Application of parallel computing in fluid simulation

MSc. High Performance Computing

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Author: Lin Li

Student ID: 20304760

Supervisor: Mike Peardon & Richie Morrin

School of Mathematics

Faculty of Engineering, Mathematics and Science

Trinity College Dublin, University of Dublin

September 17, 2021

Declaration

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have completed the Online Tutorial in avoiding plagiarism 'Ready, Steady, Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

Lin Li

September 17, 2021

Acknowledgements

I would like to thank my supervisor Mike Peardon and Richie Morrin for their continuous guidance. Their weekly meetings and emails have been very beneficial to me. Jose Refojo has also been very helpful to me. They are like a lighthouse in the sea, encouraging me and guiding me in the most difficult time. I would also like to thank all the teachers and students of High Performance Computing.

Thanks to China Scholarship Council and Department of Education and Skills in Ireland for their support.

Abstract

As people's living standards improve and science and technology develop, parallel computing becomes more and more important. Fluid simulation is one of the most relevant and important operations in people's lives and requires a lot of floating point operations. It plays a vital role in aerodynamic research such as car modelling and aircraft flow design. To explore the reduction of computing time in fluid simulation, I have used MPI to parallelize the algorithm and test it in a cluster, using a 2D oblique excitation reflection as an example. To optimize performance, I made three versions of blocking communication, RMA, and non-blocking communication, and analyzed them in detail. In addition, I have also accelerated the algorithm using CUDA and tested it to compare how well different parallel methods accelerate the algorithm.

There is little performance difference between the MPI versions of the three different communication methods. But the acceleration of the CUDA version is a significant improvement compared to them. I think GPU computing like CUDA is going to shine in the future.

All related software is located at https://github.com/LinLi-max/summer_project.git.

Keywords: MPI, CUDA, fluid simulation, parallel computing

Contents

Lists of Tables

Lists of Figures

1. Introduction

Fluid simulation is always a scenario where floating point computing power is in high demand. The importance of fluid simulation is due not only to its inherent importance in flow physics but also to its strong engineering applications. It is also an important element in the improvement of people's living standards. This paper uses two-dimensional oblique excitation reflections as a case for parallel computing.

Excitation waves are widely present in nature and human scientific and technological activities. It is one of the most characteristic and fundamental physical phenomena of gas dynamics and is an important physical process capable of inducing vortices within the flow, exhibiting strong intermittent and non-linear aerodynamic physical characteristics. During the propagation of a surge wave, when the propagation conditions change, including the appearance of walls, changes in the properties of the flow medium, the presence of other flow field structures, etc., the surge wave will interact with it and reflect, forming a surge-reflected flow field structure (Peng J, et,2017) [1].

In this paper, I will perform parallel acceleration of the two-dimensional oblique excitation reflection problem on both CPU and GPU platforms respectively, and compare their performance differences. On the CPU platform, I use MPI to implement parallelism, which is described in detail in Chapter 4. MPI (Message-Passing Interface) is a specification for message-passing library interfaces. This definition is significant in every way. MPI primarily addresses the message-passing parallel programming model, in which data is transported from one process's address space to another process's address space via cooperative operations on each process. In collective operations, remote-memory access operations, dynamic process generation, and parallel I/O, extensions to the "traditional" message-passing architecture are offered.

On the GPU platform, I use CUDA for parallel computing on an NVIDIA graphics card, which is implemented in Chapter 6. CUDA (Compute Unified Device Architecture), is a computing platform from graphics card manufacturer NVIDIA. CUDA is a general-purpose parallel computing architecture introduced by NVIDIA that enables GPUs to solve complex computational problems. It consists of the CUDA Instruction Set Architecture (ISA) and the GPU's internal parallel computing engine. CUDA programming model makes it very easy for developers to perform complex calculations, significantly reducing development time and increased efficiency.

In the MPI version of the implementation, I compared the performance differences between the blocking communication, non-blocking communication, and RMA versions. In the CUDA version of the implementation, I have tried to unify the memory model. The project is concluded in Chapter 7 with the following summary.

2. Fluid simulation algorithm

2.1 Algorithm introduction

2.1.1 Two-dimensional oblique excitation reflection problem

The problem of reflection of an oblique excitation wave on a plane rigid wall is a two-dimensional compressible inviscid flow problem with an analytical solution. It is solved numerically using a two-step difference format with second-order accuracy (Zhang, 2010) [2].

An oblique excitation wave is projected at an angle of incidence onto a plane rigid wall and collides with this plane rigid wall and is reflected on the rigid wall, forming an incident and reflected wave, as shown in Figure 2.1.

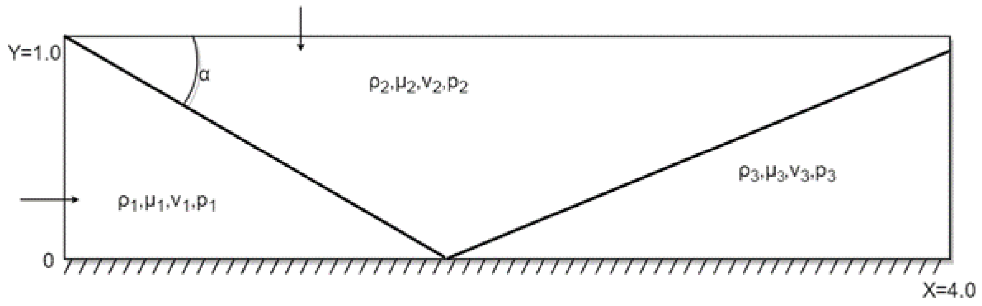


Figure 2.1: Schematic diagram of the excitation reflection problem.

2.1.2 Basic equations, initial and boundary conditions

Let the gas be ideal. The problem of reflection of an oblique excitation wave on a plane rigid wall can be described mathematically by a two-dimensional system of Euler equations for compressible inviscid flow. As Attiya and Kishk proposed in 2006, the two-dimensional system of equations with a dimension of one is showing below [3].

$$\frac{\partial \boldsymbol{u}}{\partial t}+\frac{\partial \boldsymbol{f}}{\partial x}+\frac{\partial \boldsymbol{g}}{\partial y}=\boldsymbol{0} \tag{2.1}$$

$$\boldsymbol{u}=\begin{pmatrix}\rho\\ \rho u\\ \rho v\\ E\end{pmatrix},\ \boldsymbol{f}=\begin{pmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ (E+p)u\end{pmatrix},\ \boldsymbol{g}=\begin{pmatrix}\rho v\\ \rho uv\\ \rho v^2+p\\ (E+p)v\end{pmatrix} \tag{2.1a}$$

$$p=(\gamma-1)\rho e=(\gamma-1)\left[E-\frac{1}{2}\rho\left(u^2+v^2\right)\right] \tag{2.1b}$$

Here ρ, μ, v, p and E are the density, the velocity components in the x and y directions, the pressure and the total energy per unit volume respectively (Li, 2020) [4].

At the time when the oblique excitation wave just meets the plane rigid wall, the initial flow field distribution can be calculated according to the Mach number of the excitation wave and the angle of incidence, as follows.

The left boundary (x = 0, y = 0 - 1.0) is a uniform incoming flow condition.

$$u_1 = 2.90, \quad v_1 = 0.0, \quad \rho_1 = 1.0, \quad p_1 = 0.71429 \tag{2.2a}$$

Upper boundary (x = 0 - 4, y = 1.0) for uniform incoming flow conditions.

$$\begin{aligned} & u_2 = 2.61934, v_2 = -0.50632 \\ & \rho_2 = 1.69997, \; p_2 = 1.52819 \end{aligned} \tag{2.2b}$$

Boundary conditions: right boundary (x = 4.0, y = 1.0) for free output conditions.

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} = 0 \tag{2.2c}$$

The lower boundary is a rigid wall with normal velocity on a horizontal rigid wall.

$$v_{\mathrm{w}} = 0 \tag{2.2d}$$

2.1.3 Second-order precision Lax-Wendroff two-step differential format

The problem of reflection of an oblique excitation on a plane rigid wall is a two-dimensional inviscid flow problem that can be split into two one-dimensional flow problems using an operator splitting method (Chow, 1973) [5]. After splitting, each one-dimensional flow problem is computed numerically using a second-order accurate Lax-Wendroff two-step difference format. To minimise the human factor, the following splitting calculation steps can be used.

$$\boldsymbol{u}^{n+1} = L_x\left(\frac{1}{2}\Delta t\right) L_y\left(\Delta t\right) L_x\left(\frac{1}{2}\Delta t\right)\boldsymbol{u}^n \tag{2.3}$$

Second-order accuracy Lax-Wendroff two-step differential format for each of the one inviscid flows.

$$\begin{aligned} & \boldsymbol{u}_{j+\frac{1}{2}}^{n+\frac{1}{2}} = \frac{1}{2}\left(\boldsymbol{u}_{j+1}^n + \boldsymbol{u}_j^n\right) - \frac{1}{2}\frac{\Delta t}{\Delta x}\left[\boldsymbol{f}\left(\boldsymbol{u}_{j+1}^n\right) - \boldsymbol{f}\left(\boldsymbol{u}_j^n\right)\right] \\ & \boldsymbol{u}_j^{n+1} = \boldsymbol{u}_j^n - \frac{\Delta t}{\Delta x}\left[\boldsymbol{f}\left(\boldsymbol{u}_{j+\frac{1}{2}}^{n+\frac{1}{2}}\right) - \boldsymbol{f}\left(\boldsymbol{u}_{j-\frac{1}{2}}^{n+\frac{1}{2}}\right)\right] \end{aligned} \tag{2.4}$$

Computational practice shows that the second-order accuracy Lax-Wendroff two-step difference format does not suppress unphysical oscillations in the vicinity of the excitation. Therefore, a prior artificial viscous filtering method with a switching function must be used in the calculation of the excitation.

$$\bar{\boldsymbol{u}}_{i,j}^n = \boldsymbol{u}_{i,j}^n + \frac{1}{2}\eta\theta\left(\boldsymbol{u}_{i+1,j}^n - 2\boldsymbol{u}_{i,j}^n + \boldsymbol{u}_{i-1,j}^n\right) \tag{2.5}$$

$$\theta = \left|\frac{|\rho_{i+1} - \rho_i| - |\rho_i - \rho_{i-1}|}{|\rho_{i+1} - \rho_i| + |\rho_i - \rho_{i-1}|}\right| \tag{2.5a}$$

$$\eta = \frac{\Delta t\,|a|}{\Delta x}\left(1 - \frac{\Delta t\,|a|}{\Delta x}\right) \tag{2.5b}$$

2.2 Serial code implementation.

2.2.1 Implementation

The serial code is described in Zhang Deliang's "Tutorial on Computational Fluid Dynamics", and I have implemented and optimised the code. The flow chart for the serial code is shown in Figure 2.2.

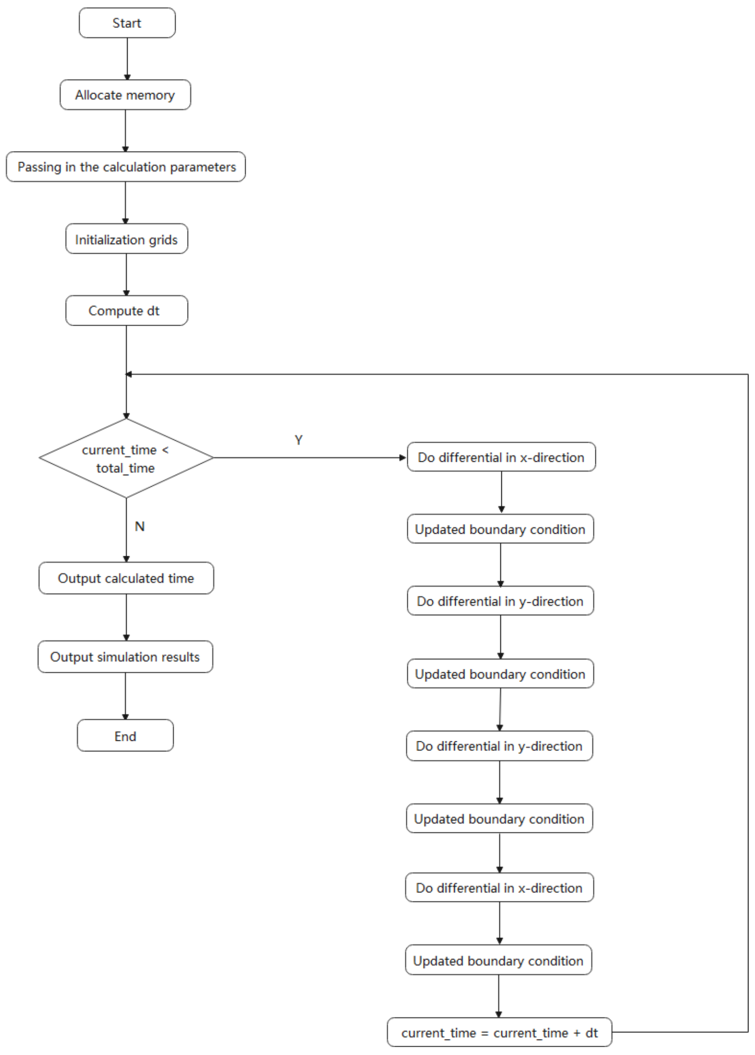

Start
Allocate memory
Passing in the calculation parameters
Initialization grids
Compute dt
current_time < total_time
Y
N
Do differential in x-direction
Updated boundary condition
Do differential in y-direction
Updated boundary condition
Do differential in y-direction
Updated boundary condition
Do differential in x-direction
Updated boundary condition
current_time = current_time + dt
Output calculated time
Output simulation results
End

Figure 2.2: Flow chart for the serial code.

