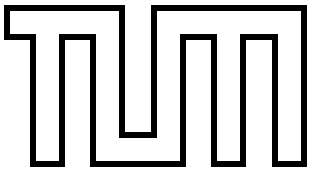


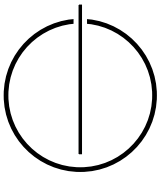
FAKULTÄT FÜR INFORMATIK

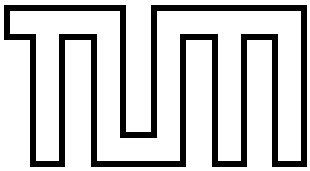
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Implementing a Molecular Dynamics Simulation for the Invasive Run-time Support System

Huaiwei Zhang





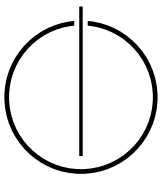
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Implementing a Molecular Dynamics Simulation for the Invasive Run-time Support System

Author:	Huaiwei Zhang
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Santiago Narváez, M.Sc.
Date:	May 6th, 2022



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, May 6th, 2022 Huaiwei Zhang

Acknowledgements

First, I want to thank my supervisor Prof. Bungartz sincerely. I really appreciate the topic he gave to me, and it is a memorable experience to complete the thesis and the project. During this work, I learned a lot about molecular dynamics simulation and invasive computing, which helps me lay a solid foundation for my career in the future. I also appreciate the favor M.Sc. Narváez did for me. Not only he gave helpful support to me about the project development, but he also provided me with guidance about the thesis writing. Without his help, it will take me longer time to finish the work.

I also want to give thanks to my family and friends. Although they did not directly help me about the thesis, they keep supporting my work during my life. All in all, I hope all of them can be lucky in their work and life.

Abstract

The invasive Run-time Support System (iRTSS) is a run-time system that supports the invasive Message Passing Interface (iMPI) API. iMPI extends the functionality of the standard MPI such that applications using it can change their resources at runtime. In the thesis, the application in iRTSS can change the number of iMPI processes during its execution.

During the thesis I ported miniMD, a C++ molecular dynamics simulation application, to iRTSS. miniMD is a simplification of the well-known LAMMPS application. Besides the default simulation scenario of miniMD, I added two collision simulation scenarios to it, which makes it easier to observe the simulation process. A Transmission Control Protocol/Internet Protocol (TCP/IP) communication scheme to gather the results of the simulation is a new part of the application. The result of this part is the invasive miniMD (*i*miniMD). *i*miniMD is executed in bare metal, which can transmit the simulation data to the visualization application.

The visualization application is developed with C++ based on MPI and OpenGL. It can receive the simulation data transmitted from *i*miniMD, visualize the simulation data, and record it in files. The visualization application runs on a Linux machine, which is connected with the bare metal executing *i*miniMD through a network cable.

Contents

Part I.

Introduction and Background

1. Introduction

1.1. Motivation

As computer systems develop higher computing speed and larger memory storage, molecular dynamics simulation that plays an important role in simulation science can be completed more efficiently. But it is sometimes hard to utilize the computing resource properly. With the help of iRTSS and iMPI [1], developers can manage the hardware and software resources before and during the process of a molecular dynamics simulation. In my thesis, a molecular dynamics simulation application will be developed to simulate one physical process with the capability of managing the resources all through the execution. Thus, the goal to test the functionalities of the iRTSS system can be achieved. Another goal is to visualize the simulation process.

1.1.1. The significance of computer simulation

The traditional scientific method to do research on natural processes is to construct an artificial model, describe appropriate variables, and develop an experiment under certain conditions in the laboratory. With the help of experiments, researchers get mathematical formulations to describe how each variable behaves and interacts with each other in the artificial model. However, it is hard to simulate some natural processes in the laboratory because of numerous variables, long simulation time, complex mathematical computation, etc. For example, there are $2.68678 \cdot 10^{25}$ atoms in a cubic meter of gas at a pressure of 101.325 kilopascal and a temperature of 273.15 Kelvin [2], which leads to a large number of computations in an experiment. Even though researchers can finish the experiment successfully, it is sometimes not easy to reproduce the experimental conditions. It is also hard to observe the experiment at a specific point in time if the process is rapid.

Researchers can create the initial experimental conditions, simulate the process, and observe the experiment at a given point in time with the help of a computer simulation. With a correct mathematical description of a natural process, its discrete simulation process can be executed in a computer, which efficiently leads to a solution with acceptable precision if researchers make a compromise between solution accuracy and time complexity. The simulation techniques can be sorted into Molecular Dynamics (MD), Monte Carlo (MC), and hybrid techniques of both of them [3]. The MD is used in my project.

1.1.2. The message passing interface

Before MPI arises, software developers needed to do inter-process communication using different library functions provided by hardware vendors, which was not convenient when

the hardware platform changed. In 1994, the first version of MPI was released. It provided a unified standard of message passing interface for different hardware vendors. Nowadays, two implementations of MPI - MPICH and Open MPI are widely used in high-performance-computing software development [4]. As a result, a wide range of hardware platforms can support a single MPI-based software without source code modification.

Implementing the simulation program by MPI can reduce the execution time effectively if the workload is distributed evenly in each process without considerable performance overheads. In order to combine the simulation application with the invasive computing better in the iRTSS platform, which will be introduced in Subsection 1.1.3, the invasive application sometimes needs to retreat allocated hardware resources and reallocate them for better resource management during the execution. Thus the iMPI library is needed in my project. The iMPI library in iRTSS includes some basic MPI operations and iMPI operations. These iMPI operations facilitate the adaptations of processes in order to adjust resources dynamically. In consequence, simulation applications in iRTSS can support elastic resource management well even with an enormous number of allocated processes and complex mechanisms of inter-process communication in iRTSS.

1.1.3. Invasive Run-Time Support System

iRTSS can work closely with a standard Unix-like host operating system. It can be scaled to a high extent, and it integrates methods, principles, and abstractions for the application-aware configuration, adaptation, and extension of invasive computing systems [1]. In parallel systems, invasive computing is intended to focus on the management of both the hardware and software, so as to provide the users with the capability of resource-aware programming. The resource-aware programming is based on the invasive programming. The invasive programming means that a program running in a parallel system is able to request and claim temporarily processor, memory, and communication resources in its computing neighborhood, to execute the program in the claimed resources, and to free the claimed resources after the execution. A typical process of invasive programming is divided into an invade step, an infect step, and a retreat step. In the invade step, the operating system is required to allocate resources. In the infect step, the program is executed in the resources allocated. In the retreat step, the allocated resource is freed and can be reallocated again [5].

An iRTSS system consists of the Operating System Abstraction Layer (OSAL), an agent system, a parallel operating system OctoPOS, and the Hardware Abstraction Layer (HAL). OSAL provides the application with the interfaces of iRTSS. The agent system is intended to assign the applications to hardware. OctoPOS constructs the invade, retreat, and infect abstractions of invasive computing, and maps operations of the run-time system onto different kinds of hardware according to configurations. With the support of the invasive computing abstractions, application developers can explicitly allocate and free computing units, memory block, and communication resources during executions. Because I need to provide a simulation application with adjustable initial resources, dynamic numbers of nodes, and other hardware for an efficient execution in a distributed system, OctoPOS can help me achieve this goal. HAL can provide OctoPOS with interfaces to access the hardware

[1] `https://invasic.informatik.uni-erlangen.de/en/tp_c1_PhIII.php`

platform [6].

The TCP interfaces implemented in iRTSS make real-time data transmission from the simulation application in OctoPOS to the client application running in the same host computer as that of the OctoPOS platform or another device possible. In my thesis, servers in OctoPOS can connect and communicate with clients outside OctoPOS after TCP connections are established. The client can visualize the dynamic simulation process over time using C++ visualization tools such as OpenGL and OpenCV.

1.1.4. A parallel molecular dynamics microapplication - miniMD

miniMD is a lightweight, scalable, parallel molecular dynamics simulation application. It is intended to run on supercomputers or other computer architectures and test their performance [2]. Users can simulate various cases of molecular dynamics processes by changing the size of the problem, the number of time steps to simulate the process, the cutoff distance between particles, the particle density, the force between particles, etc [7]. Based on the ideal extensibility of miniMD, I can add extra features. MPI is also supported in miniMD, which is helpful for me to add iMPI operations to it. The libraries available in OctoPOS can fulfill the requirements of the development and extension of miniMD. Therefore, it is reasonable to port miniMD to OctoPOS. By contrast, other more sophisticated simulation applications such as LAMMPS, which consists of over 200000 lines of code [3], typically include C++ libraries not supported in OctoPOS, therefore it is nearly impossible to reproduce them in OctoPOS.

In order to use the miniMD code in OctoPOS, I need to implement it in C language only with the support of library functions available in OctoPOS. By running miniMD in OctoPOS on the 64-bit architecture hardware, I can achieve the following goals. First, I verify whether the libraries from OctoPOS, such as the MPI library, the iMPI library and the TCP library, can perform correctly. Once I can run the adapted miniMD in OctoPOS correctly, a client application which can receive the simulation data from the miniMD application through TCP connections in normal UNIX operating system needs to be developed, in order to visualize the real-time motions of the particles and record the simulation data. After the functional verification, I can summarize the performance metrics under conditions of different simulation scenarios and numbers of processes.

1.2. Objectives

There are three objectives needed to be achieved:

1. The miniMD application based on iMPI needs to be implemented in OctoPOS, and its performance will be tested. miniMD needs to be reproduced by C language and the implementation should only depend on the libraries provided by OctoPOS. Different molecular dynamics simulation scenarios will be constructed. They should be simulated correctly and their performance metrics should be produced.

[2] `https://github.com/Mantevo/miniMD`
[3] `https://github.com/lammps/lammps`

2. A visualization application in normal UNIX operating system will be implemented. It can visualize the real-time trajectories of particles and the change of each process's simulation domain in the simulation box. At the meantime, it can record the data of particles and simulation domains per a certain number of time steps. It will also show important information of simulation such as: the number of particles, the size of the simulation box and the current simulation iteration index. Visualizing the real-time scenarios can help researchers track the simulation process during the execution. Recording the simulation data can help researchers analyze the simulation process after the execution.
3. The experiment to visualize target scenarios and the strong scaling experiment will be implemented. The default simulation case in miniMD and new scenarios will be simulated. The simulation result will be performed by the visualization application. In the strong scaling experiment, multiple executions based on dynamic scales of resources will be tested given a fixed problem. The strong scaling experiment of one problem can be finished in one time of execution with the help of iMPI.

2. Related Work

2.1. Invasive computing

Invasive computing is a new paradigm for designing and programming parallel systems. It involves a wide range of definitions: programming language, compiler, operating system, hardware, etc [5]. Based on adaptations on these components, resource-aware programs can be achieved in invasive computing systems, and high utilization of resources as well as efficiency of energy can be realized. Nowadays, the parallel system is widely applied, thus invasive computing can help developers to manage resources with more cores integrated into a single chip [1]. Figure 2.1 describes the parallelism of each level, which is common in current devices. For instance, the Fermi CUDA [8] architecture implemented in NVIDIA Graphics Processing Units (GPUs) supported by 512 thread processors led to a powerful computing capability of 6 GB Graphics Double Data Rate of version 5 (GDDR5) Random Access Memory (RAM) and 1 TFLOPS [5]. The scalable CUDA unified graphics and parallel architecture are both developed to facilitate high-performance computing and programmable graphics [9]. The parallel system can be programmed through APIs, which could lead to higher efficiency and flexibility if it is combined with invasive computing.

The research on invasive computing was already started in the Transregional Collaborative Research Center 89: Invasive Computing. It is being funded by the Deutsche Forschungsgemeinschaft. It is in its third stage from July 2018. Scientists from Technische Universität München, Karlsruher Institut für Technologie, and Friedrich-Alexander-Universität Erlangen-Nürnberg have joined in the research group. The research includes projects about the language and algorithm of invasive computing, system architecture, compiler, run-time system, and applications of the invasive computing system [2]. It will highly improve the efficiency of utilizing resources on various hardware platforms such as embedded system and high-performance distributed system.

[1] `https://invasic.informatik.uni-erlangen.de/en/index.php`
[2] `https://invasic.informatik.uni-erlangen.de/en/index.php`

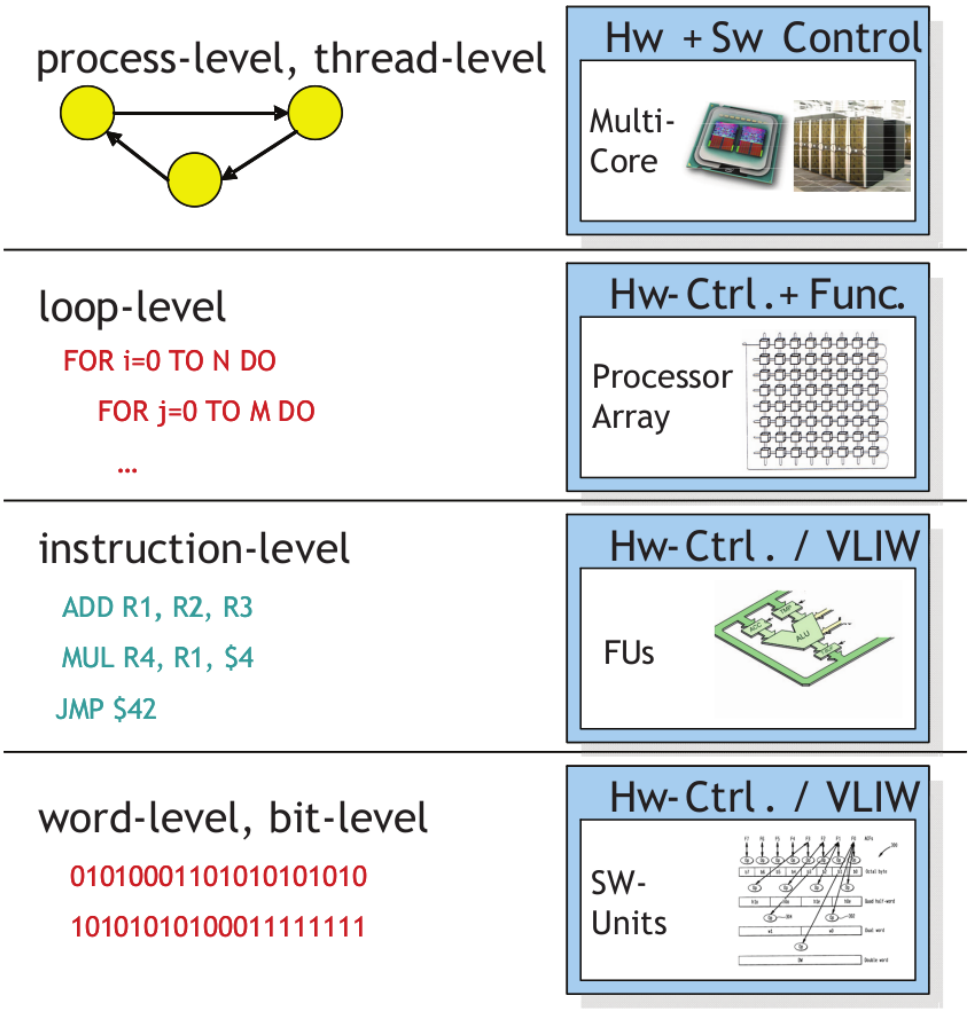


Figure 2.1.: Levels of parallelism.
Source: [10]

2.1.1. Basic theory

Parallel computing is being popular because of its powerful computing capability. It is widely applied to supercomputers, computer graphics, and single-chip devices. However, there exist problems needed to be solved before its application. First, it is not easy to map the program to thousands of cores properly. Second, the adaptive abilities to occupy and release run-time resources may be necessary for the elastic management of resources. Third, the scalability of programs, heat dissipation, reliability, and fault-tolerance of executing programs should also be provided by platforms. Last but not least, the interconnected structure between components should be able to reconstruct the component topology dynamically and efficiently during run-time period because of dynamics load changes and time-variant resource constraints [5].

In order to solve the problems above, invasive computing is introduced. It can help the executing programs to coordinate and manage the processing elements by linking the processing resource to the program [5]. It leads to invasive programming which is introduced in Subsection 1.1.3. Specifically, the following theory gives the basis to construct a typical invasive computing system.

Some notations about invasive computing should be described. No matter what the extended language or the system architecture is, the basic instructions of invasive computing are the `invade` instruction, the `infect` instruction, and the `retreat` instruction [10]. A basic invasive program is shown in Listing 2.1 [6]. A `claim` means a group of resources those can be accessed by applications. The resource can refer to processing elements, memory storage, or interconnect devices. For instance, a `claim` of processing elements in X10 constructs a partitioned global address space. Analogically, `claims` mean a group of constituents each of which can be a single `claim` or a group of `claims`. An invasive-let (`i-let`) means a piece of invasive computing program that can be executed in parallel. It is a fundamental unit of a program section with concurrent execution potential [5]. This simple program fragment means: the `invade` method returns the allocated resource `claim` according to `constraints`, the `infect` method executes the program `ilet` on `claim`, and the `retreat` method frees `claim`. The `invade` instruction declares the resource the current process needs and the invaded resource can not be invaded by any other process until one successful `retreat` instruction of the corresponding resource is completed. During the `infect` instruction, an `i-let` is copied and deployed in the assigned hardware. The `i-let` deployed in the hardware is called as the `i-let` incarnation. Once it is executed by the scheduler of the operating system, it is called as the `i-let` execution. A `team` refers to a set of `i-let` incarnations. The `infect` instruction only returns after all the programs are finished. The data of the `team` will not be deleted before a `retreat` instruction. The `retreat` instruction explicitly frees the resource that is invaded before. After the completion of `retreat`, an `i-let` can not be executed in the `claim` through the `invade` instruction anymore.

```
1  val claim = Claim.invade(constraints);
2  claim.infect(ilet);
3  claim.retreat();
```

Listing 2.1: A basic invasive program.

Problems at the architecture level need to be solved. For instance, Figure 2.2 shows how invasive computing works at tiles, each of which includes cores of Reduced Instruction Set Computer (RISC) processor, hardware accelerators, and local memory. One tile is connected with another tile by flexible network-on-a-chip (NoC) interconnects. In the invasion step, each Core i-let Controller (CIC) component in one tile transmits the information of hardware to the invasion process. Afterward, the configurations about CICs are decided, which show the infection of available cores. In the end, the distribution of the executing program is decided based on an overall optimization strategy. The CIC in each tile plays an important role in dynamically mapping the processing requests to hardware controlled by iRTSS and the operating system. This example can illustrate the potential process of invasive computing based on the loosely-coupled Multi-Processor Systems-on-a-Chip (MPSoC) platform. Examples of the invasive computing processes in Tightly-coupled Processor Arrays (TCPAs) and High Performance Computing (HPC) platforms can be found in [5].

The resource-aware programming idea should be considered when programmers design invasive algorithms and analyze their complexity. Besides the algorithmic problem, the problem to extend programming languages such as C++, X10 also needs to be solved, because the resource-aware language constructs are not common in normal programming languages [5].

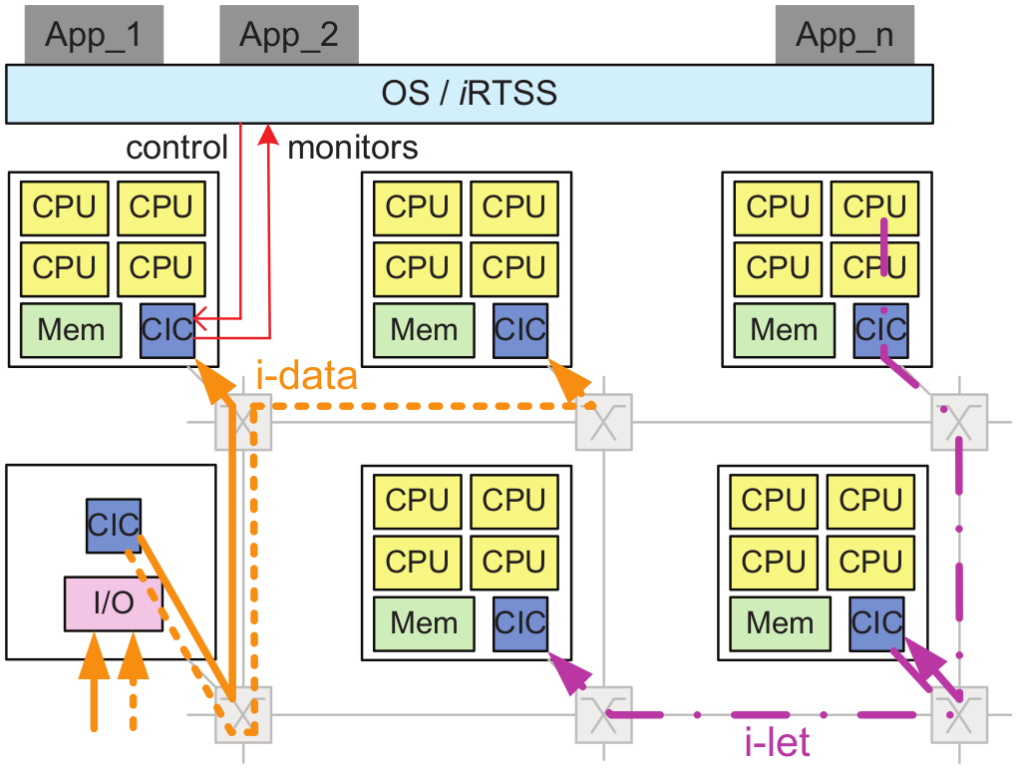


Figure 2.2.: Invasive computing in a loosely-coupled MPSoC architecture.
Source: [5]

2.1.2. Current developments

The project of invasive computing developed on the Transregional Collaborative Research Center 89 consists of different projects. Figure 2.3 shows the research fields of the project group A, B, C, and D. Each group focuses on an area intersected by specific project areas and hardware [1].

