregular grid topology is created based on the EP reallocation and EP re-indexing strategies described in previous subsections.

6.4. Mapping to Distributed Memory Machines with MPI

6.4.1. Overview

The MPI-based library in the framework is designed to implement the programming interface on distributed memory machines by a number of processes. The overview of the mapping from the machine model to a distributed memory machine is shown in Figure 6.11. In this section, the term "ensemble" and "topology" are used to represent implementation objects of *Ensemble* and *Topology* that are distributed across different processes.

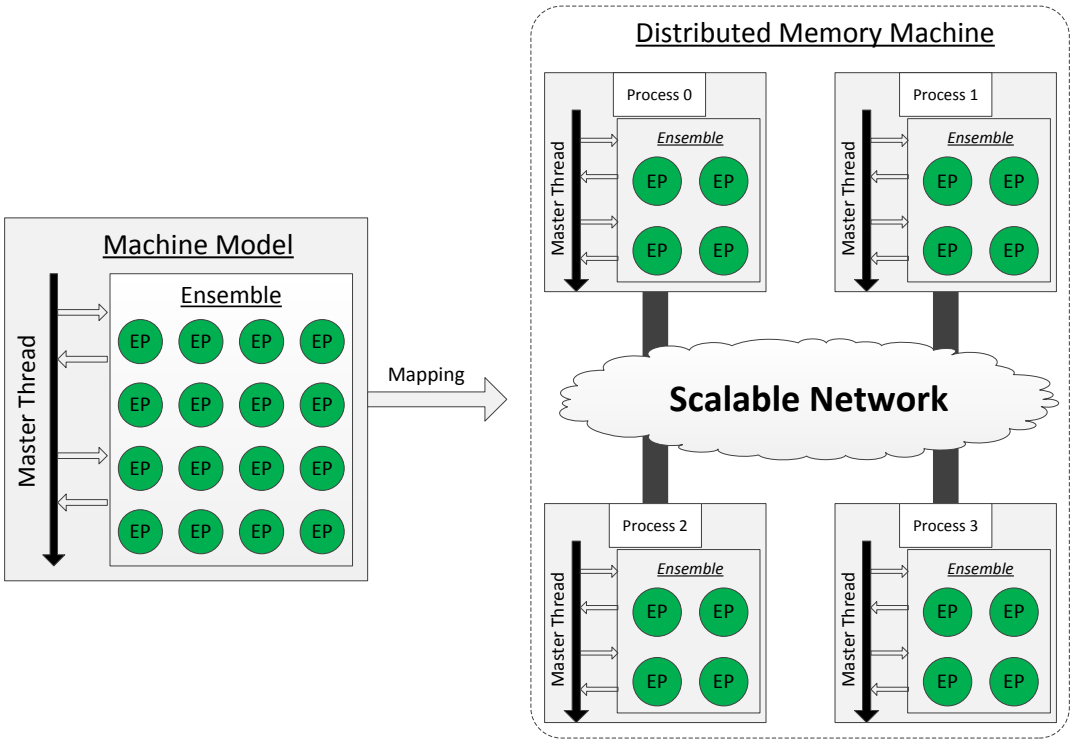


Figure 6.11.: Overview of MPI mapping

The master thread controls the communication and computation of all the EPs in the machine model. As the machine model is mapped on a distributed memory machine, the master thread is duplicated across all the processes based on a SPMD fashion. Each process keeps an ensemble, which stores a subset of the EPs in the ensemble, shadow copies of the EPs, and references to topologies, and so on. Each master thread manages the computation and communication of a local ensemble. The computation is handled by each process locally, while the communication is done by MPI among processes. Compared to the sequential and OpenMP-based library, it is more complicated for the MPI-based library to implement *Ensemble* on distributed memory systems in terms of

the EP distribution, management and distribution of topologies, communication among processes, and so on.

The entities described in the programming interface namely *ElementaryPoint*, *Ensemble*, and *Topology*, are implemented in a distributed fashion. The high-level *Ensemble* operations are implemented by the processes controlled by the duplicated master threads. For example, the *parallel* operations as well as the collective operations triggered on the EPs in the ensemble are translated into local operations done by each process respectively. The *SC-Update* operations are implemented by local memory-to-memory and communication among processes for exchanging remote EPs.

The mapping from the machine model to a distributed memory machine can keep the semantics of the programming interface and exploit parallelism of ensemble-based programs. All the low-level mapping details are handled by the MPI-based library automatically.

6.4.2. Storage of Elementary Points

The EPs are stored in a C++ *vector* named *loc_EP_Set*. The EPs in *loc_EP_Set* are referenced by the process using their local indices. Each process keeps different subsets of the EPs in the ensemble according to an EP distribution, which is determined by the root topology and decomposition algorithms.

The EPs are inserted into the ensemble sequentially by the master thread, and each process gets the EPs and stores them into its *loc_EP_Set* according to an initial EP distribution strategy, e.g., block wise, round robin, or other types of regular distribution. After the root topology is inserted into the ensemble, all the processes trigger a collective communication to get an optimal EP distribution determined by the output of METIS. The EP distribution generated from METIS balances the computational load and communication efficiency among processes for irregular applications.

Topologies of multi-body applications can be updated during execution, which may cause changes of the EP distribution. The EPs in *loc_EP_Set* can be sent to or received from other processes according to a new EP distribution. Take an MD simulation based on domain decomposition as an example, the EPs in *loc_EP_Set* change if the EPs move out of local domains.

Storage of EPs' Shadow Copies

The storage of SCs is determined by the EP distribution, the communication pattern among processes, and the root topology as well. On distributed memory machines, the SCs of the EPs in *loc_EP_Set* are stored in *loc_SC_Set*, which is kept in each process. *loc_SC_Set* is implemented as a C++ *vector*. It consists of SCs of local EPs

loc_SC_Set and SCs of EPs that are located in remote processes. All these SCs of remote EPs are received from remote processes and copied into *loc_SC_Set* so that each EP in *loc_EP_Set* can directly get its neighbor EPs in *loc_SC_Set* by using the *getNghbList* operation.

The topology is responsible for maintaining the SCs and translates the remote indices into local ones automatically. For instance, an EP requires a SC of a remote EP. It is received from a remote process and copied into *loc_SC_Set*. The global index of an EP is translated into its local index that can be referenced by the local process directly.

6.4.3. Implementations of *Ensemble* Operations

SC-Update

In the MPI-based library, a *SC-Update* operation not only copies the EPs from *loc_EP_Set* to *loc_SC_Set*, but also triggers communication among processes according to a topology specified in the operation. The implementation of local memory-to-memory copying from *loc_EP_Set* to *loc_SC_Set* is the same as the sequential library. In addition, a *SC-Update* operation triggers communication among a group of processes according to certain topologies and EP distribution algorithms, since the EPs in *loc_EP_Set* also require SCs of EPs located in remote processes. After a *SC-Update* operation is accomplished, the EPs in the receive buffer are updated to *loc_SC_Set* and can be referenced by the local process using their local indices. Details of the communication scheme among processes for both multi-body and irregular grid applications will be described in the following subsections.

getNghbList

A *getNghbList* is a local operation implemented on each process. It only references local EPs stored in *loc_SC_Set*, since it is assumed that remote EPs are already obtained by *SC-Update* operations from remote processes. A *getNghbList* operation is called to return the pointers to the EPs that are stored in *loc_SC_Set* according to the topology specified in the operation. The major steps of the implementation of a *getNghbList* operation are described below:

1. An EP pointer and a topology pointer is provided to the *getNghbList* operation;
2. The id of the EP is obtained and translated into a local index;
3. It is decided whether the provided topology is the root topology;
4. If yes, return the neighbor EPs in the topology according to the local index. The neighbor EPs are a vector of EP pointers, which is already generated in the topology after *SC-Update* is called;

5. If not, go to the base topology of the provided topology and get the neighbor EPs according to Step 4.

After the *getNghbList* operation is accomplished, each individual EP can access its neighbor EPs in the return vector of EP pointers for local computation.

Parallel Operations

The parallelism is exploited by multiple processes executing EPs' member functions in parallel. Each process executes a *for* loop that goes through all the EPs in *loc_EP_Set* and triggers sequential execution of the EPs' member functions by the process. A *parallel* operation is accomplished by all the involved processes, since the data from remote processes are copied into the shadow copies by calling *SC-Update* operations. The implementation of a *parallel* operation is shown in Figure 6.12.

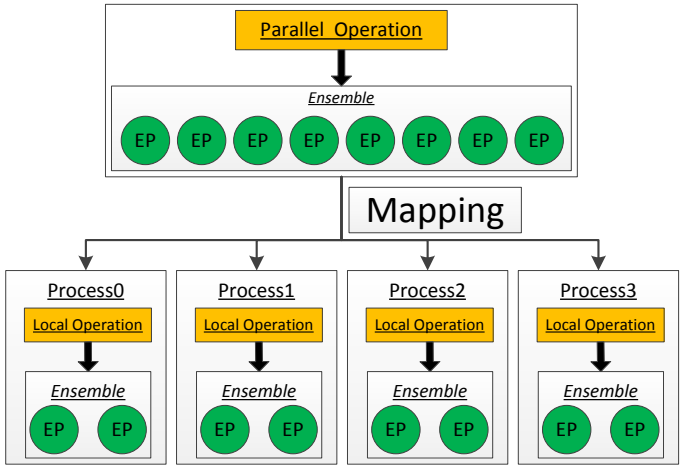


Figure 6.12.: MPI mapping of *parallel* operation

A *parallel* operation based on the root topology triggers indentical operations on each process. It goes through all the EPs in *loc_EP_Set* by a *for* loop, and triggers execution of the EPs' certain member functions. The pseudo code of the operation on a single process is shown in Figure 6.12.

Algorithm 6.8 Parallel operation implemented on a single process

```
template <typename Operation>
template <class Topology>
void parallel(Operation op, Topology*topology){
    for(i=0;i<EP_Set.size();i++)
        op(EP_Set[i]);
}
```

A *parallel* operation based on a sub topology triggers the execution of a subset of EPs' member functions by multiple processes. Each process determines whether there are

some EPs within the sub topology stored in its *loc_EP_Set* by checking the *glo2Loc* array. This array translates EPs' global indices to their local indices in the process (*glo2Loc*[*global*] = -1 if and only if the EP with index *global* is not stored in the process). The implementation of a *parallel* operation on a single process is shown in Algorithm 6.9. As can be seen from the pseudo code, the parallel operation is implemented by calling a for loop that goes through all the sub-indices of the EPs in the sub topology. It determines whether an EP is locally stored in *loc_EP_Set*, and triggers a certain operation on the EP. It is not recommended to use a *parallel* operation according to a sub topology, since the indirect array access is necessary in the operation, which is not efficient.

Algorithm 6.9 *parallel* operation based on a sub topology

```
template <typename Operation>
template <class Topology>
void parallel(Operation op, Topology*sutTop){
    for(i=0;i<subTop->subIndices().size();i++){
        //if the specified
        if(globalToLocal[topology->subIndices()[i]] == -1)
            continue;
        else
            op(EP_Set[glo2Loc[topology->subIndices()[i]]]);
    }
}
```

Collective Operations

A *collective* operation of EPs is translated into local collective operations and collective operations among MPI processes. Each process is responsible for a *collective* operation of the EPs in *loc_EP_Set*. Different processes cooperate in the collective operations. For example, broadcast and reduction operations can be implemented efficiently. The overview of the translation of a *collective* operation is shown in Figure 6.13. A *collective* operation is triggered on the EPs in the ensemble, it is translated to a hierarchical collective operation among processes and local EPs within a single process.

6.4.4. EP Distribution and Communication Management

On distributed memory machines, the EPs are distributed across different processes. The EP distribution can significantly affect the implementation efficiency of the programming interface in terms of the computational load balance and communication overhead. In principle, there are typically two major objectives that an EP distribution strategy needs to satisfy. The first objective is to keep almost an equal amount of EPs stored in each process to keep the equal computational workload. The second one is to minimize the volume of communication among processes when triggering *SC-Update* operations. Typical block-wise or round-robin EP distribution algorithms can't satisfy these objectives especially for multi-body applications and irregular grid applications.

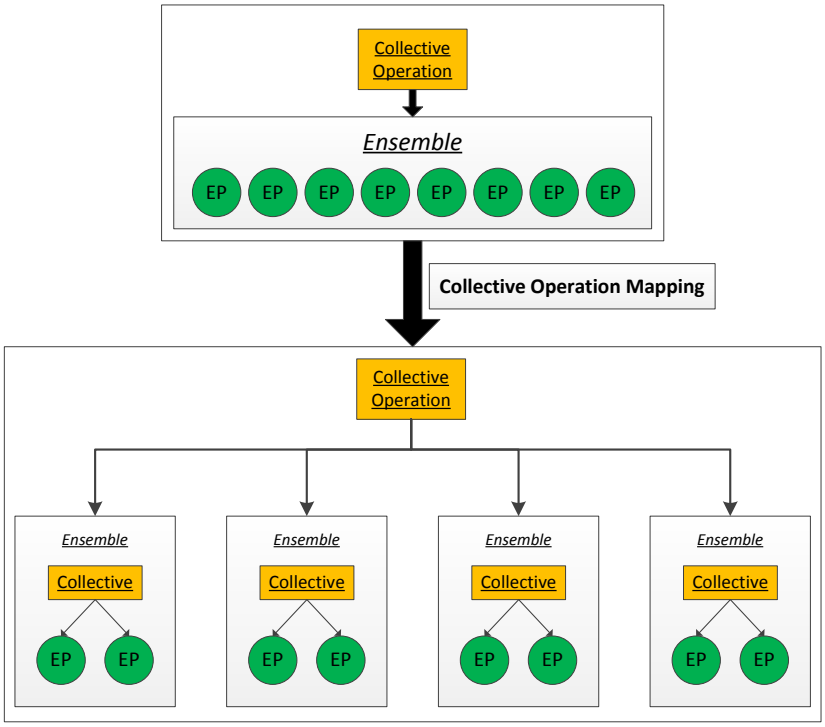


Figure 6.13.: MPI mapping of collective operation

Therefore, the MPI-based library employs METIS to partition the EPs according to the root topology specified by the programmers. METIS[116] primarily applies either the multi-level recursive bisection or the multi-level k-way partitioning paradigms to partition irregular graphs. Both these methods are able to produce high quality partitions. After the partitioning by METIS is accomplished, a collective communication among all the processes is triggered to get a target EP distribution from the initial EP distribution. The target EP distribution is kept during execution of an ensemble-based program, and changed only when a global redistribution is triggered. According to the EP distribution and the root topology, the communication among processes is managed by the MPI-based library automatically when a *SC-Update* operation is called.

EP distribution of Multi-body Ensembles

The EP distribution across different processes is based on the combination of the domain decomposition method and the parallel linked cells (PLC) algorithm, which is a parallel version of the linked cell algorithm. The domain decomposition method is applied to spatially partition the domain of a multi-body application into a number of subdomains. Each process is responsible for the computation and communication of the EPs located in a subdomain. The computation of EPs is handled locally, while the communication

among processes is determined by the linked cells. The PLC algorithm is combined with the domain decomposition method in order to manage the EP distribution and the communication pattern among processes.

As it is shown in Figure 6.14, the regular subdomain is subdivided into three types of cells, e.g., inner cells, boundary cells, and halo cells according to their spatial location. The pointers to the EPs located in the sub domain are kept in inner cells and boundary cells. Halo cells keep the pointers to the EPs that are received from other sub domains.

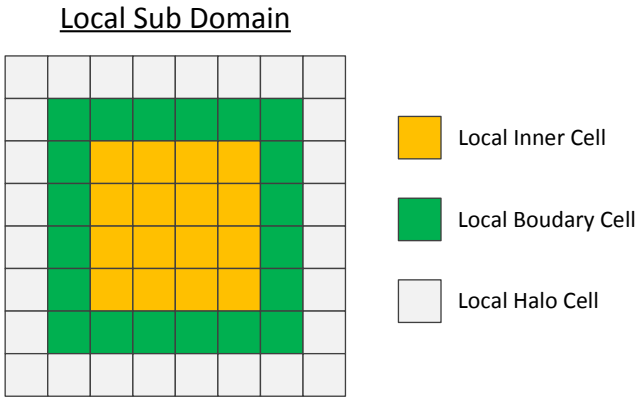


Figure 6.14.: Cell classification

Based on the domain decomposition and the PLC algorithm, each process maintains a sub domain formed by a set of linked cells described above. The pointers to the EPs in *loc_EP_Set* are kept in the inner cells and boundary cells, while the pointers to the EPs received from remote processes are kept in halo cells. According to the linked cells, each EP in is *loc_EP_Set* is able to reference its neighbor EPs for local computation.

Each process only communicates with its certain neighbor processes rather than all the other processes when a *SC-Update* operation is called. It improves the communication efficiency significantly and the complexity of communication among processes has reduced apparently. Figure 6.15 illustrates the communication scheme between two sub domains in the context of a 2D domain. As can be seen from this figure, the 2D domain is partitioned into two regular sub domains that are maintained by two processes. Both the processes maintain a subdomain formed by local inner cells, boundary cells, and halo cells for keeping the pointers of the local EPs and EPs received from remote processes. While a *SC-Update* operation is triggered, each processe sends EPs located in boundary cells to the other instance and stores the received EPs in *loc_SC_EP_Set*, which is managed by halo cells. Then, each EP is able to reference its neighbor EPs in the local process for computation.