2.2.2 Output

Based on the data output from the serial program, I plotted the density and pressure curves at y=0.5, as well as the pressure and density contour plots. Figures 2.3 - 2.6 show the pressure and density plots at y = 0.5 for grid size x * y = 1200 * 300, as well as the pressure contour and density contour plots, in that order. And figures 2.7 - 2.10 show the pressure and density plots at y = 0.5 for grid size x * y = 1600 * 400, as well as the pressure contour and density contour plots, in that order.

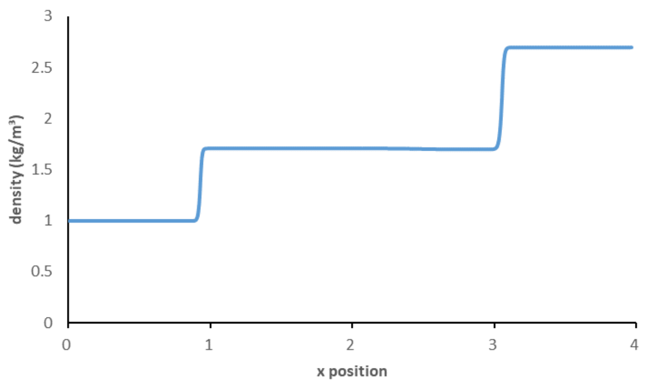


Figure 2.3: Pressure distribution map (serial version at 1200 * 300).

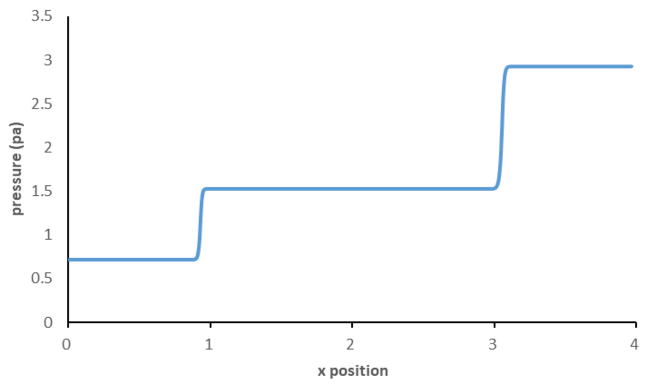


Figure 2.4: Density distribution map (serial version at 1200 * 300).

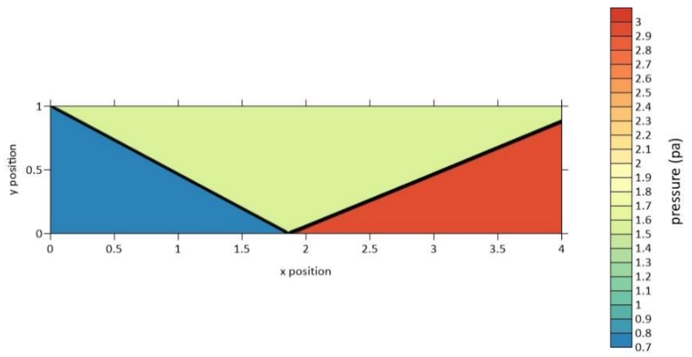


Figure 2.5: Pressure contour map (serial version at 1200 * 300).

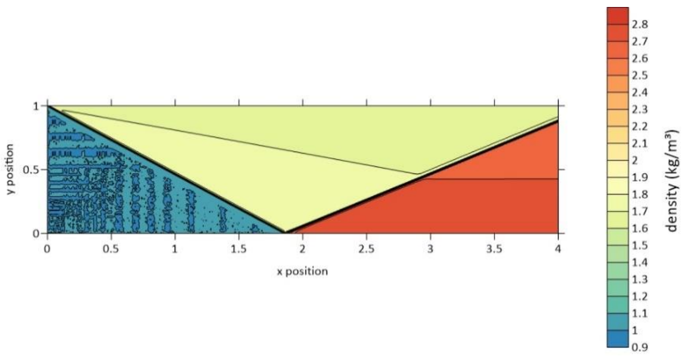


Figure 2.6: Density contour map (serial version at 1200 * 300).

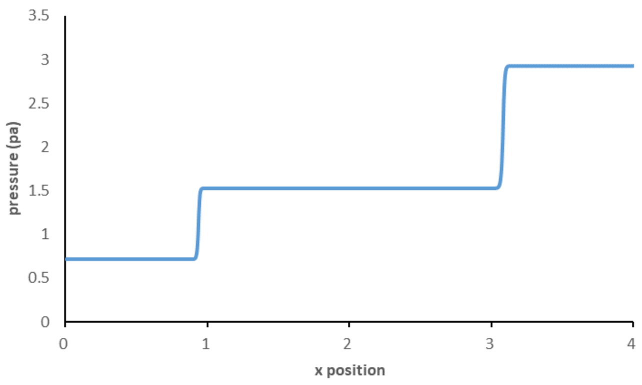


Figure 2.7: Pressure distribution map (serial version at 1600 * 400).

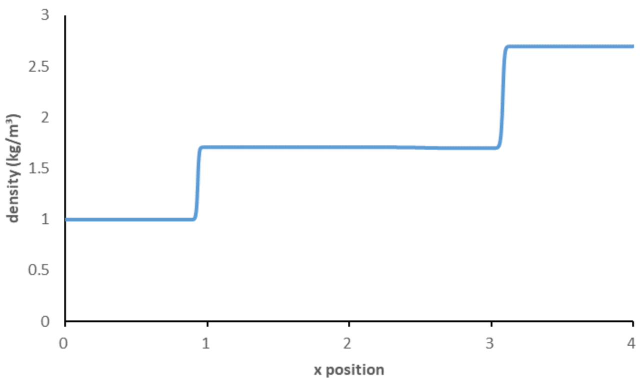


Figure 2.8: Density distribution map (serial version at 1600 * 400).

From the pressure and density maps, it is easy to see that the pressure and density are twice stepped over at almost the same position, which is the incident and reflected airflow. And as the mesh density, i.e. the number of meshes, increases, the step becomes steeper and steeper, indicating that our simulation is becoming more and more detailed and closer to the real situation.

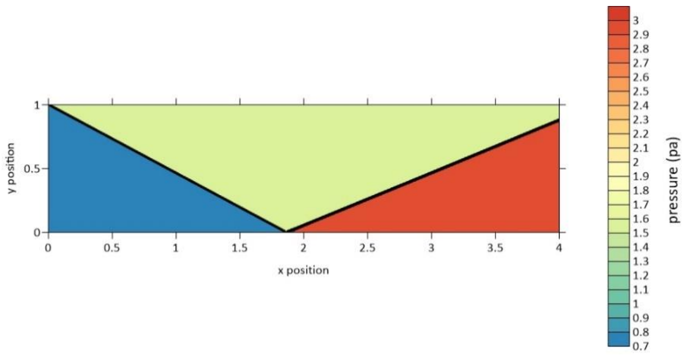


Figure 2.9: Pressure contour map (serial version at 1600 * 400).

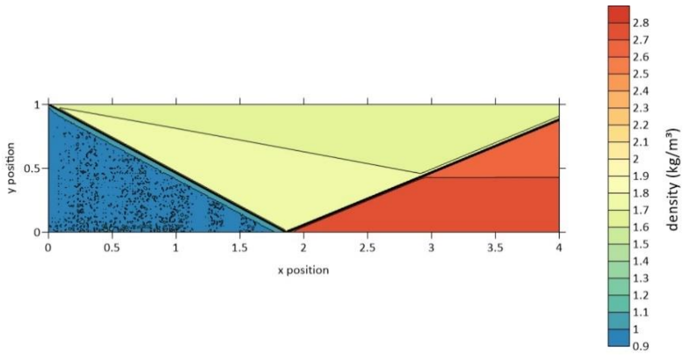


Figure 2.10: Density contour map (serial version at 1600 * 400).

From the isobaric pressure and isobaric density diagrams, we can see the incident and reflected paths of the excitation wave, as well as the variation of physical quantities around the excitation wave. And as the grid density, i.e. the number of grids, increases, both the iso-pressure and iso-density maps become smoother and less noisy. At grid size x * y = 1600 * 400, both images are clear enough to fully reflect the actual physics, so there is no need to increase the number of grids in terms of contour plots, which can be used as the standard answer to verify that the output of the parallel version of the code is correct.

2.2.3 Timing

Table 2.1: Run time of serial code.

grid size	1600 * 400	1200 * 300
time(s)	4250.633	1773.141

The test hardware environment is the same as the MPI version, which is described in detail in later chapters. Table 2.1 shows the running times of the serial code for different numbers of grids. I will use this time as a benchmark against the running time of the parallel code to calculate the speed-up ratio and optimize the performance of the program.

3. MPI introduction

3.1 MPI overview

MPI (Message-Passing Interface) is a message-passing library interface specification. As the MPI 3.1 specification mentioned, all parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Extensions to the “classical” message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O[6].

The main advantages of MPI are high efficiency or good communication performance, ease of portability, and powerful functionality (Coello C, 2006) [7]. The framework structure of one of the MPI programs can be depicted in Figure 3.1.

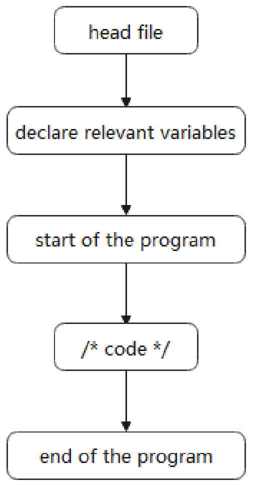


Figure 3.1: MPI program framework diagram.

As can be seen above, different processes can run on different processors (and not just on different cores of the processor, which makes it ideal for the construction of distributed systems, as shown schematically in Figure 3.2.

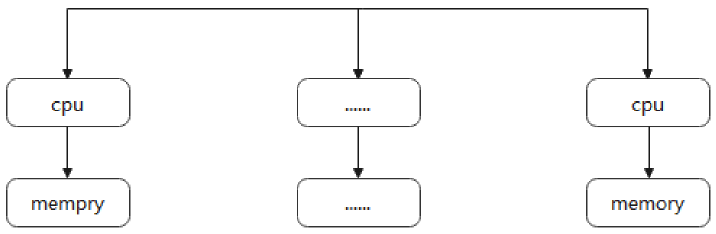


Figure 3.2: Distributed Memory Model.

The body part of the MPI program framework is the core of the entire program, mainly for calculation and communication, and its execution process can be represented in Figure 3.3.