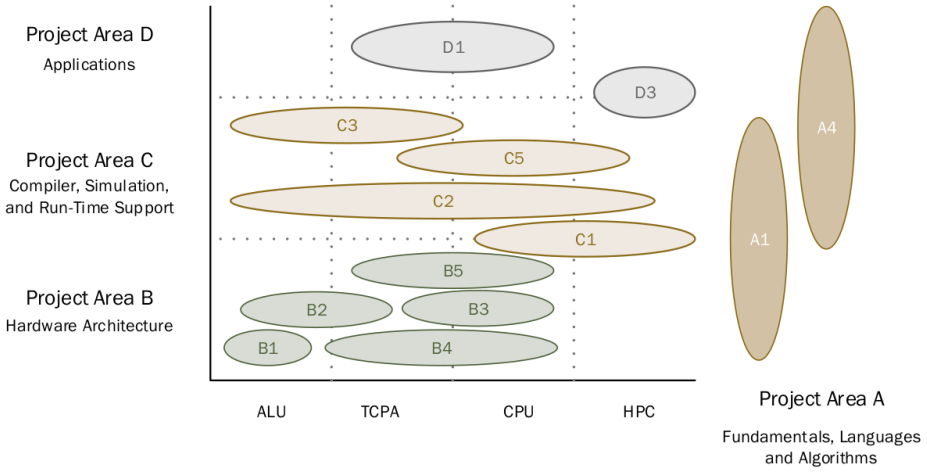


Figure 2.3.: Project groups and application fields.
Source: [1]

The main areas of project A are the basic theory of invasive computing, adding the abilities to schedule the application and to distribute the load throughout the hardware resources, and the techniques to analyze the invasive computing applications and the run-time systems. The sub project A1 developed the resource-aware programming X10: InvadeX10. Developers can develop programs by it in both the distributed system and shared memory system [1].

Project B focuses on the hardware mechanisms for adaptive invasive microarchitectures, the efficient implementation of invasive computing on TCPAs, the techniques to save energy and power on platforms of invasive computing, the mechanisms to monitor the properties of invasive hardware, and the design of the NoC equipment [1].

Project C consists of sub projects C1, C2, C3, and C5. The sub project C1 built an operating system OctoPOS, which can support the resource-aware programming in multi-core systems. In my project, the molecular dynamics simulation application can be developed in OctoPOS. Therefore, OctoPOS will be discussed in details. The sub project C2 focuses on the exploration of simulative design space, which can help developers improve the invasive hardware and resource-aware software. The sub project C3 concentrates on the techniques of compilation, code generation, program transformation, and optimization for both the procedural code and the regularly-structured as well as task-level code [3]. The sub project C5 focuses on the security problems of invasive computing systems.

Project D focuses on the applications of invasive computing. For instance, the sub project D1 applies invasive computing to robotics. The sub project D3 focuses on the following

[3] `https://invasic.informatik.uni-erlangen.de/en/tp_c3_PhIII.php`

three fields: numerical methods for resource-aware computing and their implementations, the advantages of resource-aware computing in HPC systems, and the combination of resource-aware computing with the hardware and programming models of current HPC systems [1].

2.1.3. A parallel operating system - OctoPOS

OctoPOS is a parallel operating system supporting invasive computing. As Figure 2.4 shows, OctoPOS acts as an intermediate layer between the run-time system as well as the agent system and the hardware architectures [4]. It supports operating system functionalities such as the Remote Procedure Call (RPC) mechanism for communication through the i-NoC devices and managing i-lets through CIC components [6]. The run-time system supports the three fundamental commands of invasive computing, and the commands can be mapped by OctoPOS to the underlying hardware. OctoPOS can support the management of processors, automatically hide the latency, reduce the contention on the resources, and maintain the locality [10], [11], [12], [13]. The agent system is responsible for scheduling the invasive computing operations [14], [15]. It can distribute applications on the hardware based on the exchange of the local information and keep the overhead of management stable with an increasing number of cores [6]. Two important concepts about OctoPOS: the programming model and hardware model are introduced as follow.

The programming model of OctoPOS is shown in Figure 2.5. This model illustrates the transitions between the three fundamental primitives of invasive computing. The `invade` primitive claims the allocated resources for the current process. Then, the function `assort` constructs a team, according to the invaded resources. The team consists of some related threads those execute the program of the process. After the team is made, the `infect` primitive can copy the code to the invaded resource and start the execution. During the execution, the `invade` and `retreat` primitives are used to adapt the allocated resources. More details are given in Subsection 2.1.1.

The development of a wide range of hardware with parallelisms provides more choices for the construction of the hardware model with invasive computing. As Figure 2.1 shows, the traditional multi-core CPU can give parallelism on the level of process and thread. My application supported by OctoPOS will be executed based on the iMPI model, which needs the high-level parallelism of processes, thus it is enough to deploy it on a normal system with multi-core CPUs.

OctoPOS can run in a Linux system. A project in OctoPOS is eventually built as a binary file which can be executed in normal UNIX-like operating system or bare metal. More details about OctoPOS application setup are in Section 2.3.

[4] `https://invasic.informatik.uni-erlangen.de/en/tp_c1_PhIII.php`

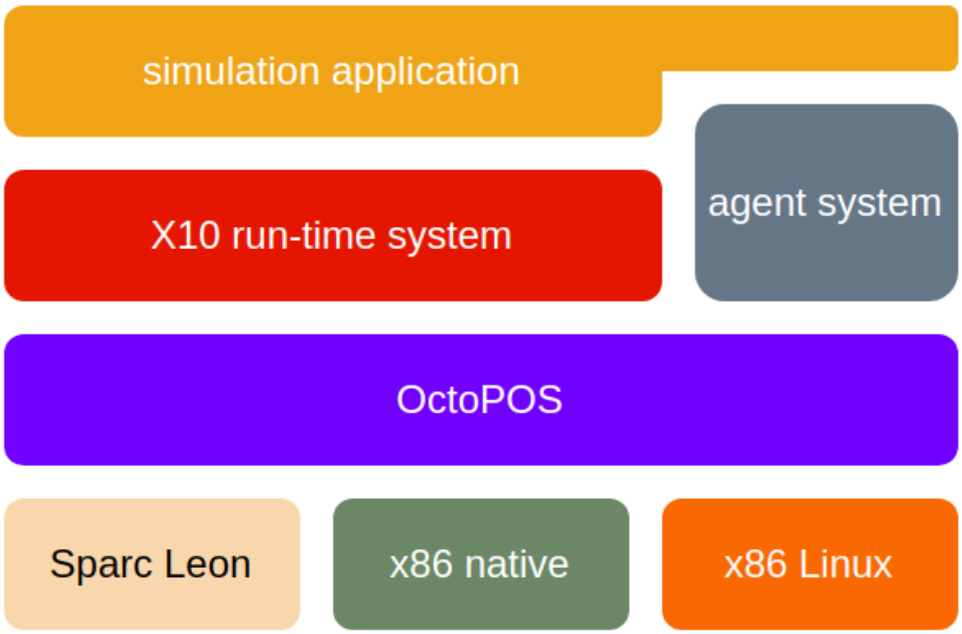


Figure 2.4.: Interaction between OctoPOS and other system components.

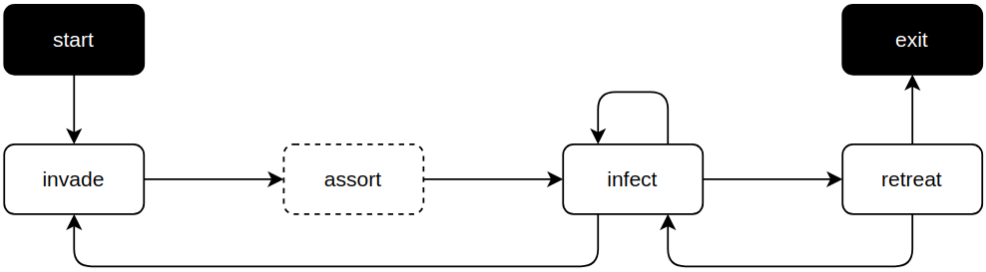


Figure 2.5.: Programming model of OctoPOS.
Source: [10]

2.2. Libraries in OctoPOS

MPI plays an important role in communication between processes in distributed systems. It was widely used in networks of workstations and parallel computers [16]. It provides developers with a unified standard to pass messages between processes. Therefore, developers can ignore the underlying mechanisms of different device vendors and develop applications conveniently. Nowadays, two implementations of MPI: Open MPI [17], [18], [19], [20], [21], [22], and MPICH [23], [24], [25], [26], [27], are the mainstream [1]. In OctoPOS, all the iMPI operations and a part of MPI operations are realized for the use of parallel programming on it. MPI and iMPI in OctoPOS are introduced in Subsection 2.2.1 and Subsection 2.2.2. Another important library in OctoPOS is the network data passing library introduced in Subsection 2.2.3.

2.2.1. MPI in OctoPOS

Before more specific knowledge about MPI is introduced, basic definitions of MPI need to be discussed. A group in MPI refers to an ordered set of processes, and each process has a unique rank number in this group [1]. The communication between processes in this group happens in a communicator. By contrast, two processes in two different communicators can not reach each other. After a MPI program is initialized, a communicator `MPI_COMM_WORLD` is established. In OctoPOS, the normal initialization interface is `MPI_Init`, and it should be called at the beginning of the MPI program. Afterward, the execution environment is finished and then the subsequent MPI operations are accessible. At the end of the MPI program, the `MPI_Finalize` operation needs to be called to clean the MPI environment. Programmers must make sure all the communication work is finished and each process has exited or is only doing its local work before the `MPI_Finalize` call [5].

In the MPI 4.0 version, the features those can be set up by programmers are communication mode, customized data type, communication topology, etc. The communication modes include the one-sided communication mode, the point-to-point communication mode, the partitioned point-to-point communication mode, and the collective communication mode. In OctoPOS, the point-to-point communication and collective communication cases are widely used.

The point-to-point communication mode means that the message of one process can be transmitted to another process directly. In OctoPOS, the general point-to-point functions are `MPI_Send` and `MPI_Recv`. Their non-blocking versions `MPI_Isend` and `MPI_Irecv` are also implemented. For instance, Listing 2.2 shows the `MPI_Send` interface. The parameter *buf* refers to the starting address of the data to be sent. The *count* means the number of *datatype* elements. *datatype* can be any kind of MPI data type. *dest* is the rank number of the process of destination. *tag* is used to match the sender and receiver. However, it can also be `MPI_ANY_TAG`, which is a wildcard and can match one `MPI_Recv` operation with any tag. The main difference between the blocking version and the non-blocking version is that the data buffer pointed by the parameter *buf* can be used immediately after the return of one blocking call, but the data buffer of one non-blocking call can only be used after the termination of a communication process. There is an additional parameter of type `MPI_Request` in the non-blocking version. This new parameter should be tested after the return of a non-blocking

[5] `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`

call. Only after the request succeeds to tell whether this communication call is successful or not through the interface `MPI_Wait` or `MPI_Test`, the data buffer can be used again.

The collective communication interfaces in OctoPOS include `MPI_Reduce`, `MPI_Allreduce`, `MPI_Bcast`, `MPI_Alltoall`, `MPI_Alltoallv`, etc. A collective communication call involves all the processes in one communicator, and all the processes wait synchronously for the return of this call. `MPI_Alltoall` means each process sends an equal number of bytes to all the other processes, and each process receives the data from all the other processes. `MPI_Alltoallv` is a more flexible version of `MPI_Alltoall`. Each process can parameterize the number and location of data sent and received. `MPI_Bcast` means a root process broadcasts its data to all the other processes. The `MPI_Reduce` operation does a mathematical operation in one data from all the processes in one communicator and copies the result to the root process. The `MPI_Allreduce` interface differs from the `MPI_Reduce` one in that it copies the mathematical result to all the processes. Figure 2.6 shows the process of a `MPI_Allreduce` operation in a three-process communicator.

```
1  int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest,
       int tag, MPI_Comm comm);
```

Listing 2.2: `MPI_Send` C interface.

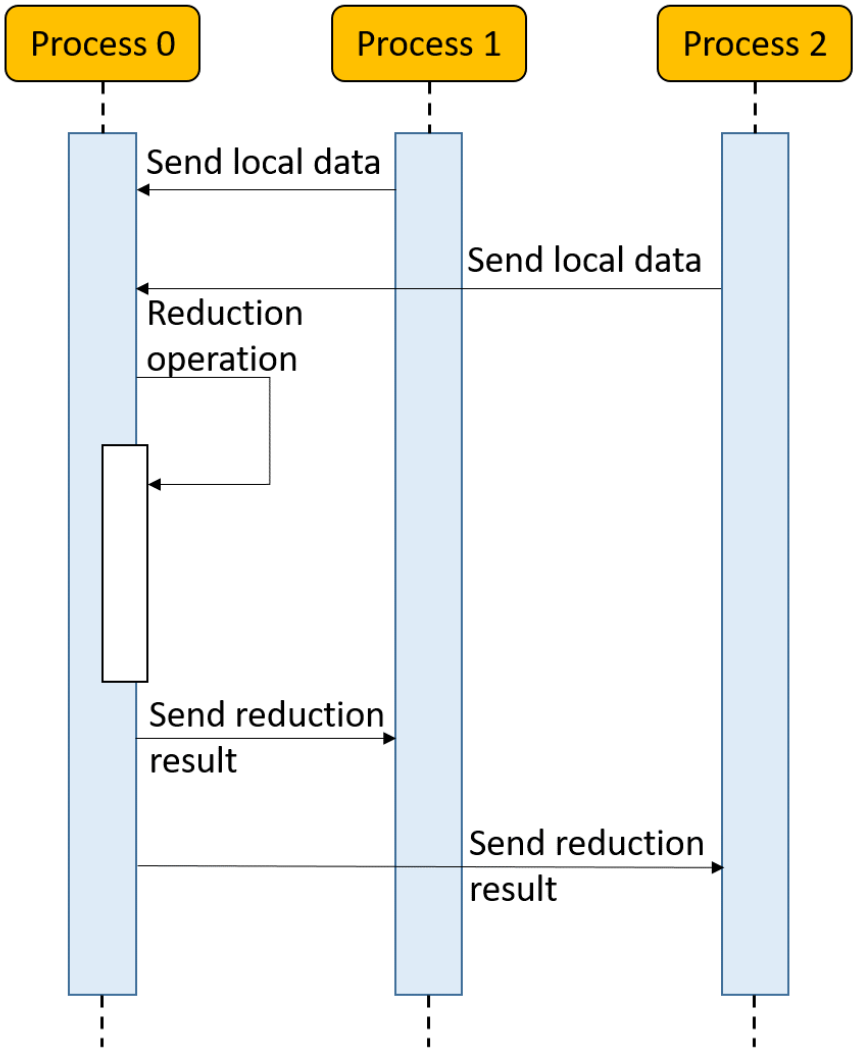


Figure 2.6.: Example of the process of `MPI_Allreduce`.
Source: [1]

A part of the MPI routines was implemented in OctoPOS, and others can not be accessed in OctoPOS. For instance, the cartesian operations such as the `MPI_Cart_create`, `MPI_Cart_shift` functions are not available in the "mpi" library. They operate with the cartesian topology information and help find the local location of each process in its process neighborhood. In such a case, a new C function in the MPI application of OctoPOS to play a similar role is needed.

2.2.2. iMPI in OctoPOS

The iMPI interfaces in OctoPOS refer to the basic four operations which are frequently used: `MPI_Init_adapt`, `MPI_Probe_adapt`, `MPI_Comm_adapt_begin`, and `MPI_Comm_adapt_commit`. These interfaces solve the problem of dynamic adaptation in the traditional MPI, and they can support the resource management during the iMPI execution to some extent. The traditional MPI interfaces to scale the program are `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`. They can spawn new processes and deploy the original program on them. A parent process and a children process are in two different communicators. They communicate with each other through an intercommunicator. However, the spawning is based on some fixed resources. It is controlled only by the application rather than the resource manager [1]. The resource manager is intended to manage the resources in a cluster [6], in order to monitor the real-time status of resources. Thus, the traditional interfaces can not realize resource-aware programming and are not proper for my project.

The `MPI_Init_adapt` operation shown in Listing 2.3 can replace the `MPI_Init` operation in a traditional MPI program. It initializes the adaptive execution environment. Besides the command line arguments, the additional parameter `local_status` points to the final status of the process. The status can be the new status or the joining status. The new status `MPI_ADAPT_STATUS_NEW` means the calling process is initialized by the normal executing command. For instance, the "mpirun" command is used to specify the parameters of execution and run a MPI program. The joining status `MPI_ADAPT_STATUS_JOINING` means the calling process is initialized by the resource manager [1].

The `MPI_Probe_adapt` operation tells whether there exists an adaptation in the resource manager and helps the current process to decide if it is time to start an adaptation window. The adaptation is launched by the resource manager, but in OctoPOS it is not the case. `MPI_Resize_request` is used to specify the new number of processes in the communicator and launch the adaptation. Listing 2.4 shows the `MPI_Probe_adapt` routine. The possible values of `pending_adaptation` are `MPI_ADAPT_TRUE` and `MPI_ADPAT_FALSE`, and its value determines if there is an adaptation happening. The `local_status` can be `MPI_ADAPT_STATUS_STAYING`, `MPI_ADAPT_STATUS_LEAVING`, or `MPI_ADAPT_STATUS_JOINING`. A staying status points out that the calling process will stay in the current group. By contrast, a leaving status means the calling process will be removed after the adaptation [1].

If there exists an adaptation, each process can start a `MPI_Comm_adapt_begin` command. Listing 2.5 shows the details of this routine. The parameter `intercomm` refers to a temporal communicator during the adaptation. The parameter `new_comm_world` means the new communicator after the adaptation. Each process with status joining or staying can access each

[6] `http://www.schedmd.com`

other in the `new_comm_world` communicator. However, a process with status leaving can not reach the `new_comm_world` communicator, but it can be reached by parent processes in the `MPI_COMM_WORLD` communicator or by children processes in the `intercomm` communicator. The `staying_count` refers to the number of processes those will stay in the current group. The `leaving_count` refers to the number of processes those will leave the current group. The `joining_count` refers to the number of new processes those will join in the current group [1]. When this routine can be called is decided by programmers, thus it improves the flexibility of programming.

After the `MPI_Comm_adapt_begin` command and other operations during the adaptation window are completed, the `MPI_Comm_adapt_commit` command can be called. It has no parameter. It will replace the old `MPI_COMM_WORLD` communicator with a `new_comm_world` communicator. The adaptations made on the communicator can be adding joining processes to it or removing leaving processes from it. Furthermore, the resource manager is notified that the current adaptation is finished, thus a new adaptation can be launched [1].

With these four iMPI routines implemented in OctoPOS, an invasive program is easier to be developed at the application level. The advantages of these routines are summarized as follow. First, dynamic resource management is controllable at the application level through these routines. Second, the probing operation makes it more flexible to decide the opportunity to launch an adaptation. Third, operations not relevant to adaptations can be started during the adaptation window, in order to improve the executing efficiency. For example, Listing 2.6 shows an iMPI program in OctoPOS. The current number of the processes `npb_nprocs` and the largest possible number of processes `max_npb_nprocs` are defined at the global variable area of the main program. This example inserts additional processes per iteration during a `for` loop of adaptations. Each process gets its status at the `MPI_Init_adapt` call. A parent process should get a new status because it is not launched by the resource manager. A new children process should get a joining status, and receive the current iteration index of loop through a `MPI_Recv` operation because it will join the `for` loop as a new member of the group after the current adaptation. A parent process that already stayed in the current group needs to probe whether there is an adaptation happening through a `MPI_Probe_adapt` call. A `MPI_Resize_request` operation is called to launch an adaptation which will generate a new group with new_size processes. Each process can start an adaptation once the probing call returns a true status. During the adaptation window, the current iteration index is sent to children processes through the `MPI_Send` operation in the process with rank 0. These operations are called during the adaptation window, in order to hide the latency of adapting the group.

```
1  int MPI_Init_adapt(int* argc, char*** argv, int* local_status);
```

Listing 2.3: MPI_INIT_ADAPT C interface.

```
1  int MPI_Probe_adapt(int* pending_adaptation, int* local_status, MPI_Info*
       info);
```

Listing 2.4: MPI_PROBE_ADAPT C interface.

```
1  int MPI_Comm_adapt_begin(MPI_Comm* intercomm, MPI_Comm* new_comm_world, int*
       staying_count, int* leaving_count, int* joining_count);
```

Listing 2.5: MPI_COMM_ADAPT_BEGIN C interface.

```
1  #include "octopos.h"
2  #include "mpi.h"
3
4  const int npb_nprocs = 4;
5  const int max_npb_nprocs = 255;
6  int num_adaptations = 5;
7  int main(int argc, char **argv) {
8    MPI_Info info;
9    MPI_Comm intercomm, new_comm_world;
10   int rank, old_size, new_size, local_status, tmp_status, pending_adapt,
11   staying_count, leaving_count, joining_count;
12   MPI_Init_adapt(argc, argv, &local_status);
13
14   int cur_iter = 0;
15   if(local_status == MPI_ADAPT_STATUS_JOINING) {
16     MPI_Comm_adapt_begin(&intercomm, &new_comm_world, &staying_count,
17     &leaving_count, &joining_count);
18     MPI_Status status;
19     MPI_Recv(&cur_iter, 1, MPI_INT, 0, MPI_ANY_TAG, intercomm, &status);
20     MPI_Comm_adapt_commit();
21   }
22
23   for(int iter = cur_iter; iter < num_adaptations; iter++) {
24     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
25     MPI_Comm_size(MPI_COMM_WORLD, &old_size);
26     if(iter != num_adaptations - 1) {
27       new_size = old_size + 2;
28       MPI_Resize_request(MPI_COMM_WORLD, new_size);
29       MPI_Probe_adapt(&pending_adapt, &tmp_status, &info);
30       while(pending_adapt != MPI_ADAPT_TRUE) {
31         MPI_Probe_adapt(&pending_adapt, &tmp_status, &info);
32       }
33       MPI_Comm_adapt_begin(&intercomm, &new_comm_world, &staying_count,
34       &leaving_count, &joining_count);
35       if(rank == 0) {
36         for(int j = old_size; j < new_size; j++) {
37           int tmp = iter + 1;
38           MPI_Send(&tmp, 1, MPI_INT, j, 0, intercomm);
39         }
40       }
41       MPI_Comm_adapt_commit();
42     }
43   }
44   MPI_Finalize();
45   return 0;
46 }
```

Listing 2.6: The code of an iMPI program example.