Both the advantages and disadvantages of the combination of the PLC algorithm and domain decomposition method are described as follows:

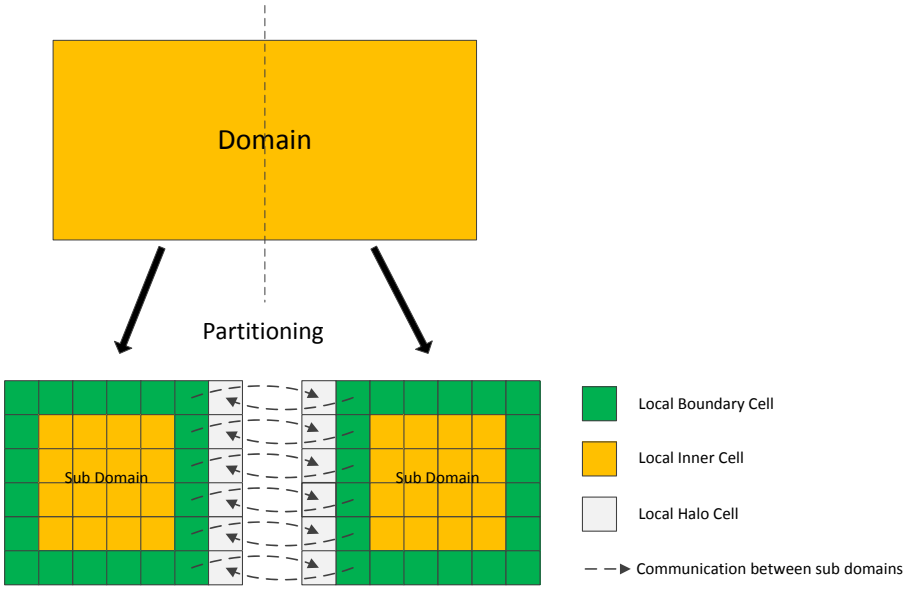


Figure 6.15.: Communication between sub domains

1. Based on the PLC algorithm, each process can easily manage its local EPs and remote EPs. For example, the process is able to determine the neighbor EPs of an EP by referencing the linked cells. The process can also receive EPs from remote processes and save them into appropriate location according to the linked cells.
2. The PLC algorithm manages the movement of EPs efficiently. If an EP moves from one cell to another cell, the local process can change the communication accordingly. If the other cell is in the region that belongs to a remote process, a point-to-point communication is triggered between two processes.
3. The communication among processes is easily determined by linked cells especially for multi-body applications with a cut-off radius.
4. The communication efficiency might be worse than the fine granular communication efficiency because of redundant communication. As it is shown in Figure 6.16, three EPs are sent from the boundary cells of P0 to halo cells of P1 according to the PLC algorithm. However, only the EP in green is really needed by the target EP. It increases the communication overhead among processes, since the MPI-library coarses the communication pattern among EPs to communication pattern among a group of EPs in cells.
5. Based on the PLC algorithm, the communication between two processes might be more than necessary, since regular cubics are not standard cycles. As can be seen from Figure 6.17, three EPs in a boundary cell of P0 are sent to a halo cell of P1 according to the PLC algorithm. However, there are no EPs in P1's boundary cells,

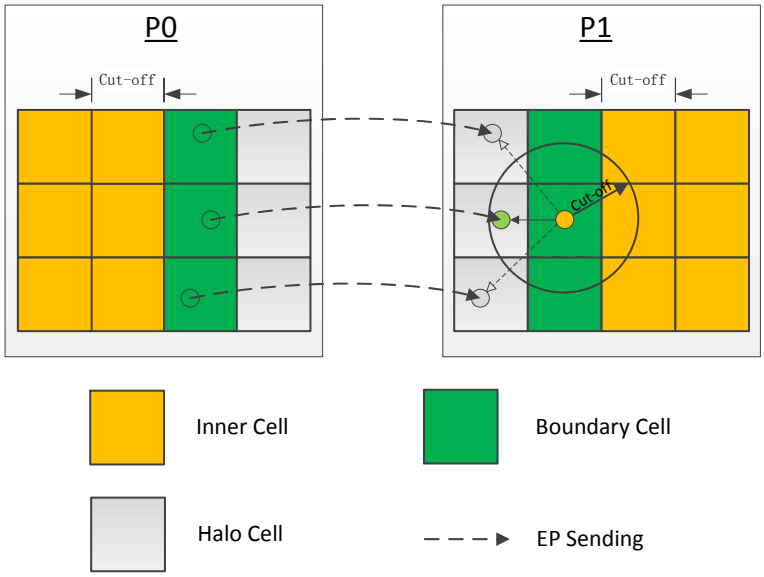


Figure 6.16.: Redundant communication (cutoff)

which mean that there is no computation happened based on the EPs received from P0. These EPs on P0 are unnecessarily sent to P1. It increases the communication overhead among processes.

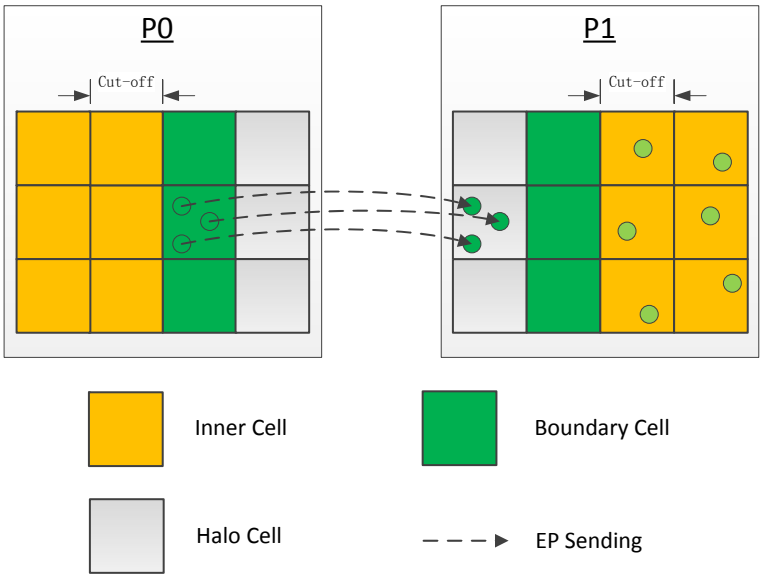


Figure 6.17.: Redundant communication (empty cells)

In spite of these disadvantages described above, the PLC algorithm is a relatively efficient algorithm for managing EPs in terms of the EP distribution, communication patterns

of EPs, movement of EPs, and so on. Therefore, we integrate the PLC algorithm and domain decomposition method into the MPI-based library.

Optimized EP Distribution of Multi-body Ensembles

Typically, the block-wise domain decomposition with linked cells is not flexible for multi-body applications in terms of load balancing, since EPs are not always evenly distributed in the simulation domain during runtime. It cannot guarantee that each process keeps an equal number of EPs according to the regular block-wised domain decomposition. Although the communication pattern among processes is regular, the load unbalancing problem cannot be avoided. Therefore, the MPI-based library applies an irregular cell-based decomposition algorithm using METIS in order to optimize communication among processes. The major steps of the algorithm are described below:

1. In principle, process 0 is responsible for the EP distribution using METIS, it collects the identifiers and coordinates of all the EPs from other processes.
2. Process 0 creates a cell-based graph according to the coordinates of the EPs. As can be seen from Figure 6.18, the 2D global domain is represented as a linked cell list with 16 cells. According to the organization of the cells and their communication scheme, a cell-based graphis organized in such a grid-based fashion. A vertex of this graph is a cell with size of the cut-off radius and maintains of a number of EPs. An edge represents the communication between two cells. The weight of a vertex in the graph is the number of EPs. The weight of an edge is the communication volume between the two cells that are connected by the edge.

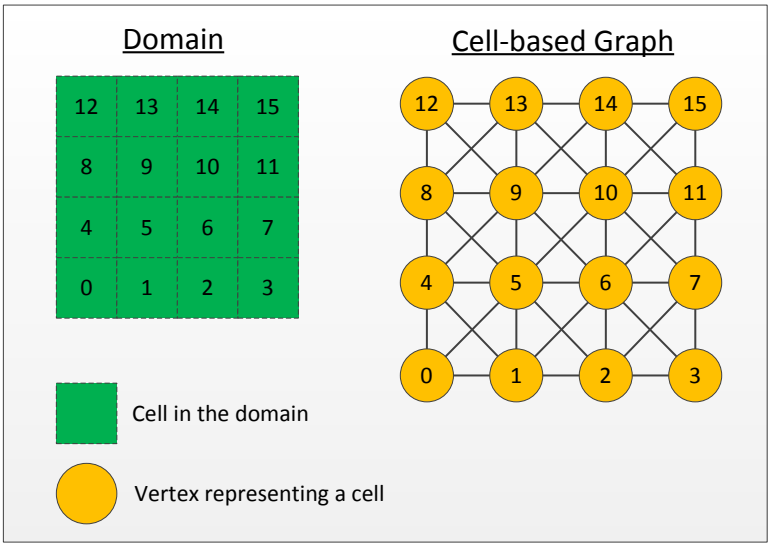


Figure 6.18.: Cell-based graph

3. Process 0 applies METIS to partition the cell-based graph, and the cell partition result is generated in an array named cell partition array. An example is shown in Figure 6.19, process 0 keeps the sub domain consisting of cell 0, cell 1, and cell 4. The shape of sub domains is irregular because the EPs are not evenly distributed in the global domain and each cell maintains a different number of EPs. The cell-based partitioning by METIS can obtain a balanced EP distribution and guarantee that the communication among processes is already optimized.

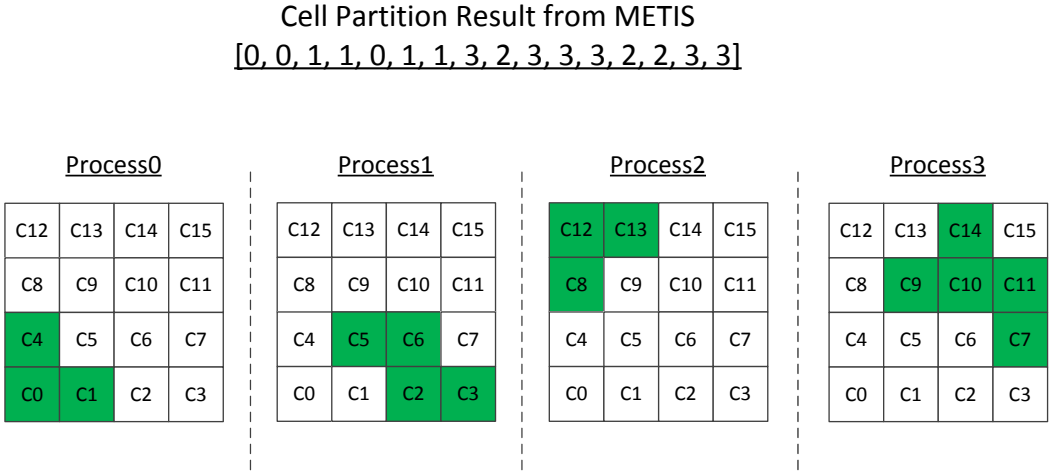


Figure 6.19.: Cell partition

4. Process 0 broadcasts the result of cell-based partitioning to all the other involved processes so that each process keeps the decomposition information of the cells.

5. According to the result of cell-based partitioning, each process is able to determine the communication scheme with its neighbor processes. As it is shown in Figure 6.20, process 0 needs to send cell 1 to process 1, and send its cell 4 to process 1, process 2, process 3 respectively. In addition, process 0 keeps cell 2, cell 5, cell 6, cell 8, and cell 9 as its halo cells. Among the halo cells, cell 2, cell 5, and cell 6 are from process 1, cell 8 and cell 9 are from process 2 and process 3 respectively.

This optimized EP distribution algorithm is based on the PLC algorithm and domain decomposition method. It generates a balanced EP distribution and optimizes the communication among processes. According to the result of cell partitioning, each process is able to manage the neighbor EP list and the communication scheme with its neighbor processes deterministically.

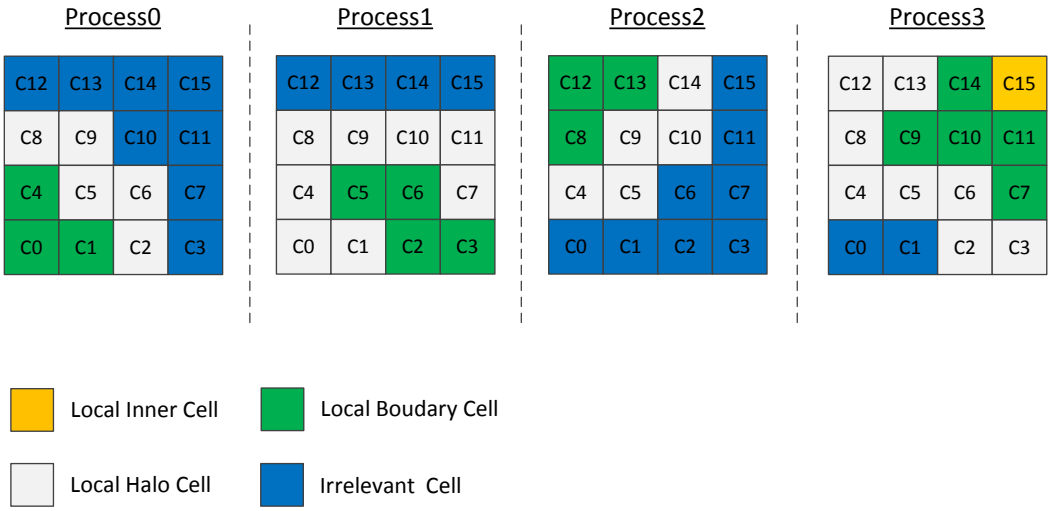


Figure 6.20.: Communication pattern based on irregular cell partitioning

EP Distribution of Irregular Grid Ensembles

The EP distribution for irregular grid applications is to partition all the EPs into multiple subsets according to the result of partitioning the id-based graph specified in the root topology. As it is described in the previous chapter, the vertices of the id-based graph represent identifiers of the EPs and the undirected edges represent EP-to-EP communication. The partitioning of the id-based graph is accomplished by the METIS library, which guarantees that each process keeps almost an equal amount of EPs and the communication among all the processes is minimized. After the id-based graph is partitioned into K sub graphs (K is the number of processes), each process gets a set of identifiers and stores the EPs with these identifiers. The edge-cuts among different sub graphs represent communication among processes.

6.4.5. *Topology* Management

Both *MultiBodyTopology* and *IrrGridTopology* implementation objects are managed by the MPI-based library in terms of the communication pattern of EPs, the communication among processes, creation of neighbor EP list for *getNghbList* operations, and so on. In addition, the MPI-based library is responsible for the update of both the multi-body and irregular grid topology during execution. The topology is managed in a distributed fashion. Each process keeps the root topology, which maintains the neighbor EP list for all the local EPs. The neighbor EP list is organized in two dimensional dynamic array described in Section 6.2. It keeps the addresses of the neighbor EPs of the EPs in *loc_EP_Set* so that each EP in *loc_EP_Set* is able to get its neighbor EPs from the local topology.

Creation of *MultiBodyTopology*

The generation of the neighbor EP list in the multi-body topology is based on the PLC algorithm. This algorithm is presented in Algorithm 6.10 illustrating that how a single process generate neighbor EP list based on the linked cells. The assumption is that the communication is already accomplished so that all the EPs are available in inner, boundary, and halo cells of all the processes.

Algorithm 6.10 Creation of neighbor EP list

1. Forall *ep* in local process
   1) Get cell id *localidCell* of *ep*
   2) Get ids of neighbor cell *localidCell*
   3) Get neighbor cells *localNghbCells*
   4) Get *EPs* in the *localnghbCells* and determine whether the distance between the EPs in *localNghbCells* and *ep* is smaller than the cut-off radius
   5) If yes, put the address of EP
   5) Create neighbor EPs for *ep*

End for

2. *updateTopology()* and go to Step 1

Each process keeps a root multi-body topology, which manages the neighbor EP list for all the local EPs according to the linked cells. It maintains the communication patterns of the EPs in *loc_EP_Set* in the local process. In addition, the multi-body topology keeps a pointer to the *Ensemble* implementation object so that it can directly reference the EPs in *loc_EP_Se*t. The EPs in *loc_EP_Set* can get its neighbor EPs by going through its neighbor cells and referencing the EPs in *loc_SC_Set.*

MultiBodyTopology Update

The multi-body topology is updated by monitoring the runtime information automatically, e.g., location of the EPs, number of EPs kept in each process, and so on. Typically, the root multi-body topology has to be updated during execution, since the spatial location of EPs can be updated by local computation. This leads to changes of EPs' communication pattern, and the multi-body topology updates the neighbor EP list accordingly. The multi-body topology is updated mainly under three circumstances, which are described as follows:

1. An EP moves within a sub domain from an inner cell to other local cells. As can be seen from Figure 6.21, the EP in cell *i0* might move to cell *b1* or cell *i2*, since local computation may update its spatial location. If the EP moves to an inner cell *i2*, it can get its neighbor EPs in boundary cells *b1*, *b*2, *b*3, and inner cells *i1*, *i*2, *i*3, *i*5, *i*6, *i*7. If the EP moves to an boundary cell *b1*, it can get its neighbor EPs in inner cells *i*0, *i*1, *i*2, boundary cells *b0*, *b*1, *b*2, and halo cells *h0*, *h*1, *h*2.

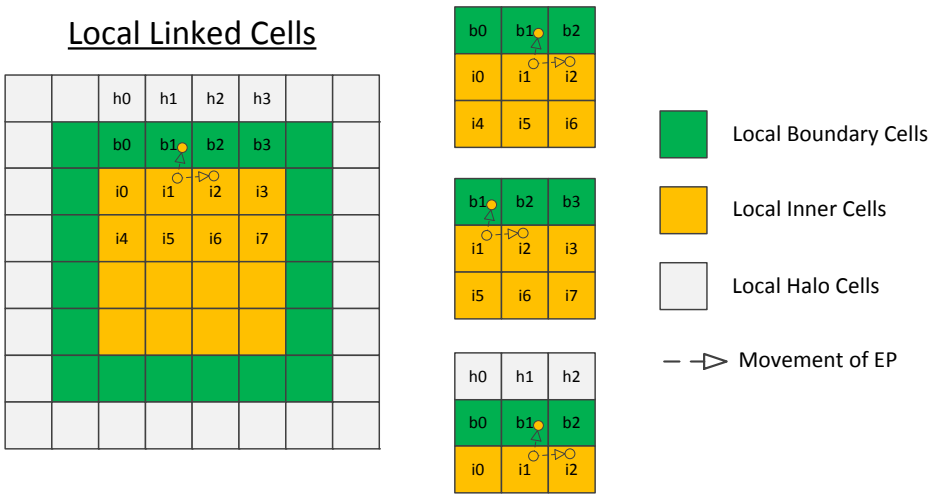


Figure 6.21.: Movement of an EP within a subdomain

2. An EP moves within a sub domain from a boundary cell to a halo cell. As it is shown in Figure 6.22, the EP in boundary cell *b1* moves to the local halo cell *h3*, which causes communication between process 0 and process 1. Process 0 needs to send it to process 1, which stores it in the local boundary cell *b1*. After the migration of the EP is done, it can get its neighbor EPs in inner cells *i*0, boundary cells *b0*, *b*1, *b*2, and halo cells *h0*, *h*1, *h*2, *h3*, *h4* in process 1.