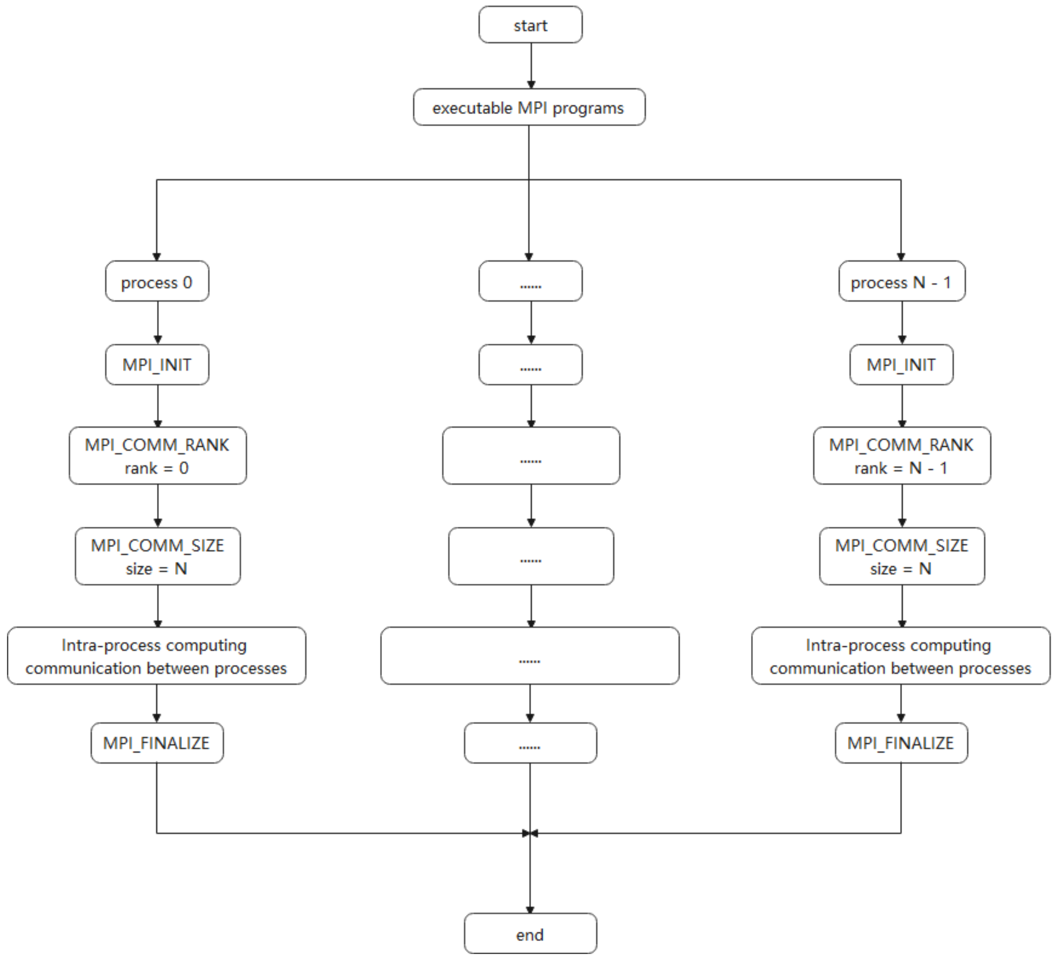


Figure 3.3: MPI program execution flow.

3.2 MPI parallel programming

The MPI has a standard programming methodology and model, and if you are familiar with the basic methodology and model, you will be able to design the parallel program you need (Corne D W, Knowles J D, Oates M J, 2000) [8]. The following section describes the structure of a parallel program and the basic calls to MPI library functions.

3.2.1 The basic program structure of MPI programming

With 128 call interfaces in MPI-1 and 287 in MPI-2, it is difficult to fully grasp this number of calls, and few people will ever use all of them in their programming. However, in theory, all the communication functions of MPI can be implemented using its six basic calls, and with these six calls, the functions of a message-passing parallel program can be implemented. The MPI communicator provides an important encapsulation mechanism for the library and the communication model, the communicator, which includes process groups and communication contexts. A process group is a collection of all processes participating in communication; a communication context provides a relatively independent communication area, where different messages are passed in different contexts without interfering with each other, and a communication context can be a different communication area.

3.2.2 Description of the parameters of the MPI calls

MPI describes the parameters in three ways IN, OUT, INOUT.

- IN (input): the parameter passed to the MPI by the calling part, the MPI is not allowed to modify this parameter other than to use the given parameter.
- OUT (output): the result parameter returned by the MPI to the calling part, the initial value of this parameter has no meaning to the MPI and can be modified at the time of the call.
- INOUT (input/output): the call first passes this parameter to the MPI, which references it, modifies it, and returns the result to the external call, the initial value of this parameter and the return result have meaning.

If a parameter does not change before or after the call, such as the handle of an implicit object, but the object to which the handle refers is modified, this parameter is still specified as OUT or INOUT. The MPI definition avoids the use of INOUT parameters whenever possible, as these are error-prone, especially for scalar parameters. In another case, an argument to an MPI function that is used as IN by some processes executing in parallel and OUT by others executing at the same time is noted as INOUT, even though it is not semantically the input and output of the same call. processes that do not care about the value of the parameter can pass any value to the parameter.

3.2.3 Basic function calls

A complete MPI program consists of the following basic functions.

- MPI initialization. MPI_INIT is the first call of the MPI program and does all the initialization of the MPI program. The function of this statement is to set up the MPI runtime environment. There can only be one MPI_Init statement per program, which returns an error code, with 0 indicating success.
- Get the rank of each processor. By calling the MPI_Comm_rank routine, each process in the group associated with the communication sub can find its sequence number rank. With this flag number, different processes can distinguish themselves from others, enabling parallelism and collaboration between processes.

- Get the number of processes. The call statement MPI_ Comm_ size returns the number of processes used in this program. The return value of the call, size, is the size of the associated group of the default communication sub-MP worker_COMM_ WORLD used by this routine. Each process serial number in the associated group is identified by an integer, counting from zero, and this integer identifier is called the serial number.
- Sending messages. MPI_ Send sends a count of data of type datatype from the send buffer to the destination process labelled dest, using a tag to distinguish this message from other messages sent by this process to the same destination. count contiguous data spaces of type datatype The send buffer is composed of count consecutive data spaces of type datatype, starting at buf.
- Receiving messages. The function MPI_ Recv receives a message from the specified process source with the same datatype and tag as the receiving process, and the maximum number of received messages and data elements is count. The length of the received message must be less than or equal to the length of the buffer because if the received data is too large and the MPI does not truncate it, an overflow error will occur in the receive buffer.
- End of MPI. MPI_Finalize is the last call of the MPI program, it ends the MPI program and it is the last executable statement of the MPI program, otherwise, the program runs with an unpredictable result.

.

3.3 The basic model of MPI parallel programming

The MPI environment is initialized and terminated as follows: before calling the MPI routines, each process should execute MPI_INIT, then call MPI_COMM_SIZE to get the size of the default group and MPI_ COMM_RANK to get the logical number of the calling process in the default group. The process can then send messages to or receive messages from other nodes as required, often calling MPI_ SEND and MPI _RECV. Finally, when no MPI routines need to be called, MPI FINALIZE is called to remove the MPI environment, and the process can then either end or continue executing MPI-unrelated statements. This is the general flow of MPI parallel computing. The peer-to-peer and master-slave modes are the two most basic design modes for MPI parallel programs. are the two most basic design patterns for MPI parallel programs. Most MPI programs are one or a combination of these two patterns. By mastering these two patterns, you have mastered the main lines of parallel programming.

3.3.1 Peer-to-peer model

The so-called peer-to-peer model means that the processes of an MPI program have the same or similar functions and status, and the code of the MPI program should also be similar, except for the objects to be processed and the data to be manipulated, for example, the MPI program allows each process to assign initial values to different parts of an array in parallel, the relationship between the processes is a typical peer-to-peer relationship.

3.3.2 Master-slave mode

In master-slave mode, the processes of an MPI program do not have the same role and status, one or some processes perform one type of task, while others perform others, and the code corresponding to these processes with different functions or status differs considerably (Corne D, 2001) [9]. In master-slave mode, a group of processes work together on the same computing task, with one of them being the master process, responsible for the generation and initialization of the other slave processes, and for assigning the load to the slave processes. After the slave process has completed the actual calculation, it transmits the results to the master process, which finally collects the results and outputs them.

Let matrix C = A x B. Multiply matrix A and matrix-vector to get matrix C. Implementation method: The master process broadcasts vector B to all slave processes, then sends each row of matrix A to the slave processes, in turn, the slave processes calculate the result of multiplying one row and B, then send the result to the master process. Once the master process has sent all the rows of A, it sends an end flag to the corresponding slave process for each result received, and the slave process exits after receiving the end flag. The master process also finishes when it has collected all the results (Du Zhihui, et al. 2001) [10]. As shown in Figure 3.4.

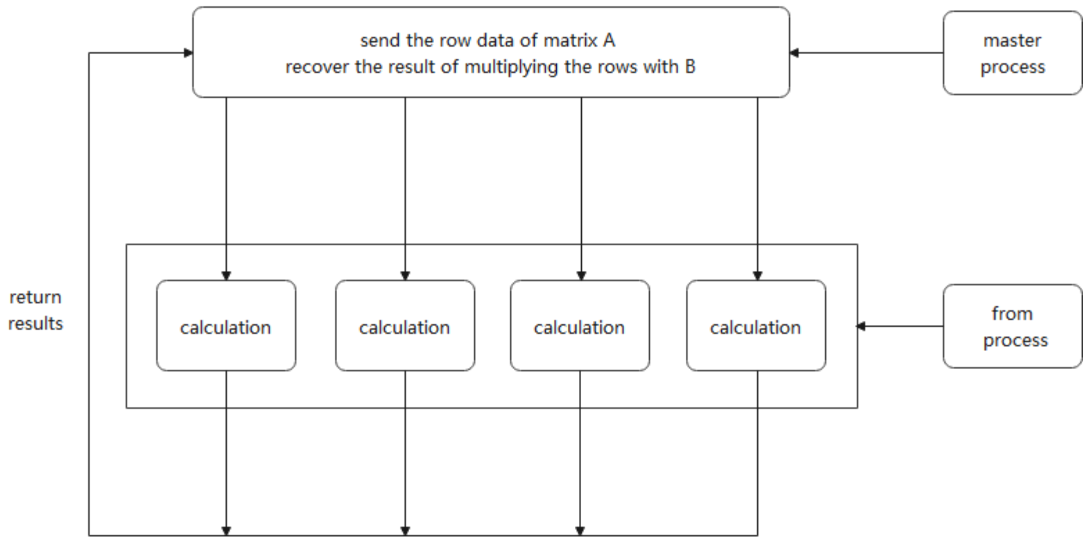


Figure 3.4: Matrix vector multiplication in master-slave mode.

3.4 Execution of MPI programs

The execution of the MPI program is shown in Figure 3.5. There are several specific steps [11].

- Compile the source program to obtain the MPI executable. For execution on a homogeneous system, only one compilation is required. If the system is heterogeneous, the MPI source program needs to be compiled on each heterogeneous system.
- Copy the executable program to each node machine.
- Execute this MPI program in parallel via the MPI command.

3.5 Parallel debugging methods and steps

Debugging parallel programs is much more complex than debugging serial programs, mainly because parallel programs are difficult to duplicate due to parallel execution and the state of affairs when errors occur. This difficulty is mainly due to uncertainty and the probe effect.