2.2.3. TCP in OctoPOS

TCP transmits data with high reliability and low error rate [28]. A critical feature of TCP is its three-way handshake connection. During the process of a connection, the flags `SYN`, `ACK`, and `SYN-ACK` are used. At the first step, the client endpoint sets an initial sequence number, and packs the data packet with this number and a `SYN` flag. After receiving this packet, the server endpoint not only inserts the `SYN` and `ACK` flags into the responding data packet, but also packs the packet with the acknowledgement number which is equal to the sequence number plus one. Finally, the client endpoint sends a `ACK` flag with a responding acknowledgement number to the server endpoint, and then the connection is established. This three-way handshake process makes sure that the client endpoint can match the server endpoint in case of the data missing and out-of-order segments [29]. Figure 2.7 shows the whole process of three-way handshake.

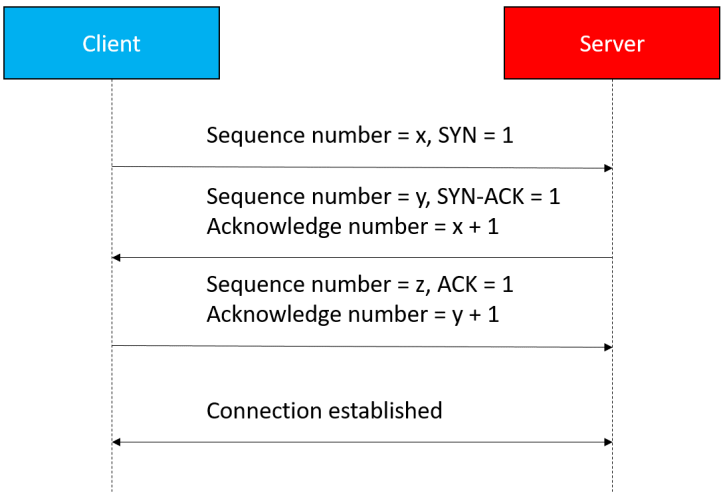


Figure 2.7.: TCP three-way handshake process.

The TCP operations in OctoPOS are implemented in the "octo_tcp" library. A general process in the server consists of creating the socket, binding the socket to an IP address and a port number, listening to the socket, and accepting the connection from the client. The setup of the client is connecting the server according to the IP address and the port number. The operation of creating a socket creates a socket of the type `octo_tcp_socket_t` when the function call returns a status of success. Functions to bind a socket to an ipv4 or ipv6 address and a port number are supported. If the ipv4 address can not be confirmed, an argument of "0.0.0.0" can be used to bind the socket to any address available on the current system. Similarly, "[::0]" can match any accessible ipv6 address. The listening operation keeps listening to possible clients until it is finished. This operation can specify the maximum number of connections. The accepting operation is intended to accept a number of connections based on the socket created previously. An operation to abort the accepting process is also provided. Once the connection is established, the server and client

can use the `octo_tcp_send` and `octo_tcp_recv` operations to send and receive information. Table 2.1 shows common TCP synchronous operations. The full set of the TCP operations in OctoPOS can be viewed in the directory "irtss/src/os/krn/cface/header". The "irtss" refers to the directory of iRTSS. The common return values of these operations can be `ESUCCESS`, `ENOMEM`, or `EINVAL`. These values mean that the call is successful, that not enough memory is available, or that some parameters are invalid. The `octo_tcp_sync_shutdown` operation can close the data transmission channels in one socket. The parameter *modes* is a bitmask that can specify the channels to close.

One TCP operation may have an asynchronous version. An asynchronous operation is an operation with the same aim as that of the synchronous operation and a following operation waiting for the completion of the previous one. For instance, Table 2.2 describes the interface of asynchronously creating a new socket and the interface of waiting for its completion. The parameter *handle* stores an asynchronous handle which is checked in the `octo_tcp_create_wait` call. The waiting operation will be blocked until the `octo_tcp_create` call is finished.

Description	Interface
Creating a new socket	enum Errors __must_check octo_tcp_sync_create (octo_tcp_socket_t *socket);
Binding a socket to ipv4:port	enum Errors __must_check octo_tcp_sync_bind4 (octo_tcp_socket_t socket, const octo_ip4_t *ipv4, uint16_t port);
Binding a socket to ipv6:port	enum Errors __must_check octo_tcp_sync_bind6 (octo_tcp_socket_t socket, const octo_ip6_t *ipv6, uint16_t port);
Listening on a socket	enum Errors __must_check octo_tcp_sync_listen (octo_tcp_socket_t socket, uint16_t backlog);
Accepting a connection	enum Errors __must_check octo_tcp_sync_accept (octo_tcp_socket_t socket, octo_tcp_socket_t *accepted);
Connecting a socket to ipv4:port	enum Errors __must_check octo_tcp_sync_connect4 (octo_tcp_socket_t socket, const octo_ip4_t *ip, uint16_t port);
Connecting a socket to ipv6:port	enum Errors __must_check octo_tcp_sync_connect6 (octo_tcp_socket_t socket, const octo_ip6_t *ip, uint16_t port);
Writting data to a socket	enum Errors __must_check octo_tcp_sync_send (octo_tcp_socket_t socket, const void *src, uintptr_t *size);
Reading data from a socket	static inline enum Errors __must_check octo_tcp_sync_recv (octo_tcp_socket_t socket, void *dst, uintptr_t *size);
Shutting down data channels	enum Errors __must_check octo_tcp_sync_shutdown (octo_tcp_socket_t socket, unsigned modes);
Closing a socket	enum Errors __must_check octo_tcp_sync_close (octo_tcp_socket_t socket);

Table 2.1.: Synchronous TCP interfaces in OctoPOS.

Operation to create a socket	**Waiting operation**
enum Errors __must_check octo_tcp_create (octo_tcp_create_t *handle);	enum Errors __must_check octo_tcp_create_wait (octo_tcp_create_t handle, octo_tcp_socket_t *socket);

Table 2.2.: Asynchronous creation of a new socket and its waiting operation for the completion.

2.3. OctoPOS applications

Based on OctoPOS, invasive applications can be developed. My project is based on iMPI and developed by C. The following sections give the setup of an OctoPOS application and two examples of the iMPI application in OctoPOS.

2.3.1. The setup of an OctoPOS application

The setup of an OctoPOS application includes two parts: building an iRTSS release and building the application in OctoPOS.

After the installation of iRTSS, an iRTSS release needs to be built and used by OctoPOS. The fundamental step is to generate the platform configuration of resources. The platform configuration is defined in a Perl module file. The Perl module file includes the name of architecture, the name of the release variant, the number of tiles, the features of devices such as the shared memory and interconnect network device, and the specific features of the tile such as the number of processes and memory device in the tile. It is critical that the size of the configured hardware resources must be big enough so as to fulfill the requirements of the OctoPOS application. iRTSS supports various executing architectures such as x64native and x86guest. For instance, an OctoPOS application running as a guest on the top of a Linux operating system uses the x86guest variant. In my project, the OctoPOS application uses the x64native variant and runs in the 64-bit architecture bare metal.

After the iRTSS configuration, an iRTSS release can be built by a Perl script based on the configuration file. A link in OctoPOS needs to point to the built release, therefore iRTSS can be used by applications in OctoPOS. In an OctoPOS application, the "Makefile" file defines the project name, the OctoPOS release version, architecture, and the release variant. After building the source code with the "Makefile" file, the execution file can be generated.

2.3.2. An iMPI application to compute the heat matrix

This is an iMPI application to compute the heat matrix which can show an example of numerical simulation in OctoPOS. The first part is the data initialization. The root process initializes the global data and constructs the grid population according to the heat sources at the beginning. Then the root process determines the local data grid size for each process, and the local data is broadcast through `MPI_Bcast`. After that, the global matrix is split and distributed to other processes through `MPI_Send`. The main loop of computing starts after the data preparation. In each iteration, all the processes calculate the Jacobi matrix collectively and synchronize the overlapping data between neighbor processes. Whether to expand the group is according to if there are enough resources left which is decided by the comparison between the new potential number of processes and `max_npb_nprocs`. If the remaining processes are enough to support the adaptation, a `MPI_Resize_request` operation is started. After that, all the data is collected and updated in the root process. If there is an adaptation in the current iteration, an adaptation window is started. After the adaptation, the local data in each process gets initialized and the global data is distributed from the root process to other processes in the new group. Then a new iteration is ready.

2.3.3. The invasive swe-x10

A software package to solve the Shallow Water Equation (SWE) problem shown by Equation 2.1 is a great example of invasive simulation in OctoPOS. SWE refers to a nonlinear system of conservation laws for momentum and depth. It plays an important role in the research of the coastal inundation and the propagation of tsunamis [30]. As Equation 2.1 shows, u and v note the velocities in two dimensions, and h refers to the height in the remaining dimension. The right side $S(t, x, y)$ is a source term that can be modeled as other effects such as the friction of the ocean floor [31]. The notation $[g]_x$ means the partial derivative with respect to the independent variable x.

$$[h, hu, hv]_t + [hu, hu^2 + \frac{1}{2}gh^2, huv]_x + [hv, huv, hv^2 + \frac{1}{2}gh^2]_y = S(t, x, y) \tag{2.1}$$

The invasive X10-based swe application performs well in both the shared memory system and the distributed system, compared with previous C++ applications based on a hybrid model of MPI and OpenMP [31]. A critical algorithm of the invasive swe-x10 is the actor model. As Figure 2.8 illustrates, one actor noted by a node in the figure represents an equal-size local domain of the unknowns needed to be solved. Each actor communicates directly with its direct neighbors by channels. A channel containing a data buffer connects the output port of one actor with the input port of another port. Data passing the channel is First In First Out (FIFO). During one iteration of the simulation, an actor gets the result of its neighbor at the previous time step, and the result in the current time step is updated according to the result received. Therefore, the unknowns can be updated per iteration.

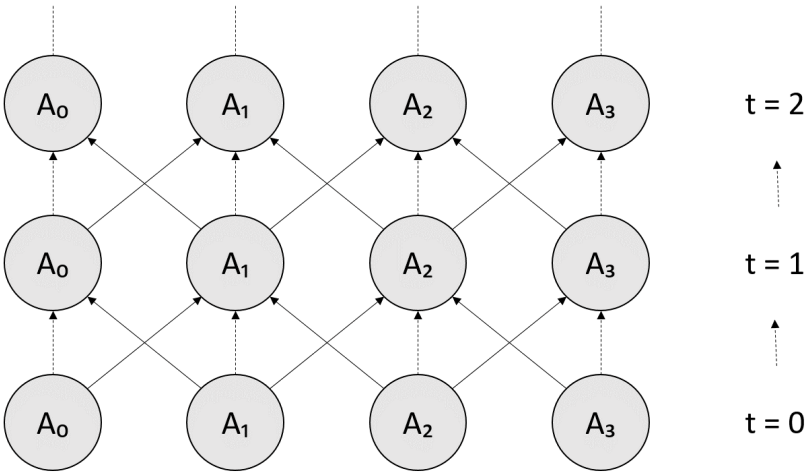


Figure 2.8.: The communication between actors and their neighbors.
Source: [31]

Another important feature of the invasive swe-x10 is the "ActorX10" library [32]. It is based on the Asynchronously Partitioned Global Address Space (APGAS) programming model of X10, which allows a running program to dynamically create and deploy threads on

a cluster of nodes. In order to lower the difficulty of programming with APGAS, the actor model discussed previously is combined with PGAS, and the resulting model is implemented in X10 to develop the "ActorX10" library.

The communication part of the invasive swe-x10 is an example of transmitting data in OctoPOS. The Input/Output (IO) tile, which includes IO and ethernet devices, can transmit data to a visualization application: a normal C++-based project in a Linux system that can receive network data and configure the visualization of simulation with colorization, movement, and scaling. The real-time scenario can be then presented during the simulation process. This application gives an example of visualizing molecular dynamics simulation based on OctoPOS and UNIX-like systems.

2.4. Molecular dynamics simulation

Besides traditional experiments to simulate scientific scenarios, computer simulation is a new powerful tool to help do research on microscopic interactions between particles and their structures. Basic theory about molecular dynamics simulation is introduced as follows.

2.4.1. Basic theory

The first important theory about MD is the interaction between particles. The total force in one particle can be calculated by Equation 2.2.

$$\boldsymbol{f}_i = -\frac{\partial}{\partial \boldsymbol{r}_i} V \tag{2.2}$$

The numerator $V(\boldsymbol{r}^N)$ where $\boldsymbol{r}^N = (\boldsymbol{r}_1, \boldsymbol{r}_2, ...\boldsymbol{r}_N)$ refers to the potential between particles [2], and the Lennard-Jones (LJ) variant of pair-potential is used in my project.

$$v^{\boldsymbol{LJ}}(r) = 4 \cdot \epsilon[\Big(\frac{\sigma}{r}\Big)^{12} - \Big(\frac{\sigma}{r}\Big)^{6}] \tag{2.3}$$

Equation 2.3 describes the computing of the LJ potential. The parameter σ denotes the zero crossing of the potential [33]. The parameter ϵ denotes the depth of the potential, thus larger ϵ means stronger bond between particles. The parameter r denotes the distance between particles. The diagram of the LJ potential is described in Figure 2.9. The potential and force decay rapidly, which leads to the idea to neglect the minor effects of those particles far from the current particle.

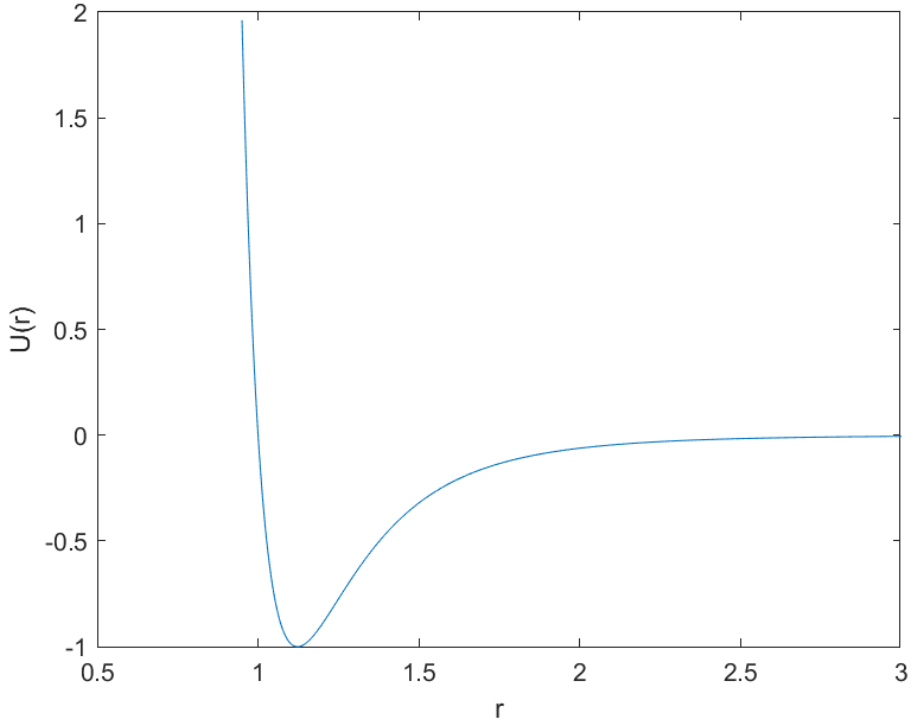


Figure 2.9.: Lennard-Jones potential with σ=1 and ϵ=1.
Source: [2]

In order to decrease the time complexity of computing the particle-to-particle force from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$, the LJ potential is approximated to zero if the distance between particles exceeds a radius cutoff r_{cut}. The r_{cut} is typically chosen as $2.5 \cdot \sigma$. Neglecting the LJ forces of the particles from a distance over r_{cut} leads to an error in computing the total LJ force in the current particle and changes the total energy of the current simulation domain isolated mechanically and thermically [2]. Another error comes from the numerical scheme, which will be introduced in the next segment. Combining Equation 2.2 and Equation 2.3, Equation 2.4 showing the calculation of the total LJ force on one particle i is got.

$$\boldsymbol{F}_i \approx 24 \cdot \epsilon \sum_{\substack{j=1, j \neq i \\ 0 < r_{ij} \leq r_{cut}}}^{N} \frac{1}{r_{ij}^2} \cdot \left(\frac{\sigma}{r_{ij}}\right)^6 \cdot \left(1 - 2 \cdot \left(\frac{\sigma}{r_{ij}}\right)^6\right) \boldsymbol{r}_{ij} \tag{2.4}$$

In my project, only the short-range force such as the LJ force is calculated. Therefore, the cell list or the neighbor list can be used to compute the forces between particles [34]. In miniMD, the neighbor list is used, and it stores less particles those should be checked for the force computing than the cell list. The simulation box is evenly split into local domains. Only particles in the domain whose distance from the local domain does not exceed the neighbor cutoff n_{cut} are considered into the force computing between particles. The parameter n_{cut} is bigger than r_{cut} considering the motions of particles in the domain between two neighboring operations.

In miniMD, the Embedded Atom Method (EAM) potential [35], [36], [37] is also included. It is used to compute the interactions between atoms of metals or their alloys. For example, the EAM potential has been applied to the fields of surface reactions, phase transitions of solids as well as other materials, and crack information [38]. The fundamental idea of EAM is the electron density induced by energy potentials [2]. For a single atom i, its energy is decided by the potential which is contributed to by all the surrounding atoms in the position x_i. The atom i can be regarded as embedded in a bulk of electron density. This embedding energy V_i^{emb} is described in Equation 2.5 [2].

$$\begin{aligned} V_i^{emb} &= \mathcal{F}_i(\rho_i^{host}) \\ \rho_i^{host} &= \sum_{j=1,j\neq i}^{N} \rho_j^{atom}(\parallel r_{ij} \parallel) \end{aligned} \tag{2.5}$$

ρ_i^{host} refers to the electron density induced by all the other atoms in the position x_i. The parameter r_{ij} denotes the distance between the atoms in x_i and x_j. Besides Equation 2.5, the pair potential described by Equation 2.6 should be computed. ϕ_{ij} is a function that only depends on the types of atom i and j [2]. The combination of the embedded energy and pair potential leads to the total potential in Equation 2.7. The function Z_i refers to the "effective charges" [2]. Z_i is constructed properly so as to feature a decaying function value as r_{ij} grows. The typical method to model Z_i and $\mathcal{F}_i$ is to use cubic splines to describe them. The coefficients of the cubic splines are computed through a weighted least square approach, thus common physical properties can be modeled accurately [2]. ρ_j^{atom} is computed by the Hartree-Fock approximation method [39]. In my project, the EAM mode is removed because the LJ interaction is enough to simulate desired scenarios.

$$V_i^{pair} = \frac{1}{2} \sum_{j=1,j\neq i}^{N} \phi_{ij}(\parallel r_{ij} \parallel) \tag{2.6}$$

$$V = \sum_{i=1}^{N} \mathcal{F}_i(\sum_{j=1,j\neq i}^{N} \rho_j^{atom}(\parallel r_{ij} \parallel)) + \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1,j\neq i}^{N} \frac{Z_i(\parallel r_{ij} \parallel) Z_j(\parallel r_{ij} \parallel)}{\parallel r_{ij} \parallel} \tag{2.7}$$

In order to get discrete mathematical results, numerical methods need to be applied to approximate the locations and velocities of particles. A simple example of the numerical method is the explicit Euler method. Equation 2.8 shows its scheme.

$$\begin{aligned} \vec{v}(t + \triangle t) &\doteq \vec{v}(t) + \triangle t \vec{a}(t) \\ \vec{r}(t + \triangle t) &\doteq \vec{r}(t) + \triangle t \vec{v}(t) \end{aligned} \tag{2.8}$$

The first step is to compute the velocity $\vec{v}(t + \triangle t)$ in the next time step $t + \triangle t$ with the current acceleration $\vec{a}(t)$. The second step is to compute the location $\vec{r}(t + \triangle t)$ in the next time step with the current velocity $\vec{v}(t)$. But this method lacks the stability of computing, and its accuracy of the result is not high. Other methods include the implicit Euler method, Crank Nicolson method, Leapfrog method, etc. The method used in miniMD is the Velocity Störmer Verlet method. Equation 2.9 shows its routine. It uses two separate steps to get the

velocity $\vec{v}(t+\triangle t)$ in the next time step. At the beginning, the velocity in the intermediate time step $\vec{v}(t+\frac{\triangle t}{2})$ is computed. The next location is got based on the velocity in $t+\frac{\triangle t}{2}$. Then, the velocity in the intermediate time step and the acceleration in the next time step lead to the velocity $\vec{v}(t+\triangle t)$. From the theoretical perspective, this method has second order, which leads to higher accuracy than the explicit Euler method. It is more stable when errors happen.

$$\begin{aligned}
\vec{v}(t+\frac{\triangle t}{2}) &= \vec{v}(t)+\frac{\triangle t}{2}\vec{a}(t) \\
\vec{r}(t+\triangle t) &= \vec{r}(t)+\triangle t\vec{v}(t+\frac{\triangle t}{2}) \\
\vec{v}(t+\triangle t) &= \vec{v}(t+\frac{\triangle t}{2})+\frac{\triangle t}{2}\vec{a}(t+\triangle t)
\end{aligned} \tag{2.9}$$

In a typical molecular dynamics simulation application such as miniMD, the physical properties of simulation such as the temperature of the simulation box is computed based on the number and velocities of particles. How the temperature T is computed is given in Equation 2.10 [2]. k_B is the Boltzmann constant. E_{kin} refers to the kinetic energy. N denotes the number of particles. The temperature sometimes needs to be adjusted to keep the balance between the kinetic energy and the potential energy [2]. Equation 2.11 shows that the target temperature T^D is got by scaling the velocities with a factor β [2].

$$T = \frac{2}{3Nk_B}E_{kin} = \frac{2}{3Nk_B}\sum_{i=1}^{N}\frac{m_i}{2}\mathbf{v}_i^2 \tag{2.10}$$

$$\begin{aligned}
\beta &:= \sqrt{E_{kin}^D/E_{kin}} = \sqrt{T^D/T} \\
\mathbf{v}_i^n &:= \beta\mathbf{v}_i^n
\end{aligned} \tag{2.11}$$

In molecular dynamics simulation, general boundary conditions are needed. Some widely-used examples can be the Dirichlet boundary conditions or Neumann boundary conditions. For materials with regular structures such as crystals, the periodic boundary conditions can be used to extend the simulated domain into infinity [2]. In the case of periodic boundary conditions, when a particle moves out of the subdomain along the direction of its velocity, an image particle in the opposite direction will move into this subdomain through the boundary on the opposite side. The outflow boundary conditions refer to boundaries that particles can cross before they leave the simulation box. Similarly, the inflow boundary conditions refer to boundaries that particles can cross before they enter the simulation box. As the simulation process goes on, the number of particles in the simulation box may decrease in the case of outflow boundary conditions. By contrast, the reflecting boundary conditions can make sure that each particle that comes close to the boundary will be bounced back. There are two methods to realize this principle. The first one is to set a mirror virtual particle with the same mass for each particle outside the boundary. Thus there exists a repulsive force between the real particle and the virtual particle, and the force will tend to infinity when the particle gets close enough to the boundary. The second one is to move the particles outside the simulation box into the simulation box. The position and velocity of the new particle are mirrored across the boundary [2]. Figure 2.10 presents a sketch of the reflecting boundary conditions.