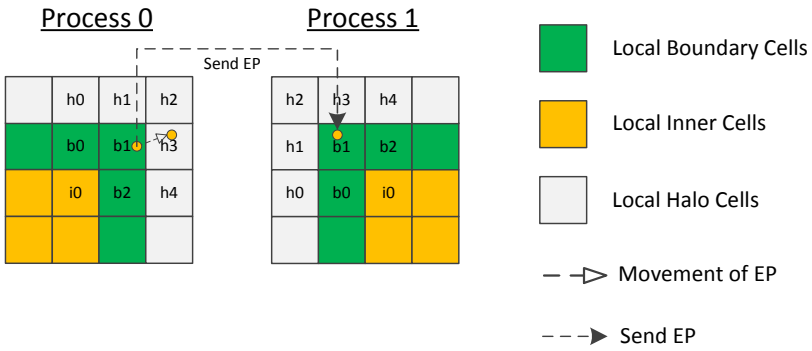


Figure 6.22.: Movement of an EP cross subdomains

3. In case of a significant load imbalance among processes, all the EPs in the ensemble needs to be redistributed, and the *MultiBodyTopology* implementation object on each process needs to be updated accordingly. The runtime system of the MPI-based library monitors the number of EPs that are kept in each process. If a process has much more EPs than the other processes, the EP redistribution step is triggered automatically and the root multi-body topology is updated. Both the global

redistribution among all the processes and partial adaption between neighbor processes are supported by the MPI-based library. The global redistribution triggers a collective communication operation, which redistributes all the EPs across different processes according to the output of METIS. The global redistribution can get an optimal EP distribution that both the load balancing and communication are already optimized by METIS. However, the global redistribution is expensive because all the processes have to involve in the communication. Apart from the global redistribution, the partial adaption merely triggers a communication between two neighbor processes and both processes exchange EPs and cell-based sub domains accordingly. The partial adaption can guarantee that both processes get an equal amount of EPs by cell-based partial adjustment, but the communication among all the processes is not yet optimized.

Creation of Irregular Grid Topology

Similar to the multi-body topology, each process keeps the root irregular grid topology, which maintains the neighbor EP list for the EPs in *loc_EP_Set* according to the id-based graph and EP distribution as well. Each process only keeps its local EPs and shadow copies of the EPs that are stored in remote processes. The memory organization of local EPs and their SCs in a single process is shown in Figure 6.23. The organization reuses the idea of the PARTI/CHAOS library[87]. The shadow copies of local EPs are allocated in the memory as an array, the SCs of remote EPs are stored after the local shadow copies with the indices from n to $n + m$ (the overall number of remote shadow copies is $m + 1$). The $m + 1$ remote copies are organized following the sequence of process ids from process i to process $i + j$ (j is the overall number of neighbor processes). Therefore, the accesses to the SCs of remote EPs are transformed to the accesses to the SCs stored in *loc_SC_Set*.

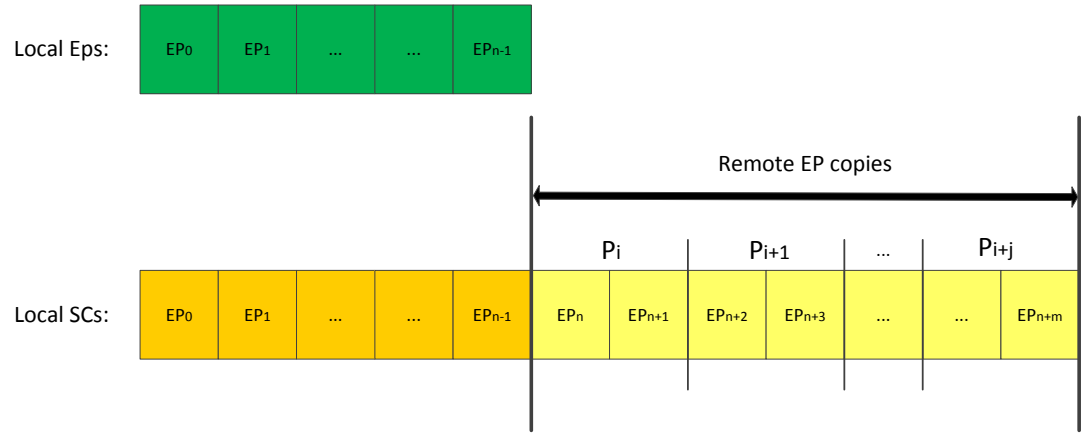


Figure 6.23.: Local EPs and SC organization

A simple example of a data distribution and the communication pattern among processes is shown in Figure 6.24. There are eight EPs that are distributed across four processes. Take process 2 as an example, it has to receive EPs from all the other processes and the SCs are organized as described above. Then, a global index to local index translation needs to be implemented in order to reference the neighbor EPs of the local EPs. Each process manages a translation table to handle this index transformation.

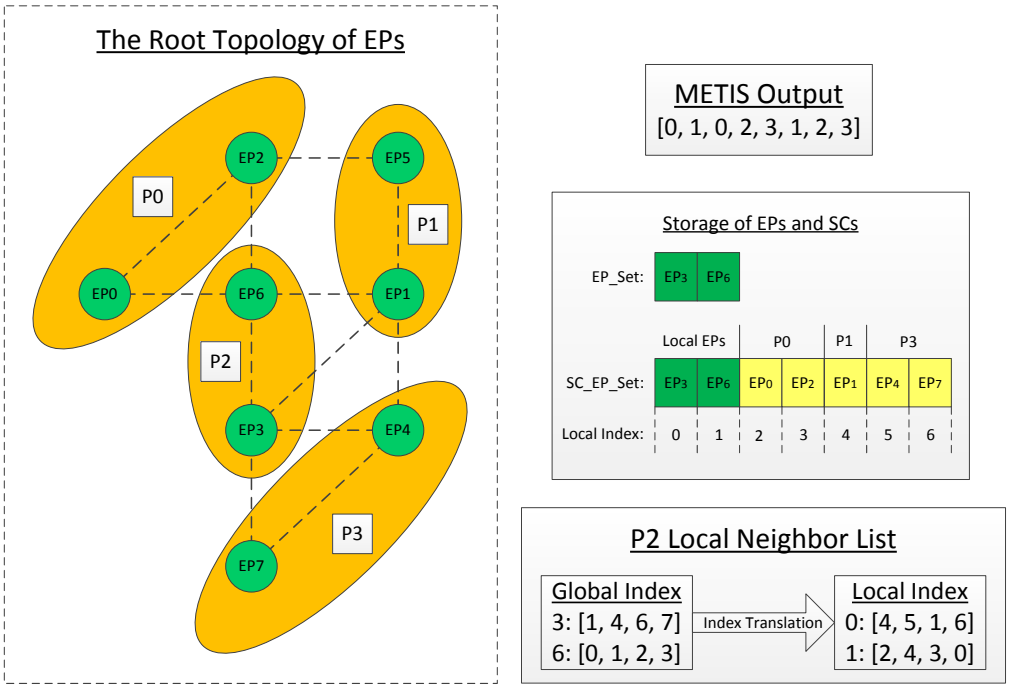


Figure 6.24.: EP distribution based on graph partitioning

IrrGridTopology Update

The irregular grid topology is updated by revising the structure of the id-based graph by calling the *Ensemble* operations, e.g., *addNode*, *addEdge*, and so on. All the changes of the id-based graph are accepted by the root irregular grid topology when an *update-Topology* operation is called. Then, a new neighbor EP list is created and will be reused in the next computational steps. Typically, although the creation of a new neighbor EP list is expensive, the newly created neighbor EP list can be reused for a long period of time until the next *updateTopology* operation is called. This scheme is efficient and managed by the MPI-based library automatically.

6.4.6. Communication Optimization

On distributed memory machines, the fine granular communication pattern among EPs has to be transformed to coarse granular communication among processes to avoid the too many small messages problem. It improves the communication efficiency significantly. Different communication optimizations are applied in the implementation of the MPI-based library. Details of the optimizations are shown below:

1. Aggregated send receive buffer management: Each process keeps an aggregated send and receive buffer for storing all the EPs that need to be sent to or received from its neighbor processes. The EPs in *loc_EP_Set* are copied into the aggreated send buffer, while EPs received from remote processes are stored in the aggregated receive buffer.

2. Communication reduction: A group of EPs usually require the same EP as one of their shadow copies for local computation. Based on this context, if the EP is located on a remote process, it has to be sent to the processes that keep the EPs. As it is shown in Figure 6.25, the EPs are distributed on process 0 and process 1 (EP_0, EP_1, and EP_3 are located on process 0, while EP_2, EP_4, EP_5, and EP_6 are located on process 1). Based on this EP distribution, both EP_1 and EP_3 require EP_4 as their shadow copies, and EP_4 has to be sent from process 1 to process 0. The communication reduction can guarantee that EP_4 is sent only once rather twice as the topology specifies. It is the process 0's responsibility to manage the access pattern to EP_4 by EP_1 and EP_3. This communication reduction strategy can reduce the communication volume significantly.

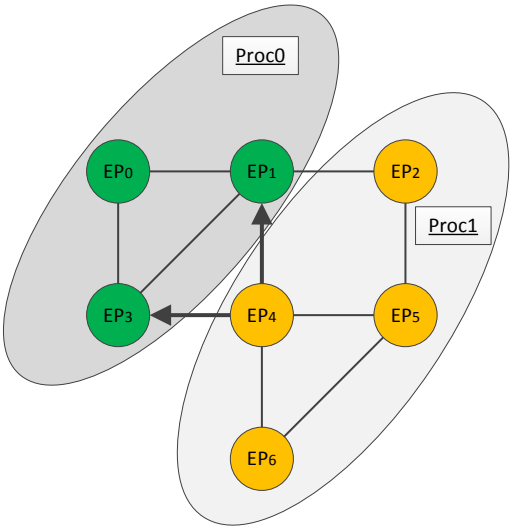


Figure 6.25.: Communication reduction

3. Communication coalescing: A process collects many EPs destined for the same process into a single message, which is stored in the aggregated send buffer. This optimization is called communication coalescing. The objective of communication coalescing is to reduce the number of message startups in order to avoid the "too many short messages" problem, since there is a substantial latency associated with message passing on many distributed memory machines.
4. Automatic adjustment of communication patterns according to the update of topologies: For multi-body and irregular grid applications, the communication pattern is usually irregular and adaptive. The changes of the communication pattern can be monitored by the processes when the topology is updated. The communication pattern can be adjusted accordingly based on the runtime information, i.e., spatial locations of EPs or changes of the id-based graphs, and so on.

6.5. Summary

The implementation framework implements the programming interface on different types of architectures. An ensemble-based program can be translated to different executables for target machines by linking to the machine-specific libraries. The major advantages of the framework are described as follows:

1. The programming interface can be implemented on different architectures based on a high-level programming approach. The source code of an ensemble-based program can run on the architectures without being revised.
2. On sequential and shared memory machines, the memory overhead is automatically optimized. The implementation framework preserves the semantic of the programming interface and achieves efficient memory utilization.
3. On distributed memory machine, communication among processes is handled by the MPI-based library, the programmers don't have to define low-level communication operations. The high-level *Ensemble* operations are implemented by a number of processes cooperatively.
4. Although the creation of the root topology is expensive, the implementation framework can efficiently reuse the high-level topology information for a long period of time until the next *updateTopology* is called.

7. Experimental Results

7.1. Overview

This chapter presents the experimental results of two irregular applications implemented by a manual program and an ensemble-based program. Both programs are tested on a sequential, shared memory, and distributed memory system on SuperMUC. The major objective of the experiments is to show that the implementation framework can be efficiently applied to scientific applications including irregular grid applications and MD simulations. We compare the performance of both programs in order to find out the major overheads originating from the implementation framework.

The rest of this chapter is organized as follows: Section 7.2 introduces the experimental platform including the hardware and software environment; Section 7.3 compares the performance and scalability of two implementations of an irregular grid application; Section 7.4 compares the implementations of an MD simulation on the experimental platform.

7.2. Experimental Platform

The experimental platform is SuperMUC[36], which is a new supercomputer at Leibniz-Rechenzentrum (Leibniz Supercomputing Centre) in Garching near Munich. With a peak performance of more than 3 petaflops, SuperMUC is one of the top 10 supercomputers in the Top500 list. It has more than 155.000 processor cores in 9400 compute nodes and 300 terabytes of memory.

The experiment platform uses the fat nodes on SuperMUC. A fat node is based on the Intel Westmere-EX processor[117]. It is a shared memory NUMA machine with four sockets, each of which has one Intel Xeon Processor E7-4870 processor and 64 GB of memory. The processor has 10 cores running at the frequency 2.4GHz with a peak performance of 9.6 GFlops. Each fat node has 40 processor cores and 256 GB of memory, and its peak performance is 384 GFlops. The operating system on the nodes is Suse Linux Enterprise Server, and the compiler currently used in the implementation framework is the Intel compiler with OpenMP support. The MPI-based library of the implementation framework uses IBM MPI.

7.3. Irregular Grid Applications

7.3.1. Overview

In order to test the performance and scalability of the implementation framework for irregular grid applications, we compare the execution time of a manual irregular gird implementation and an ensemble-based one. The manual program is implemented in C++, and parallelized with OpenMP and MPI on a shared memory node as well as a distributed memory system. The ensemble-based program is implemented by the sequential, OpenMP-based, and MPI-based library of the implementation framework on the same target systems.

The computational kernel of the programs is a simplified version of the FIRE benchmark[118], which is a general purpose computational fluid dynamics program package. It was developed specially for computing compressible and incompressible turbulent fluid flows as encountered in engineering environments. The benchmark consists of the solver of the resulting linear equation system. The computational domain is discretized on an irregular grid with a finite volume approach.

We use three different irregular grid data sets to evaluate the program. One of these data sets is called Cojack from the FIRE benchmark. In order to evaluate the scalability of the implementation framework, we build two larger data sets called Grid64 and Grid128. They reprensent 3D cubic grids with $262,144$ and $2,097,152$ points. They have regular geometrical shape, but the connectivity of points in the grids is specified by a neighbor list.

The computational kernel is an iterative kernel. The local values of a point at a time step are determined by the values of its neighbor points at its previous time step according to certain computational rules. The connectivity of points in the grids is described by a 2D array called a neighbor list. To simplify the experiments, each point only keeps a double floating point value as its local value, and the computational rule is a simple arithmetic mean operation. The local value of a point at time step $N+1$ is calculated by the arithmetic means of the local values of its neighbor points and its own local value at time step N. The maximum number of the iterations N is set to 128. The iterative operation is mathematically described as follows:

$$value^{N+1} = \frac{1}{numOfNeighbors+1}(\sum_{i=1}^{i=numOfNeighbors} Neighbor[i].value^{N} + value^{N})$$

After the arithmetic mean operation, there is a reduction operation on the local value of all the points. The reduction is a simple geometrical mean operation, and the result of this operation is returned to the master thread globally. The geometrical mean operation is mathematically described as follows:

$$reduceResult = \sqrt{\sum_{i=1}^{i=numOfPoints} Point[i].value^{2}}$$

The computation is a simlified version from FIRE code including a local computation and global reduction operation. In the evaluation, we only present the execution time of the communication and iterative kernel computation. The evaluation of other sections, e.g., I/O, initialization, and data distribution are ignored.

7.3.2. Data Sets

We choose two classes of irregular grids as test data sets. In such data sets, each point only requires data from its near neighbor points rather than randomly selected points in the grid.

Cojack Data

The Cojack data set represents a specific shape in the FIRE benchmark. It consists of $318,044$ points, each of which has 6 nearest neighbor points. The neighborhood information is kept in an input file.

3D Cubic Grids

In order to test the scalability of the implementation framework, we build Grid64 and Grid128, which are 3D cubic grids with the size of $64 \times 64 \times 64$ and $128 \times 128 \times 128$. The number of points in each grid is $262,144$ and $2,097,152$. Each point has 26 nearest neighbors based on the Moore neighborhood[57]. Although the data sets represent regular cubic grids, the accesses of neighbor points require one level of array indirection.

7.3.3. Sequential Comparison

In this subsection, we compare the execution time of the manual sequential program and the ensemble-based program implemented on a single CPU. Both programs are compiled by the Intel compiler with O3 optimization. Their execution time is shown in Figure 7.1. As can be seen from this figure, the ensemble-based program is around 10-30% slower than the manual sequential program on the three data sets. The overhead of the ensemble-based programming mainly originates from two factors. The first one is the branch in the implementation of the *getNghbList* operation defined in *Ensemble*. A topology pointer is a parameter of this operation, and its implementation has to decide whether this topology is the root topology or a sub topology. This branch takes some execution time, since this operation is called by all the EPs in the ensemble for a large number of times. The second factor is the memory overhead. In the implementation of the ensemble-based program, each point has to reference the ensemble and get its neighbor points from the root topology in the ensemble. It causes much more level 2

and level 3 cache accesses than the manual program. From the information of the CPU's hardware counters, the ensemble-based implementation causes more than twice as many cache misses as the manual program.

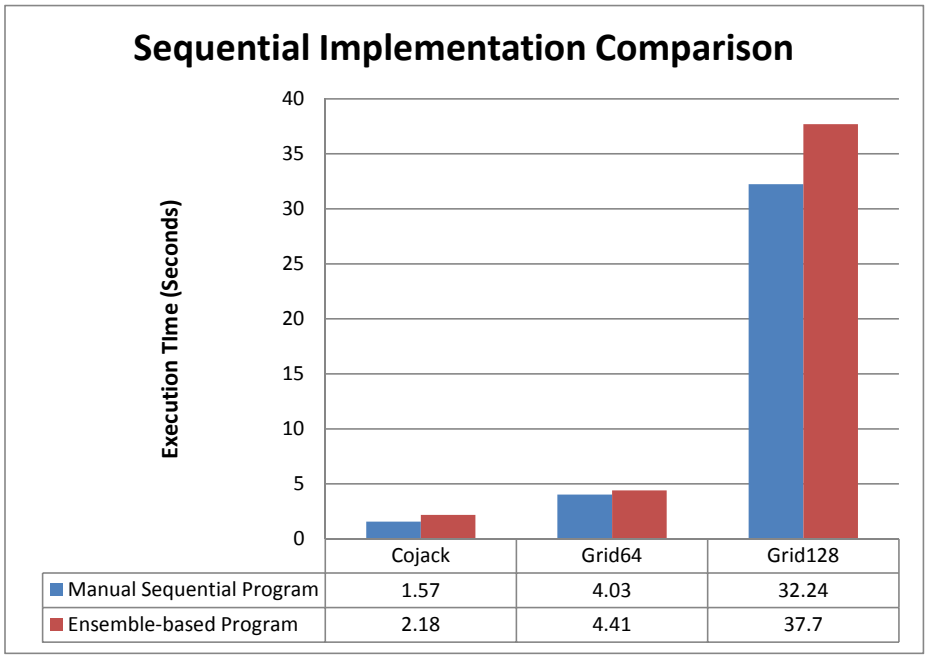


Figure 7.1.: Execution time of sequential programs

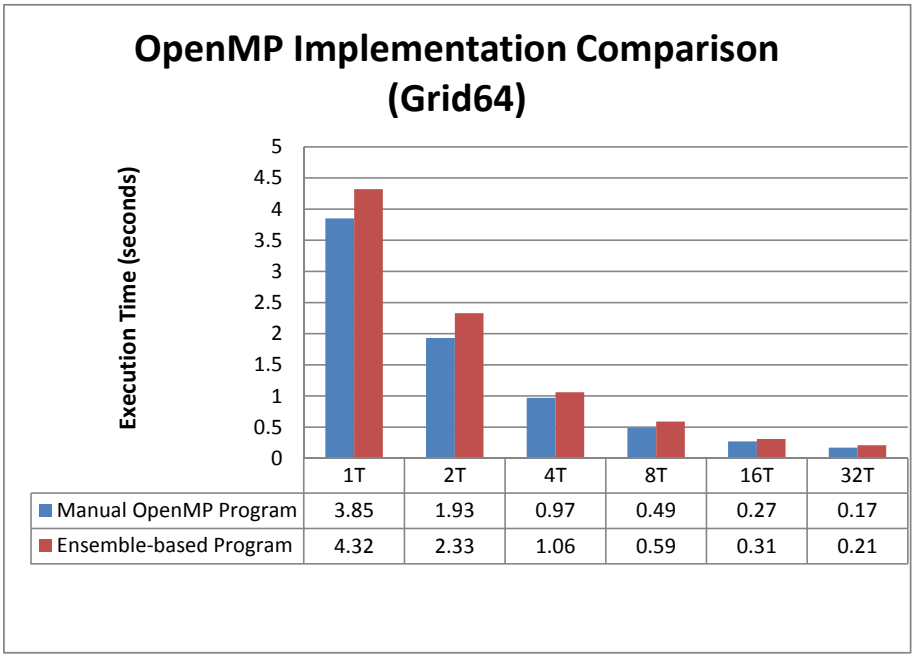
7.3.4. OpenMP Comparison

In this subsection, the execution time and speedup of the two programs are presented in order to test the scalability of the ensemble-based implementation on a shared memory system. The manual irregular grid program is parallelized by OpenMP, while the ensemble-based program is implemented by the OpenMP-based library. In order to simplify our discussion, only the data sets Grid64 and Grid128 are used. In this subsection, we only evaluate the execution time of the computation step excluding the reduction operation.