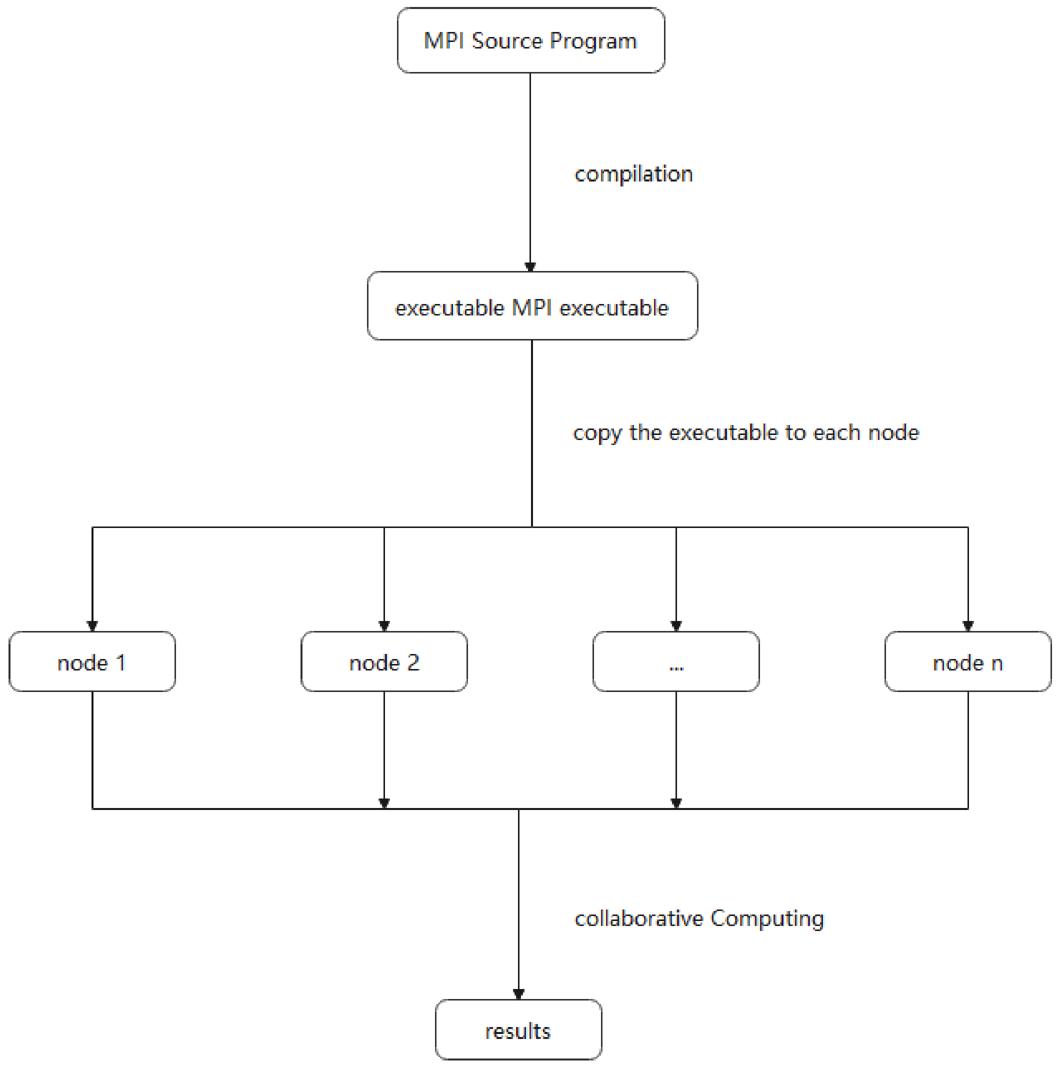


Figure 3.5: The MPI implementation process.

3.5.1 The difficulties of parallel debugging

So far, serial debugging has become a more mature technique, compared to parallel debugging, which is still in its infancy [12]. The parallelization introduced in the operation of parallel programs has made debugging parallel programs extremely difficult. This difficulty is mainly due to uncertainty and the probe effect.

1. Uncertainty

This uncertainty means that the same program input does not necessarily produce the same output in different executions, thus making the circular debugging often used in serial debugging ineffective. In serial debugging, when the programmer finds an error, he or she can execute the program again, trace its progress and find the error, which is called cycling debugging. Serial programs are usually single-tasked and run with uncertainty, thus ensuring that cyclic debugging is effective. There are many reasons for the uncertainty of parallel program operation.

- Shared variable contention: It is mainly for systems with shared storage, where read and write operations are performed on heaps of shared variables. Reads and writes to shared variables need to be guaranteed by proper synchronization operations.
- Message-passing contention: This is primarily for distributed storage systems that rely on message sending and receiving for communication. Due to the asynchronous nature of messaging, messages may be sent and received in different orders in different program executions.
- Dynamic process scheduling: The purpose of dynamic process scheduling is to achieve load balancing on parallel machines.
- The order of scheduling may vary in different program executions.
- Indeterminate system calls: the interaction of a program with its runtime environment may vary from program to program.

2. Probe effect

The probe effect is a problem caused by the introduction of parallel debugging tools. The introduction of debugging tools can mask timing errors in the program being debugged (Cheng, 2013) [13]. For example, when some output statements are added to a program as debug instructions, they may change the order of execution of the original statements. It is easy to see that because of the single-task execution method, serial programs are largely immune to the probe effect, whereas for parallel programs, the order of execution of statements between tasks is directly related to the resultant output in the program, so extra debugging statements should be added with extreme caution.

3.5.2 Parallel debugging approach

Despite the difficulties of parallel debugging, several parallel debugging methods have been accumulated. It is worth note that the various techniques do not stand alone, but complement and support each other. From a practical point of view, the implementation of breakpoints and replay is the basis of most parallel debuggers.

1. Reply

Replay was proposed to address the uncertainty of parallel programs so that cyclic debugging could effectively serve parallel debugging. Typical replay techniques can be found in distributed storage systems that support message passing. Relevant message passing can be logged during program runtime so that these messages can be used to control synchronous communication during replay and ensure effective loop debugging. It is worth noting that in a distributed storage system, the message passing statements in the program are separate from the computation, and thus the amount of information logged is relatively small and easily controlled in size; in a shared storage system, however, where the sending and receiving of messages is replaced by the reading and writing of shared variables, the communication is inextricably linked to the computation, and thus the additional overhead of logging will be difficult to control. To resolve this difficulty, higher-level synchronization controls, such as locks and roadblocks, should be developed in conjunction with shared variable systems, based on which logging and retransmission can be achieved.

Probe effects should be borne in mind in replay implementations. The extra overhead of a replay implementation is concentrated around synchronous communication, so inappropriate logging overhead may affect the timing relationships between statements and create a probing effect. When designing the debugger, consideration should be given to implementing the logging and replay implementations in a library of communication statements rather than adding statements to the source program at debug time, which is more effective in eliminating most of the probing effects.

2. Breakpoint debugging

Breakpoint debugging is the basic approach in serial program debugging. Breakpoints are usually set at the statement level at the start of debugging. In parallel debugging, multi-tasking replaces single-tasking and the corresponding multiple serial breakpoints replace individual breakpoints. Parallel debugging has more types of breakpoints than serial debugging due to its characteristics.

- Control flow-based breakpoints, which are similar to serial debugging breakpoints.
- Event-based breakpoints, which take effect when an exception or user-defined event occurs.
- Predicate-based breakpoints, which take effect when a conditional expression holds. In addition, due to the multitasking nature of parallel programs, breakpoints are also known as global breakpoints; they interrupt the execution of all parallel tasks when a certain condition holds. Global breakpoints are more complex and often require hardware support.

3.5.3 Steps for parallel debugging

1. Algorithm correctness check

Ensures that the parallel processing itself is correct at the algorithm level, and if the parallel algorithm used for the lock is rewritten from a serial algorithm, then the correctness of the serial algorithm should be ensured first. This step is the basis for correctness debugging and can usually only be carried out manually.

2. The boundary runs debugging

The simplest case, usually with one number of processors, is tested to eliminate some simple errors and to ensure that most of the remaining errors are related only to coordinated communication in the processor case.

3. Joint debugging

A small parallel environment is required and can usually start by testing the program running under both processors to troubleshoot errors.

4. Scalable debugging

Debugging by progressively increasing the number of processors to the extent that the algorithm and program allow.

3.6 MPI performance analysis

3.6.1 Calculation of parallel execution time

For parallel algorithms, the main concern is how fast the program can run, i.e. how long the entire computation takes, including computation, communication, etc. Therefore, for an algorithm, we do not only have to determine the number of computation steps, but we also have to calculate the communication time and the time required to wait for synchronization [14]. In a message-passing system, the time taken to solve the problem must include the time taken to send the message. The parallel execution time T(p) consists of two parts: the computation part T(comp) and the simultaneous communication part T(comm). T(p) = T(comp) + T(comm) : total time = calculation time + comm time.

The computation time can be extrapolated as in the serial algorithm and is usually calculated under the assumption that all processors are identical and have the same processing speed. This condition can be satisfied for many parallel machines, but it is often not satisfied for machines because the formation of a fleet of machines does not require all machines to be the same, and many of the machines used in the laboratory were purchased at different times and their performance can vary considerably. However, when doing analysis, it can be difficult to get results if we consider heterogeneous cases, so even when using a cluster, we assume that the machines at each node are the same.

3.6.2 Performance analysis tools

Performance analysis tools do not help us to increase the speed of our programs, but they can help us to find bottlenecks in performance. Analysis tools from compiler or accelerator suppliers are usually limited to the programming models supported by the device. Almost all vendor tools do not record MPI activity, so we have no way of knowing how well a hybrid application performs in terms of parallelism.

Research-oriented performance tools remedy this problem, with HPC toolkit and Score-P being the better third-party analysis tools that also support hardware accelerator analysis. Of these, Score-P supports the largest number of parallel examples, can record the most concurrent activity, and provides the most complete performance graphs for very complex applications, it is output formats are OTF2 and CUBE4, so I use Score-P as a performance analysis tool. And I use Vampir for visualizing performance data as it is by far the most capable trace visualizer and profile generator. Score-P and Vampir can be downloaded from http://www.score-p.org and http://www.vampir.eu respectively. Figure 3.6 is an overview of Score-P and figure 3.7 is an overview of Vampir.

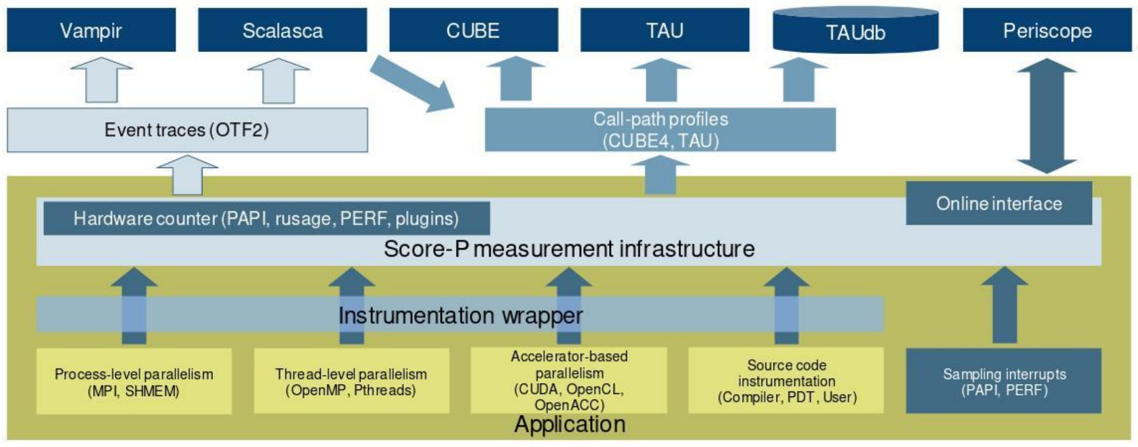

Vampir
Scalasca
CUBE
TAU
TAUdb
Periscope
Event traces (OTF2)
Call-path profiles (CUBE4, TAU)
Hardware counter (PAPI, rusage, PERF, plugins)
Online interface
Score-P measurement infrastructure
Instrumentation wrapper
Process-level parallelism (MPI, SHMEM)
Thread-level parallelism (OpenMP, Pthreads)
Accelerator-based parallelism (CUDA, OpenCL, OpenACC)
Source code instrumentation (Compiler, PDT, User)
Sampling interrupts (PAPI, PERF)
Application

Figure 3.6: An overview of Score-P.[1]

Following are the detail of the Vampir performance charts.

- Master Timeline: Detailed information on functions, communications, and synch-ronization events for the collection of processes.
- Summary Timeline: Fractions of the number of processes that are actively involved in given activities at a certain point in time.
- Process Timeline: Detailed information on different levels of function calls in a stacked bar chart for an individual process.
- Counter Timeline: Detailed counter information over time for an individual process.
- Function Summary: Overview of accumulated information across all functions and for a collection of processes.
- Communication Matrix View: Overview of information on messages sent between processes for detecting communication imbalances.

[1] Dr. Jessica, High Performance Computing Software-MPI, *Performance Optimization Section*, p.32.

- Process Summary: Overview of accumulated information across all functions and for every process independently.