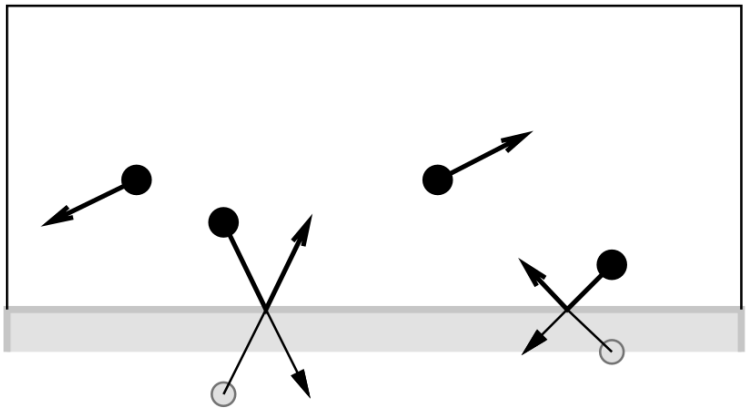


Figure 2.10.: Reflecting Boundary Conditions.
Source: [2]

2.4.2. Examples of molecular dynamics simulation

LAMMPS is a molecular parallel simulator that can simulate scenarios with a large diversity of features. New external constraints, diagnostics, and interatomic potentials can be added to LAMMPS [40]. Fundamental simulation methods used in LAMMPS are distributed neighbor lists, spatial decomposition in parallel, parallel Fast Fourier Transforms (FFTs) which is used to compute the interactions of long-range Coulomb force [41], and the Störmer Verlet symplectic method [42]. Because of its versatility, LAMMPS is widely applied to various simulation fields such as material science, biology, and geography [7]. The substances it can simulate range from solid and soft materials to mesoscopic and coarse-grained systems [8]. The source code of LAMMPS was developed at Sandia National Laboratories. The languages it consists of are C++, C, Fortran, and Python. It can not only be executed in one CPU core, but it can also run in a parallel system based on message passing with accelerators.

miniMD was developed in the Mantevo project at Sandia National Laboratories. It is a simplified version of LAMMPS [41], [43]. It shares lots of features of LAMMPS. For instance, it uses a parallel spatial decomposition similar to LAMMPS. But not all the features of LAMMPS are included in miniMD. For example, the models of the interaction between particles only include the LJ and EAM models. Only short-range force can be computed in miniMD. The miniMD code supports weak scaling and strong scaling, which is perfect to test the performance of machines in the cases of different problem sizes and configurations of hardware resources. Compared with the complicated LAMMPS code, it only consists of 5000 lines of C++ code. Thus, developments based on miniMD will be convenient. There are different variants of miniMD. The most proper one for my project is the "miniMD_ref" one, because the MPI feature is implemented in its code and it does not refer to complicated external libraries such as Kokkos [44]. The features users can set up include the size of the simulation box, the number of processes and threads, the mode of interaction between particles, the density of particles, the number of time steps to run, cutoff radius, etc. The features it owns can fulfill the basic requirements of molecular dynamics simulation [45]. All

[7] `https://www.olcf.ornl.gov`
[8] `https://www.lammps.org/`

in all, miniMD is much easier to be reproduced and modified in OctoPOS than LAMMPS.

Part II.

Thesis Development

3. Design and Implementation of *i*miniMD

The adaptations made in miniMD include three parts: basic changes such as the reconstruction of functions, the development of extra content, and the visualization content. The work of the first two parts leads to *i*miniMD in OctoPOS. The result of the third part is a C++ program based on OpenGL and MPI in Linux. Figure 3.1 shows the basic process of *i*miniMD. The "Simulation process" in Figure 3.1 means a whole process of the molecular dynamics simulation in miniMD. The loop means that the simulation process is executed for several times with different numbers of iMPI processes. At the end of each iteration, except the last one, an iMPI adaptation is necessary to expand the communicator and new children processes are prepared for the next iteration. The details about the "Simulation process" of Figure 3.1 are shown in Figure 3.2. The main part of the process is a loop of simulation. The real-time simulation data is sent to the client program per a number of iterations. The reneighboring, which consists of the reallocation of local particles, the communication of ghost particles, and the construction of neighbor lists, is called per $neigh\_every$ iterations shown in Table 3.1. Otherwise, only the information of ghost particles is communicated between processes. Then the force and the velocity of each particle, the total energy, the temperature, and the pressure of the simulation box are computed. At the end of an iteration, the location of each particle is updated according to its velocity. There is a visualization program intended to receive the data sent from the *i*miniMD application through TCP. It starts with two MPI processes, one of which is responsible for visualizing the data and the other of which aims at receiving the data and printing them in files. The visualization tools are provided by the OpenGL library. The specific implementations of *i*miniMD are described in this Chapter. The details about the visualization application are in Chapter 4.

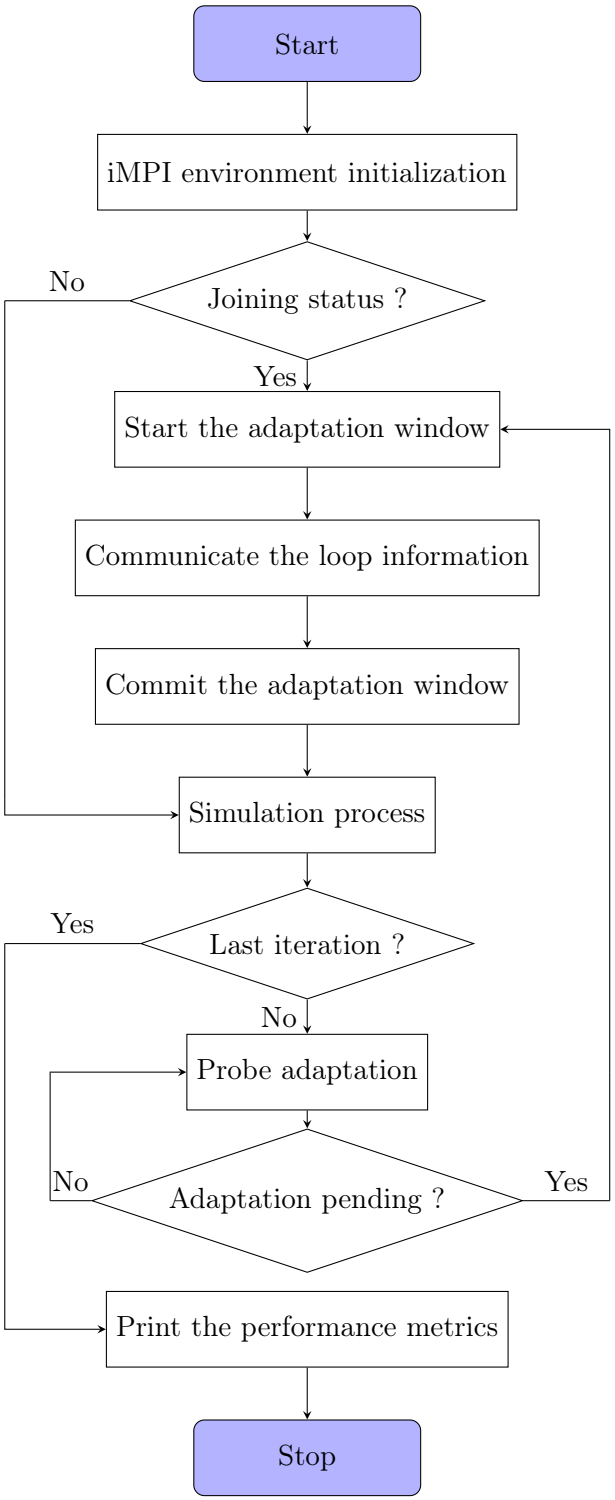


Figure 3.1.: Flow chart of *i*miniMD.

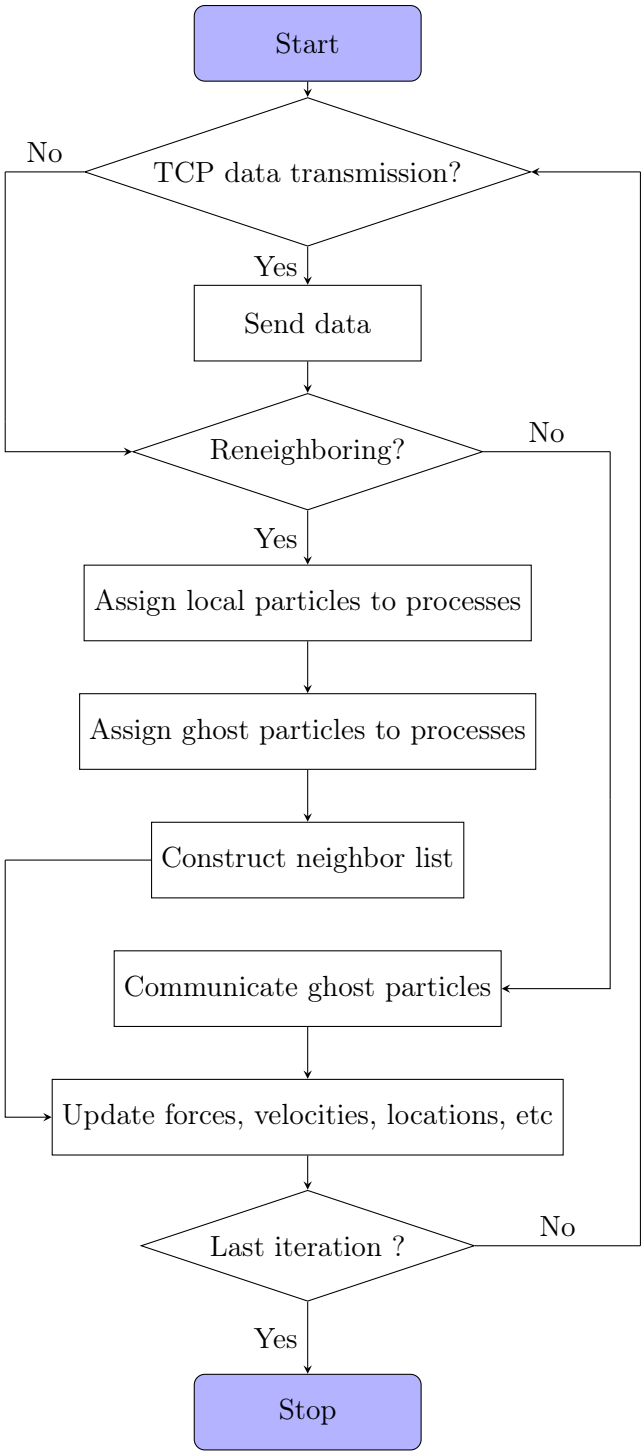


Figure 3.2.: Flow chart of the molecular dynamics simulation process.

3.1. Basic adaptations

In order to run miniMD in OctoPOS, the first objective is to reconstruct the source code and to make it fit into OctoPOS.

The first adaptation is the change of the programming language. It is reasonable to deploy a C-version miniMD in OctoPOS. The routines not supported by both C and C++ must be modified. C++ is based on object-oriented programming that is not available in C. All the C++ `class` definitions in miniMD were replaced with the C `struct` definitions in my project. Listing 3.1 shows the "Integrate" `class` in miniMD, which is responsible for simulating the process and computing the performance metrics. Correspondingly, the "Integrate" `struct` is defined in *i*miniMD. As Listing 3.2 describes, the old member variables keep the same, and some new member variables are inserted to add new features. *nrk* stores the color data of particles. *send_every* refers to the interval of sending data. *init_scenario* refers to the simulation scenario type. *is_server* can be set to decide whether the simulation data should be sent through TCP. *tcp_connection* refers to the socket used to send the data through TCP. *invasive_iter* refers to the current iMPI iteration index. *num_adaptations* means the number of iMPI iterations. It is equal to the number of iMPI adaptations plus one, because the first simulation does not need the iMPI adaptation. Because C does not support member functions in a `class`, member functions in the "Integrate" `class` are moved outside its `class` definition, and a pointer to an "Integrate" `struct` acts as a new parameter in these functions. These functions access the member variables of the "Integrate" `struct` through this pointer. The function *run* is moved to another `struct` in *i*miniMD. Similar adjustments were also made in other `class` definitions of miniMD. Another important transformation of language is about the memory management. Dynamic allocation and deallocation are supported by C++ through the `new` and `delete` operations. These two operations can separately allocate and free storage space in a pool which is called the free store [1]. In C, the `malloc` and `free` operations can be the substitutions of the `new` and `delete` operations. It is up to programmers to manage the memory [2]. But programmers must be careful to allocate a reasonable size of memory and remember to free the memory after its usage. In Subsection 2.3.1, it is pointed out that the size of memory in each tile is predefined in the configuration file in OctoPOS, thus any excessive allocation of memory in the code may lead to a memory overflow. Additionally, a `free` operation to match a previous `malloc` operation will help the execution to get rid of memory leaks. Especially in one pass of execution for several iMPI adaptations aiming at strong scaling, any memory should be deallocated if it was allocated using the `malloc` operation and will not be accessed anymore.

[1] `https://docs.microsoft.com/en-us/cpp`
[2] `https://www.cplusplus.com/reference`

```
1  class Integrate
2  {
3    public:
4      MMD_float dt;
5      MMD_float dtforce;
6      MMD_int ntimes;
7      MMD_int nlocal, nmax;
8      MMD_float* x, *v, *f, *xold;
9      MMD_float mass;
10     MMD_int sort_every;
11     ThreadData* threads;
12
13   Integrate();
14   ~Integrate();
15   void setup();
16   void initialIntegrate();
17   void finalIntegrate();
18   void run(Atom&, Force*, Neighbor&, Comm&, Thermo&, Timer&);
19 };
```

Listing 3.1: Definition of `class` Integrate in C++.

```
1  typedef struct{
2    MMD_float dt;
3    MMD_float dtforce;
4    MMD_int ntimes;
5    MMD_int nlocal, nmax;
6    MMD_float* x, *v, *f, *xold;
7    int* nrk;
8    int invasive_iter, num_adaptations;
9    MMD_float mass;
10   MMD_int send_every, sort_every;
11   enum ScenarioStyle init_scenario;
12   enum IsServer is_server;
13   octo_tcp_socket_t* tcp_connection;
14   ThreadData* threads;
15 }Integrate;
16
17 void init_integrate(Integrate*);
18 void destroy_integrate(Integrate*);
19 void integrate_setup(Integrate*);
20 void initial_integrate(Integrate*, Atom*, int);
21 void final_integrate(Integrate*, Atom*);
```

Listing 3.2: Definition of `struct` Integrate in C.

The second necessary adaptation is to make the replacements of the inaccessible functions of miniMD in OctoPOS and delete the features those can not be reproduced in OctoPOS. For instance, the MPI operations about cartesian grid are not available in OctoPOS. In miniMD, they tell the location of each process in the cartesian grid. Then each process can locate the processes to communicate with. The function replacements responsible for computing the rank number according to the three-dimensional location in the three-dimensional cartesian grid and computing the location according to the rank number, are introduced in Listing 3.3 and Listing 3.4. The algorithm used here is similar to that of the cartesian functions in Open MPI. Listing 3.3 shows the function *get_location* to compute the three-dimensional location by the rank number. The array *procgrids* stores the numbers of processes in three dimensions. The location of the process is computed and then loaded into *myloc*. In Listing 3.4, the function *get_neighbor* describes how the cartesian grid location is used to compute the rank number. The parameter *dims* is equivalent to the parameter *procgrids* in the previous function, and the parameter *periodic* tells the periodicity of coordinate in each dimension. Another important example is about the performance metrics. The library used to compute the execution time is the "timer" library in miniMD, which is not fully supported in OctoPOS. As a result, the function `MPI_Wtime` is used to the compute the execution time. `MPI_Wtime` aims at acting as a wall clock or elapsed clock with high accuracy, whose resolution is determined by another function `MPI_Wtick` [3]. Based on the `MPI_Wtime` function, a "Timer" `struct` was implemented in *i*miniMD. It includes a function intended to return the current double-precision time in seconds and structures storing the performance metrics of each execution.

[3] `https://www.mpich.org/static/docs/v3.3/www3/MPI_Wtime.html`

```
1  void get_location(const int procgrids[], int rank, int myloc[])
2  {
3    int nnodes = procgrids[0] * procgrids[1] * procgrids[2];
4    for(int i = 0; i < 3; i++)
5    {
6      nnodes = nnodes / procgrids[i];
7      int tmp_coord = rank / nnodes;
8      myloc[i] = tmp_coord;
9      rank = rank % nnodes;
10   }
11 }
```

Listing 3.3: Function: getting the cartesian grid location according to the rank number.

```
1  void get_neighbor(const int dims[], const int periodic[], const int coords
       [], int* rank)
2  {
3    int coord, multiplier, ndims = 3;
4    *rank = 0;
5    multiplier = 1;
6    for(int i = ndims - 1; i >= 0; i--)
7    {
8      coord = coords[i];
9      if(periodic[i])
10     {
11       if(coord >= dims[i])
12         coord = coord % dims[i];
13       else if(coord < 0)
14       {
15         coord = coord % dims[i];
16         if(coord)
17           coord = dims[i] + coord;
18       }
19     }
20     *rank += multiplier * coord;
21     multiplier *= dims[i];
22   }
23 }
```

Listing 3.4: Function: getting the rank number according to the cartesian grid location.

There are features in miniMD those can not be reproduced in OctoPOS. OpenMP is currently not supported in OctoPOS, so it was removed in *i*miniMD. It is not enough to only remove the OpenMP directives. In miniMD, the total work in each process is split evenly into parts, and these parts are loaded into threads. In *i*miniMD, the multi-threading model is not implemented. Thus the work distribution within one process and the synchronization between threads are not needed anymore. To understand this case easier, it can be imagined there is only one thread in each process. The directives such as the `pragma omp parallel for` were removed, and local variables in the local domain of each thread were replaced with the global variables in each process. In short, only one thread in each process occupies all the resources in this process. Although there is only one thread in each process, the `class` "ThreadData" storing the constructions of thread in miniMD is preserved. The reason to keep it is that the OpenMP directives and their corresponding adaptations could be added to OctoPOS if OpenMP is supported in OctoPOS in the future. After the work above was finished, other minor changes only include removing unnecessary input and output operations and deleting the features not needed such as the EAM interaction mode. The initial parameters of simulation are no longer in files, and they are configured in the source code.

By now, *i*miniMD is able to be executed in OctoPOS, and it is reasonable to insert the iMPI operations into it. The iMPI operations are mainly responsible for adapting the number of processes for strong scaling. Given a fixed problem of simulation, after each iMPI adaptation, new children processes start an adaptation window to join the current communicator. In each iteration of the invasive loop, all the processes in the current communicator simulate the scenario together, then launch an adaptation window together with new children processes. In Listing 3.5, the simulation code is included in the function $main\_func$, which represents the content of "Simulation process" in Figure 3.1. The variable cur_iter tells in which iteration the new children process should join, and this value is sent from a parent process to children processes through `MPI_Send`. Besides the iMPI content, the new content includes additional scenarios of simulation, the performance metrics part, and the data transmission part. These will be introduced in details in Section 3.2 and Section 3.3.

```
1  (...)
2  int main(int argc, char** argv)
3  {
4    MPI_Info info;
5    MPI_Comm intercomm, new_comm_world;
6    int rank, old_size, new_size, local_status, tmp_status, pending_adapt,
         staying_count, leaving_count, joining_count;
7    MPI_Init_adapt(argc, argv, &local_status);
8    int cur_iter = 0;
9    if(local_status == MPI_ADAPT_STATUS_JOINING)
10   {
11     MPI_Comm_adapt_begin(&intercomm, &new_comm_world, &staying_count, &
           leaving_count, &joining_count);
12     MPI_Status status;
13     MPI_Recv(&cur_iter, 1, MPI_INT, 0, MPI_ANY_TAG, intercomm, &status);
14     MPI_Comm_adapt_commit();
15   }
16   for(int iter = cur_iter; iter < num_adaptations; iter++)
17   {
18     main_func(argc, argv, local_status);
19     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20     MPI_Comm_size(MPI_COMM_WORLD, &old_size);
21     if(iter != num_adaptations - 1)
22     {
23       new_size = old_size + 2;
24       MPI_Resize_request(MPI_COMM_WORLD, new_size);
25       MPI_Probe_adapt(&pending_adapt, &tmp_status, &info);
26       while(pending_adapt != MPI_ADAPT_TRUE)
27       {
28         MPI_Probe_adapt(&pending_adapt, &tmp_status, &info);
29       }
30       MPI_Comm_adapt_begin(&intercomm, &new_comm_world, &staying_count, &
             leaving_count, &joining_count);
31       if(rank == 0)
32       {
33         for(int j = old_size; j < new_size; j++)
34         {
35           int tmp = iter + 1;
36           MPI_Send(&tmp, 1, MPI_INT, j, 0, intercomm);
37         }
38       }
39       MPI_Comm_adapt_commit();
40     }
41   }
42   (...)
43 }
```

Listing 3.5: Code of the iMPI adaptations in *i*miniMD.