The execution time of both programs on Grid64 using 2, 4, 8, 16, 32 threads is shown Figure 7.2. This figure tells that both programs scale well up to 32 threads and the ensemble-based program has relatively constant executional overhead using different number of threads. Then, we increase the data size to Grid128 with more than 2 million points. The execution time of the two programs on Grid128 is shown in Figure 7.3. This figure tells that neither of the programs scale well when the number of threads is increased to 32. The main reason is that non-continuous memory accesses cause a large number of L2 and L3 cache misses especially when the number of threads running

on a single socket increases. The time for memory accesses rather than the arithmetic calculations dominates the overall execution time.

In order to test the scalability, we present the speedup curves on Grid64 and Grid128 using 2, 4, 8, 16, 32 threads in Figure 7.4. From this figure, we can see that the ensemble-based program scales well on Gird64 by using up to 32 threads. However, the scalability on Gird128 is not as good as expected. It means that without special optimization, it is difficult to get good speedup when irregular grids becomes larger and larger.



	1T	2T	4T	8T	16T	32T
Manual OpenMP Program	3.85	1.93	0.97	0.49	0.27	0.17
Ensemble-based Program	4.32	2.33	1.06	0.59	0.31	0.21

Figure 7.2.: Execution time of OpenMP programs (Grid64)

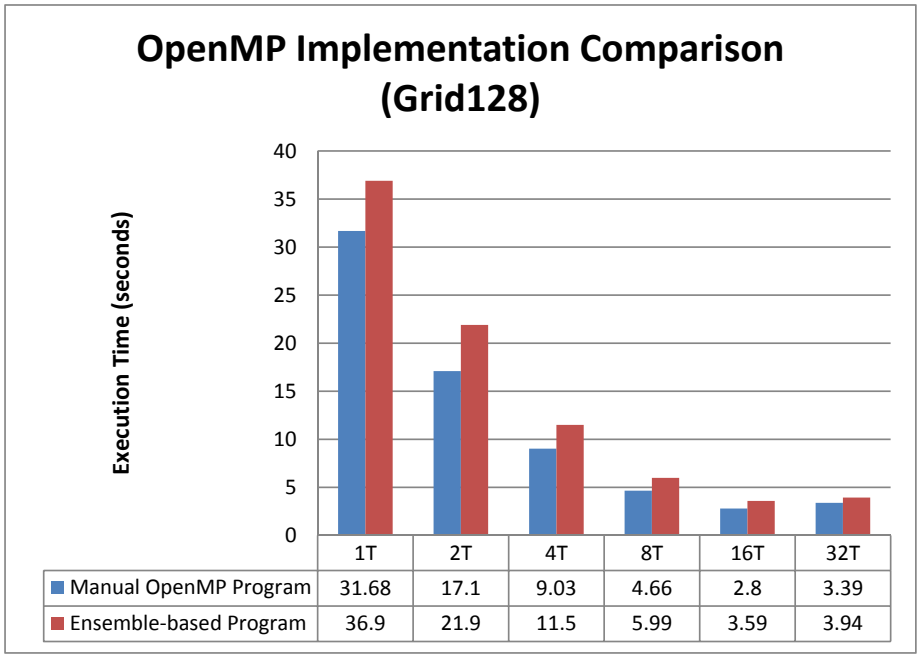


Figure 7.3.: Execution time of OpenMP programs (Grid128)

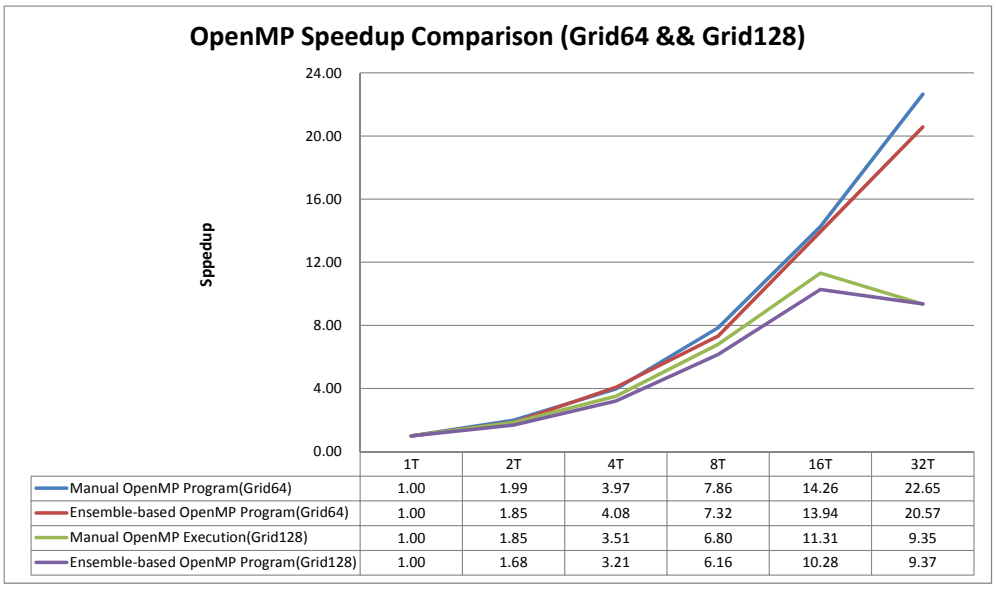


Figure 7.4.: OpenMP speedup curves on Grid64 and Grid128

Therefore, the OpenMP-based library of the implementation framework applies the re-indexing and data reallocation strategy to improve the performance on large irregular grids. It distributes all the EPs across different sockets of a shared memory

node with the METIS-based optimal EP distribution algorithm. In addition, threads are also evenly distributed across different sockets by setting the environment variable GOMP_CPU_AFFINITY. From the experimental result shown in Figure 7.5, we can see that the ensemble-based program implemented by the OpenMP-based library with the re-indexing and reallocation strategy scales well up to 32 threads and achieves much better performance than the ensemble-based implementation without re-indexing.

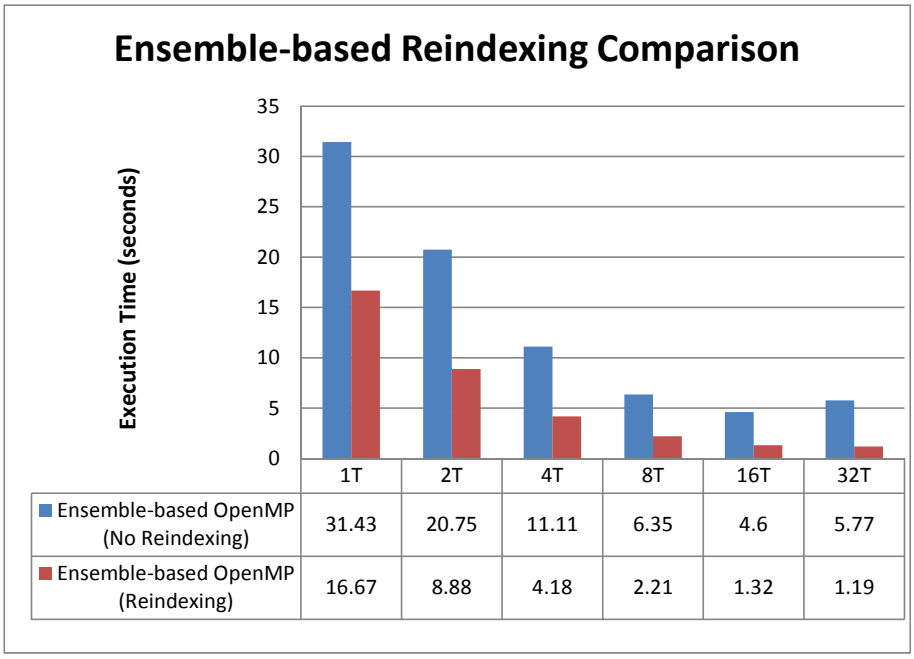


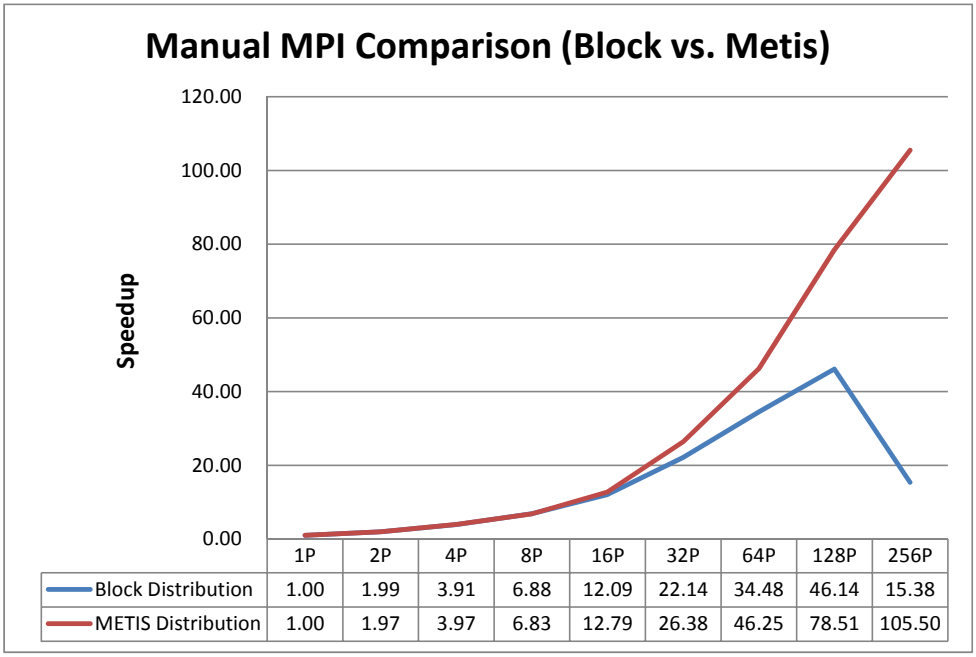
Figure 7.5.: Ensemble-based re-indexing

7.3.5. MPI Comparison

On a distributed memory system, the manual irregular grid program is parallelized in C++ with MPI, while the ensemble-based program is implemented by the MPI-based library of the framework. The execution time of these two programs is evaluated in order to test the performance and scalability of the MPI-based library on distributed memory systems.

First of all, we compare the speedup of two manual MPI programs based on the block and METIS decomposition method. The block decomposition is to decompose the point set into multiple subsets according to the indices of the points. For example, points $[0, 1, 2, ..., P-1]$ are assigned to process 0, points $[P, P+1, P+2, ..., 2P-1]$ are assigned to process 1, and so on and so forth ($P = N/numOfProcesses$). The METIS decomposition is to decompose the point set according to graph-based algorithms with optimized communication among different subsets. The speedup curves of the two MPI programs

is shown in Figure 7.6. This figure tells that the METIS decomposition leads to better scalability and performance especially when the number of processes is more than 64. The major reason is that the METIS decomposition can obtain efficient communication among processes especially for irregular grids. Thus, it is worth integrating METIS into the MPI-based library of the framework. The METIS library is efficient and doesn't take much time on decomposing millions of points.



	1P	2P	4P	8P	16P	32P	64P	128P	256P
Block Distribution	1.00	1.99	3.91	6.88	12.09	22.14	34.48	46.14	15.38
METIS Distribution	1.00	1.97	3.97	6.83	12.79	26.38	46.25	78.51	105.50

Figure 7.6.: MPI speedup curves (Block vs. METIS)

Then, we compare the execution time of the manual MPI program based on METIS decomposition and the ensemble-based program using different numbers of processes. The execution time of the two programs using one process is shown in Figure 7.7. This figure shows that the execution of the MPI-based library is around 30% slower than the sequential library, since there are some branch statements added into the MPI-based library. For example, the implementation of the *getNghbList* operation needs to guarantee that the pointer of an EP provided to the operation is a local EP, which increases the overhead but detects possible errors automatically.

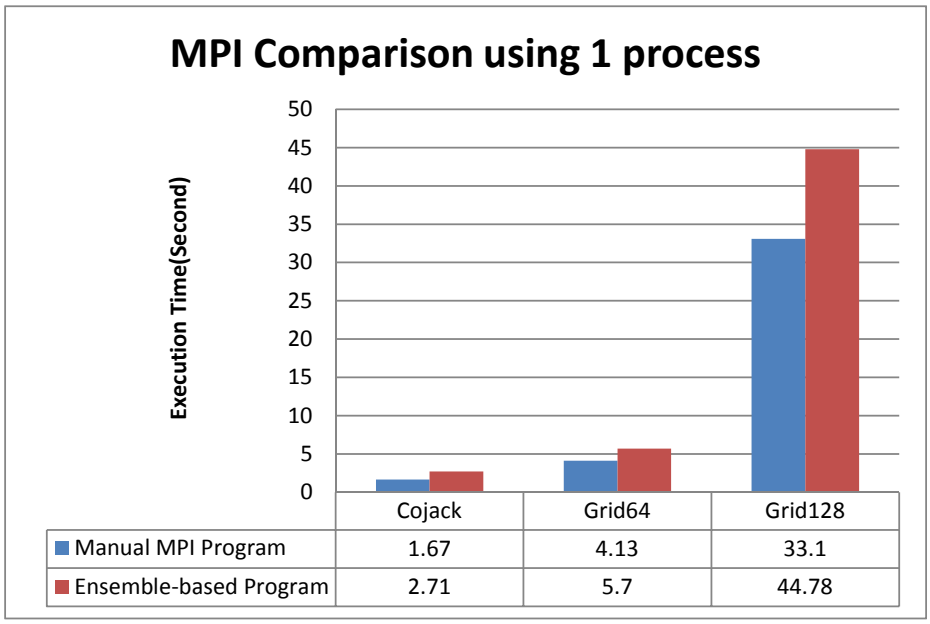


Figure 7.7.: MPI comparison using 1 process

The comparison between the manual MPI program using METIS and the ensemble-based implementation is shown in Figure 7.8. This figure tells that with 20-30% of overhead compared to the manual MPI implementation, the MPI-based library can get good performance while the number of processes increases. Based on the execution time of both programs using one process, the speedup curves of these two programs are shown in Figure 7.9. From this figure, we can see that the ensemble-based program scales well up to 256 processes. Compared to the MPI program based on METIS, the ensemble based program can obtain comparative performance and its overhead is stable while using different numbers of processes.

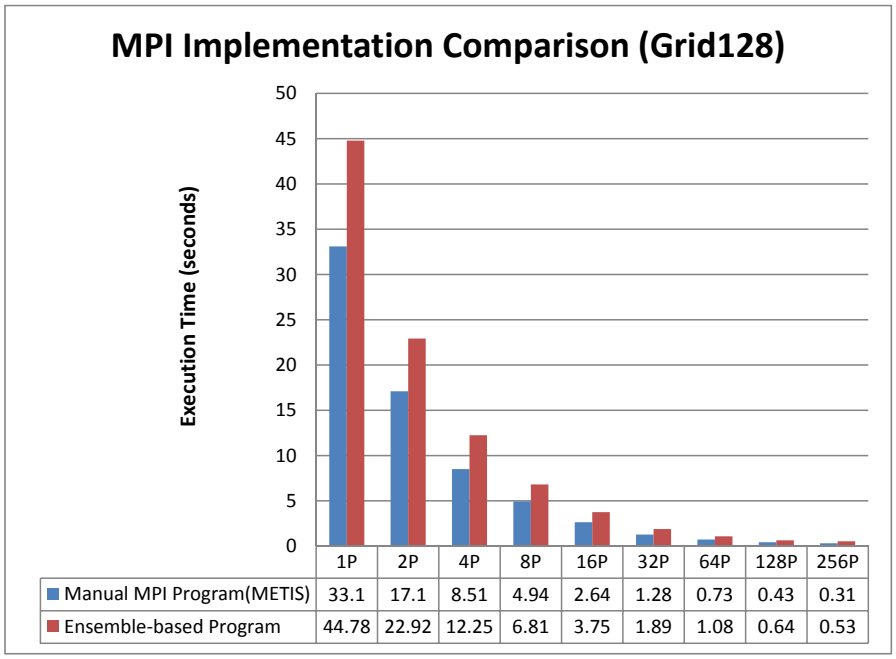


Figure 7.8.: Execution time comparison with MPI

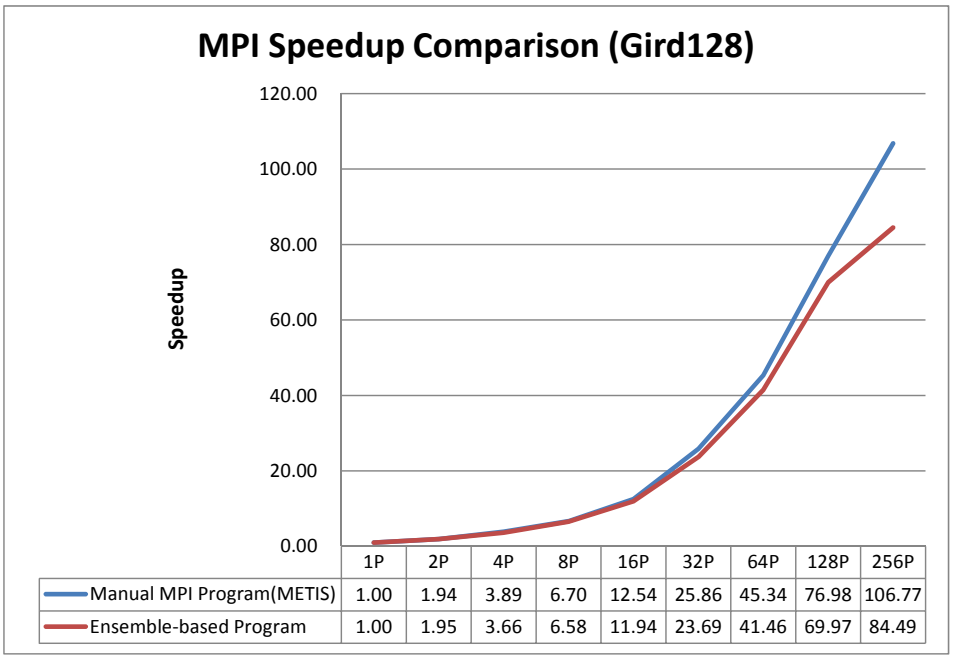


Figure 7.9.: Speedup curve comparison with MPI

7.3.6. Summary for Irregular Grid Applications

In this section, we compared the execution time of the manual C++ program and ensemble-based program for an irregular grid application on different data sets. Both programs are implemented on a sequential, shared memory, and distributed memory system. The experimental results show that with reasonable execution overhead, the ensemble-based implementation framework achieves acceptable performance on the irregular grid application. Meanwhile, it is easier and straightforward to implement the irregular application using the ensemble-based programming, which reduces so much programming overhead. For example, the MPI program requires nearly $1,000$ lines of code, while the ensemble-based program only has less than 200 lines including high level operations like *updateShadowCopy*, *parallel* operation, and so on.