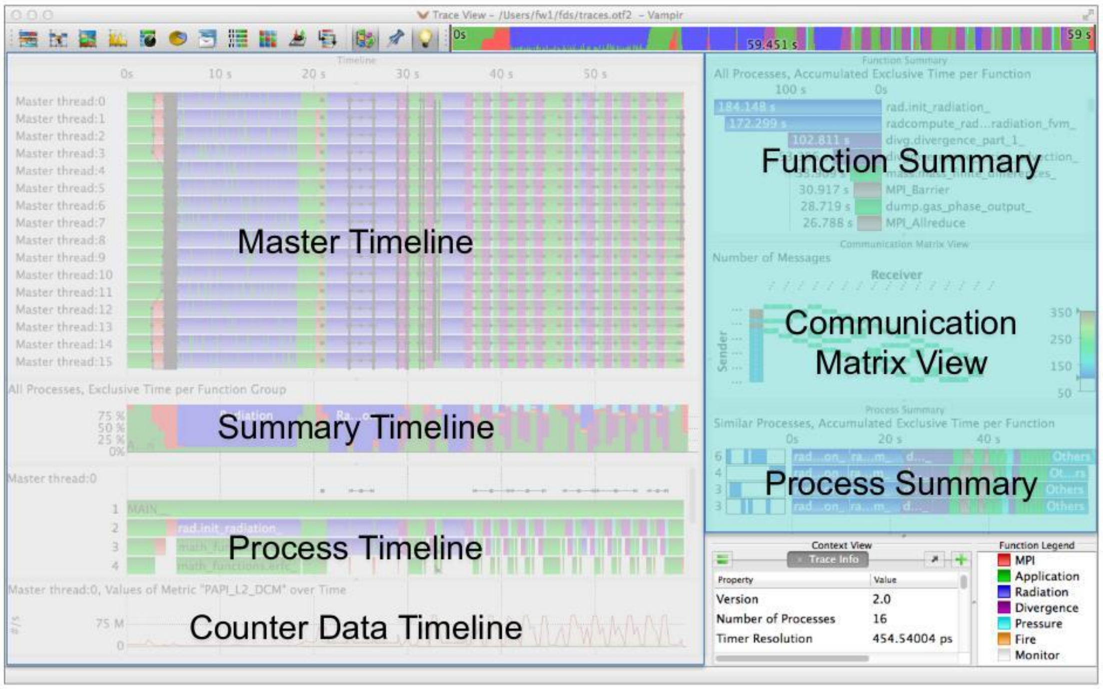


Figure 3.7: An overview of Vampir.[2]

[2] Dr. Jessica, High Performance Computing Software-MPI, *Performance Optimization Section*, p.52.

4. MPI implementation

4.1 Hardware of the test system

Table 4.1 lists the hardware configuration of the test system on kelvin01(one node). All MPI programs are tested in this hardware environment. In my tests, I requested up to 6 nodes and used 64 of these processors.

Table 4.1: Hardware configuration on kelvin01.

cluster	kelvin01
CPU Model	Intel(R) Xeon(R) CPU X5650 @ 2.67GHz
CPU(s)	12
Thread(s) per core	1
Core(s) per socket	6
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	12288K
Operating System	Scientific Linux release 7.8 (Nitrogen)

4.2 Blocking communication implementation

4.2.1 Implementation

Figure 4.1 is the diagram of the blocking communication MPI code. To simplify my calculations, I have allocated the entire grid memory for each processor. Each processor does only the operations to which it is allocated. Since the dt is the same for each processor, to save computational resources I only compute dt in rank0 and broadcast it to each processor.

As for the message exchange equation, I use MPI_Sendrecv instead of MPI_Send and MPI_Recv. This simplifies the interface, avoids possible deadlocks, and reduces sequentialization to a certain extent. Essentially, the use of MPI_Sendrecv implements a kind of interleaved Sends and Recvs. This is used so often that a special MPI interface has been created.

This report shows the concrete implementation of the code based on the flowchart idea. I will explain and introduce the key parts of the implementation. To make the computational load as even as possible across processors, I use the code to assign computational tasks to each processor as figure 4.2.

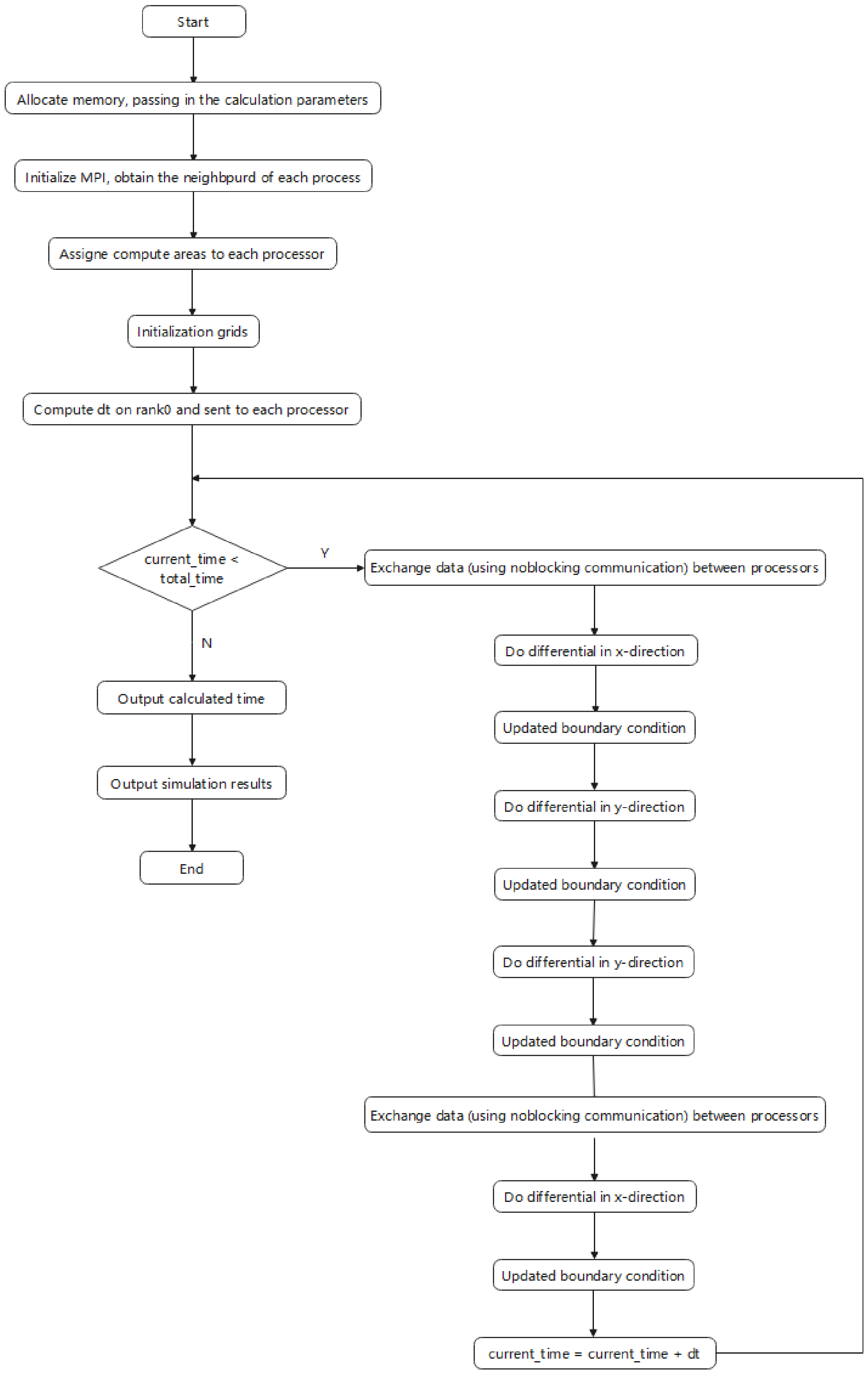
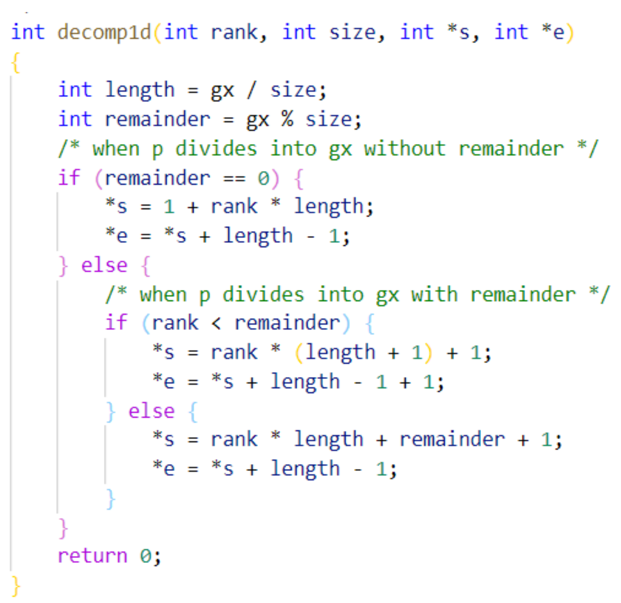

Start
Allocate memory, passing in the calculation parameters
Initialize MPI, obtain the neighbpurd of each process
Assigne compute areas to each processor
Initialization grids
Compute dt on rank0 and sent to each processor
current_time < total_time
Y
N
Exchange data (using noblocking communication) between processors
Do differential in x-direction
Updated boundary condition
Do differential in y-direction
Updated boundary condition
Do differential in y-direction
Updated boundary condition
Exchange data (using noblocking communication) between processors
Do differential in x-direction
Updated boundary condition
current_time = current_time + dt
Output calculated time
Output simulation results
End

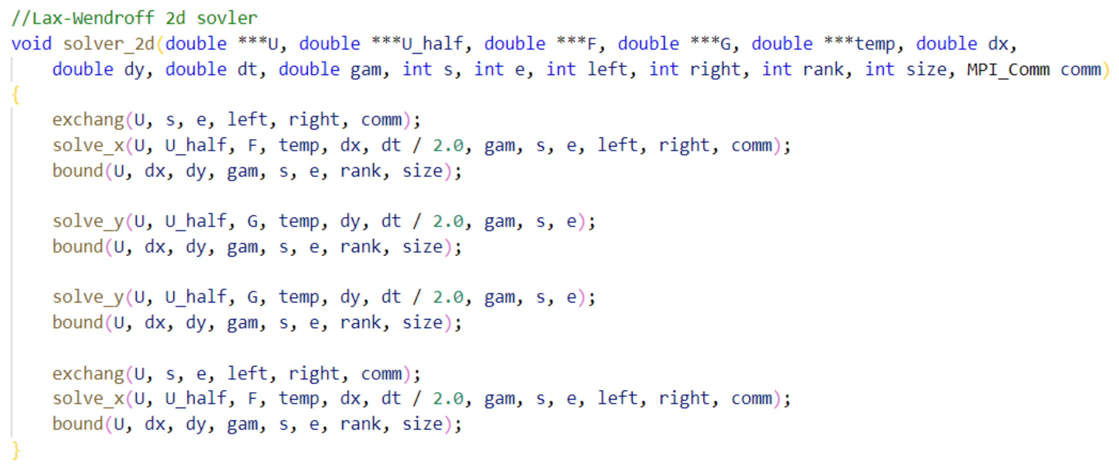
Figure 4.1: Diagram of the blocking communication MPI code.



```
int decomp1d(int rank, int size, int *s, int *e)
{
    int length = gx / size;
    int remainder = gx % size;
    /* when p divides into gx without remainder */
    if (remainder == 0) {
        *s = 1 + rank * length;
        *e = *s + length - 1;
    } else {
        /* when p divides into gx with remainder */
        if (rank < remainder) {
            *s = rank * (length + 1) + 1;
            *e = *s + length - 1 + 1;
        } else {
            *s = rank * length + remainder + 1;
            *e = *s + length - 1;
        }
    }
    return 0;
}
```

Figure 4.2: The code to assign computational tasks to each processor.