3.2. *i*miniMD extensions

There is only one way to initialize the particles in miniMD. It generates particles according to parameters in the configuration file such as the predefined density and the problem size. The generated particles are randomly distributed into processes, but the initial distribution of particles is approximately even in the simulation box. Each process has its local domain, and it manages the particles located at its local domain. The properties of each particle are stored in the arrays. During the simulation, the forces on particles are computed, and then physical properties of particles such as the velocities and locations are updated per iteration. According to visualization experiments, the scenario in miniMD looks chaotic. Thus, a new collision scenario should be developed. It is easier to observe the particles while one object strikes another object. Two scenarios of collision between objects with outflow boundary conditions and reflecting boundary conditions were added to *i*miniMD. The initial configurations of a collision simulation example can be checked in Table 3.1. In the outflow boundary conditions scenario, a smaller object with a given velocity $\mathbf{v}$ collides with a larger resting object, and both of them are superimposed with a thermal motion based on the Maxwell-Boltzmann distribution [2]. A possible method to generate a Maxwell-Boltzmann distribution is to generate a number according to a $N(0,1)$ normal distribution and multiply it with a $factor$. The result can be a velocity component of the thermal motion. $factor$ is described in Equation 3.1. $\langle v_d^2 \rangle$ means the mean squared velocity of each component of a velocity v. Another collision case includes the reflecting boundary conditions and adds the gravity force to the dropping object. The t_target parameter in Table 3.1 means the temperature the simulation box keeps during the execution. In order to keep the temperature t_target, the velocities of particles are scaled per a number of iterations.

$$\boldsymbol{factor} = \sqrt{\langle v_d^2 \rangle} \tag{3.1}$$

Symbol	Value	Meaning
$L_1 * L_2 * L_3$	250*250*10	sizes of simulation box
$boundarytype$	OBC	outflow boundary conditions
ϵ	5.0	ϵ in LJ
σ	1.0	σ in LJ
m	1	mass of particle
N	2000	total number of particles
N_1	400	number of particles in dropping object
N_2	1600	number of particles in resting object
$\mathbf{v}$	(0, 10, 0)	initial velocity of dropping object
f_{cut}	2.5*σ	cutoff radius of force
δt	0.005	time step
$ntimes$	20000	number of iterations
$neigh\_every$	20	interval of constructing the neighbor list
$has\_gravity$	0	gravity not available
t_target	100.00	the constant temperature in the simulation box

Table 3.1.: Initial parameters for one collision simulation example.

The performance metrics part is a modified version of that in miniMD. By the functions of the "Timer" `struct` in *i*miniMD, the time interval from one starting point to the current point in time can be computed by the function `MPI_Wtime`. miniMD computes the time of the communication between processes, constructing the neighbor list, computing the forces, and the whole simulation. Similarly, splitting the total execution time into several parts is also achieved in *i*miniMD. Furthermore, the time used in the iMPI adaptation and the time used in the network data transmission are computed in *i*miniMD. Therefore, it can be summarized how the invasive operations and the data transmission affect the whole simulation. Clear statistics of the time cost in each part can be generated after each execution.

3.3. Data transmission

In order to visualize the real-time motions of particles and record the simulation data, the data needs to be transmitted to another visualization application running in a Linux system. In the swe-x10 application described in Subsection 2.3.3, the data is transmitted using a socket through TCP. In this case, the swe application is a server, and another C++ visualization application acts as a client. Fortunately, OctoPOS provides developers with a well-integrated TCP library. By including "octo_tcp.h", the TCP interfaces are available for developers.

In the source code of *i*miniMD, there is a variable $send\_every$ that shows the iteration interval of transmitting the simulation data. It is time-consuming to transmit data, thus frequent communication may make the execution time too long. Listing 3.6 describes the function of sending data to clients. The $init\_flag$ shows whether it is the first time to send data, because only the first time to call the communication function needs creating a socket, binding the socket to an IP address and a port number, listening to clients, and accepting connections. Once the socket $client\_connection$ is accepted, the interface `octo_tcp_sync_send` can be used to send data to clients. *rank* refers to the rank number of the caller process, and this parameter is common in *i*miniMD because it is indispensable to debug the iMPI programs according to the output of the program. *step* shows the number of times needed to send all the data in the $send\_buffer$ array, and the $step\_size$ is the number of bytes sent per `octo_tcp_sync_send` operation. Because there is an upper limit of the size of data that can be stored in the data packet of TCP, splitting the data and sending them for several times could be necessary. $step\_size$ should be chosen carefully in order to match the speed of sending data in *i*miniMD and the speed of receiving data in the visualization application, and this will be introduced in details in Section 4.2. *port* is a global variable that is equal to 8080, and the local port of each process is calculated by $port + rank$. The macro $local\_ip$ is equal to "192.168.132.100", which is the IP address of OctoPOS and defined in the file "Device.cc" of iRTSS. With this IP address, external applications can locate the OctoPOS application. If the current communication ends, the socket must be closed through the interface `octo_tcp_sync_close`.

```
1  void send_data(int init_flag, octo_tcp_socket_t* client_connection, int rank
       , char* send_buffer, int step, int step_size)
2  {
3    enum Errors err;
4    uint16_t local_port = port + rank;
5    if(init_flag != 0) {
6      octo_tcp_socket_t listen_socket;
7      err = octo_tcp_sync_create(&listen_socket);
8      if(err != ESUCCESS) {
9        printf("ERROR: Could not create socket\n");
10       return;
11     }
12     octo_ip4_t ip4;
13     err = octo_ip4(&ip4, local_ip);
14     if(err != ESUCCESS) {
15       printf("ERROR: Could not convert the IP %s\n", local_ip);
16       return;
17     }
18     err = octo_tcp_sync_bind4(listen_socket, &ip4, local_port);
19     if(err != ESUCCESS) {
20       printf("ERROR: Could not bind socket to %s:%d\n", local_ip, local_port
             );
21       return;
22     }
23     uint16_t numc = 10;
24     err = octo_tcp_sync_listen(listen_socket, numc);
25     if(err != ESUCCESS) {
26       printf("ERROR: Could not listen to socket %s:%d\n", local_ip,
             local_port);
27       return;
28     }
29     err = octo_tcp_sync_accept(listen_socket, client_connection);
30     if(err != ESUCCESS) {
31       printf("ERROR: Could not accept clients in socket %s:%d\n", local_ip,
             local_port);
32       return;
33     }
34   }
35   if(init_flag == 0 || err == ESUCCESS) {
36     for(int i = 0; i < step; i++) {
37       uintptr_t send_size = step_size * (sizeof(char));
38       err = octo_tcp_sync_send(*client_connection, (void*)(send_buffer + i *
             step_size), &send_size);
39       if(err != ESUCCESS) {
40         printf("ERROR: Could not send data to client %s:%d\n", local_ip,
               local_port);
41         return;
42       }
43     }
44   }
45   else
46     printf("ERROR: Client connection failed %s:%d\n", local_ip, local_port);
47 }
```

Listing 3.6: Function: sending data to client through TCP.

4. Design and Implementation of the Visualization Application

In the case of visualization, *i*miniMD plays the role of server in communication. Thus, a corresponding client application is needed. In order to develop the application without the restrictions in OctoPOS mentioned in the previous chapters, the application is designed to be a C++ application in Linux. Figure 4.1 describes the outline of the visualization process. Each process in *i*miniMD sends the data of local particles to the root process. Then the root process sends the data to the data transmission process in the client. After that, the data is sent to the visualization process where the simulation data is visualized. More details are introduced in the next sections.

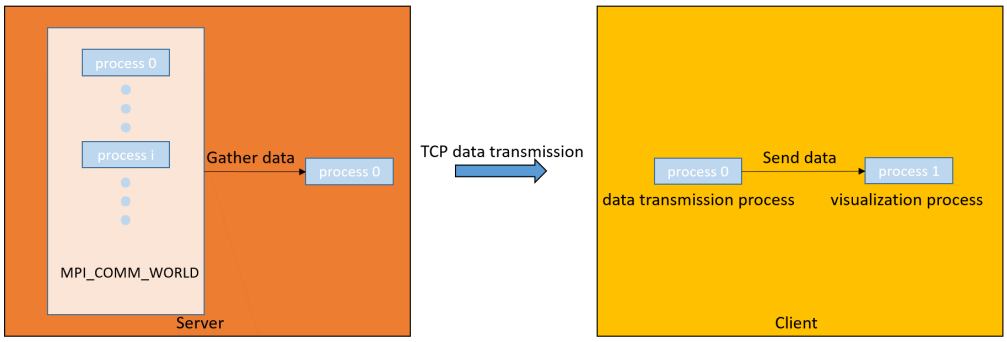


Figure 4.1.: Visualization process.

4.1. Basic design and implementation

The data in the server is sent to one data transmission process in the client, and another visualization process is needed to visualize the real-time motions of particles. The multi-process mechanism is needed in the implementation. Therefore, it is reasonable to design the client based on the MPI routines. The data transmission process connects with OctoPOS by the IP address and port number of OctoPOS. Then the simulation data can be sent from OctoPOS to the data transmission process. With MPI, the simulation data can be sent to the visualization process. The OpenGL library provides the interfaces to visualize the simulation. OpenGL is a kind of widely used two-dimensional and three-dimensional graphics API library. It is independent of operating systems and hardware, which makes it flexible and convenient to use this tool [1]. The objects of the simulation are represented by two-dimensional and three-dimensional instances in OpenGL. For instance, I used eight points and their connecting lines to represent the simulation box in OpenGL. The particles moving in it are solid spheres whose properties such as their radius, color, and position can be adjusted by specific OpenGL interfaces.

4.2. Data transmission

The flow chart of the data transmission process is shown in Figure 4.2. In Figure 4.1, the data transmission process possesses a socket, which has the same IP address and port number as those in OctoPOS. For instance, a process in the server sends data through a socket bound to the IP address "192.168.132.100" and the port number "8080". Correspondingly, a socket with the same IP address and port number in the client undertakes the responsibility of receiving the simulation data. The connection between the client and the server keeps alive until the current simulation ends. Frequent termination of the connection and establishment of it will lead to high overhead, thus keeping the connection alive improves the execution efficiency. The simulation data is packed into a byte array before it is sent and is converted into the desired data format after it is received. For instance, an *unpack_int* function copies the received data bytes to the storage address of an `int` variable, and then this variable can be used further. In *i*miniMD and the client, other data formats include `float` and `double`. In *i*miniMD, the default format is the `float` one, but it can be adjusted to the `double` one if the `double` precision is chosen in the initial configuration.

After the connection between the server and the client is built, the general simulation data is transmitted in Figure 4.2. The general simulation data includes the number of simulation iterations, the number of visualization iterations, the number of iMPI iterations, the iMPI iteration index, the number of processes, the size of the simulation box, the simulation scenario type, and the data precision. Then, the visualization loop in Figure 4.2 can start. In each iteration, the locations, colors, and velocities of particles are transmitted. Besides, the sizes and locations of all the local domains in current simulation iteration are received. However, the data is likely to be lost through TCP. Listing 4.1 describes the code of receiving the location data of particles. num_rank refers to the number of processes in *i*miniMD. num_data_arr stores the number of particles in each process. DIM_NUM is the number

[1] `https://www.khronos.org/opengl`

of dimensions. $FLOATSTEP$ refers to the number of bytes needed to store a float value. $DOUBLESTEP$ refers to the number of bytes needed to store a double value. Within each iteration, the location data of the particles in one process is received. But the size of data sent per transmission is only $group\_size$ bytes, in order to match the speed of sending the data and the speed of receiving the data. In my experiment, a too high $group\_size$ may lead to frequent data loss. For instance, in the hardware experiment of Chapter 5, some simulation data is lost if 1200 bytes are sent per sending operation. If the number of received data bytes cur_num is not equal to $group\_size$, it means that a part of the simulation data is lost and the visualization program must be terminated. The data of all the particles is sent to the visualization process through the `MPI_Send` interface, and then the visualization process can collect all the data necessary to visualize the scene of the current iteration.

In order to verify the correctness of the data received, analyze the history simulation, and reproduce the simulation in a professional software of molecular dynamics simulation, the data of particles and the data of simulation domains are written to visualization toolkit (vtk) files. During the simulation, a temporary file is intended to store point cloud data. Each line of the temporary file includes the position and color data of one particle. The point cloud data of each process in one iteration is stored in one file, and all the files for one iteration are integrated into one vtk file that stores all the point cloud data in the same iteration. The vtk file can store the information of structured points, structured grid, rectilinear grid, polygonal data, and unstructured grid [2]. There is no professional vtk structure to store point cloud data. A polygonal structure is used to store the point cloud data. The example of point cloud data in vtk format is described in Listing 4.2. The first few lines define the version of file, the information of data, and the type of file. Then the real data follows. In this example, the dataset `POLYDATA` is used. The line including the keyword `POINTS` means that the coordinates of n points with the float precision are given. The precision descriptor can also be double. The keyword `VERTICES` is used to describe the polygonal topology. Each line represents a cell. n refers to the length of the cell list. $2n$ refers to the total number of integers in the cell list. The last part defines the color of each point and each color is represented by an integer. It is also proper to store the construction of the simulation box in a vtk polygonal structure [3]. Listing 4.3 shows an example to construct a simulation box. The data after the line "`POINTS 8 float`" refers to the eight vertices of the box, and 30 values following the keyword `POLYGONS` are used to describe the graph topology of the box. Each line following `POLYGONS` depicts a square with 4 vertices. Besides the global simulation box, the local simulation domain of each process is also recorded in a single vtk file. In the future work, the dynamic change of the local simulation domain is needed to reach the load balancing of work, thus the local simulation domain may be different in different points in time.

The vtk files can be visualized in ParaView, which is a data visualization and analysis application [4]. ParaView can help reproduce the simulation and analyze it after the simulation is completed. For example, developers can choose the point cloud data of one process or choose the data of multiple processes to present. A group consisting of vtk files in several iterations can be played in succession to show a consecutive process of simulation. Besides

[2] `https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf`
[3] `https://kitware.github.io/vtk-examples`
[4] `https://www.paraview.org`

the visualization result of the point cloud data, the statistics of data such as the number of points and the range of their positions can also be presented in ParaView. The viewpoint and scale of the visualizing window, the opacity, color, and radius of particle, and the opacity as well as the color of the simulation box can be adjusted to make the simulated scenario clearer.

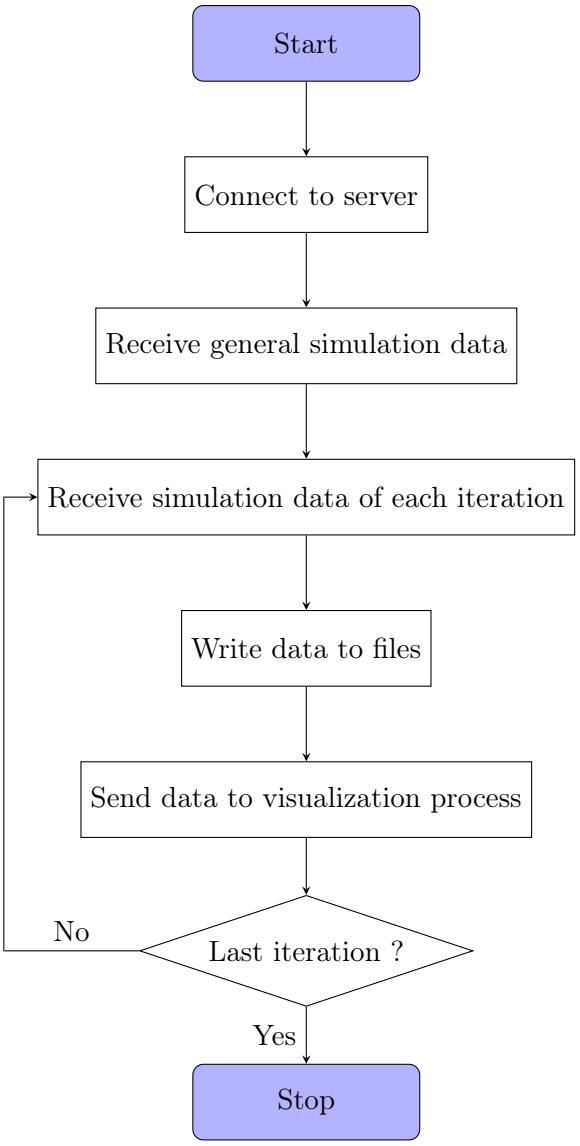


Figure 4.2.: Flow chart of the data transmission process.

```
1   int buf_start = 0;
2   for(int k = 0; k < num_rank; k++) {
3     if(num_data_arr[k] == 0)
4       continue;
5
6     group_size = GROUPSIZE1;
7     total_size = num_data_arr[k] * DIMNUM * ((precision == 0)?FLOATSTEP:
          DOUBLESTEP);
8     group = total_size / group_size;
9     start = 0;
10
11    for(int i = 0; i < group; i++) {
12      int cur_num = recv(sockfd, recv_buf + buf_start + start, group_size, 0);
13      if(cur_num != group_size) {
14        printf("ERROR: It loses location data of process %d in range %d-%d,
              and receives %d bytes instead of %d bytes!\n", k, start, start +
              group_size, cur_num, group_size);
15        exit(0);
16      }
17      start += group_size;
18    }
19
20    if(start < total_size) {
21      int cur_num = recv(sockfd, recv_buf + buf_start + start, total_size -
            start, 0);
22      if(cur_num != total_size - start) {
23        printf("ERROR: It loses location data of process %d in range %d-%d,
              and receives %d bytes instead of %d bytes!\n", k, start,
              total_size, cur_num, total_size - start);
24        exit(0);
25      }
26    }
27
28    buf_start += total_size;
29  }
```

Listing 4.1: Code of receiving the location data in client.

```
1  # vtk DataFile Version 2.0
2  Simulation Point Cloud Data
3  ASCII
4  DATASET POLYDATA
5  POINTS n float
6  p_(0x), p_(0y), p_(0z)
7  p_(1x), p_(1y), p_(1z)
8  ...
9  p_((n-1)x), p_((n-1)y), p_((n-1)z)
10 VERTICES n 2n
11 1 0
12 1 1
13 ...
14 1 n-1
15 CELL_DATA n
16 SCALARS cell_scalars int 1
17 LOOKUP_TABLE default
18 color_(0)
19 color_(1)
20 ...
21 color_((n-1))
```

Listing 4.2: VTK file format to store point cloud data.

```
1  # vtk DataFile Version 2.0
2  Simulation box
3  ASCII
4  DATASET POLYDATA
5  POINTS 8 float
6  0.000000 0.000000 0.000000
7  250.000000 0.000000 0.000000
8  250.000000 250.000000 0.000000
9  0.000000 250.000000 0.000000
10 0.000000 0.000000 10.000000
11 250.000000 0.000000 10.000000
12 250.000000 250.000000 10.000000
13 0.000000 250.000000 10.000000
14 POLYGONS 6 30
15 4 0 1 2 3
16 4 4 5 6 7
17 4 0 1 5 4
18 4 2 3 7 6
19 4 0 4 7 3
20 4 1 2 6 5
```

Listing 4.3: VTK file format to store a simulation box.

4.3. Visualization

To start the visualization step, the visualization process initializes the GLUT environment, configures general displaying modes, and configures the size and the position of the GLUT window. GLUT refers to the OpenGL Utility Toolkit, which provides developers with OpenGL API and can be used to write OpenGL programs. The simulation box is drawn in this window. The simulated scenario takes place in the simulation box. The synchronization between the visualization process and the data transmission process is achieved through the `MPI_Send` and `MPI_Recv` operations. After the data transmission operations in the current iteration are completed, the data transmission in the next iteration and the visualization process of the current iteration will then be in progress in parallel. Figure 4.3 shows the flow chart of the visualization process. The main work of it consists of receiving the simulation data, drawing the simulation box, drawing the particles, and printing general simulation information. These steps are integrated into a callback function called *update_box*. A time interval in milliseconds to trigger the callback function, the callback function pointer, and arguments of the callback function are passed to the function *glutTimerFunc*, then *update_box* can be called automatically per the time interval [5]. The length of the time interval should be chosen carefully because this interval should be close to the time used to receive the data from the server, copy it into its target storage space, and send the data to the visualization process. Too short or too long time interval to trigger a new *update_box* call will make the visualization process or the data transmission process wait for the other one and then prolong the execution time.

A part of the callback function *update_box* is shown in Listing 4.4. Firstly, the scenario type is received, which will be used to decide whether the three-dimensional mode or the two-dimensional mode is chosen. The evenly distributed particles are based on a three-dimensional case. As is discussed before, it is not easy to observe the collision in a three-dimensional space, because different initial shapes constructed by particles may be mixed after some iterations. However, the two-dimensional box can make the collision more clear. The particles of the collision cases have a zero velocity in the z direction of the coordinate system, thus a two-dimensional box to show the motions of particles is reasonable. Then other simulation data is received. According to the size of the simulation box and the fixed size of the GLUT window, the scaling parameter of each dimension can be computed. The scaling parameters are also used to scale the positions of particles. Once the data is ready, the particles and the simulation box will be drawn by *draw_atoms* and *draw_box*. `MPI_Wtime` is needed to compute the time interval of drawing the objects. The simulation box is composed of eight points and twelve lines. A particle is represented by a solid sphere, whose features include its location, radius, color, etc. Figure 4.4 and Figure 4.5 show the GLUT windows at the beginning and at the 4000th iteration of a 9000-iteration collision simulation. The general information printed in the GLUT window includes the current simulation iteration index, the number of the simulation iterations, the current number of particles, and the size of the simulation box. It is important to choose a proper radius and a distinctive color of particle for clear observation, especially when two objects mix together and the trajectories of their particles are pretty close to each other. In the end of *update_box*, it calls *glutTimerFunc* again until the visualization process ends. *num_iter* refers to the

[5] `https://www.opengl.org/resources/libraries/glut/spec3/node64.html`

visualization iteration index, and *max_iter* means the number of visualization iterations in the current simulation. In the strong scaling case, the same simulation can be executed for multiple times. After each iMPI adaptation in *i*miniMD, the visualization of one whole simulation will be presented again. *invasive_iter* notes the current iMPI iteration index, and *num_adaptations* notes the number of iMPI iterations. After the current simulation is finished, *update_box* is called if the next iMPI iteration is available. Otherwise the visualization process is synchronized with the data transmission process through a `MPI_Recv` operation and a corresponding `MPI_Send` operation, and then the application exits.