7.4. Molecular Dynamics Simulation

7.4.1. Overview

In order to test the implementation framework for multi-body applications with a truncated approximation, we compare the performance of the manual MD program and the ensemble-based program on a sequential, shared memory, and distributed memory system. The computational kernel of the MD programs is based on the truncated LJ potential formula, and the simulation domain is a 3D cubic domain. Each molecule in the domain keeps a randomly generated position and interacts with its neighbor molecules located within the cut-off radius region. The positions of all the molecules are updated according to the molecule-to-molecule interactions and the equations of motion.

In the experiments, the number of the molecules is set to $64K$ $(65,536)$, and $128K$ $(131,072)$, the number of iteration steps is 8. The cut-off radius is set to 1, the size of the simulation domain is $8 \times 8 \times 8$. It is extremely time-consuming for such MD simulations if the number of molecules is $64K$ to $128K$, thus we set a small number to the number of iterations in order to evaluate the programs in a reasonable period of time. We only present the execution time of the communication and iterative kernel computation. The evaluation of other sections, e.g., I/O, initialization, and data distribution are ignored.

7.4.2. Sequential Comparison

In this subsection, we evaluate the performance of an MD program based on the linked cell algorithm and the ensemble-based program on a sequential machine. The execution time of the two programs is shown in Figure 7.10. This figure tells that the ensemble-based program is around 15% slower than the basic sequential program. The major overhead is the creation of the neighbor list in the root topology. Different from irregular

grid applications, the creation of the neighbor list is triggered by calling *updateTopology* operations in each time step because of the non-predictive movement of the molecules in the simulation domain. This operation is relatively expensive with memory overhead and more execution time as well. However, once the neighbor list is created, each molecule is able to access its neighbor molecules directly and efficiently.

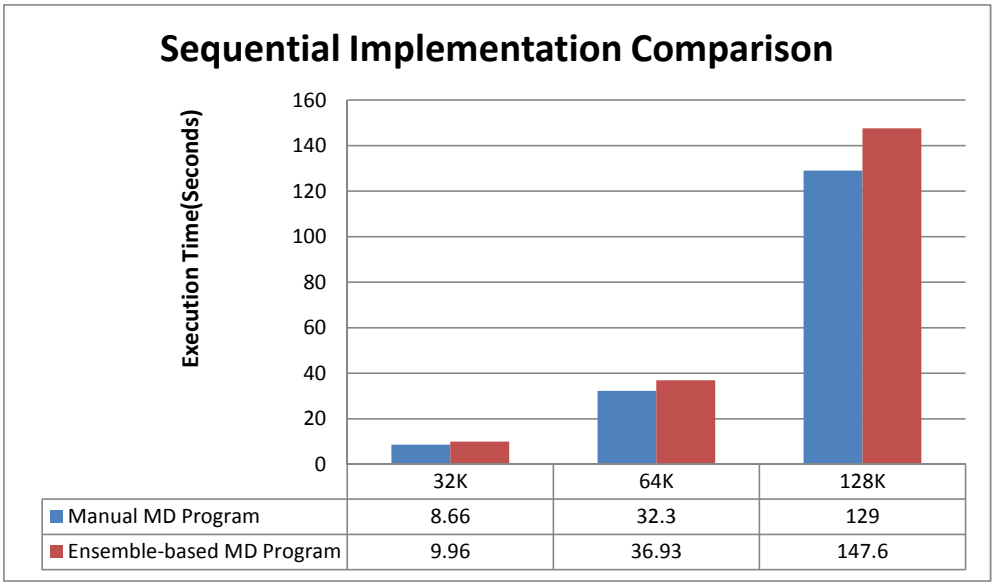


Figure 7.10.: Execution time of sequential MD programs

7.4.3. OpenMP Comparison

In this subsection, we evaluate the performance of the basic OpenMP program and the ensemble-based program on a single node on SuperMUC. The execution time of these two programs is shown in Figure 7.11, and the speedup curves are shown in Figure 7.12.

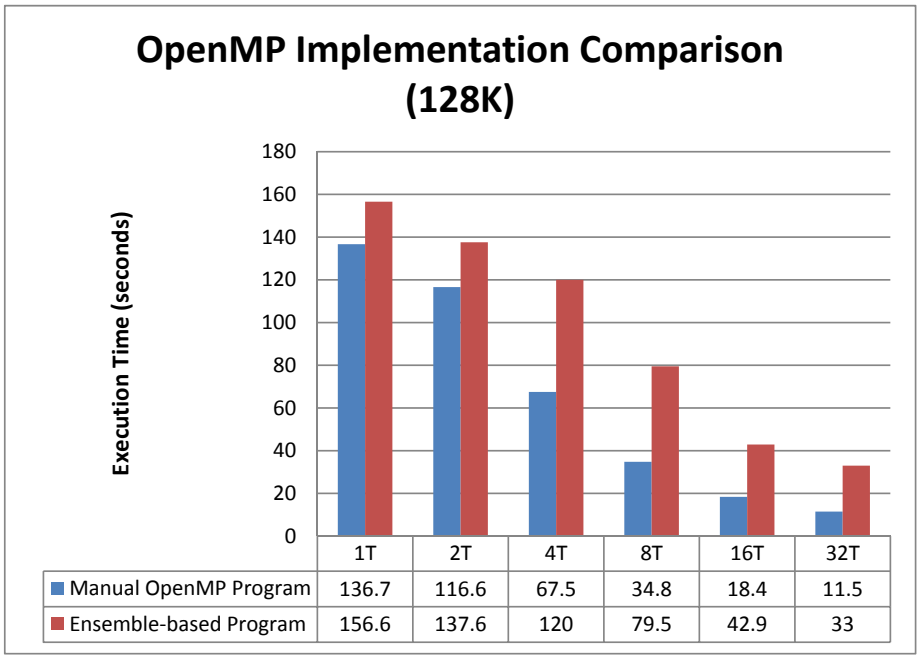


Figure 7.11.: Execution time of OpenMP MD programs

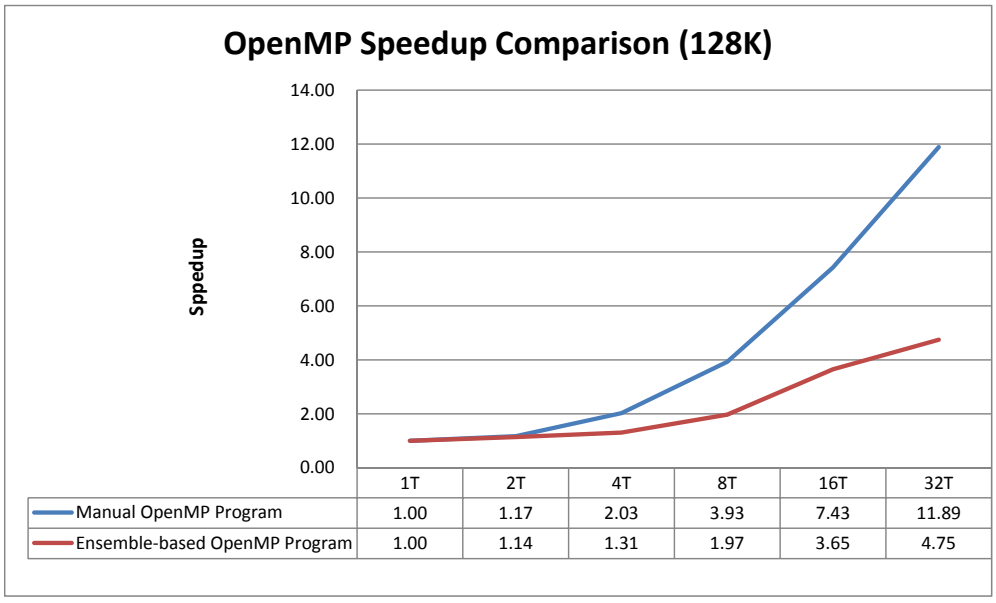


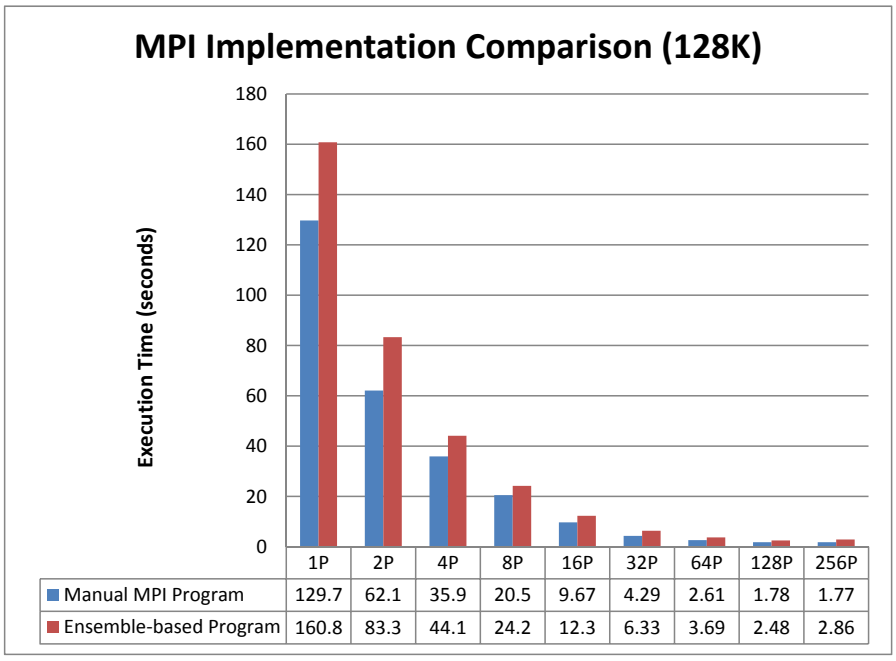
Figure 7.12.: Speedup curves of OpenMP MD Programs

From the results shown in these figures, we can see that the overhead of the ensemble-based program becomes higher and higher when the number of threads increases. The overhead mainly originates from two aspects. The first one is the creation of the neighbor

list is not as efficient as expected because of its vector-based data structure. The second one is the parallel operation that doesn't scale very well because of memory bandwidth of the nodes on SuperMUC. Different from irregular grid applications, each molecule in an MD simulation usually has hundreds of neighbor molecules, which greatly increases the memory overhead. The discussion of thread distribution was already presented in the experimental results of irregular grid applications, so we don't repeat it in the subsection.

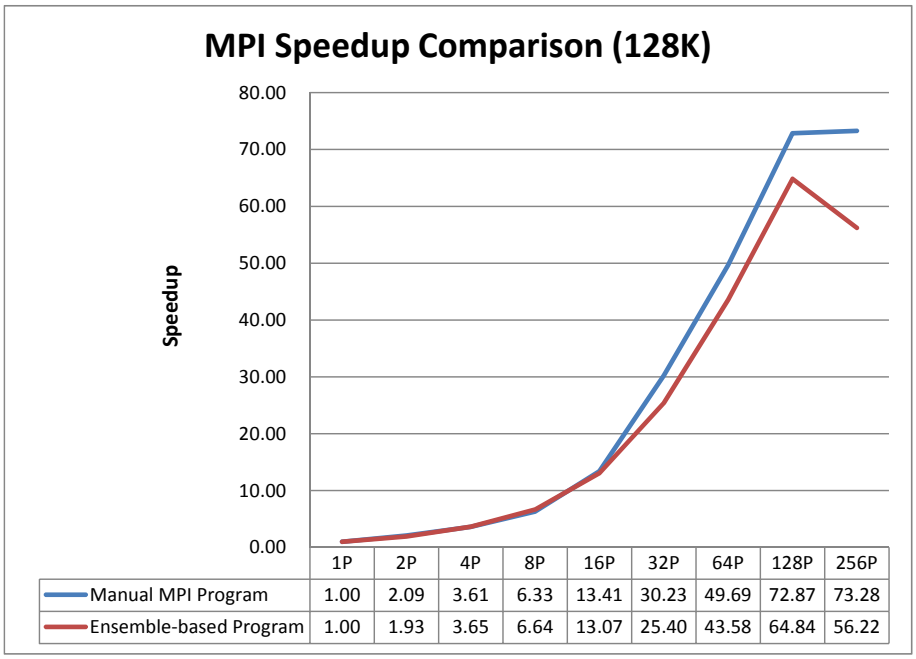
7.4.4. MPI Comparison

On a distributed memory system, the manual MD program is written in C++ with MPI, while the ensemble-based program is implemented by the MPI-based library of the implementation framework. In order to balance the computational load, the manual program uses METIS to decompose the molecules according to the linked cells. The MPI-based library integrates METIS automatically. We compare the execution time of both programs in order to evaluate the performance and scalability of the MPI-based library on distributed memory systems. The data set used is 128K. The execution time of both programs is shown in Figure 7.13, and the speedup curves are shown Figure 7.14.



	1P	2P	4P	8P	16P	32P	64P	128P	256P
Manual MPI Program	129.7	62.1	35.9	20.5	9.67	4.29	2.61	1.78	1.77
Ensemble-based Program	160.8	83.3	44.1	24.2	12.3	6.33	3.69	2.48	2.86

Figure 7.13.: Execution time of MPI MD programs



	1P	2P	4P	8P	16P	32P	64P	128P	256P
Manual MPI Program	1.00	2.09	3.61	6.33	13.41	30.23	49.69	72.87	73.28
Ensemble-based Program	1.00	1.93	3.65	6.64	13.07	25.40	43.58	64.84	56.22

Figure 7.14.: Speedup curves of MPI MD programs

From the experimental results shown in these figures, the ensemble-based program scales as well as the manual MPI program. The execution time of one process is a bit larger than the sequential implementation because of the branches used to decide local and non-local EPs. This overhead keeps constant when the number of processes increases.

The major advantage of the ensemble-based programming is the productivity. It is really difficult to implement an MD simulation using MPI. For exmple, the manual MPI program has almost 3,000 lines of code, which takes care of all the implementation details including, the data decomposition, communication among processes, migrating molecules between processes, and so on. However, the ensemble-based program only has a few hundred of lines. It mainly consists of the definition of the molecule including fine granular local attributes and operations. The main function contains three high-level operations namely *update*, *parallel*, and *updateTopology*. The experimental results show that the ensemble-based program is only a bit slower than the MPI program. It means the ensemble-based programming paradigm is relatively efficient and easy to use.

7.5. Summary

The ensemble-based implementation framework can be applied to implement irregular grid applications and MD simulations. The experimental results show that with reasonable and acceptable overhead, it reduces a large number of lines of code and improves

the programming productivity especially for these irregular applications. The implementation framework manages the root topology with a neighbor list of all the EPs based on C++ vectors and pointers. It implements the high-level operations efficiently on a sequential, shared memory, and distributed memory system on SuperMUC.

8. Conclusion and Future Work

8.1. Ensemble-based Programming Approach

The ensemble-based programming approach is applied to implement parallel scientific applications in a fine granular and SPMD fashion. Its major objective is to improve the programming productivity and make parallel programming easier and more straightforward. Different from current programming approaches starting from the global data structure, the ensemble-based approach is based on specifying the finest granular elements and their interactions. These fine granular elements are organized as an ensemble, which manages the elements, their topologies, and high-level operations. By using the high-level operations explicitly, programmers can control the actions of the elements including the communication, synchronization, parallel operations, and so on.

The ensemble-based programming approach tries to combine the major advantages of current programming models and runtime strategies. Similar to HPF and GA, the ensemble-based programming approach provides a shared memory programming view, and can be implemented on distributed memory platforms. It applies the template-based OOP approach like TBB and Charm++ in order for the flexibility and extensibility. The support for irregular grid applications reuses the idea of the PARTI/CHAOS library and manages the data storage and communication among processes efficiently on distributed memory systems.

8.2. Programming Scheme

8.2.1. Machine Model

The ensemble-based machine model is different from a concrete architecture with CPUs, main memory, and I/O devices. It is a simple and fine-granular platform, which hides optimizations such as cache optimizations, data locality, communication overlapping, and so on. The ensemble-based machine model is an abstract architecture consisting of a control processor (CP) and a large number of fine granular processors (FGPs). The execution model of the machine is similar to SIMD. The CP issues a “single instruction”, which represents a specific parallel operation on FGPs. Multiple FGPs then perform user-defined coarse-grained computations in parallel.

8.2.2. Programming Paradigm

The programming paradigm is designed on top of the machine model in order to program such an abstract architecture. It consists of software entities and their relations. The software entities are Elementary Points (EPs), the ensemble, topologies, and the master thread. An EP represents the finest granular computational object in the domain of an application. Each EP is mapped to a single FGP in the machine model. Multiple EPs are formed into an ensemble with their relations kept in a topology. A topology defines communication patterns resulting from the need for the information of a set of EPs in the ensemble. The EPs can exchange their local attibutes based on certain topologies. The ensemble is a software container that stores a set of EPs and manages their information, communication, computation, and so on. The main constituents in the ensemble are a set of EPs, topologies, high-level operations for managing EPs and topologies, parallel operations, shadow copy update, and collective operations. The master thread is executed on the CP in a fork-join fashion. It is responsible for globally managing the ensemble-based program. The major tasks of the master thread include performing I/O, creating the ensemble and topologies, creating and initializing EPs, managing topologies in the ensemble, local computations, and triggering parallel operations, and so on.