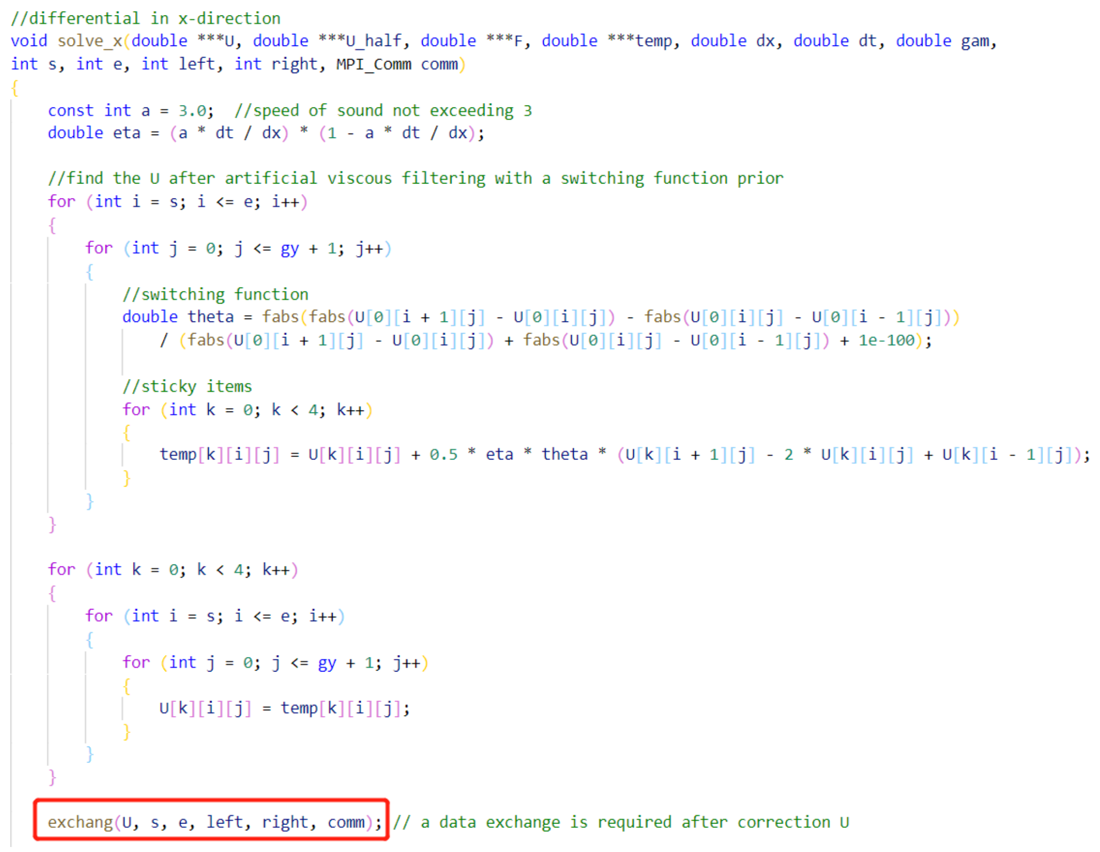
For the data exchange session, as I am partitioning the grid along the x-direction using the number of processors, I need to do a data exchange before each x-directional differencing to ensure that each processor gets the correct boundary value. For each x-directional difference, the boundary values of individual processors are exchanged before the calculation, and the boundary values of the whole grid are updated after the exchange. The entire 2D differential code is shown in figure 4.3.



```
//Lax-Wendroff 2d sovler
void solver_2d(double ***U, double ***U_half, double ***F, double ***G, double ***temp, double dx,
    double dy, double dt, double gam, int s, int e, int left, int right, int rank, int size, MPI_Comm comm)
{
    exchang(U, s, e, left, right, comm);
    solve_x(U, U_half, F, temp, dx, dt / 2.0, gam, s, e, left, right, comm);
    bound(U, dx, dy, gam, s, e, rank, size);

    solve_y(U, U_half, G, temp, dy, dt / 2.0, gam, s, e);
    bound(U, dx, dy, gam, s, e, rank, size);

    solve_y(U, U_half, G, temp, dy, dt / 2.0, gam, s, e);
    bound(U, dx, dy, gam, s, e, rank, size);

    exchang(U, s, e, left, right, comm);
    solve_x(U, U_half, F, temp, dx, dt / 2.0, gam, s, e, left, right, comm);
    bound(U, dx, dy, gam, s, e, rank, size);
}
```

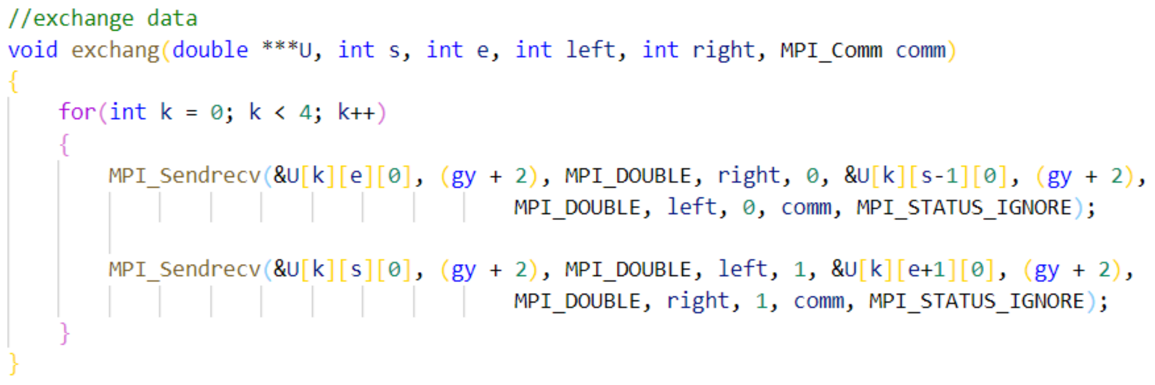
Figure 4.3: The 2D differential code.

Each iteration step does two 0.5 step x-directions of differencing, which means that data needs to be exchanged twice per iteration step. It is particularly important to note that data exchange is also required after the manual processing of the sticky quantities using the switch function, in function solve_x in solve.c, as shown in figure 4.4.



```
//differential in x-direction
void solve_x(double ***U, double ***U_half, double ***F, double ***temp, double dx, double dt, double gam,
int s, int e, int left, int right, MPI_Comm comm)
{
    const int a = 3.0;  //speed of sound not exceeding 3
    double eta = (a * dt / dx) * (1 - a * dt / dx);

    //find the U after artificial viscous filtering with a switching function prior
    for (int i = s; i <= e; i++)
    {
        for (int j = 0; j <= gy + 1; j++)
        {
            //switching function
            double theta = fabs(fabs(U[0][i + 1][j] - U[0][i][j]) - fabs(U[0][i][j] - U[0][i - 1][j]))
                / (fabs(U[0][i + 1][j] - U[0][i][j]) + fabs(U[0][i][j] - U[0][i - 1][j]) + 1e-100);

            //sticky items
            for (int k = 0; k < 4; k++)
            {
                temp[k][i][j] = U[k][i][j] + 0.5 * eta * theta * (U[k][i + 1][j] - 2 * U[k][i][j] + U[k][i - 1][j]);
            }
        }
    }

    for (int k = 0; k < 4; k++)
    {
        for (int i = s; i <= e; i++)
        {
            for (int j = 0; j <= gy + 1; j++)
            {
                U[k][i][j] = temp[k][i][j];
            }
        }
    }

    exchang(U, s, e, left, right, comm);// a data exchange is required after correction U
```

Figure 4.4: The exchange function in solve.c.



```
//exchange data
void exchang(double ***U, int s, int e, int left, int right, MPI_Comm comm)
{
    for(int k = 0; k < 4; k++)
    {
        MPI_Sendrecv(&U[k][e][0], (gy + 2), MPI_DOUBLE, right, 0, &U[k][s-1][0], (gy + 2),
                                MPI_DOUBLE, left, 0, comm, MPI_STATUS_IGNORE);

        MPI_Sendrecv(&U[k][s][0], (gy + 2), MPI_DOUBLE, left, 1, &U[k][e+1][0], (gy + 2),
                                MPI_DOUBLE, right, 1, comm, MPI_STATUS_IGNORE);
    }
}
```

Figure 4.5: The exchange function code.

Essentially, the processing of artificial viscous quantities with switching functions is also a difference algorithm, equivalent to a single x-directional difference calculation. So a data exchange is needed to update the vector U and then carry out the subsequent calculation. In MPI parallel computing, be sure to analyze the algorithm structure yourself to ensure that the bounds of each processor are updated appropriately for each calculation, otherwise you will get a problem that is hard to find.

The exchange function code is shown in figure 4.5. And as shown in the figure 4.6 data exchange diagram, each processor passes data from its own boundary to the neighbouring processor.

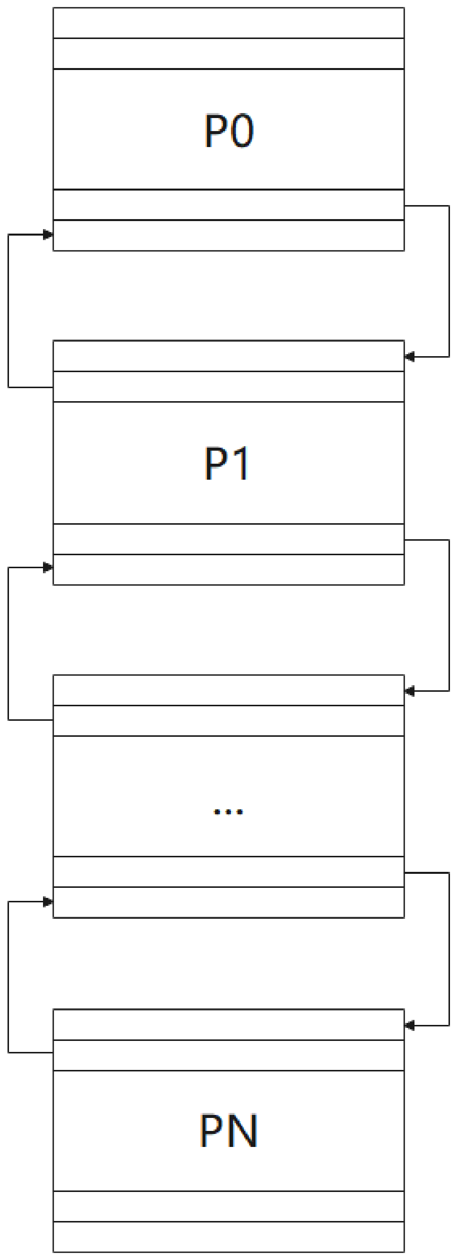


Figure 4.6: Data exchange diagram.

4.2.2 Output

I used the txt file output from the program at grid size x * y = 1600 * 400 to plot the pressure distribution and density distribution at y=0.5, as well as the pressure and density contour plots. The output of the parallel program was compared with the graphs from the serial txt file to confirm that the output of the parallel program was correct.

After comparison, the parallel program outputs the same image as the serial program, verifying that the parallel program's calculations are correct.

4.2.3 Timing

Table 4.2 shows the running time and speedup of the blocking parallel program for grid size x * y = 1600 * 400. And figure 4.7 shows the speedup for this case.