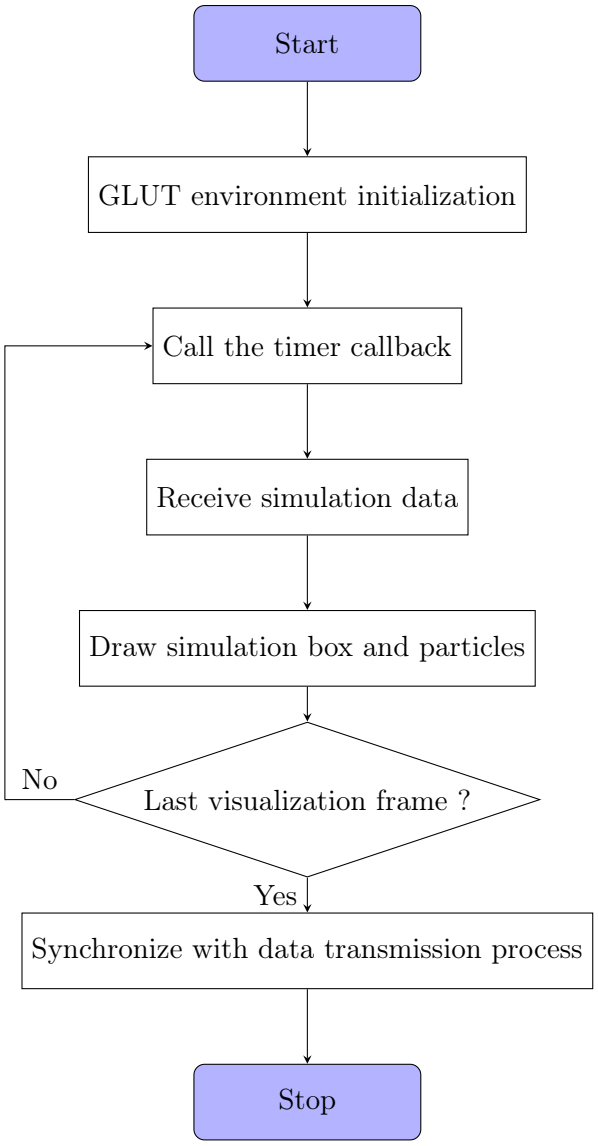


Figure 4.3.: Flow chart of the visualization process.

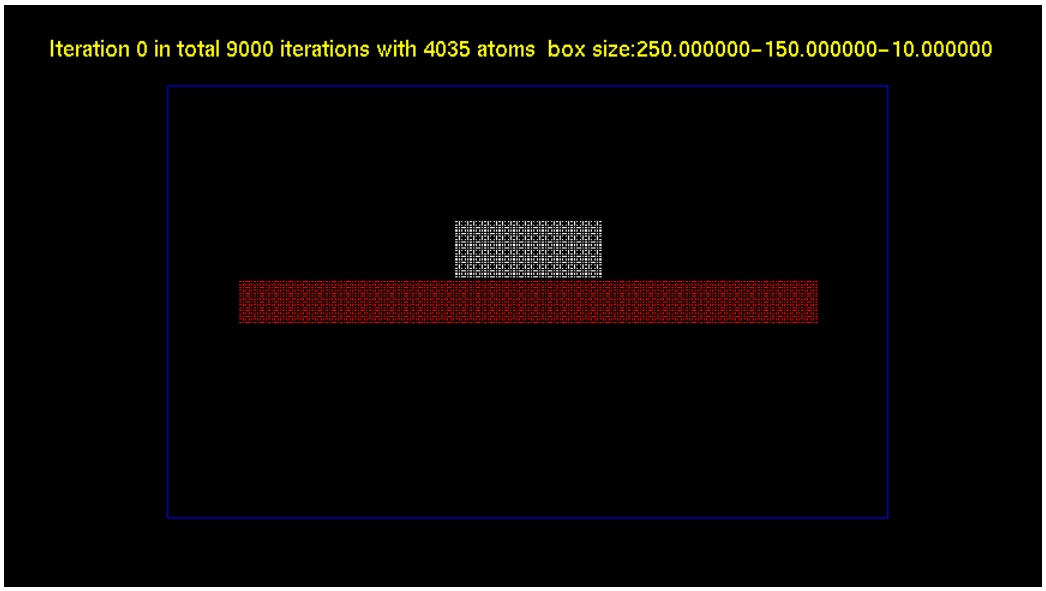


Figure 4.4.: Visualization window in the client application (1).

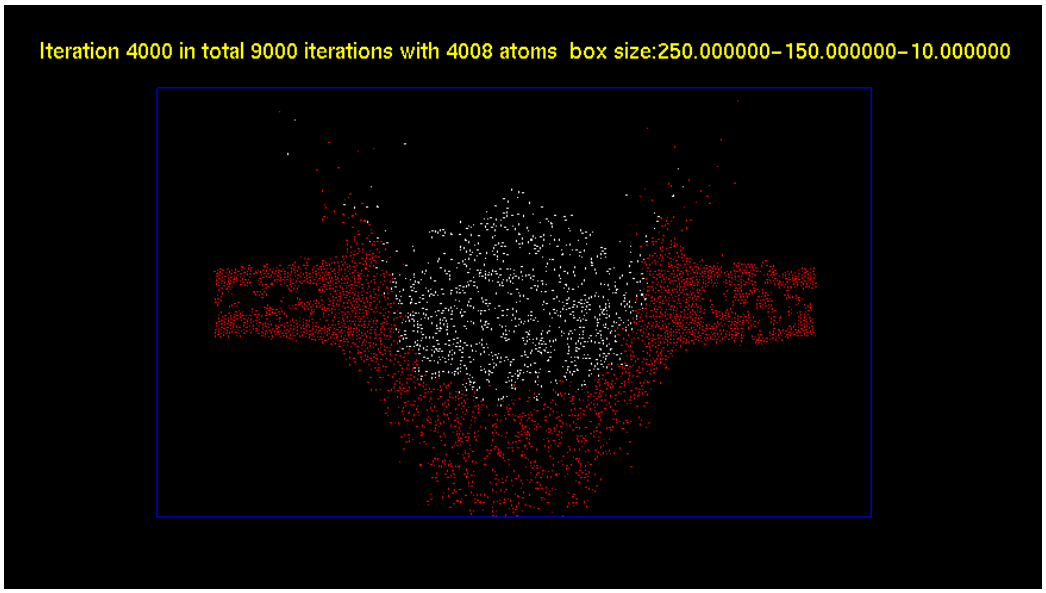


Figure 4.5.: Visualization window in the client application (2).

```
1  void update_box(int arg)
2  {
3    (...)
4
5    double start_time = MPI_Wtime();
6
7    char* text = (char*)malloc(CHARSIZE * TEXTSIZE);
8    sprintf(text, "Iteration %d in total %d iterations with %d atoms\n box
         size:%f-%f-%f", iter_info[0], iter_info[1], num_total, boxes[0], boxes
         [1], boxes[2]);
9    textout(-0.2, -0.05, -0.02, text);
10   free(text);
11
12   draw_atoms(locations, nranks, num_total);
13   draw_box();
14   double end_time = MPI_Wtime();
15   printf("Draw atoms at iteration %d with %lf seconds...\n", num_iter,
         end_time - start_time);
16
17   num_iter++;
18
19   (...)
20
21   if(num_iter < max_iter)
22     glutTimerFunc(3000, update_box, -1);
23   else {
24     if(invasive_iter == num_adaptations - 1) {
25       printf("INFO: Visualization process is waiting for the end of the
             program!!!\n");
26       int end_flag;
27       MPI_Recv(&end_flag, 1, MPI_INT, 0, rank, MPI_COMM_WORLD, &status);
28       exit(0);
29     }
30     else {
31       num_iter = 0;
32       glutTimerFunc(3000, update_box, -1);
33     }
34   }
35 }
```

Listing 4.4: Function: callback function of updating simulated scenario.

Part III.

Results and Conclusions

5. Evaluation

5.1. Experiments in QEMU

5.1.1. Evaluation setup

During the development of *i*miniMD, it is convenient to run the application using the QEMU tool, which can help me virtualize the target hardware. A binary file is generated according to the code of *i*miniMD in OctoPOS, and this binary file can be executed in QEMU or hardware. It is more convenient to run the simulation in QEMU than hardware, and the result can help verify the code before the experiment is moved to hardware.

Figure 5.1 shows how to connect *i*miniMD and the visualization application. The virtual `tap` device is used as the network device in QEMU, and its IP address is in the same subnetwork as that of OctoPOS. The visualization application runs in Linux and can receive the data from the `tap` device.

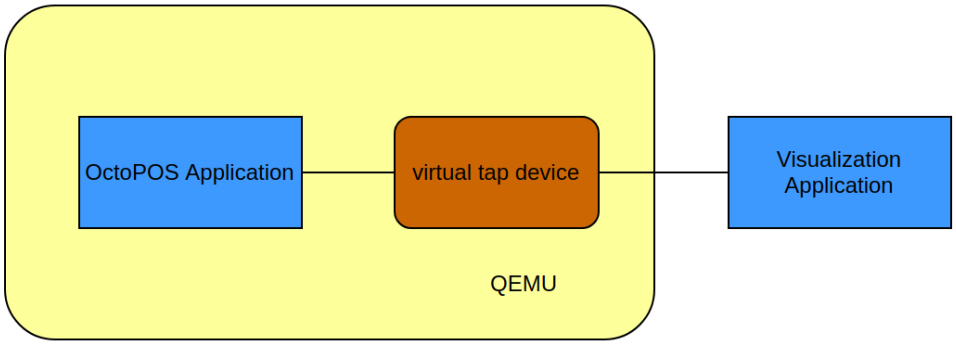


Figure 5.1.: Diagram of the QEMU experiment setup.

The experiment setup consists of four parts. The first part is the QEMU setup. The QEMU command needs to configure the number of nodes, the number of CPU cores, the memory size, the network device, etc. Each core will be occupied by one process in OctoPOS. The `tap` device needs to be created by the `tunctl` command. A `ifconfig` command should be used to configure the IP address of the `tap` device.

The second part is building iRTSS and the setup of OctoPOS. This part is already introduced in Subsection 2.3.1.

The third part is the simulation setup in *i*miniMD. In each simulation, `init_scenario` and `num_adaptations` are two important parameters. The `init_scenario` can be `EVENLY`, `RBCCOLLISION`, or `OBCCOLLISION`. The first one refers to the default simulation case in miniMD, and the last two cases refer to two new collision cases. The parameter `num_adaptations`

means the number of iMPI iterations through the iMPI interfaces. Before each iMPI iteration except the first one, an iMPI window is started and new children processes are added to the communicator.

The last part is about the visualization application. A "mpirun" command is used to boot the MPI-based visualization application after the output log of *i*miniMD shows that OctoPOS is accepting the connection of it. The output log is shown in Figure 5.2. Early booting of the visualization application may lead to an error because it can not find the server.

```
INFO: We are accepting sockets, and PLEASE START THE CLIENT!!!
```

Figure 5.2.: Output log of requiring a client in *i*miniMD.

The experiment in three scenarios will be carried on. In each scenario, the simulation will be executed in one setup with `num_adaptations=1` for getting the simulation data in the client and another strong scaling one. In the strong scaling simulation, the network data transmission between OctoPOS and the visualization application is closed in order to get more stable performance metrics, because the network condition varies in different points in time. The simulation data visualized by ParaView and the performance metrics will be given in Subsection 5.1.2.

5.1.2. Results

The first visualization simulation executed by QEMU is the `EVENLY` scenario. The initial configurations of this scenario are shown in Table 5.1. 4 iMPI processes are used to simulate 2048 particles with 40000 iterations. The time step δt is 0.0005 seconds, thus the total simulation time is 20 seconds with 40000 iterations. The locations of particles are generated according to an algorithm in miniMD, in order to distribute the particles in the simulation box almost evenly. For each particle, a unique random seed is used to generate a unique velocity vector for it. Then the velocities are recomputed to reach the initial temperature $t\_request$ shown in Table 5.1 and zero the velocity of center of mass in the simulation box.

The simulation result is shown in Figure 5.3. The simulation box is split into 4 independent local domains, each of which is occupied by one process. One box with one color refers to a process. The local particles in one process are in the same color. At the iteration 0, all the particles are initialized and located at its local domain. When more iterations of simulation are completed, particles from different processes mix with each other.

The second visualization simulation is the `RBCCOLLISION` scenario. The initial configurations of this scenario are shown in Table 5.2. 4 iMPI processes are used to simulate a collision process with 4361 particles for 20000 iterations. The time step δt is 0.0004 seconds, thus the total simulation time is 8 seconds. The particles are in two groups. One group constructs a dropping disk with the initial velocity $\mathbf{v}$ and the gravity $\mathbf{G}$, and the other group constructs a resting rectangle. For the velocity of each particle, a thermal motion generated according to the Maxwell-Boltzmann distribution is added to it. The squared velocity of the thermal motion in the x and y dimension is 0.0049. The squared velocity in the z dimension is 0. More details about this part are in Section 3.2. The velocities are scaled per a number of iterations to make the simulation box keep the constant temperature t_target.

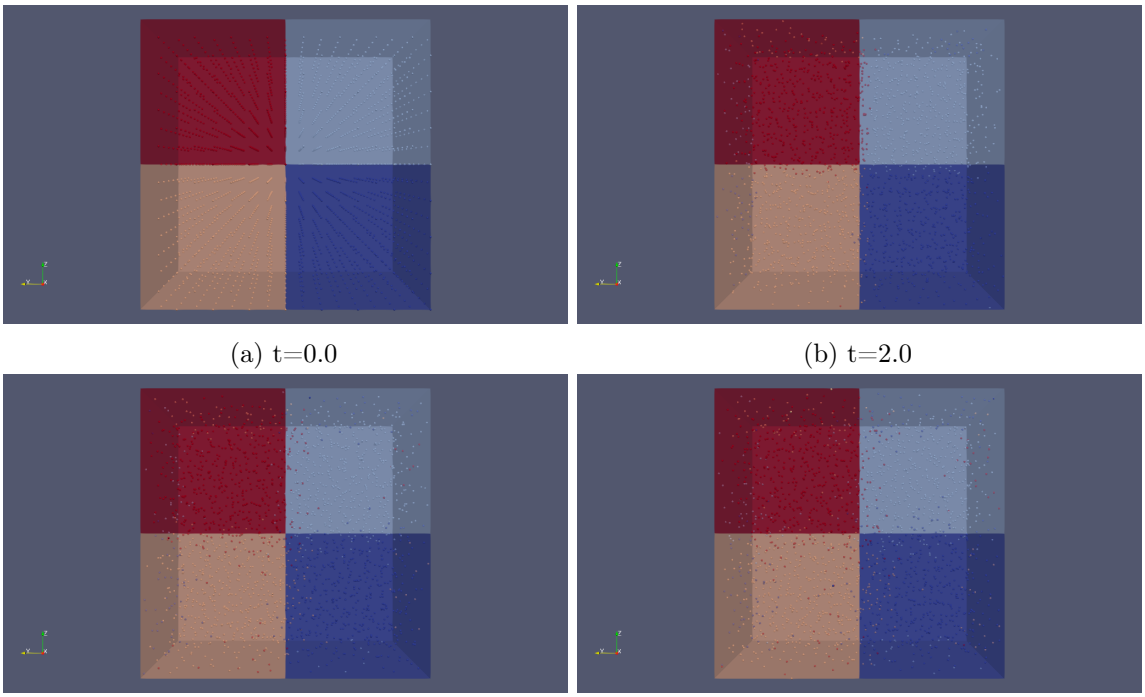
Figure 5.4 describes the simulation pictures in iteration 0, iteration 4000, iteration 8000, and iteration 20000. The particles in one object are in the same color, and they can be in different processes. When the dropping disk enters into the resting rectangle, it begins to dissolve in the rectangle. The collision between the two objects leads to a wave. The wave spreads from the middle of the simulation box to the boundaries gradually. Particles hitting the boundaries of simulation box are bounced back because of the reflecting boundary conditions, and the total number of particles in the simulation box keeps the same.

The third visualization simulation is the `OBCCOLLISION` scenario. The initial configurations of `OBCCOLLISION` scenario are shown in Table 5.3. 4 iMPI processes are used to simulate 4035 particles with 12000 iterations. The time step δt is 0.0005 seconds, thus the total simulation time is 6 seconds. The particles are also split into two groups. One is a dropping rectangle with the initial velocity $\mathbf{v}$, and the other group constructs a rectangle without the initial velocity. The squared velocity of the thermal motion in the x and y dimension is 0.01. The squared velocity in the z dimension is 0. The gravity $\mathbf{G}$ is not implemented in this scenario. A constant temperature is also kept in this simulation process.

Figure 5.5 shows the simulation pictures in iteration 0, iteration 4000, iteration 8000, and iteration 12000. When the dropping object strikes the still object, the particles constructing the bodies begin to break apart because of the strong collision. Particles moving outside the boundaries are deleted and the number of particles in the simulation box decreases.

Symbol	Value	Meaning
$L_1 * L_2 * L_3$	13.436769*13.436769*13.436769	sizes of simulation box
boundarytype	PBC	periodic boundary conditions
ϵ	1.0	ϵ in LJ
σ	1.0	σ in LJ
m	1	mass of particle
N	2048	total number of particles
ρ	0.8442	density of particles
f_{cut}	2.5*σ	cutoff radius of force
δt	0.0005	time step
ntimes	40000	number of time steps
neigh_every	20	interval of constructing the neighbor list
has_gravity	0	gravity not available
t_request	1.44	initial temperature

Table 5.1.: Initial parameters for the simulation of the `EVENLY` scenario.



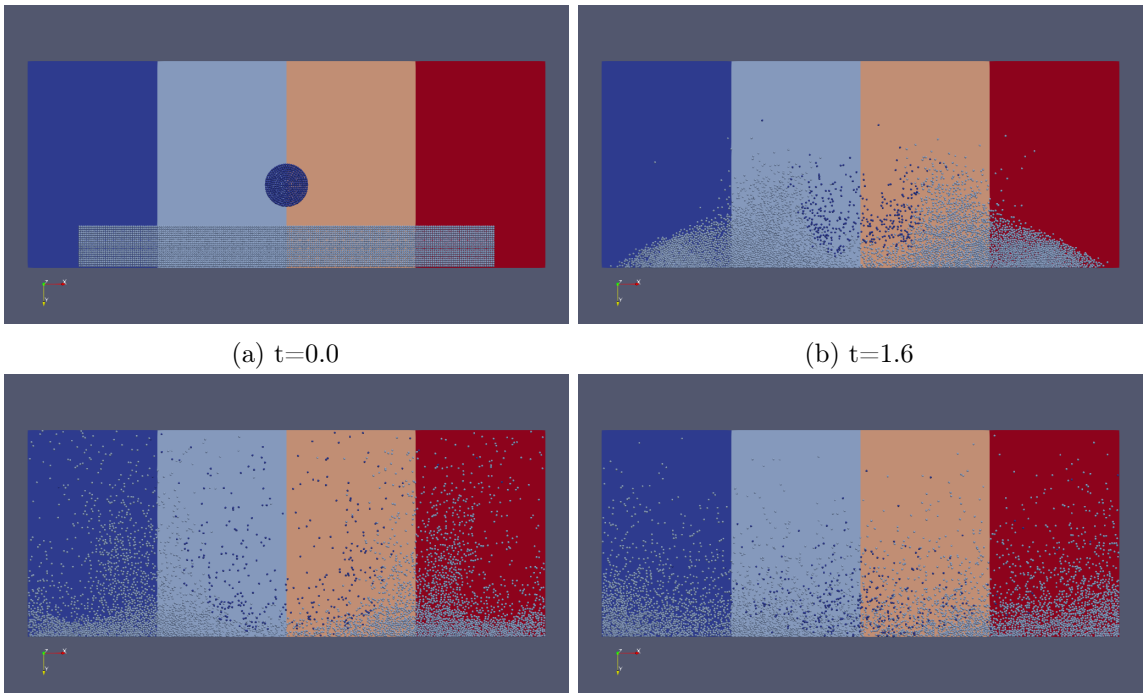
(a) t=0.0 (b) t=2.0

(c) t=10.0 (d) t=20.0

Figure 5.3.: Simulation pictures of the `EVENLY` scenario in QEMU.

Symbol	Value	Meaning
$L_1 * L_2 * L_3$	250*100*10	sizes of simulation box
boundarytype	RBC	reflecting boundary conditions
ϵ	1.0	ϵ in LJ
σ	1.0	σ in LJ
m	1	mass of particle
N	4361	total number of particles
N_1	341	number of particles in dropping object
N_2	4020	number of particles in resting object
$\mathbf{v}$	(0, 20, 0)	initial velocity of dropping object
f_{cut}	4.0*σ	cutoff radius of force
δt	0.0004	time step
ntimes	20000	number of time steps
neigh_every	20	interval of constructing the neighbor list
has_gravity	1	gravity $\mathbf{G} = (0, 12, 0)$ [2] available
t_target	100.00	the constant temperature in the simulation box

Table 5.2.: Initial parameters for the simulation of the `RBCCOLLISION` scenario.