8.2.3. Programming Interface

The programming interface is based on the object oriented programming approach in C++. It supports the mapping from the software entities, i.e., EPs, the ensemble, and topologies, to concrete C++ objects. It consists of a template hierarchy with predefined class templates. The top level of the template hierarchy includes three base templates namely *ElementaryPoint*, *Ensemble*, and *Topology*. These templates have derived templates called application-level templates, which target to three application areas, i.e., multi-body, irregular grid, and regular grid applications. The user-defined software entities with local attributes and operations can be defined as C++ classes derived from the application-level templates. In addition, the template hierarchy is flexible and can be extended to support other application areas by adding new application-specific templates into the template hierarchy. The master thread is expressed in the main function of an ensemble-based program using C++ syntax. It controls the behavior of the EPs in the ensemble by calling high-level operations defined in *Ensemble*.

Based on the programming interface, the ensemble-based programming primarily includes defining the EPs, specifying the EPs' topologies, inserting the EPs and the topologies into the ensemble, creating the master thread that manages the computation and communication of the EPs using high-level operations, and so on.

8.3. Implementation Framework

The basic idea of implementation framework is to aggregate fine granular EPs into appropriate blocks based on the description of the hardware. The blocks are then bound to threads or processes. The EP-to-EP communication is coarsened to the communication among blocks. For different types of architectures, the implementation framework consists of machine-specific libraries namely a sequential, an OpenMP-based, and an MPI-based library. They are implemented on sequential, shared memory, and distributed memory systems respectively. The implementation framework currently supports both multi-body applications and irregular grid applications. A platform-independent and ensemble-based program can be translated into different executables by linking these libraries with different compiler options. The implementation framework currently supports multi-body and irregular grid applications. Regular grid applications can usually be implemented and optimized by static compiler extensions. Thus, the support for regular grid applications is not implemented in the framework.

The sequential-based library is a standard OOP implementation. Both the communication and parallel operations of the EPs in the ensemble are handled by a single-threaded process. It is designed to demonstrate the basic implementation of the programming interface on a single process.

The OpenMP-based library implements the programming interface on top of OpenMP. It translates an ensemble-based program to an OpenMP program executed on shared memory systems. The communication and parallel operations of EPs in the ensemble are done by multiple threads in parallel. The computation of a group of EPs is aggregated and bound to a single thread and the communication among EPs is handled by accessing the shared memory. In order to minimize non-local accesses on NUMA nodes, the OpenMP-based library employs EP reallocation strategy combined with re-indexing method for irregular grid applications.

The MPI-based library implements the programming interface in C++ with MPI. It employs optimized EP distribution strategies based on the METIS library to distribute computational workload across multiple processes. The communication among processes is optimized by aggregating fine granular communication among EPs into coarser MPI messages. The communication pattern among EPs is determined by topologies, which are managed by the MPI-based library automatically. For multi-body applications, the MPI-based library employs both the domain decomposition and the parallel linked cells (PLC) algorithm for EP distribution and communication management, while for irregular grid applications; it applies graph partitioning algorithms to achieve optimal EP distribution and communication efficiency.

8.4. Evaluation

The ensemble-based implementation framework is ported on SuperMUC[36], the peta-scale supercomputer at LRZ (Leibniz Supercomputing Centre) in Germany. We implement an irregular grid and an MD program based on different data sets in order to test the performance and scalability of framework on different types of systems including a sequential, shared memory, and distributed memory system. The experimental results show that the execution of then ensemble-based programs is a bit slower than the standard implementations using C++ with OpenMP or MPI. On a sequential machine, the execution overhead of the framework is around 10%-30% for both applications. On a distributed memory machine, the re-indexing and reallocation strategy integrated in the OpenMP-based library achieves good performance and speedup. On a distributed memory machine, the executional overhead is around 30% using different number of processes.

Therefore, with acceptable and reasonable overhead, the ensemble-based programming improves the programming productivity in terms of the source code size, the coding method, and the implementation difficulty. Take the MD simulation as an example, the standard MPI program has almost 3,000 lines of code, which takes care of all the complicated implementation details. However, the ensemble-based program only has a few hundred of lines with the definition of molecules and the main function that is responsible for triggering high-level ensemble operations defined in the template hierarchy. Compared to current parallel programming models, the ensemble-based programming scheme manages the granularity of computation, data distribution, communication and load balance for irregular applications efficiently.

8.5. Future Work

The ensemble-based programming scheme is only a prototype that supports two application areas with a single ensemble and multiple topologies. The topologies are organized in a two-level hierarchy, which consists of only a single root topology and its sub topologies. In the future work, it is possible to extend the specification to support and multiple ensembles with a more complicated topology hierarchy. The programming interface needs to provide specifications to support defining relations between ensembles.

In this dissertation, we present basic implementations of the programs without detailed optimizations to different types of architectures. In order to improve the efficiency of the implementation framework, application oriented and architecture oriented optimization strategies can be integrated into the framework in the future. For example, efficient algorithms for building neighbor list can be implemented in the framework to get better performance for MD simulations.

The template hierarchy is currently designed for multi-body and irregular grid applications. The support for other application areas can be implemented by adding application-specific templates into the template hierarchy. One possible extension of the support for adaptive grid applications. The ensemble needs to support dynamic insertion and deletion of EPs at runtime. The topologies in the ensemble need to be updated by appropriate algorithms when the interaction of EPs changes.

The implementation framework is currently ported on three types of architectures including sequential systems, and parallel system like shared memory and distributed memory systems. Porting the implementation framework on heterogenous architectures with CPUs and GPUs is an interesting and challenging topic. In addition, the support for hybrid architectures can also be exploited in the future in order to take advantages of hybrid programming with MPI and OpenMP.

A. Appendix

A.1. Compiler Commands and Options

On sequential machines, the implementation framework applies a standard C++ compiler and the sequential library to translate the ensemble-based program into an executable. On shared memory machines, it translates the program into an executable by a standard C++ compiler with OpenMP. On distributed memory machines, the program is compiled to a multi-process executable with MPI. Table A.1 presents the compiler commands and required options to generate executables for different target machines.

	Compiler Command	Required Options
Sequential	g++ / icpc	——
Shared Memory	g++ / icpc	-fopenmp
Distributed Memory	mpicxx / mpiCC	——

Table A.1.: Basic compiler commands and required options

Besides to the basic compiler commands and options, users need to specify the location of header files when compiling the source files of the master thread and fine granular entities. In addition, it is necessary to specify the location of a machine-specific library and -l option when linking all the object files to an executable. The additional compiler options are presented in Table A.2.

	Header Location	Library Location	Linking
Sequential	-I ../Seq/include	-L ../Seq/lib	-lseq_e
Shared	-I ../OpenMP/include	-L ../OpenMP/lib	-lopenmp_e
Distributed	-I ../MPI/include	-L ../MPI/lib	-lmpi_e

Table A.2.: Machine-specific compiler options

A.2. Full Declaration of Template Hierarchy

A.2.1. *ElementaryPoint*

Listing A.1: Full declaration of ElementaryPoint

```
#ifndef ELEMENTARYPOINT_H_
#define ELEMENTARYPOINT_H_

#include <iostream>

using namespace std;

template<class Ensemble>
class ElementaryPoint {

protected:
    //Identifier
    unsigned long id;

    //Ensemble pointer
    Ensemble*ensemblePointer;

public:
    //Default constructor
    ElementaryPoint();

    //Constructor with id
    ElementaryPoint(unsigned long id);

    //Constructor with id, ensemble pointer
    ElementaryPoint(unsigned long id, Ensemble*ensemblePointer);

    //Constructor with Ensemble pointer
    ElementaryPoint(Ensemble*ensemblePointer);

    //Default destructor
    virtual ~ElementaryPoint();

    //Get the id of ElementaryPoint
    inline unsigned long getId();

    //Set the id of ElementaryPoint
    inline void setId(unsigned long id);

    //Get the Ensemble pointer of ElementaryPoint
    inline Ensemble*getEnsemble();

    //Set the Ensemble pointer of ElementaryPoint
    inline void setEnsemble(Ensemble*ensemblePointer);
};
```

```
#endif /* ELEMENTARYPOINT_H_ */
```

A.2.2. *Topology*

Listing A.2: Full declaration of Topology

```
#ifndef TOPOLOGY_H_
#define TOPOLOGY_H_

#include <iostream>
#include <vector>

using namespace std;

template<class EP>
class Topology {
protected:
    //id of the topology
    unsigned short id;

    //Number of EPs
    unsigned long numOfEPs;

    //The set of EPs' identifiers
    vector<unsigned long> idSetOfEP;

    //A tag shows whether it is a sub topology
    bool isRootTop;

    //If a topology is a sub topology, a pointer to its root topology
    Topology<EP>* baseTop;

public:
    //Default constructor
    Topology();

    //Default destructor
    virtual ~Topology();

    //Get the pointer of the base topology
    Topology<EP> *getBaseTop() const;

    //Get the id
    unsigned short getId() const;
```

```
    //Get the id set of EPs
    vector<unsigned long> getIdSetOfEP() const;

    //Get isRootTopology
    bool getIsRootTop() const;

    //Get number of EPs
    unsigned long getNumOfEPs() const;

    //Set the pointer of the base topology
    void setBaseTop(Topology<EP> *baseTop);

    //Set number of EPs
    void setNumOfEPs(unsigned long numOfEPs);

    //update operation
    virtual void updateTopology();
};
```

A.2.3. *Ensemble*

Listing A.3: Full declaration of Ensemble

```
#ifndef ENSEMBLE_H_
#define ENSEMBLE_H_

#include <iostream>
#include <vector>

using namespace std;

template<class EP, class Topology>
class Ensemble {
public:
    /************************************************
     *  Constructors and Destructor
     ************************************************/
    //Constructor
    Ensemble();

    //Constructor with command arguments
    Ensemble(int *argc, char ***argv);

    //Destructor
    virtual ~Ensemble();
```

```
    //Number of EPs
    unsigned long numOfEPs;

    //Number of Topologies
    unsigned short numOfTops;

    //Local set of EPs
    vector<EP> EP_Set;

    //Topology Set
    vector<Topology*> topPtrSet;

    /************************************************
     *   Finalization
     ************************************************/
    //Finalization
    virtual void Finalize();

    /************************************************
     *  Insert an EP into the Ensemble
     *  Remove an EP from the Ensemble
     ************************************************/
    //Insert an Elementary Point into the Ensemble
    virtual void insertEP(EP*ep);

    //Remove an EP from the Ensemble
    virtual void removeEP(EP*ep);

    //Remove an EP from the Ensemble
    virtual void removeEP(unsigned long id);

    /************************************************
     *  Insert a Topology into the Ensemble
     *  Remove a Topology from the Ensemble
     ************************************************/
    //Insert a Topology into the Ensemble
    virtual void insertTopology(Topology*topology);

    //Remove a Topology from the Ensemble
    virtual void removeTopology(Topology*topology);

    //Remove a Topology from the Ensemble
    virtual void removeTopology(unsigned short id);

    /************************************************
     *  Parallel Execution
```

```
 ************************************************/
//Parallel execution of all elementary points in the ensemble
template<typename Operation>
void parallel(Operation op);

//Parallel execution of all elementary points in the ensemble
template<typename Operation>
void parallel(Operation op, Topology*topology);

/************************************************
 *    Get NeighborList Operations
 ************************************************/
//Get neighborList of the current Elementary Point
virtual vector<EP*>& getNghbPtrs(EP*current);

//Get neighborList of the current Elementary Point
virtual vector<EP*>& getNghbPtrs(EP*current, Topology*topology);

/************************************************
 *    Updating Shadow Copies
 ************************************************/
//Update shadow copies using a given Topology
virtual void update(Topology*topology);

//Update the shadow copies of specific properties of all
   elementary points in the Ensemble
template<typename GetOperation>
void update(GetOperation getOp, Topology*topology);
/************************************************
 *    AllReduction Operations on all the
 *    Elementary Points in the Ensemble
 ************************************************/
/* Collective operation on a specific property using predefined
   operations and
 * put the result to the the same property place
 * ------- reduceOpt (A.property) = A.property ----------
 */
template<typename GetOperation>
void allReduceOp(GetOperation op, int reduceOpType);

/* Collective operation on a specific property using user defined
    operations and
 * put the result to the same property place
 * ------- (user Defined)reduceOpt (A.property) = A.property
    ----------
 */
```

```
template<typename GetOperation, typename ReduceOperation>
void allReduceOp(GetOperation getOp, ReduceOperation reduceOpType
    );

/* Collective operation on a specific property using predefined
    operations and
 * put the result to the place of another property
 * ———— reduceOpt (A.propertyGet) = A.propertyPut ————————
 */
template<typename GetOperation, typename PutOperation>
void allReduceOp(GetOperation getOp, PutOperation putOp, int
    reduceOpType);

/* Collective operation on a specific property using user defined
     operations and
 * put the result to the place of another property
 * ———— (user Defined)reduceOpt (A.propertyGet) = A.
     propertyPut ————————
 */
template<typename GetOperation, typename PutOperation,
        typename ReduceOperation>
void allReduceOp(GetOperation getOp, PutOperation putOp,
        ReduceOperation reduceOpType);

/* Predefined collective operation on a predefined operation
    between the first property and
 * the second property then put the result to the place of
     another property
 * ——reduceOpt [(EP.propertyFirst) optBetween (EP.propertySecond)
     ] = EP.property ——
 */
template<typename GetFirstOperation, typename GetSecondOperation,
        typename GetOutPut>
void allReduceOp(GetFirstOperation getFirst, GetSecondOperation
    getSecond,
        int opBetween, int reduceOpType, GetOutPut getOutPut);

/* User defined defined collective operation on a user defined
    operation between the first property
 * and the second property then put the result to the place of
     another property
 * ——reduceOpt [(EP.propertyFirst) optBetween (EP.propertySecond)
     ] = EP.property ——
 */
template<typename GetFirstOperation, typename GetSecondOperation,
```

```
        typename OperationBetween, typename Operation, typename
            GetOutPut>
void allReduceOp(GetFirstOperation getFirst, GetSecondOperation
    getSecond,
        OperationBetween opBetween, Operation op, GetOutPut
            getOutPut);

/************************************************
 *   Reduction Operations on all the
 *   Elementary Points in the Ensemble
 *   and store the result to a specific place
 ************************************************/
/* Collective operation on a specific property using predefined
    operations and
 * put the result to the a user specified place
 * ------- reduceOpt (A.property) = result ----------
 */
template<typename GetOperation>
void reduceOp(GetOperation op, int reduceOpType, void* result);

/* Collective operation on a specific property using user defined
     operations and
 * put the result to a user specified place
 * ------- (user Defined)reduceOpt (A.property) = result
    ----------
 */
template<typename GetOperation, typename ReduceOperation>
void reduceOp(GetOperation op, ReduceOperation reduceOpType, void
    * result);

/* Predefined collective operation on a predefined operation
    between the first property and
 * the second property then put the result to a user specified
     place
 * --reduceOpt [(EP.propertyFirst) optBetween (EP.propertySecond)
     ] = result --
 */
template<typename GetFirstOperation, typename GetSecondOperation>
void reduceOp(GetFirstOperation getFirst, GetSecondOperation
    getSecond,
        int opBetween, int reduceOpType, void* result);

/* User defined defined collective operation on a user defined
    operation between the first property
 * and the second property then put the result to the place of
     another property
```

```
     * --reduceOpt [(EP.propertyFirst) optBetween (EP.propertySecond)
         ] = result --
     */
    template<typename GetFirst, typename GetSecond, typename
       OperationBetween,
            typename ReduceOperation>
    void reduceOp(GetFirst getFirst, GetSecond getSecond,
            OperationBetween opBetween, ReduceOperation reduceOpType,
            void* result);

    /************************************************
     *   Get Operations
     ************************************************/
    //get Topology pointer
    Topology* getTopPtr(unsigned short topId) const;

    //get Topology pointer set
    vector<Topology*>& getTopPtrSet();

    //get GlobalSize
    unsigned long getGlobalSize();

    //get local set of elementary points
    vector<EP>& getEP_Set();
};

#endif /* ENSEMBLE_H_ */
```

A.3. Ensemble-based Programs

A.3.1. Irregular Grid Program

The point definition is shown in Listing A.4.

Listing A.4: Point Definition

```
#include "Point.h"

Point::Point() {
    this->id = 0;
    this->buffer = 0;
}

Point::Point(int id) {
    this->id = id;
```

```
    this->buffer = 0;
}

Point::Point(int id, double value) {
    this->id = id;
    this->value = value;
    this->buffer = 0;
}

Point::Point(int id, double value, Ensemble<Point, Topology<Point> >*
    pEnsemble) {
    this->id = id;
    this->value = value;
    this->buffer = 0;
    this->pEnsemble = pEnsemble;
}

Point::~Point() {
    // TODO Auto-generated destructor stub
}

int Point::getId() {
    return this->id;
}

double Point::getValue() {
    return this->value;
}

void Point::setEnsemble(Ensemble<Point, Topology<Point> >* pEnsemble)
    {
    this->pEnsemble = pEnsemble;
}

Ensemble<Point, Topology<Point> >* Point::getEnsemble() {
    return this->pEnsemble;
}

void Point::localCompute() {
    double sum = this->value;
    for (int i = 0; i
            < pEnsemble->getNghbs(this, pEnsemble->getTopSet()[0]).
                size(); i++) {
        sum
                += pEnsemble->getNghbs(this, pEnsemble->getTopSet()
                    [0])[i]->getValue();
```

```
    }
    if (pEnsemble->getNghbs(this, pEnsemble->getTopSet()[0]).size()
        != 0)
        sum = sum
                / (pEnsemble->getNghbs(this, pEnsemble->getTopSet()
                    [0]).size()
                        + 1);
    this->value = sum;
}

void Point::print() {
    cout << " id: " << this->id << " value: " << this->value << endl;
}
```

The main function of the irregular program is shown in Listing A.5.