Table 4.2: Time and speedup of the blocking parallel program.

np	2	4	8	16	32	64
time(s)	2350.100	1310.905	843.483	586.354	451.438	379.448
speedup(time)	1.809	3.243	5.039	7.249	9.416	11.202

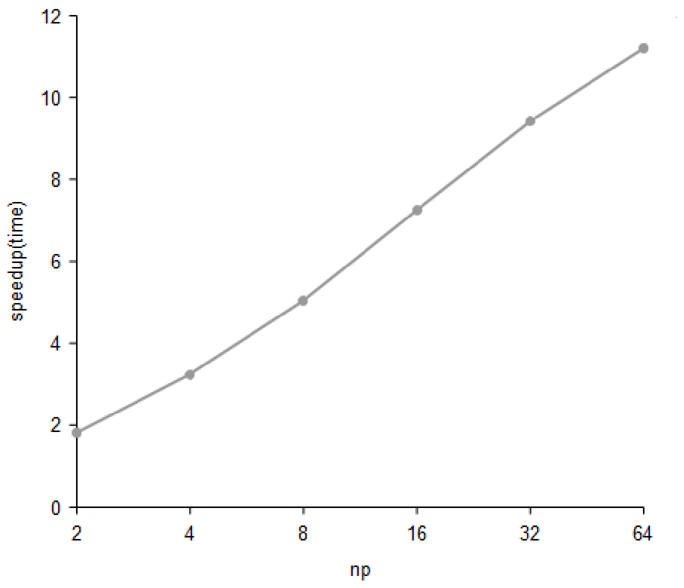
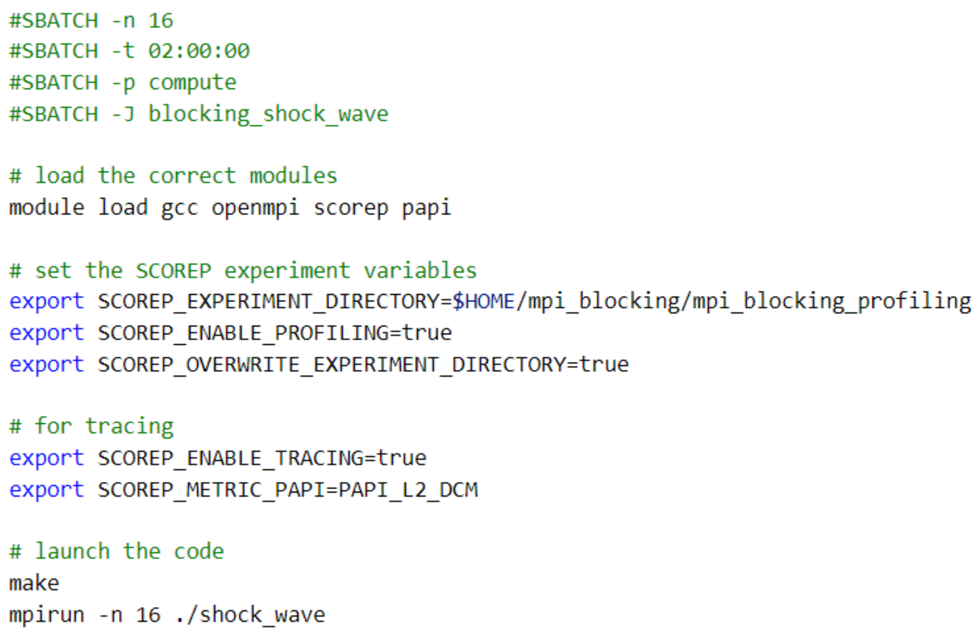


Figure 4.7: Speedup of the blocking parallel program.

We can see that the parallel program has the same performance for two different grid sizes. A maximum speed of 11 was obtained with 64 processors. Although the speedup rises with the number of processors, it is not linear and the speedup rises gradually and slowly, approximately quadratically with the number of processors. Because the MPI overhead gradually increases as the number of processors increases, the communication time gradually increases.

4.2.4 Profiling



```
#SBATCH -n 16
#SBATCH -t 02:00:00
#SBATCH -p compute
#SBATCH -J blocking_shock_wave

# load the correct modules
module load gcc openmpi scorep papi

# set the SCOREP experiment variables
export SCOREP_EXPERIMENT_DIRECTORY=$HOME/mpi_blocking/mpi_blocking_profiling
export SCOREP_ENABLE_PROFILING=true
export SCOREP_OVERWRITE_EXPERIMENT_DIRECTORY=true

# for tracing
export SCOREP_ENABLE_TRACING=true
export SCOREP_METRIC_PAPI=PAPI_L2_DCM

# launch the code
make
mpirun -n 16 ./shock_wave
```

Figure 4.8: The sbatch for Score-P.

To find bottlenecks in parallel programs, I use Score-P and Vampir to profile programs. And to save pages, only the case of grid size x * y = 1600 * 400 at np = 16 will be profiled. To use Score-P on the kelvin01, I created a sbatch to run automatically on the server as figure 4.8.

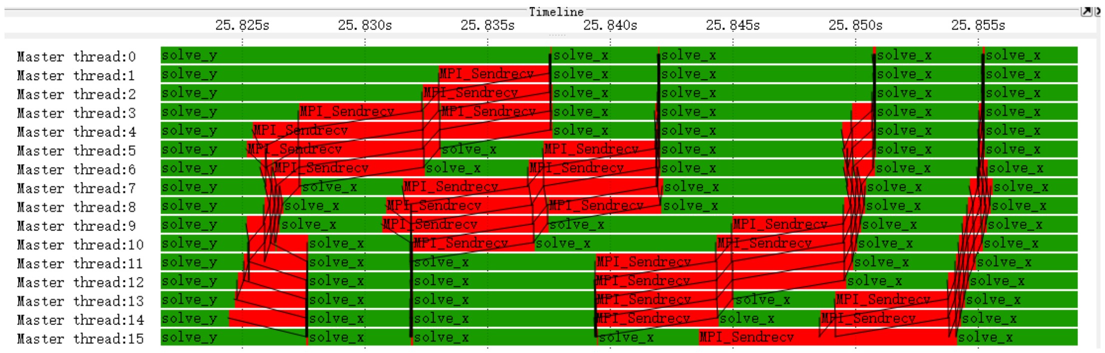
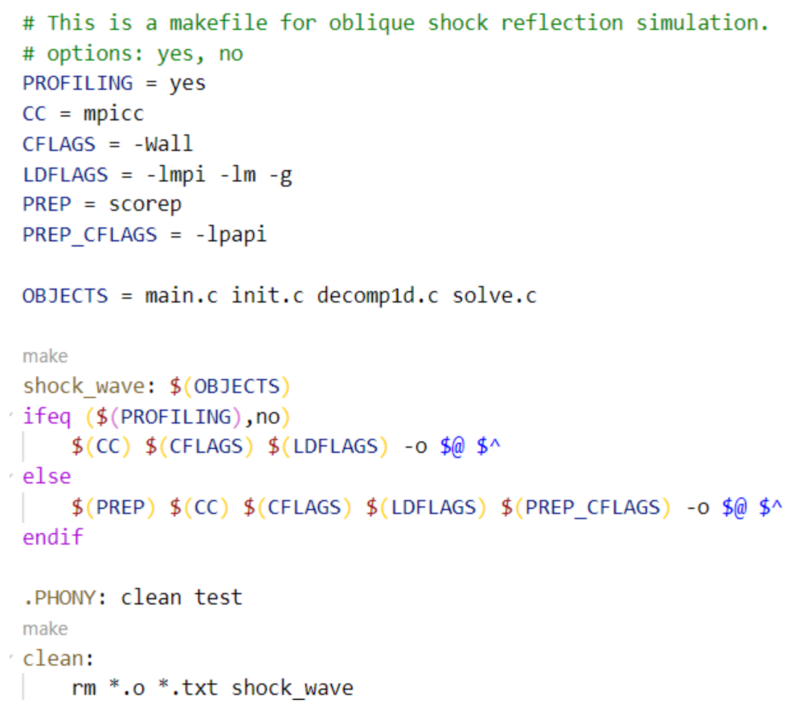


Figure 4.9: Timeline of the blocking parallel program.



```
# This is a makefile for oblique shock reflection simulation.
# options: yes, no
PROFILING = yes
CC = mpicc
CFLAGS = -Wall
LDFLAGS = -lmpi -lm -g
PREP = scorep
PREP_CFLAGS = -lpapi

OBJECTS = main.c init.c decomp1d.c solve.c

make
shock_wave: $(OBJECTS)
ifeq ($(PROFILING),no)
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^
else
    $(PREP) $(CC) $(CFLAGS) $(LDFLAGS) $(PREP_CFLAGS) -o $@ $^
endif

.PHONY: clean test
make
clean:
    rm *.o *.txt shock_wave
```

Figure **4.10**: The Makefile for Score-P.

After Score-P has been successfully run, the otf2 file is generated and can be opened with Vampir for visual profiling. Figure 4.9 shows the timeline of this program and we can clearly see that the data is sent and received sequentially and that the communication takes up a significant part of the time. In addition, to use Score-P, the Makefile needs to be modified as figure 4.10.

Figure 4.11 shows the function summary of this program. We can find that MPI_Sendrecv spent 551.618s.

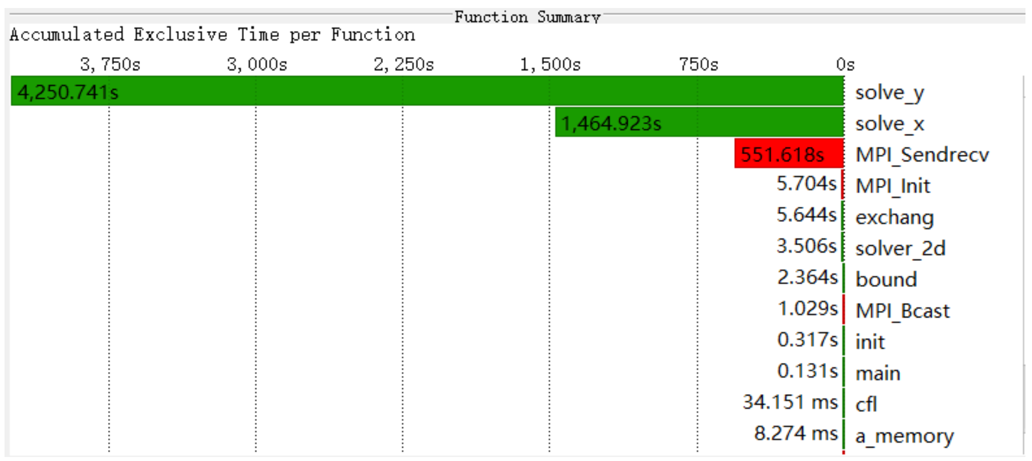


Figure 4.11: Function summary of the blocking parallel program.

4.3 Non blocking communication implementation

4.3.1 Implementation

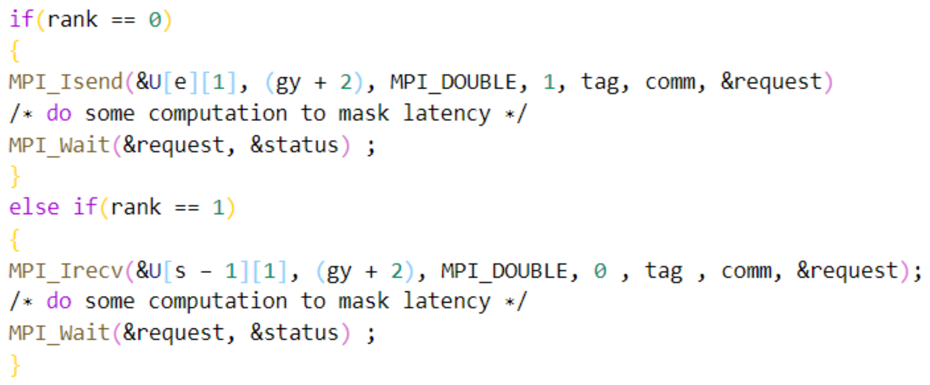
To solve the problem of sequentialization and idle waiting during data transfer, I use non-blocking communication to improve parallel program performance. Non blocking sends separate send start from send complete and receives separate recv start from recv complete. In non blocking MPI, the functions return immediately after being called while the data may not yet have been transferred. During this period, we can do the computation of the local data for each processor. When the complete transfer of the data is detected, the boundary value calculation is then performed. This allows data transfer and calculation to overlap and prevents one communication operation from stopping other communication operations from finishing.

New structure MPI_Request contains the information about the state of the transfer. We can use MPI_Test(request, flag, status) or MPI_Wait(request, status) to find out if the transfer is completed. A simple example of non blocking messaging is shown in Figure 4.12. We can do calculations of processor local data between MPI_Isend and MPI_Wait or between MPI_Irecv and MPI_Wait. This will even out the latency of message delivery. As for the implementation of the non blocking parallel code, it is mostly the same as the blocking parallel code. The main differences are in the splitting of the algorithms and the calls to the communication interface, which I will mainly describe. Figure 4.13 is a diagram of the blocking communication MPI code.

I split the algorithm so that I could exchange boundary data between processors while performing local data computation. Essentially, the calculation is divided into three parts.

- Artificial viscous fluid processing with switching functions.
- X directed grid update.
- Y directed grid update.

Since the grid is split along the x direction, we only need to focus on the first and second points. I use the functions exchange1 and exchange2 to process them separately. The overall structure of the solver code is as figure 4.14.



```
if(rank == 0)
{
MPI_Isend(&U[e][1], (gy + 2), MPI_DOUBLE, 1, tag, comm, &request)
/* do some computation to mask latency */
MPI_Wait(&request, &status) ;
}
else if(rank == 1)
{
MPI_Irecv(&U[s - 1][1], (gy + 2), MPI_DOUBLE, 0 , tag , comm, &request);
/* do some computation to mask latency */
MPI_Wait(&request, &status) ;
}
```

Figure 4.12: A simple example of non blocking messaging.