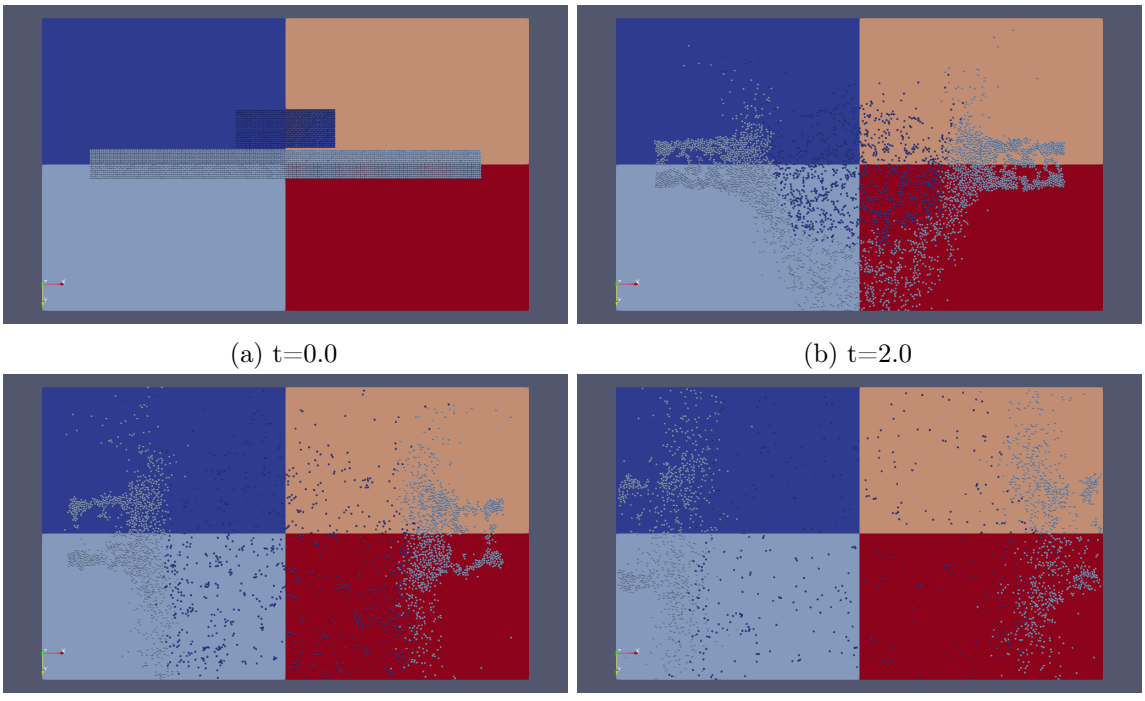
(a) t=0.0 (b) t=1.6

(c) t=3.2 (d) t=8.0

Figure 5.4.: Simulation pictures of the `RBCCOLLISION` scenario in QEMU.

Symbol	Value	Meaning
$L_1 * L_2 * L_3$	250*150*10	sizes of simulation box
boundarytype	OBC	outflow boundary conditions
ϵ	5.0	ϵ in LJ
σ	1.0	σ in LJ
m	1	mass of particle
N	4035	total number of particles
N_1	1020	number of particles in dropping object
N_2	3015	number of particles in resting object
$\mathbf{v}$	(0, 30, 0)	initial velocity of dropping object
f_{cut}	2.5*σ	cutoff radius of force
δt	0.0005	time step
ntimes	12000	number of time steps
neigh_every	20	interval of constructing the neighbor list
has_gravity	0	gravity not available
t_target	100.00	the constant temperature in the simulation box

Table 5.3.: Initial parameters for the simulation of the `OBCCOLLISION` scenario.



(a) t=0.0 (b) t=2.0

(c) t=4.0 (d) t=6.0

Figure 5.5.: Simulation pictures of the `OBCCOLLISION` scenario in QEMU.

The strong scaling experiment is also an important part. In Figure 5.6, the performance metrics of the simulation with 9000 iterations and 7 iMPI adaptations in `EVENLY` scenario are shown. The total execution time is split into several parts. The force time means the time used to compute the potential energy, pressure, temperature, and force. The communication time is for the communication between processes. As Figure 5.6 shows, the communication time becomes longer as the number of processes grows. It leads to an important overhead in the parallel computing of the simulation. The neighboring time is for building the neighbor lists of particles, which are used to compute the potential energy and force between particles. The optimal number of processes for this simulation is 3 according to the metrics. More processes may lead to higher overhead of communication and synchronization between processes. The other time refers to the time that can not be sorted into a specific variety. In each iMPI adaptation, one process is inserted into the current group. Compared with the total execution time, the invasive time shown in Figure 5.7 is quite short. For instance, the time interval used to insert one process into the communicator with 3 processes is only 0.0381 seconds in QEMU while the total execution time is 26.4917 seconds. When the program boots, the process is not initialized by the resource manager, thus the invasive time for "New number of processes = 1" is 0. The invasive time length is almost the same in each adaptation. The performance metrics of the `RBCCOLLISION` and `OBCCOLLISION` scenarios are in Figure 5.8 and Figure 5.9. The simulation of the `RBCCOLLISION` scenario is executed for 9000 iterations and 7 iMPI adaptations. The simulation of the `OBCCOLLISION` scenario is executed for 9000 iterations and 7 iMPI adaptations. Both of them have the optimal number of processes to get the best performance, and the iMPI overhead is inevitable. The initial particles are distributed in the same positions for different numbers of processes and they are not located in the simulation box evenly, thus the load balancing is not realized and it can lower the execution efficiency.

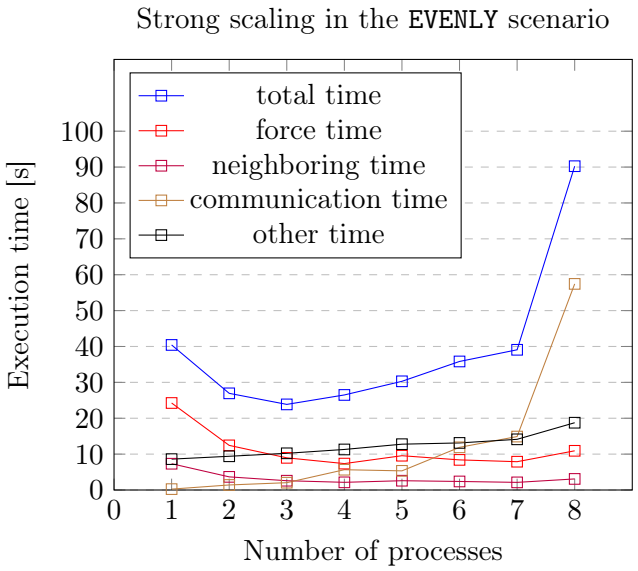


Figure 5.6.: Performance metrics of strong scaling in the `EVENLY` scenario.

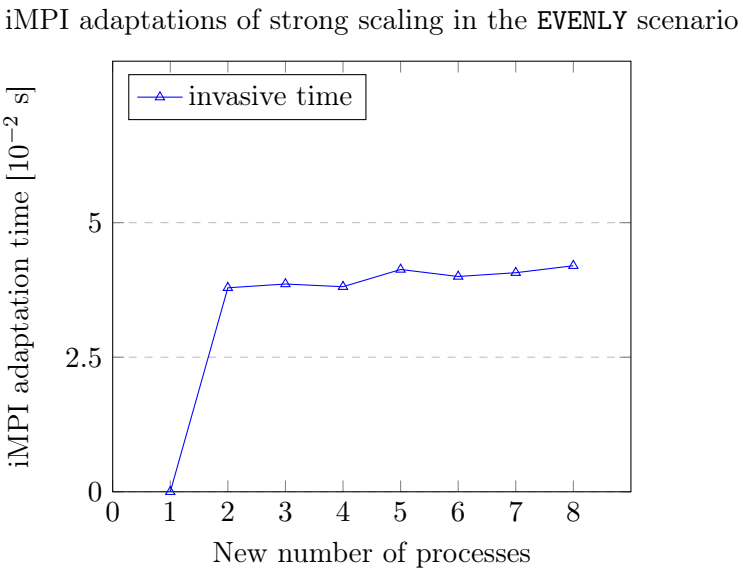


Figure 5.7.: Invasive time of consecutive iMPI adaptations in the `EVENLY` scenario.

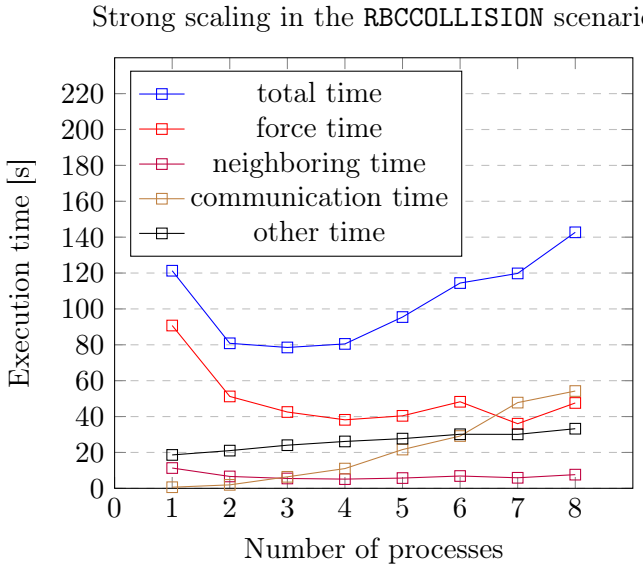


Figure 5.8.: Performance metrics of strong scaling in the `RBCCOLLISION` scenario.

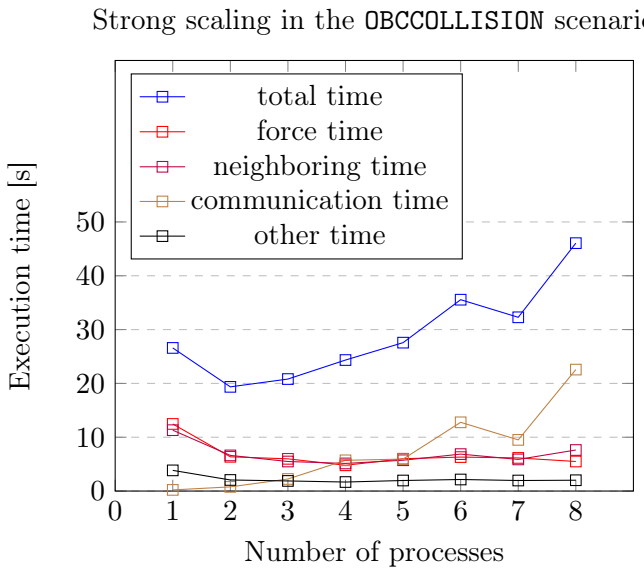


Figure 5.9.: Performance metrics of strong scaling in the `OBCCOLLISION` scenario.

5.2. Experiments in bare metal

5.2.1. Evaluation setup

After the verification in virtual machines, it is time to do the same experiment in the hardware. The hardware in the experiment consists of two computers shown in Figure 5.10. Linux operating system is installed in the client computer. The visualization application can be executed on it. The client computer is configured with Dynamic Host Configuration Protocol (DHCP) and Trivial File Transfer Protocol (TFTP), and it is connected with the server computer which executes the binary file of *i*miniMD. The client computer is booted in the Preboot Execution Environment (PXE) mode, which is the model to boot the client connected with the server via a local network. The server computer is based on x86_64 architecture and can execute 64-bit binary files. It owns two Non-Uniform Memory Access (NUMA) nodes and 88 CPUs. One process in OctoPOS runs in one CPU. The server computer gets an IP address assigned by DHCP and receives the binary file of *i*miniMD transmitted by TFTP from the client computer. After that, OctoPOS starts and *i*miniMD is executed. The visualization application can receive the simulation data from OctoPOS through TCP.

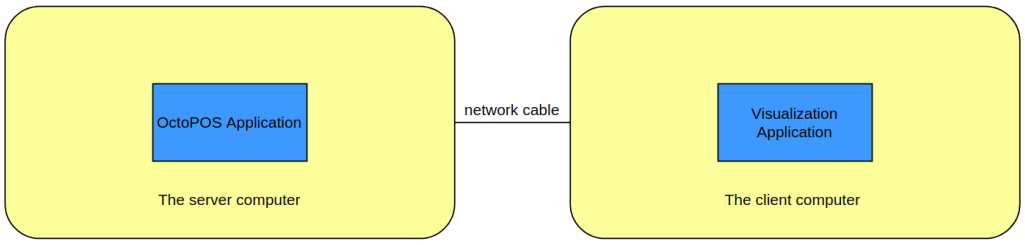
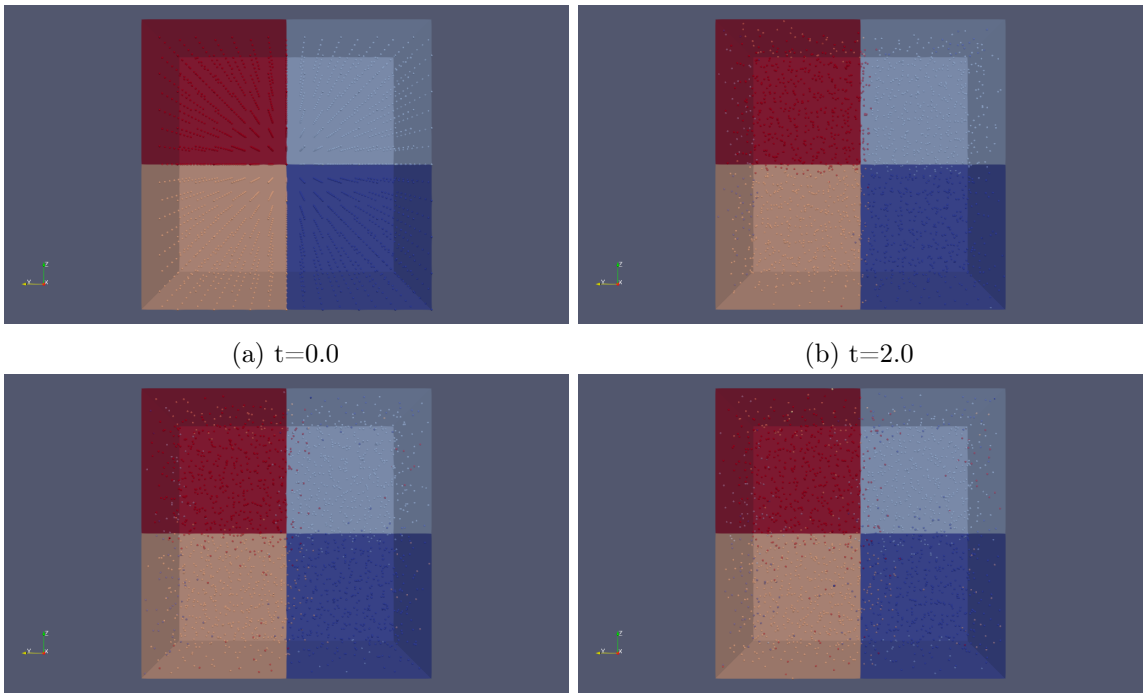


Figure 5.10.: Diagram of the hardware experiment setup.

The experiment setup of OctoPOS, *i*miniMD, and the visualization application remains the same as that of the QEMU experiment. There is no need to virtualize the hardware anymore. The experiment also consists of the visualization experiment for the three simulation scenarios and the strong scaling experiment.

5.2.2. Results

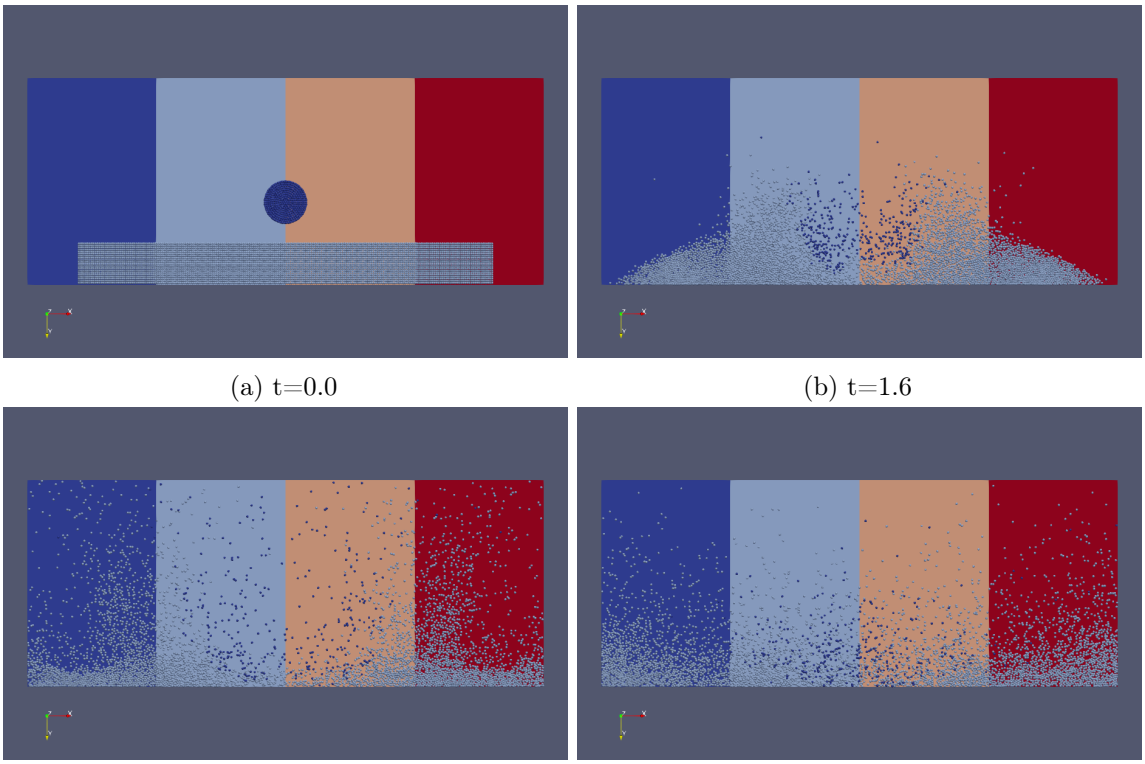
The visualization experiment uses the initial configurations in Table 5.1, Table 5.2, and Table 5.3. The visualization results are almost the same with those in the QEMU experiment. Although the same initial configuration is used, the slight difference of the simulation comes from the random generation of the initial positions and velocities of particles. The visualization scenarios are shown in Figure 5.11, Figure 5.12, and Figure 5.13.



(a) t=0.0 (b) t=2.0

(c) t=10.0 (d) t=20.0

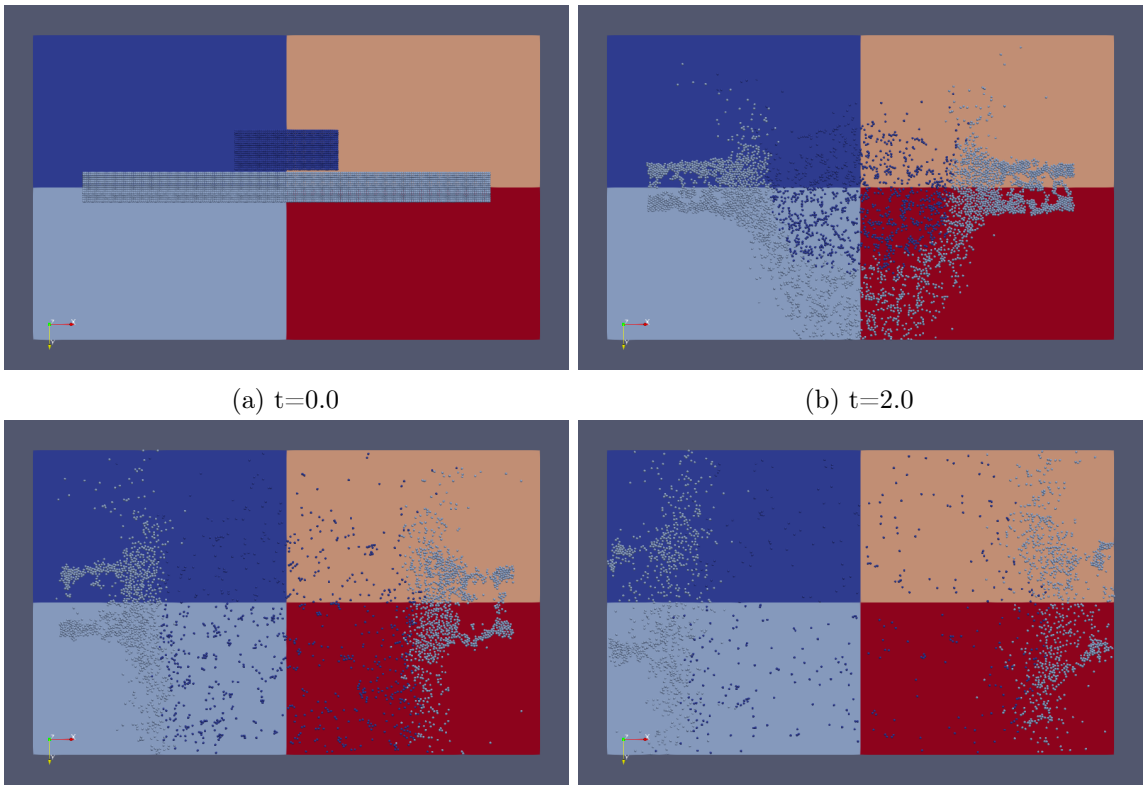
Figure 5.11.: Simulation pictures of the `EVENLY` scenario in hardware.



(a) t=0.0 (b) t=1.6

(c) t=3.2 (d) t=8.0

Figure 5.12.: Simulation pictures of the `RBCCOLLISION` scenario in hardware.



(a) t=0.0 (b) t=2.0

(c) t=4.0 (d) t=6.0

Figure 5.13.: Simulation pictures of the `OBCCOLLISION` scenario in hardware.

The performance metrics of these three experiment are described in Table 5.4, Table 5.5, and Table 5.6. As shown in Figure 5.14, Figure 5.15, and Figure 5.16, it is obvious that the IO time which means the data transmission time makes up the main portion of the execution time.

Total time [s]	**Other time [s]**	**Force time [s]**	**Communication time [s]**	**Neighboring time [s]**	**IO time [s]**
296.835766	17.843868	24.216906	15.989113	7.099721	231.686158

Table 5.4.: Visualization experiment performance metrics of the `EVENLY` scenario.