Listing A.5: Main function of irregular grids

```
#include <map>
#include <vector>
#include <iostream>
#include <algorithm>
#include <sys/time.h>
#include "Point.h"

#define FINAL_ITER_STEP 128

#include "Ensemble.h"
#include "Topology.h"

using namespace std;

double dispEllapsedTime(struct timeval startTime);
void non_dupInsert(int in, vector<int>& dupVec);
unsigned long cellIndexOf3DIndex(int xIndex, int yIndex, int zIndex);
/************************************************
 *  Main Function
 ************************************************/
int main() {
    double start_time, end_time;
    struct timeval startTv;
    /************************************************
     *  Ensemble Creation
     ************************************************/
    Ensemble<Point, Topology<Point> > pointEnsemble;
    /************************************************
     *  Insert EPs into the Ensemble
     ************************************************/
```

```
    Point* point;
    for (int i = 0; i < NUM_OF_POINT; i++) {
        point = new Point(i, double(i / 1000), &pointEnsemble);
        pointEnsemble.insert(point);
    }
    /*******************************************
     *  create the root topology
     *******************************************/
    Topology<Point> pointTop(NUM_OF_POINT, nghbList, &pointEnsemble);
    /*******************************************
     *  Insert the root topology into the Ensemble
     *******************************************/
    pointEnsemble.insertTopology(&pointTop);

    start_time = dispEllapsedTime(startTv);
    for (int iter = 0; iter < FINAL_ITER_STEP; iter++) {
        /*************************************
         *  update Shadow copy Operations
         *************************************/
        pointEnsemble.update(pointEnsemble.getTopSet()[0]);
        /*************************************
         *  Parallel Operations
         *************************************/
        pointEnsemble.parallel(mem_fun_ref(&Point::localCompute),
                pointEnsemble.getTopSet()[0]);
    }
    end_time = dispEllapsedTime(startTv);
    cout << "elapse time is: " << end_time - start_time << endl;
    pointEnsemble.finalize();
    return 0;
}

double dispEllapsedTime(struct timeval startTime) {
    struct timeval endTime;
    gettimeofday(&endTime, NULL);
    int diff_usec = endTime.tv_usec - startTime.tv_usec;
    int diff_sec = endTime.tv_sec - startTime.tv_sec;
    double ellapsed_time = diff_sec + (diff_usec / 1000000.0);
    return ellapsed_time;
}
```

A.3.2. Molecular Dynamics Program

The molecule definition is shown in Listing A.6.

Listing A.6: Molecule Definition

```
#include "Molecule.h"
#include "Macros.h"

Molecule::Molecule() {
    this->id = 0;
    this->mass = 0;
    for (int i = 0; i < 3; i++) {
        this->position[i] = 0.0;
        this->velocity[i] = 0.0;
        this->force[i] = 0.0;
        this->accelerate[i] = 0.0;
    }
}

Molecule::Molecule(int id, double mass, double position[3], double
   velocity[3],
        Ensemble<Molecule, Topology<Molecule> >* pEnsemble) {
    this->id = id;
    this->mass = mass;
    for (int i = 0; i < 3; i++) {
        this->position[i] = position[i];
        this->velocity[i] = velocity[i];
        this->force[i] = 0.0;
        this->accelerate[i] = 0.0;
    }
    this->pEnsemble = pEnsemble;
}

Molecule::~Molecule() {
    // TODO Auto-generated destructor stub
}

double Molecule::getPositionNdim(int dimension) {
    return this->position[dimension];
}

double* Molecule::getPosition() {
    return this->position;
}

int Molecule::getId() {
    return this->id;
}

void Molecule::setEnsemble(Ensemble<Molecule, Topology<Molecule> >*
```

```
    pEnsemble) {
    this->pEnsemble = pEnsemble;
}

void Molecule::requestNghbs() {
    vector<Molecule*> myNghbs;
    myNghbs = pEnsemble->getNghbs(this, pEnsemble->getTopSet()[0]);
}

void Molecule::localCompute() {
    for (int i = 0; i < pEnsemble->getNghbs(this, pEnsemble->
        getTopSet()[0]).size(); i++) {
        interact(pEnsemble->getNghbs(this, pEnsemble->getTopSet()[0])
            [i]);
    }
    updateLocation();
    limitVelocity();
    wrapAround();
}

void Molecule::interact(Molecule*mp) {
    double rx, ry, rz, r, fx, fy, fz, f;
    //computing base values
    rx = position[0] - mp->position[0];
    ry = position[1] - mp->position[1];
    rz = position[2] - mp->position[2];
    r = sqrt(rx * rx + ry * ry + rz * rz);

    //if(r < 0.000001 || r >= DEFAULT_RADIUS)
    if (r < 0.000001 || r > CUT_OFF_RADIUS)
        return;

    f = A / pow(r, 12) - B / pow(r, 6);
    force[0] = f * rx / r;
    force[1] = f * ry / r;
    force[2] = f * rz / r;

    //updating particle properties
    force[0] += fx;
    force[1] += fy;
    force[2] += fz;
}

void Molecule::updateLocation() {
    // applying kinetic equations
    accelerate[0] = force[0] / MASS;
```

```
    accelerate[1] = force[1] / MASS;
    accelerate[2] = force[2] / MASS;

    velocity[0] = velocity[0] + accelerate[0] * DELTA;
    velocity[1] = velocity[1] + accelerate[1] * DELTA;
    velocity[2] = velocity[2] + accelerate[2] * DELTA;

    limitVelocity();

    position[0] = position[0];
    +velocity[0] * DELTA;
    position[1] = position[1];
    +velocity[1] * DELTA;
    position[2] = position[2];
    +velocity[2] * DELTA;

    force[0] = 0.0;
    force[1] = 0.0;
    force[2] = 0.0;
}

void Molecule::limitVelocity() {
    //if( fabs(p.vx * DEFAULT_DELTA) > DEFAULT_RADIUS ) {
    if (fabs(velocity[0]) > MAX_VELOCITY) {
        //if( p.vx * DEFAULT_DELTA < 0.0 )
        if (velocity[0] < 0.0)
            velocity[0] = -MAX_VELOCITY;
        else
            velocity[0] = MAX_VELOCITY;
    }
    //if( fabs(p.vy * DEFAULT_DELTA) > DEFAULT_RADIUS ) {
    if (fabs(velocity[1]) > MAX_VELOCITY) {
        //if( p.vy * DEFAULT_DELTA < 0.0 )
        if (velocity[1] < 0.0)
            velocity[1] = -MAX_VELOCITY;
        else
            velocity[1] = MAX_VELOCITY;
    }
    //if( fabs(p.vy * DEFAULT_DELTA) > DEFAULT_RADIUS ) {
    if (fabs(velocity[2]) > MAX_VELOCITY) {
        //if( p.vy * DEFAULT_DELTA < 0.0 )
        if (velocity[2] < 0.0)
            velocity[2] = -MAX_VELOCITY;
        else
            velocity[2] = MAX_VELOCITY;
    }
```

```
}

void Molecule::wrapAround() {
    if (position[0] < 0.0)
        position[0] += DOMAIN_X;
    if (position[1] < 0.0)
        position[1] += DOMAIN_Y;
    if (position[2] < 0.0)
        position[2] += DOMAIN_Z;

    if (position[0] > DOMAIN_X)
        position[0] -= DOMAIN_X;
    if (position[1] > DOMAIN_Y)
        position[1] -= DOMAIN_Y;
    if (position[2] > DOMAIN_Z)
        position[2] -= DOMAIN_Z;
    return;
}
```

The main function of the MD program is shown in Listing A.7.

Listing A.7: Main function of MD simulation

```
#include <sys/time.h>
#include <map>
#include <vector>
#include <iostream>
#include <algorithm>
#include "Molecule.h"
#include "Macros.h"

#include "Ensemble.h"
#include "Topology.h"

using namespace std;

double dispEllapsedTime(struct timeval startTime);
/*****************************************
 *  Main Function
 *****************************************/
int main(int argc, char **argv) {
    double start_time, end_time;
    struct timeval startTimeval;
    double domainMin[3] = { 0, 0, 0 };
    double domainMax[3] = { DOMAIN_SIZE, DOMAIN_SIZE, DOMAIN_SIZE };
    /*************************************
     *  Ensemble Creation
     *************************************/
```

```
Ensemble<Molecule, Topology<Molecule> > moleculeEnsemble(&argc, &
    argv,
        domainMin, domainMax);
/*************************************
 *  Insert EPs into the Ensemble
 *************************************/
Molecule* molecule;
double position[3];
double velocity[3] = { 0, 0, 0 };
for (int i = 0; i < NUM_OF_PARTICLE; i++) {
    for (int dim = 0; dim < 3; dim++) {
        position[dim] = ((double) (rand() % RAND_MAX) / RAND_MAX)
                * DOMAIN_SIZE;
    }
    //cout << endl;
    molecule = new Molecule(i, i, position, velocity, &
        moleculeEnsemble);
    moleculeEnsemble.insertEP(molecule);
}
/*************************************
 *  Topology Creation
 *************************************/
Topology<Molecule> moleculeTop(0, CUT_OFF_RADIUS, &
    moleculeEnsemble);
/*************************************
 *  Insert Topologies into the Ensemble
 *************************************/
moleculeEnsemble.insertTopology(&moleculeTop);

start_time = dispEllapsedTime(startTimeval);
for (int iter = 0; iter < FINAL_STEP; iter++) {
   /*************************************
    *  update shadow copy Operations
    *************************************/
    moleculeEnsemble.update(&moleculeTop, moleculeEnsemble.
        getTopSet()[0]);
    /*************************************
     *  Parallel Operations
     *************************************/
    moleculeEnsemble.parallel(mem_fun_ref(&Molecule::localCompute
        ),
            moleculeEnsemble.getTopSet()[0]);
    /*************************************
     *  update topology
     *************************************/
    moleculeEnsemble.getTopSet()[0]->update();
```

```
    }
    end_time = dispEllapsedTime(startTimeval);
    cout << "elapse time is: " << end_time - start_time << endl;
    moleculeEnsemble.finalize();
    return 0;
}

double dispEllapsedTime(struct timeval startTime) {
    struct timeval endTime;
    gettimeofday(&endTime, NULL);
    int diff_usec = endTime.tv_usec - startTime.tv_usec;
    int diff_sec = endTime.tv_sec - startTime.tv_sec;
    double ellapsed_time = diff_sec + (diff_usec / 1000000.0);
    return ellapsed_time;
}
```

A.4. Acronyms

FLOPS: Floating Point Operations Per Second)

HPC: High Performance Computing

SISD: Single Instruction Single Data stream

SIMD: Single Instruction Multiple Data streams

MISD: Multiple Instruction Single Data stream

MIMD: Multiple Instruction Multiple Data streams

UMA: Uniform Memory Access

SMP: Symmetric Multi-Processor

NUMA: Non Uniform Memory Access architecture

ccNUMA: Non Uniform Memory Access architecture

GPU: Graphics Processing Unit

GPGPU: General Purpose Graphics Processing Unit

DSP: Digital Signal Processor

ASIC: Application Specific Integrated Circuit

FPGA: Field Programmable Gate Array

MPI: Message Passing Interface

Pthreads: POSIX Threads

A.4 Acronyms

OpenMP: Open Multi Processing

OpenCL: Open Computing Language

CUDA: Compute Unified Device Architecture

API: Application Programming Interface

CPU: Central Processing Unit

HPF: High Performance Fortran

GA: Global Arrays

TBB: Threading Building Blocks

UE: Unit of Execution

EP: Elementary Point

MD: Molecular Dynamics

CFD: Computational Fluid Dynamics

FEM: Finite Element Method

FEA: Finite Element Analysis

LJ-potential: Lennard-Jones potential

SAMRAI : Structured Adaptive Mesh Refinement Application Infrastructure

Bibliography

[1] J. Board, J.A., Z. Hakura, W. Elliott, D. Gray, W. Blanke, and J. Leathrum, J.F., "Scalable implementations of multipole-accelerated algorithms for molecular dynamics," in *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, may 1994, pp. 87 –94.

[2] D. Boyd and S. Milosevich, "Supercomputing and drug discovery research," *Perspectives in Drug Discovery and Design*, vol. 1, pp. 345–358, 1993. [Online]. Available: http://dx.doi.org/10.1007/BF02174534

[3] E. Clementi, S. Chin, G. Corongiu, J. Detrich, M. Dupuis, D. Folsom, G. Lie, D. Logan, and V. Sonnad, "Supercomputing and super computers: for science and engineering in general and for chemistry and biosciences in particular," in *Spectroscopy of Inorganic Bioactivators*, ser. NATO ASI Series, T. Theophanides, Ed. Springer Netherlands, 1989, vol. 280, pp. 1–112. [Online]. Available: http://dx.doi.org/10.1007/978-94-009-2409-3_1

[4] K. Kremer, "Supercomputing in polymer research," in *High-Performance Computing and Networking*, ser. Lecture Notes in Computer Science, W. Gentzsch and U. Harms, Eds. Springer Berlin Heidelberg, 1994, vol. 796, pp. 244–253. [Online]. Available: http://dx.doi.org/10.1007/BFb0020381

[5] J. Moreira, "Blue Gene: A massively parallel system," in *Computational Science ICCS 2001*, ser. Lecture Notes in Computer Science, V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C. Tan, Eds. Springer Berlin Heidelberg, 2001, vol. 2073, pp. 10–10. [Online]. Available: http://dx.doi.org/10.1007/3-540-45545-0_8

[6] Padua, "Blue Gene/P," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 175–175. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-09766-4_2409

[7] P. Gepner, D. Fraser, and M. Kowalik, "Second generation quad-core intel Xeon processors bring 45 nm technology and a new level of performance to HPC applications," in *Computational Science ICCS 2008*, ser. Lecture Notes in Computer Science, M. Bubak, G. Albada, J. Dongarra, and P. Sloot, Eds. Springer Berlin Heidelberg, 2008, vol. 5101, pp. 417–426. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69384-0_47

[8] Dongarra, "Top500 list," Jun. 2010. [Online]. Available: http://www.top500.org/lists/2012/06/

[9] T. Agerwala, "Keynote: Challenges on the road to exascale computing," in *High Performance Embedded Architectures and Compilers*, ser. Lecture Notes in Computer Science, E. J. M. B. Seznec, Andress, M. Martonosi, and T. Ungerer, Eds. Springer Berlin Heidelberg, 2009, vol. 5409, pp. 1–1. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92990-1_1

[10] T. Scogland, B. Subramaniam, and W.-c. Feng, "The green500 list: escapades to exascale," *Computer Science - Research and Development*, vol. 1, pp. 1–9, 2012. [Online]. Available: http://dx.doi.org/10.1007/s00450-012-0212-6

[11] IBM, "Blue Gene/Q: by co-design," *Computer Science Research and Development*, vol. 1, pp. 1–9, 2012. [Online]. Available: http://dx.doi.org/10.1007/s00450-012-0215-3

[12] J. Dongarra, "The Linpack benchmark: An explanation," in *Supercomputing*, ser. Lecture Notes in Computer Science, E. Houstis, T. Papatheodorou, and C. Polychronopoulos, Eds. Springer Berlin Heidelberg, 1988, vol. 297, pp. 456–474. [Online]. Available: http://dx.doi.org/10.1007/3-540-18991-2_27

[13] M. Louter-Nool, "Linpack routines based on level 2 blas," *The Journal of Supercomputing*, vol. 3, pp. 331–349, 1989. [Online]. Available: http://dx.doi.org/10.1007/BF00128169

[14] X.-J. Yang, X.-K. Liao, K. Lu, Q.-F. Hu, J.-Q. Song, and J.-S. Su, "The Tianhe-1A supercomputer: Its hardware and software," *Journal of Computer Science and Technology*, vol. 26, pp. 344–351, 2011. [Online]. Available: http://dx.doi.org/10.1007/s02011-011-1137-8

[15] N.-H. Sun, J. Xing, Z.-G. Huo, G.-M. Tan, J. Xiong, B. Li, and C. Ma, "Dawning nebulae: A petaflops supercomputer with a heterogeneous structure," *Journal of Computer Science and Technology*, vol. 26, pp. 352–362, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11390-011-1138-3

[16] O. A. R. Board, "OpenMP Application Program Interface," OpenMP, Specification, 2011. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP3.1.pdf

[17] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[18] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.

[19] P. S. Pacheco, *Parallel programming with MPI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[20] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press, 1994.

[21] N. Corporation, *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[23] R. Schreiber, "An introduction to HPF," in *The Data Parallel Programming Model*, ser. Lecture Notes in Computer Science, G.-R. Perrin and A. Darte, Eds. Springer Berlin Heidelberg, 1996, vol. 1132, pp. 27–44. [Online]. Available: http://dx.doi.org/10.1007/3-540-61736-1_41

[24] K. Kennedy and C. Koelbel, "High Performance Fortran 2.0," in *Compiler Optimizations for Scalable Parallel Systems*, ser. Lecture Notes in Computer Science, S. Pande and D. Agrawal, Eds. Springer Berlin Heidelberg, 2001, vol. 1808, pp. 3–43. [Online]. Available: http://dx.doi.org/10.1007/3-540-45403-9_1

[25] L. V. Kale and S. Krishnan, "Charm++: a portable concurrent object oriented system based on c++," *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, Oct. 1993. [Online]. Available: http://doi.acm.org/10.1145/167962.165874

[26] L. V. Kale, B. Ramkumar, A. B. Sinha, and A. Gursoy, "The CHARM Parallel Programming Language and System: Part I – Description of Language Features," *Parallel Programming Laboratory Technical Report #95-02*, vol. 1, pp. 1–15, 1994.

[27] L. V. Kale, B. Ramkumar, A. B. Sinha, and V. A. Saletore, "The CHARM Parallel Programming Language and System: Part II – The Runtime system," *Parallel Programming Laboratory Technical Report #95-03*, vol. 1, pp. 1–14, 1994.

[28] Intel, "TBB (Intel Threading Building Blocks)," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 2029–2029. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-09766-4_2080

[29] G. Russell, P. Keir, A. Donaldson, U. Dolinsky, A. Richards, and C. Riley, "Programming heterogeneous multicore systems using threading building blocks," in *Euro-Par 2010 Parallel Processing Workshops*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6586, pp. 117–125. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21878-1_15

[30] C. Ierotheou, C. Forsey, and U. Block, "Parallelisation of a novel 3d hybrid structured-unstructured grid cfd production code," in *High-Performance Computing and Networking*, ser. Lecture Notes in Computer Science, B. Hertzberger and G. Serazzi, Eds. Springer Berlin Heidelberg, 1995, vol. 919, pp. 831–836. [Online]. Available: http://dx.doi.org/10.1007/BFb0046722

[31] S. Krajnovic, "CFD applications for high performance computing: Minisymposium abstract," in *Applied Parallel Computing. State of the Art in Scientific Computing*, ser. Lecture Notes in Computer Science, B. Kagstrom, E. Elmroth, J. Dongarra, and J. Warniewski, Eds. Springer Berlin Heidelberg, 2007, vol. 4699, pp. 167–167. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75755-9_20

[32] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, "Simulations of the formation, evolution and clustering of galaxies and quasars," *nat*, vol. 435, pp. 629–636, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1038/nature03597

[33] CGNS, "CFD general notation system," Jun. 2010. [Online]. Available: http://en.goldenmap.com/CGNS#

[34] S. J. Aarseth, *Gravitational N-Body Simulations*. Cambridge University Press, 2003. [Online]. Available: http://dx.doi.org/10.1017/CBO9780511535246

[35] S. P. Molner, "The art of molecular dynamics simulation (rapaport, d. c.)," *Journal of Chemical Education*, vol. 76, no. 2, p. 171, 1999. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ed076p171

[36] LRZ, "SuperMuc petascale system," Jun. 2012. [Online]. Available: https://www.lrz.de/services/compute/supermuc/systemdescription/

[37] M. J. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, no. 9, pp. 948 –960, sept. 1972.