Total time [s]	**Other time [s]**	**Force time [s]**	**Communication time [s]**	**Neighboring time [s]**	**IO time [s]**
321.066898	23.875709	22.202931	8.225496	3.799221	262.963541

Table 5.5.: Visualization experiment performance metrics of the `RBCCOLLISION` scenario.

Total time [s]	**Other time [s]**	**Force time [s]**	**Communication time [s]**	**Neighboring time [s]**	**IO time [s]**
179.602533	41.828064	2.139494	3.467272	1.309806	129.857897

Table 5.6.: Visualization experiment performance metrics of the `OBCCOLLISION` scenario.

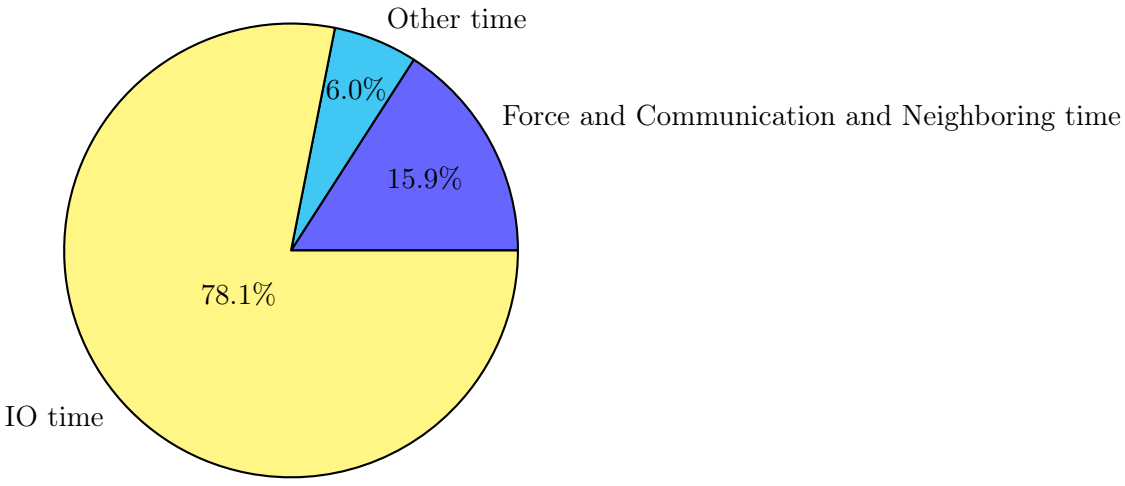


Figure 5.14.: Percentage of time used in each part of the visualization experiment of the `EVENLY` scenario.

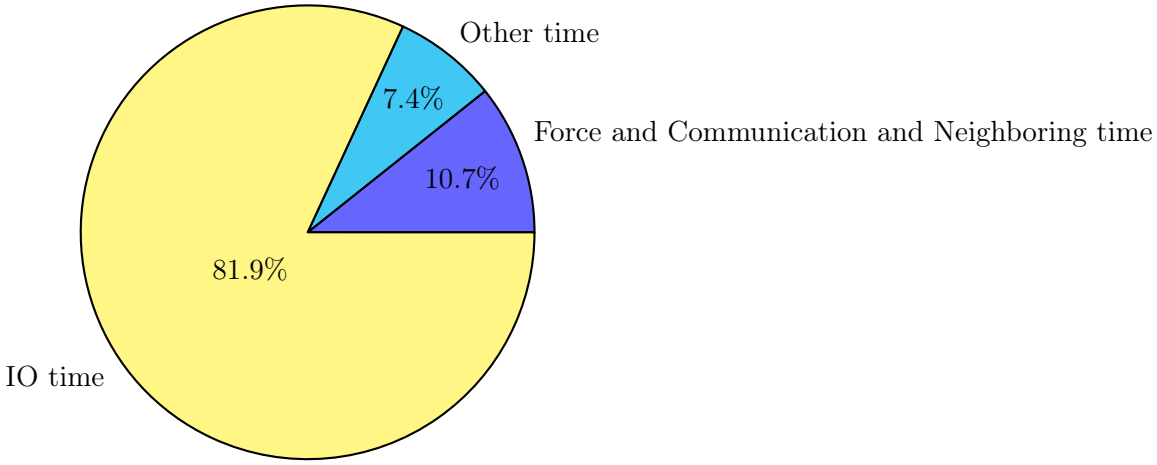


Figure 5.15.: Percentage of time used in each part of the visualization experiment of the `RBCCOLLISION` scenario.

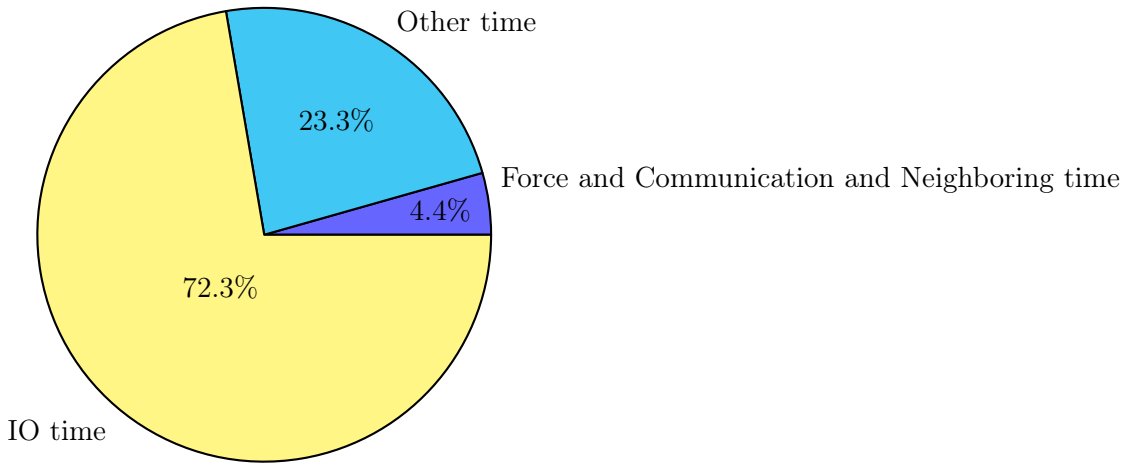


Figure 5.16.: Percentage of time used in each part of the visualization experiment of the `OBCCOLLISION` scenario.

The total execution time of the strong scaling experiment for `EVENLY` is shown in Table 5.7. The simulation configured in Table 5.1 is executed for 8 times, and 9000 iterations are simulated for each time. Each strong scaling execution includes 7 times of iMPI adaptations, and the execution should be completed for 3 times, in order to get the average execution time. The same setup is applied to the `RBCCOLLISION` and `OBCCOLLISION` scenarios. Figure 5.17 shows the change of total execution time of the three scenarios according to different numbers of processes. When the number of processes is 5, the execution time of `EVENLY` is the shortest. The overhead of MPI synchronization and communication makes the simulation time longer in the other numbers of processes.

Number of process	**The time of the 1st execution [s]**	**The time of the 2nd execution [s]**	**The time of the 3rd execution [s]**	**Average execution time [s]**
1	30.054693	30.043211	30.048298	30.048734
2	17.558879	17.551351	17.546096	17.552109
3	13.986825	13.950851	13.933834	13.957170
4	14.238795	14.497033	14.578159	14.437996
5	12.232582	12.085666	12.054135	12.124128
6	14.868155	15.255555	15.201959	15.108556
7	15.123108	16.109290	16.914537	16.048978
8	20.029437	20.182976	20.604569	20.272327

Table 5.7.: Strong scaling experiment result of the `EVENLY` scenario.

The total execution time of the strong scaling experiment for `RBCCOLLISION` is shown in Table 5.8. The simulation configured in Table 5.2 is also executed for 8 times, and 9000 iterations are simulated for each time. The optimal number of processes is 7. The initial positions of particles are fixed and one process is in charge of the particles located in its local domain, thus the workload could not be distributed to each process evenly. The workload distribution may be more even when the number of processes is 7, and it can to some extent explain why the optimal number of processes is 7. By contrast, the initial particles are distributed to each process more evenly in the `EVENLY` scenario, thus its change of execution time with different numbers of processes is more smooth. The result of the total execution time for `OBCCOLLISION` is shown in Table 5.9. The optimal number of processes is 4. The problem of imbalanced workload distribution also exists.

Number of process	**The time of the 1st execution [s]**	**The time of the 2nd execution [s]**	**The time of the 3rd execution [s]**	**Average execution time [s]**
1	48.031937	48.026710	48.032957	48.030535
2	26.191915	26.175602	26.183919	26.183812
3	23.577565	23.520012	23.539988	23.545855
4	19.910044	19.830752	19.794636	19.845144
5	20.217733	20.178668	19.935885	20.110762
6	26.130325	26.438657	26.335962	26.301648
7	19.220771	19.250932	18.738457	19.070053
8	22.528887	23.189829	22.056508	22.591741

Table 5.8.: Strong scaling experiment result of the `RBCCOLLISION` scenario.

Number of process	**The time of the 1st execution [s]**	**The time of the 2nd execution [s]**	**The time of the 3rd execution [s]**	**Average execution time [s]**
1	18.892991	18.891494	18.891858	18.892114
2	11.460446	11.512442	11.512897	11.495262
3	11.505218	11.399156	11.441426	11.448600
4	10.700879	11.025259	11.067419	10.931186
5	12.033263	12.604530	12.112412	12.250068
6	13.614653	14.469517	13.554784	13.879651
7	12.357655	12.559938	12.031661	12.316418
8	14.245429	16.185356	14.734550	15.055112

Table 5.9.: Strong scaling experiment result of the `OBCCOLLISION` scenario.

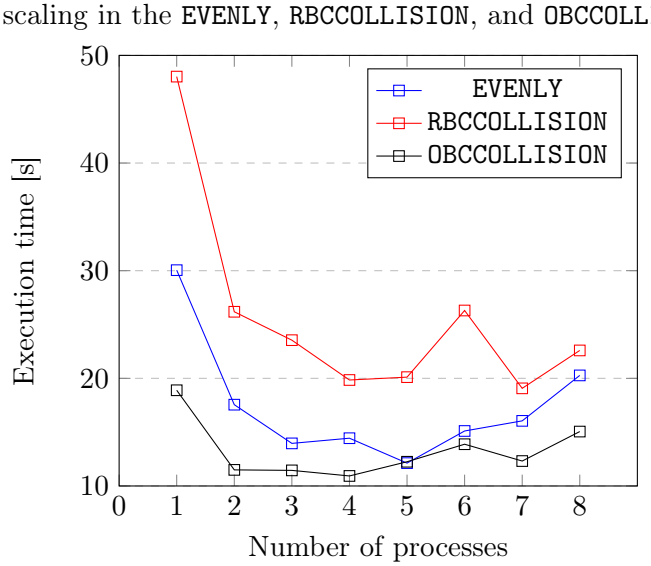


Figure 5.17.: Total execution time of strong scaling in the EVENLY, RBCCOLLISION, and OBCCOLLISION scenarios.

6. Conclusions

6.1. Achievements

The basic three objectives are described in Chapter 1. The first objective is to develop *i*miniMD. With the limitation of libraries in OctoPOS, a reconstruction of miniMD was already completed. After the reconstruction, new simulation scenarios, data transmission, and performance metrics were added to the code. In order to help the application dynamically adjust the resources used, the iMPI content was developed in the code. The second objective is to visualize *i*miniMD. The product for this is a C++ application, which is based on OpenGL and MPI. Real-time data through TCP can be received in this application, and the data is visualized through the OpenGL interfaces. For better analysis of the simulation, the data is written into vtk files, which can be visualized and analyzed in software such as ParaView. The third objective is the experiment. The first experiment is to visualize the default scenario in miniMD and two collision cases. The second experiment is to simulate one fixed problem with different numbers of processes. With the help of iMPI, the number of processes can be changed during the execution, therefore a strong scaling experiment can be completed with one execution of the code. The performance metrics are generated after each execution.

As is shown in Chapter 3, my main work in miniMD is to convert the code to a C-based miniMD by the interfaces in OctoPOS and to develop new features. It is not easy to do the reconstruction because of the language gap and limited functions available in OctoPOS. New features are based on the great understanding of miniMD. They include two collision scenarios, the invasive functionality based on iMPI, the data transmission functionality based on the "octo_tcp" library, and the performance metrics based on `MPI_Wtime`. Features of miniMD impossible to be implemented in OctoPOS such as OpenMP are not reproduced in *i*miniMD.

Without the limitation of libraries in OctoPOS, the visualization application can use the C and C++ libraries. I needed to design a structure of the application. There are only two processes in the application, one of which is for receiving the simulation data and the other of which is for visualizing the simulation data. My previous design of the visualization application is to use the same number of processes as that in *i*miniMD to receive the data. One process is responsible for receiving the data transmitted from one process in OctoPOS. In this case, each process in the visualization application possesses its own socket. Correspondingly, each process in *i*miniMD also has a socket. This design is complex and is not easy to be implemented. Another disadvantage is that the number of the processes in OctoPOS should be known prior to the execution of the visualization application launched by the "mpirun" command. In the strong scaling experiment, the number of processes in the "mpirun" command needs to be adjusted before each iMPI adaptation of *i*miniMD, thus the visualization application needs to be restarted for several times. In the 2-process design, a fixed number of processes makes the implementation of the source code easier, and it

also makes the execution more convenient. A strong scaling experiment including several consecutive iMPI adaptations can be completed with only one execution of the visualization application. The real-time displaying of simulation in the GLUT window helps researchers to find the potential problems during the simulation process. The visualization application can write the simulation data, the initial configurations, and performance metrics to files. With these archived files, the history simulation can be reproduced in ParaView, and the performance metrics data can be used to analyze one simulation after its execution.

The experiment part is shown in Chapter 5. It is convenient to test the project in QEMU before the experiment in hardware. Modification of the code can be completed according to the result in QEMU. The visualization experiment presented the simulation correctly and perfectly, which can verify the effect of *i*miniMD directly. The strong scaling experiment gave a method to verify the iMPI functionality in OctoPOS and showed the performance metrics efficiently.

6.2. Future work

6.2.1. New content in *i*miniMD

The load balancing between processes is not supported in *i*miniMD, which may lower the execution efficiency because some processes can deal with more work than the other processes. In miniMD, one process is responsible for an equal-volume local domain and in charge of the local particles. If the particles are not distributed evenly, the workload varies in different processes. Some processes are responsible for much more work than others while the other processes are almost idle in the meantime. For example, the numbers of particles in different processes are not close to each other in Figure 5.12. One possible method is that one process is responsible for a group with the same number of particles, but it will change the logic of source code a lot because the management of particles is based on the spatial decomposition parallelism. However, it really helps to improve the execution efficiency.

New simulation features can be developed in *i*miniMD. For example, only the LJ interaction is supported in *i*miniMD. The EAM interaction can be added to the project, and it can help to simulate the scenario in metals and alloys. A long-range force feature is also a great complement to construct more scenarios. Three boundary conditions are supported now in *i*miniMD, and new boundary conditions such as the inflow boundary conditions can be implemented.

6.2.2. New content in the visualization application

There exist many potential features to be added to the visualization application with the help of C++ libraries. The main work could be the improvement of the visualization effect. The real-time window in OpenGL can not be scaled, rotated, or moved. Developers can choose more versatile libraries to realize a more flexible window. More features of particles can be presented in the window. It only includes the location and color now, but the velocity, type, or energy of particle can be shown in the particle. For example, different types of particles can be shown by objects with different shape, texture, color, and spatial

structure. Additional work could be to print more simulation information in the window. It includes the iteration information, the size of the simulation box, and the number of particles. More information such as the temperature, pressure, potential, and dynamic energy of the simulation box can also be presented. Besides, the display of the simulation information can be dynamic in different levels. With the dynamic and hierarchical display of information, information can be shown or hidden with specific actions of the user. For example, if the user clicks one particle with the mouse, the physical properties of this particle are shown. If the user clicks the domain of one process, the number of local particles and other information of this process appear in the window. By clicking the particle or domain again, the information is hidden. This feature could be realized by a monitoring function which keeps listening to the registered events and calls one callback function if one of these events happens.

List of Figures

List of Tables

List of Codes

Bibliography

[1] Comprés Ureña and Isaías Alberto. *Resource-elasticity support for distributed memory hpc applications*. PhD thesis, Technische Universität München, 2017.

[2] M Griebel S Knapek and G Zumbusch. Numerical simulation in molecular dynamics, numerics, algorithms, parallelization, applications, 2007.

[3] Michael P Allen et al. Introduction to molecular dynamics simulation. *Computational soft matter: from synthetic polymers to proteins*, 23(1):1–28, 2004.

[4] Peter Pacheco and Matthew Malensek. *An introduction to parallel programming*. Morgan Kaufmann, 2021.

[5] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive computing: An overview. *Multiprocessor System-on-Chip*, pages 241–268, 2011.

[6] Sascha Roloff, Frank Hannig, and Jürgen Teich. *Modeling and Simulation of Invasive Applications and Architectures*. Springer, 2019.

[7] PAUL CROZIER and STEVEN PLIMPTON. minimd v. 1.0. Technical report, Sandia National Laboratories, 2009.

[8] David Patterson. The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges. *Nvidia Whitepaper*, 47, 2009.

[9] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[10] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Octopos: A parallel operating system for invasive computing. In *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*, pages 9–14. Citeseer, 2011.

[11] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Tappan Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue-hua Dai, et al. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[12] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, volume 27, 2008.

[13] David Wentzlaff and Anant Agarwal. Factored operating systems (fos) the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[14] Mohammad Al Faruque, Janmartin Jahn, Thomas Ebi, and Jörg Henkel. Runtime thermal management using software agents for multi-and many-core architectures. *IEEE Design & Test of Computers*, 27(6):58–68, 2010.

[15] Hyacinth S Nwana. Software agents: An overview. *The knowledge engineering review*, 11(3):205–244, 1996.

[16] David W Walker and Jack J Dongarra. Mpi: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[17] Dieter Kranzlmüller, Dieter Kranzlmuller, Peter Kacsuk, and Jack Dongarra. Recent advances in parallel virtual machine and message passing interface: 11th european pvm/mpi users' group meeting, budapest, hungary, september 19-22, 2004, proceedings. 2004.

[18] Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wasniewski. *Parallel Processing and Applied Mathematics: 6th International Conference, PPAM 2005, Poznan, Poland, September 11-14, 2005, Revised Selected Papers*, volume 3911. Springer, 2006.

[19] Jeffrey M Squyres and Andrew Lumsdaine. The component architecture of open mpi: Enabling third-party collective algorithms. In *Component Models and Systems for Grid Applications*, pages 167–185. Springer, 2005.

[20] Richard L Graham, Galen M Shipman, Brian W Barrett, Ralph H Castain, George Bosilca, and Andrew Lumsdaine. Open mpi: A high-performance, heterogeneous mpi. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9. IEEE, 2006.

[21] Richard L Graham, Brian W Barrett, Galen M Shipman, Timothy S Woodall, and George Bosilca. Open mpi: A high performance, flexible implementation of mpi point-to-point communications. *Parallel Processing Letters*, 17(01):79–88, 2007.

[22] Nathan Hjelm. Optimizing one-sided operations in open mpi. In *Proceedings of the 21st European MPI Users' Group Meeting*, pages 123–124, 2014.

[23] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[24] William Gropp. Mpich2: A new start for mpi implementations. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 7–7. Springer, 2002.

[25] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[26] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, volume 1, pages 10–pp. IEEE, 2006.

[27] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in mpich2 using the nemesis communication subsystem. *Parallel Computing*, 33(9):634–644, 2007.

[28] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H Katz. Improving tcp/ip performance over wireless networks. In *Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 2–11, 1995.

[29] Behrouz A Forouzan. *TCP/IP protocol suite*. McGraw-Hill Higher Education, 2002.

[30] Randall J LeVeque, David L George, and Marsha J Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, 2011.

[31] Alexander Pöppl and Michael Bader. Swe-x10: An actor-based and locally coordinated solver for the shallow water equations. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, pages 30–31, 2016.

[32] Sascha Roloff, Alexander Pöppl, Tobias Schwarzer, Stefan Wildermann, Michael Bader, Michael Glaß, Frank Hannig, and Jürgen Teich. Actorx10: An actor library for x10. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, pages 24–29, 2016.

[33] Jenn Huei Lii and Norman L Allinger. Molecular mechanics. the mm3 force field for hydrocarbons. 3. the van der waals' potentials and crystal data for aliphatic and aromatic hydrocarbons. *Journal of the American Chemical Society*, 111(23):8576–8582, 1989.

[34] Ulrich Welling and Guido Germano. Efficiency of linked cell algorithms. *Computer Physics Communications*, 182(3):611–615, 2011.

[35] Michael I Baskes. Modified embedded-atom potentials for cubic materials and impurities. *Physical review B*, 46(5):2727, 1992.

[36] Murray S Daw and Michael I Baskes. Semiempirical, quantum mechanical calculation of hydrogen embrittlement in metals. *Physical review letters*, 50(17):1285, 1983.

[37] Murray S Daw and Michael I Baskes. Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals. *Physical Review B*, 29(12):6443, 1984.

[38] Kai Kadau. *Molekulardynamik-Simulationen von strukturellen Phasenumwandlungen in Festkorpern, Nanopartikeln und ultrad unnen Filmen*. PhD thesis, Ph. D. thesis, Gerhard-Mercator-Universitat Duisburg, 2001.

[39] E Clementi and C Roetti. Atomic and nuclear data tables, vol. 14, 1974.

[40] Aidan P Thompson, H Metin Aktulga, Richard Berger, Dan S Bolintineanu, W Michael Brown, Paul S Crozier, Pieter J in't Veld, Axel Kohlmeyer, Stan G Moore, Trung Dac Nguyen, et al. Lammps-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271:108171, 2022.

[41] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[42] Loup Verlet. Computer" experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.

[43] Steve Plimpton, Roy Pollock, and Mark Stevens. Particle-mesh ewald and rrespa for parallel molecular dynamics simulations. In *PPSC*. Citeseer, 1997.

[44] H Carter Edwards and Christian R Trott. Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 18–24. IEEE, 2013.

[45] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.