[38] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming - a POSIX standard for better multiprocessing*. O'Reilly, 1996.

[39] M. Sato, "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors," in *System Synthesis, 2002. 15th International Symposium on*, oct. 2002, pp. 109 –111.

[40] D. Luebke, "CUDA: Scalable parallel programming for high-performance scientific computing," in *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, may 2008, pp. 836 –838.

[41] D. Zlotrg, N. Nosovic, and A. Huseinovic, "Utilizing cuda architecture for improving application performance," in *Telecommunications Forum (TELFOR), 2011 19th*, nov. 2011, pp. 1458 –1461.

[42] N. Karunadasa and D. Ranasinghe, "Accelerating high performance applications with CUDA and MPI," in *Industrial and Information Systems (ICIIS), 2009 International Conference on*, dec. 2009, pp. 331 –336.

[43] W. Wei and Y. Huang, "CUDA framework for turbulence flame simulation," in *Electronics and Signal Processing*, ser. Lecture Notes in Electrical Engineering, W. Hu, Ed. Springer Berlin Heidelberg, 2011, vol. 97, pp. 791–796. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21697-8_101

[44] A. Heck, "Linear algebra: Applications," in *Introduction to Maple*. Springer US, 1993, pp. 435–467. [Online]. Available: http://dx.doi.org/10.1007/978-1-4684-0519-4_18

[45] D. Scott, "The performance of linear algebra algorithms on Intel parallel supercomputers," in *Parallel Computing on Distributed Memory Multiprocessors*, ser. NATO ASI Series, F. zguener and F. Eral, Eds. Springer Berlin Heidelberg, 1993, vol. 103, pp. 143–150. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-58066-6_7

[46] F. Hehl, J. Fleischer, M. Steinhauser, G. Weiglein, J. Vermaseren, C. Heinicke, I. Kotsireas, E. Schruefer, Y. Obukhov, S. Tertychniy, T. Wolf, G. Baumann, A. Dolzmann, T. Sturm, V. Weispfenning, L. Lambe, J. Apel, I. Heckenberger, A. Schueler, W. Koepf, K. Gatermann, T. Beth, K. Homann, A. Klappenecker, J. Mueller-Quade, A. Nueckel, M. Roggenbach, V. Strehl, K. Behnke, K. Roesner, J. Grabmeier, M. Clausen, F. Kurth, P. Kovacs, L. Gonzalez-Vega, A. Dress, H. Melenk, B. Waits, P. Drijvers, J. Berry, T. Graham, J. Sharp, S. Townend, A. Watkins, N. Boston, D. Fowler, and O. Gloor, "Applications of computer algebra," in *Computer Algebra Handbook*, J. Grabmeier, E. Kaltofen, and V. Weispfenning, Eds. Springer Berlin Heidelberg, 2003, pp. 163–260. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-55826-9_3

[47] J. Dongarra, S. for Industrial, and A. Mathematics, *LINPACK Users' Guide*, ser. Miscellaneous Bks. Society for Industrial and Applied Mathematics, 1979. [Online]. Available: http://books.google.de/books?id=AmSm1n3Vw0cC

[48] C. Loan, M. Kilmer, and D. O¡¯Leary, "Matrix decompositions: Linpack and beyond," in *G.W. Stewart*, ser. Contemporary Mathematicians, M. E. Kilmer and D. P. O Leary, Eds. Birkhuser Boston, 2010, pp. 27–44. [Online]. Available: http://dx.doi.org/10.1007/978-0-8176-4968-5_4

[49] H. Stokes, "Matrix operations using Linpack," in *Management and Office Information Systems*, S.-K. Chang, Ed. Springer US, 1984, pp. 415–434. [Online]. Available: http://dx.doi.org/10.1007/978-1-4613-2677-9_24

[50] W. F. Tichy, R. Ass, and W. F. Tichy, "Parallel matrix multiplication on the connection machine," In Proceedings of the Conference on Scientific Applications of the CM, Tech. Rep., 1988.

[51] M. Beare, "The southampton - east anglia (sea) model: A general purpose parallel ocean circulation model," in *High-Performance Computing*, R. Allan,

M. Guest, A. Simpson, D. Henty, and D. Nicole, Eds. Springer US, 1999, pp. 337–346. [Online]. Available: http://dx.doi.org/10.1007/978-1-4615-4873-7_36

[52] J. S. Shang, "Computational electromagnetics," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 97–99, Mar. 1996. [Online]. Available: http://doi.acm.org/10.1145/234313.234357

[53] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner, "Compiling stencils in high performance fortran," in *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '97. New York, NY, USA: ACM, 1997, pp. 1–20. [Online]. Available: http://doi.acm.org/10.1145/509593.509605

[54] Y. Che, Z. Wang, X. Li, and L. Yang, "Locality optimizations for jacobi iteration on distributed parallel systems," in *Parallel and Distributed Processing and Applications*, ser. Lecture Notes in Computer Science, J. Cao, L. Yang, M. Guo, and F. Lau, Eds. Springer Berlin Heidelberg, 2005, vol. 3358, pp. 91–104. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30566-8_15

[55] P. Letourneau, *Statistical Mechanics of Cellular Automata with Memory*, ser. Canadian theses. University of Calgary (Canada), 2006. [Online]. Available: http://books.google.de/books?id=0wqZIxcBmh4C

[56] MathWorld, "Von Neumann neighborhood," Jun. 2012. [Online]. Available: http://mathworld.wolfram.com/vonNeumannNeighborhood.html

[57] ——, "Moore neighborhood," Jun. 2012. [Online]. Available: http://mathworld.wolfram.com/MooreNeighborhood.html

[58] J. Rai and P. Xirouchakis, "Fem-based prediction of workpiece transient temperature distribution and deformations during milling," *The International Journal of Advanced Manufacturing Technology*, vol. 42, pp. 429–449, 2009. [Online]. Available: http://dx.doi.org/10.1007/s00170-008-1610-6

[59] Y. Deshayes, Y. Ousten, and L. Bechou, "Three-dimensional techniques for fem simulations in laser modules and their applications," in *MEMS/NEMS*, C. Leondes, Ed. Springer US, 2006, pp. 1746–1835. [Online]. Available: http://dx.doi.org/10.1007/0-387-25786-1_43

[60] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott, "Large scale parallel structured AMR calculations using the SAMRAI framework," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '01. New York, NY, USA: ACM, 2001, pp. 6–6. [Online]. Available: http://doi.acm.org/10.1145/582034.582040

[61] R. I. Klein, "Star formation with 3-d adaptive mesh refinement: the collapse and fragmentation of molecular clouds," *Journal of Computational and Applied Mathematics*, vol. 109, no. 12, pp. 123 – 152, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377042799001569

[62] M. Russell, G. Allen, G. Daues, I. Foster, E. Seidel, J. Novotny, J. Shalf, and G. von Laszewski, "The astrophysics simulation collaboratory: A science portal enabling community software development," *Cluster Computing*, vol. 5, pp. 297–304, 2002. [Online]. Available: http://dx.doi.org/10.1023/A%3A1015629422149

[63] S. Wan, P. Coveney, and D. Flower, "Molecular dynamics simulations," in *Immunoinformatics*, ser. Methods in Molecular Biology, D. Flower, Ed. Humana Press, 2007, vol. 409, pp. 321–339. [Online]. Available: http://dx.doi.org/10.1007/978-1-60327-118-9_24

[64] A. Pai, Y.-i. Choo, and M. Chen, "Distributed tree structures for N-body simulation," in *Languages, Compilers and Run-Time Systems for Scalable Computers*, B. Szymanski and B. Sinharoy, Eds. Springer US, 1996, pp. 307–310. [Online]. Available: http://dx.doi.org/10.1007/978-1-4615-2315-4_28

[65] E. Darve, "The fast multipole method: Numerical implementation," *Journal of Computational Physics*, vol. 160, no. 1, pp. 195 – 240, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999100964519

[66] J. A. Board, W. D. Elliott, W. T. Rankin, and Z. S. Hakura, "Scalable Variants of Multipole-based Algorithms for Molecular Dynamics Applications," in *Parallel Processing for Scientific Computing*, 1995, pp. 295–300.

[67] W. Rankin and J. Board, J.A., "A portable distributed implementation of the parallel multipole tree algorithm," in *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on*, aug 1995, pp. 17 –22.

[68] V. Springel, N. Yoshida, and S. D. M. White, "GADGET: a code for collisionless and gasdynamical cosmological simulations," *New Astron*, 2001.

[69] V. Springel, "The cosmological simulation code GADGET-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, 2005.

[70] G. Luckhurst and N. Veracini, *The Molecular Dynamics of Liquid Crystals*, ser. NATO ASI series. Ser. C. : Mathematical and physical sciences. Kluwer Acad. Publ., 1994. [Online]. Available: http://books.google.de/books?id=JMUDH1tzh-0C

[71] J. E. Jones, "On the determination of molecular fields. I. from the variation of the viscosity of a gas with temperature," *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, vol. 106, no. 738, pp. pp. 441–462, 1924. [Online]. Available: http://www.jstor.org/stable/94264

[72] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *JOURNAL OF COMPUTATIONAL PHYSICS*, vol. 117, pp. 1–19, 1995.

[73] W. Smith, "Molecular dynamics on hypercube parallel computers," *Computer Physics Communications*, vol. 62, pp. 229 – 248, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0010465591900975

[74] D. Okunbor and R. Murty, "Parallel molecular dynamics using force decomposition," in *Computational Molecular Dynamics: Challenges, Methods, Ideas*, ser. Lecture Notes in Computational Science and Engineering, P. Deuflhard, J. Hermans, B. Leimkuhler, A. Mark, S. Reich, and R. Skeel, Eds. Springer Berlin Heidelberg, 1999, vol. 4, pp. 483–494. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-58360-5_29

[75] M. Laradji, S. Toxvaerd, and O. Mouritsen, "Spinodal decomposition in three-dimensional binary fluids: A large-scale molecular dynamics simulation," in *Computer Simulation Studies in Condensed-Matter Physics IX*, ser. Springer Proceedings in Physics, D. Landau, K. Mon, and H.-B. Schuettler, Eds. Springer Berlin Heidelberg, 1997, vol. 82, pp. 150–155. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-60597-0_16

[76] G. Fox, "Domain decomposition in distributed and shared memory environments," in *Supercomputing*, ser. Lecture Notes in Computer Science, E. Houstis, T. Papatheodorou, and C. Polychronopoulos, Eds. Springer Berlin Heidelberg, 1988, vol. 297, pp. 1042–1073. [Online]. Available: http://dx.doi.org/10.1007/3-540-18991-2_62

[77] A. Eeciolu, Srinivasan, "Domain decomposition for particle methods on the sphere," in *Parallel Algorithms for Irregularly Structured Problems*, ser. Lecture Notes in Computer Science, A. Ferreira, J. Rolim, Y. Saad, and T. Yang, Eds. Springer Berlin Heidelberg, 1996, vol. 1117, pp. 119–130. [Online]. Available: http://dx.doi.org/10.1007/BFb0030102

[78] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, "NAMD: biomolecular simulation on thousands of processors," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=762761.762810

[79] K. Kennedy and U. Kremer, "Automatic data layout for distributed-memory machines," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 4, pp. 869–916, Jul. 1998. [Online]. Available: http://doi.acm.org/10.1145/291891.291901

[80] J. Ramanujam and P. Sadayappan, "Compile-time techniques for data distribution in distributed memory machines," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 2, no. 4, pp. 472 –482, oct 1991.

[81] K. Kennedy and U. Kremer, "Automatic data layout for high performance Fortran," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*

(CDROM), ser. Supercomputing '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/224170.224495

[82] J. Nieplocha, M. Krishnan, B. Palmer, V. Tipparaju, R. Harrison, and D. Chavara-Miranda, "Global Arrays: Parallel programming toolkit," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 779–787. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-09766-4_403

[83] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[84] D. R. Musser and A. Saini, *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1995.

[85] P. Pirkelbauer, S. Parent, M. Marcus, and B. Stroustrup, "Runtime concepts for the C++ Standard Template Library," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 171–177. [Online]. Available: http://doi.acm.org/10.1145/1363686.1363734

[86] C++, "An introduction to the STL/CLR library," in *Foundations of C++/CLI.* Apress, 2008, pp. 333–381. [Online]. Available: http://dx.doi.org/10.1007/978-1-4302-1024-5_12

[87] R. Das, Y. shin Hwang, M. Uysal, J. Saltz, and A. Sussman, "Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics," in *Mississippi State University, Starkville, MS.* IEEE Computer Society Press, 1993, pp. 45–56.

[88] J. Saltz, R. Ponnusamy, S. D. Sharma, B. Moon, Y. shin Hwang, M. Uysal, and R. Das, "A manual for the CHAOS runtime library," 1995.

[89] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, and G. Fox, "Runtime support and compilation methods for user-specified irregular data distributions," *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. 6, no. 8, pp. 815–831, 1995.

[90] R. Ponnusamy, J. Saltz, and A. Choudhary, "Runtime compilation techniques for data partitioning and communication schedule reuse," in *PROCEEDINGS OF THE 1993 ACM/IEEE CONFERENCE ON SUPERCOMPUTING.* ACM, 1993, pp. 361–370.

[91] S. D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz, "Runtime and compile-time support for adaptive irregular problems," 1994.

[92] L. Kale, M. Bhandarkar, R. Brunner, N. Krawetz, J. Phillips, and A. Shinozaki, "NAMD: A case study in multilingual parallel programming," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, Z. Li,

P.-C. Yew, S. Chatterjee, C.-H. Huang, P. Sadayappan, and D. Sehr, Eds. Springer Berlin Heidelberg, 1998, vol. 1366, pp. 367–381. [Online]. Available: http://dx.doi.org/10.1007/BFb0032705

[93] L. Kale, A. Bhatele, E. Bohm, and J. Phillips, "NAMD (NAnoscale Molecular Dynamics)," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 1249–1254. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-09766-4_505

[94] T. El-Ghazawi and L. Smith, "UPC: Unified Parallel C," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188483

[95] I. Chivers and J. Sleightholme, "Coarray Fortran," in *Introduction to Programming with Fortran*. Springer London, 2012, pp. 459–469. [Online]. Available: http://dx.doi.org/10.1007/978-0-85729-233-9_30

[96] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," in *In ACM*, 1998, pp. 10–11.

[97] fortress, "Fortress language specification," 2008.

[98] G. Steele, "Parallel programming and parallel abstractions in fortress," in *Functional and Logic Programming*, ser. Lecture Notes in Computer Science, M. Hagiya and P. Wadler, Eds. Springer Berlin Heidelberg, 2006, vol. 3945, pp. 1–1. [Online]. Available: http://dx.doi.org/10.1007/11737414_1

[99] chaple, "Chapel language specification," 2005.

[100] V. Saraswat, "X10: Concurrent programming for modern architectures," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, Z. Shao, Ed. Springer Berlin Heidelberg, 2007, vol. 4807, pp. 1–1. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-76637-7_1

[101] R. Karrenberg and S. Hack, "Improving performance of OpenCL on CPUs," in *Compiler Construction*, ser. Lecture Notes in Computer Science, M. O Boyle, Ed. Springer Berlin Heidelberg, 2012, vol. 7210, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28652-0_1

[102] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "OpenCL as a programming model for GPU clusters," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, S. Rajopadhye and M. Mills Strout, Eds. Springer Berlin Heidelberg, 2013, vol. 7146, pp. 76–90. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36036-7_6

[103] T. Cickovski, C. Sweet, and J. A. Izaguirre, "MDL, a domain-specific language for molecular dynamics," in *Proceedings of the 40th Annual Simulation Symposium*,

ser. ANSS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 256–266. [Online]. Available: http://dx.doi.org/10.1109/ANSS.2007.26

[104] Python, "Python programming language," Jun. 2012. [Online]. Available: http://www.python.org/

[105] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: a domain specific language for building portable mesh-based PDE solvers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 9:1–9:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063396

[106] W. Luzhou, K. Sano, and S. Yamamoto, "Domain-specific language and compiler for stencil computation on FPGA-based systolic computational-memory array," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, O. Choy, R. Cheung, P. Athanas, and K. Sano, Eds. Springer Berlin Heidelberg, 2012, vol. 7199, pp. 26–39. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28365-9_3

[107] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 4:1–4:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413375

[108] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir stencil compiler," in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989508

[109] C. Leiserson, "Programming irregular parallel applications in Cilk," in *Solving Irregularly Structured Problems in Parallel*, ser. Lecture Notes in Computer Science, G. Bilardi, A. Ferreira, R. Luling, and Rolim, Eds. Springer Berlin Heidelberg, 1997, vol. 1253, pp. 61–71. [Online]. Available: http://dx.doi.org/10.1007/3-540-63138-0_6

[110] C. E. Leiserson, "The Cilk++ concurrency platform," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 522–527. [Online]. Available: http://doi.acm.org/10.1145/1629911.1630048

[111] C. Lukacs and E. Tarján, *Mathematical games.* Barnes & Noble Books, 1996. [Online]. Available: http://books.google.com.hk/books?id=OImRoZBt2McC

[112] L. Li, "The Java language," in *Java: Data Structures and Programming*. Springer Berlin Heidelberg, 1998, pp. 57–102. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-95851-9_2

[113] A. Bundy and L. Wallen, "Smalltalk," in *Catalogue of Artificial Intelligence Tools*, ser. Symbolic Computation, A. Bundy and L. Wallen, Eds. Springer Berlin Heidelberg, 1984, pp. 123–124. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-96868-6_236

[114] D. Abrahams and A. Gurtovoy, *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*, ser. C++ in-depth series. Addison-Wesley, 2005. [Online]. Available: http://books.google.de/books?id=wbNQAAAAMAAJ

[115] P. Plauger, *The C++ standard template library*. Prentice Hall, 2001. [Online]. Available: http://books.google.de/books?id=ELJQAAAAMAAJ

[116] G. Karypis and V. Kumar, *MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.376

[117] Intel. (2011) Intel Xeon processor E7-4870. [Online]. Available: http://ark.intel.com/products/53579/Intel-Xeon-Processor-E7-4870-(30M-Cache-2_40-GHz-6_40-GTs-Intel-QPI)

[118] F. J, I. Bericht, M. Gerndt, and M. Gerndt, "Parallelization of the AVL FIRE benchmark with SVM-Fortran," 1995.