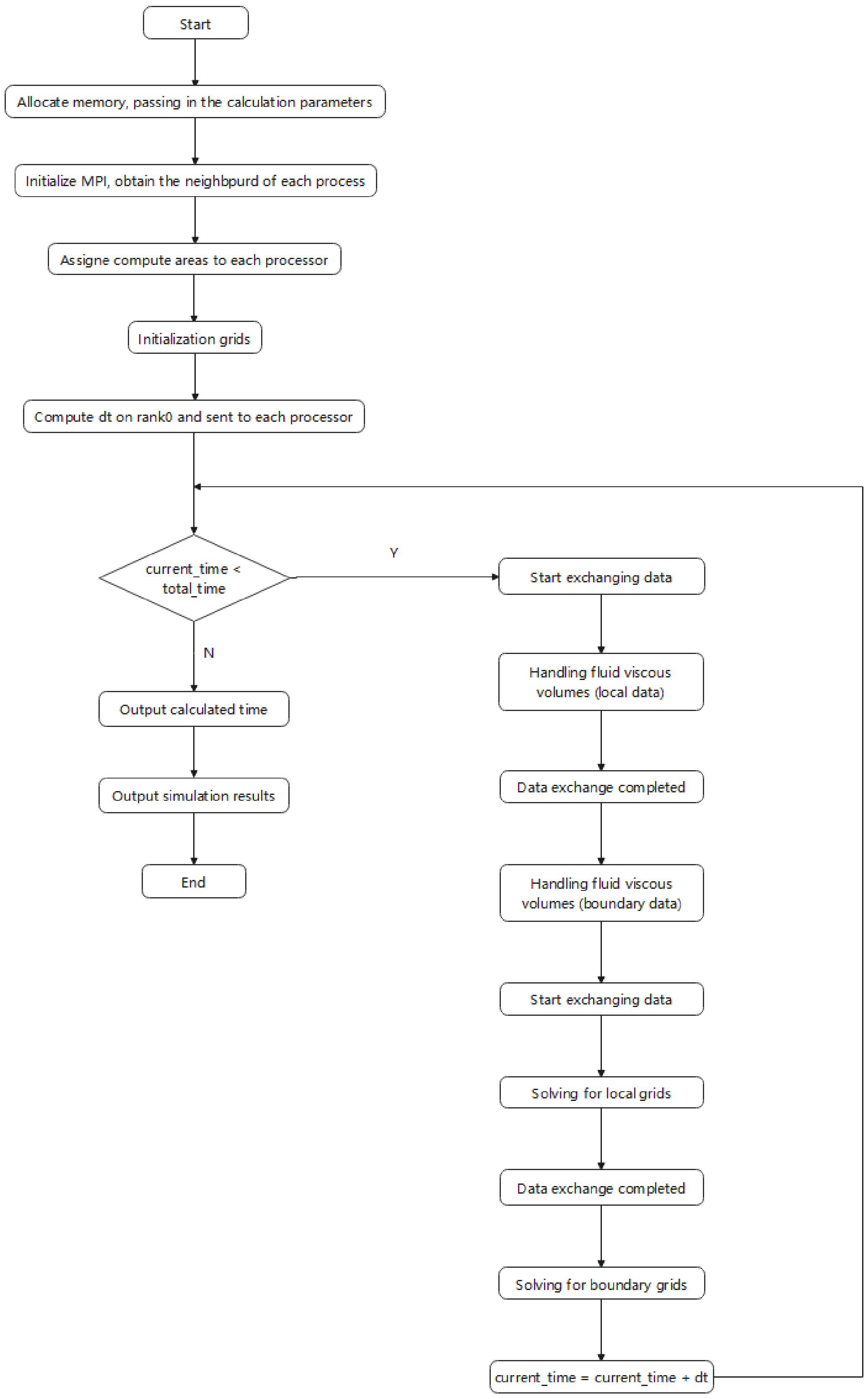
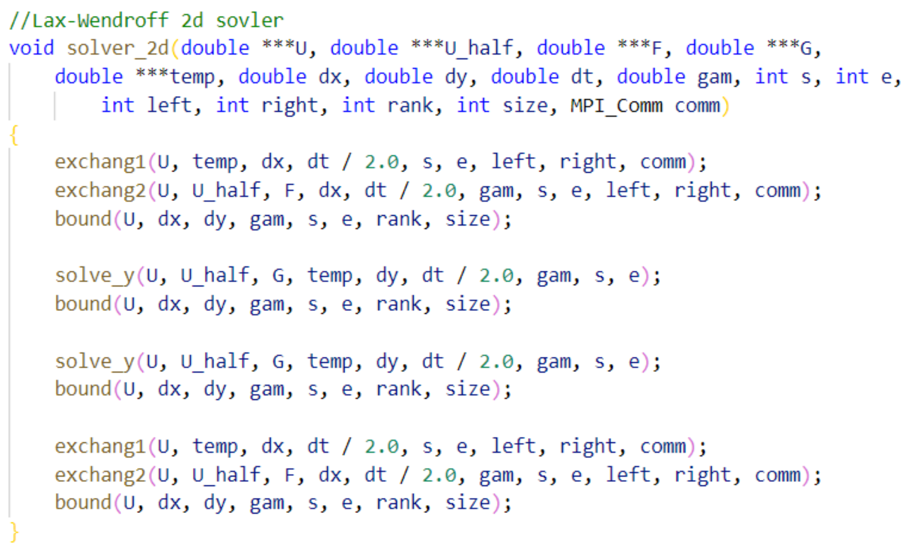
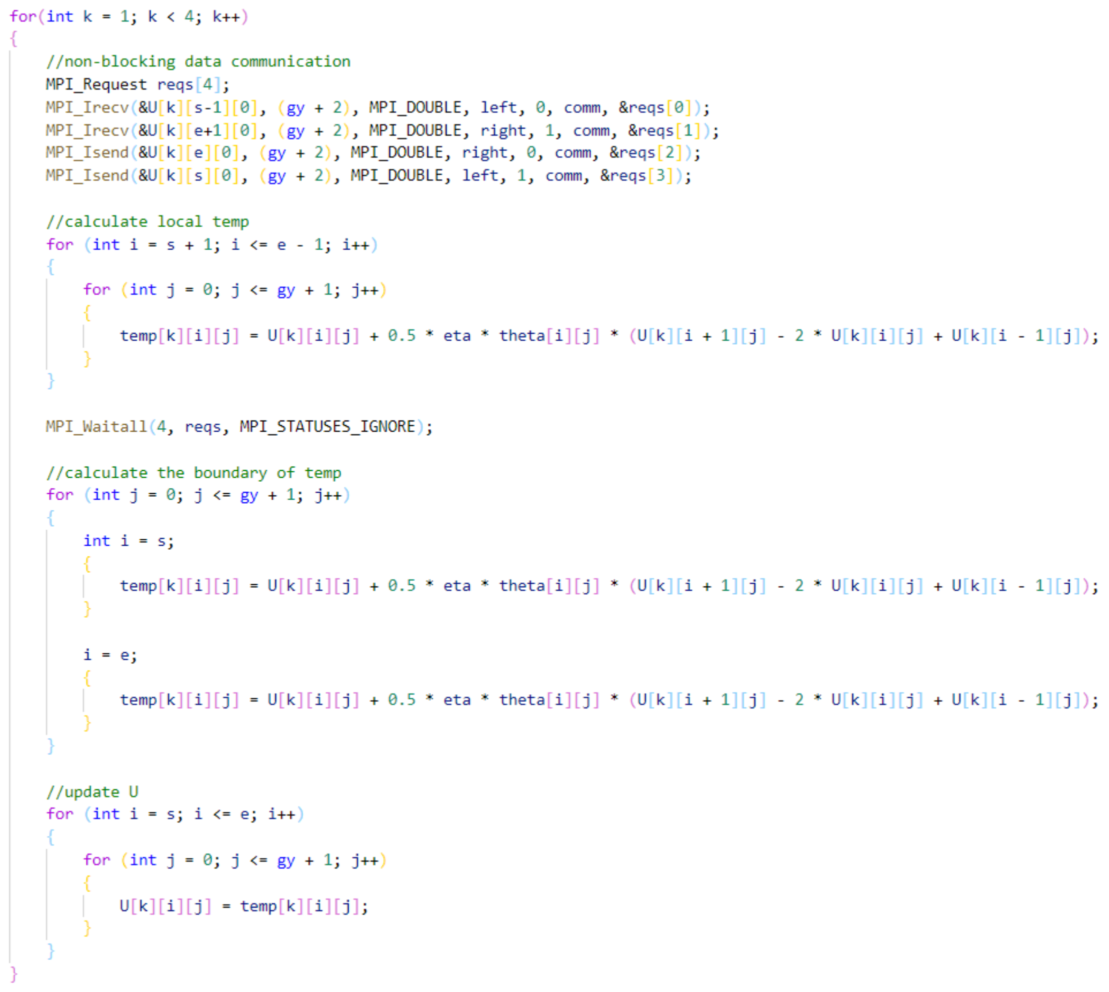

Start
Allocate memory, passing in the calculation parameters
Initialize MPI, obtain the neighbpurd of each process
Assigne compute areas to each processor
Initialization grids
Compute dt on rank0 and sent to each processor
current_time < total_time
Y
N
Start exchanging data
Handling fluid viscous volumes (local data)
Data exchange completed
Handling fluid viscous volumes (boundary data)
Start exchanging data
Solving for local grids
Data exchange completed
Solving for boundary grids
current_time = current_time + dt
Output calculated time
Output simulation results
End

Figure 4.13: Diagram of the non blocking communication MPI code.



```
//Lax-Wendroff 2d sovler
void solver_2d(double ***U, double ***U_half, double ***F, double ***G,
    double ***temp, double dx, double dy, double dt, double gam, int s, int e,
        int left, int right, int rank, int size, MPI_Comm comm)
{
    exchang1(U, temp, dx, dt / 2.0, s, e, left, right, comm);
    exchang2(U, U_half, F, dx, dt / 2.0, gam, s, e, left, right, comm);
    bound(U, dx, dy, gam, s, e, rank, size);

    solve_y(U, U_half, G, temp, dy, dt / 2.0, gam, s, e);
    bound(U, dx, dy, gam, s, e, rank, size);

    solve_y(U, U_half, G, temp, dy, dt / 2.0, gam, s, e);
    bound(U, dx, dy, gam, s, e, rank, size);

    exchang1(U, temp, dx, dt / 2.0, s, e, left, right, comm);
    exchang2(U, U_half, F, dx, dt / 2.0, gam, s, e, left, right, comm);
    bound(U, dx, dy, gam, s, e, rank, size);
}
```

Figure 4.14: The overall structure of the solver code.



```
for(int k = 1; k < 4; k++)
{
    //non-blocking data communication
    MPI_Request reqs[4];
    MPI_Irecv(&U[k][s-1][0], (gy + 2), MPI_DOUBLE, left, 0, comm, &reqs[0]);
    MPI_Irecv(&U[k][e+1][0], (gy + 2), MPI_DOUBLE, right, 1, comm, &reqs[1]);
    MPI_Isend(&U[k][e][0], (gy + 2), MPI_DOUBLE, right, 0, comm, &reqs[2]);
    MPI_Isend(&U[k][s][0], (gy + 2), MPI_DOUBLE, left, 1, comm, &reqs[3]);

    //calculate local temp
    for (int i = s + 1; i <= e - 1; i++)
    {
        for (int j = 0; j <= gy + 1; j++)
        {
            temp[k][i][j] = U[k][i][j] + 0.5 * eta * theta[i][j] * (U[k][i + 1][j] - 2 * U[k][i][j] + U[k][i - 1][j]);
        }
    }

    MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE);

    //calculate the boundary of temp
    for (int j = 0; j <= gy + 1; j++)
    {
        int i = s;
        {
            temp[k][i][j] = U[k][i][j] + 0.5 * eta * theta[i][j] * (U[k][i + 1][j] - 2 * U[k][i][j] + U[k][i - 1][j]);
        }

        i = e;
        {
            temp[k][i][j] = U[k][i][j] + 0.5 * eta * theta[i][j] * (U[k][i + 1][j] - 2 * U[k][i][j] + U[k][i - 1][j]);
        }
    }

    //update U
    for (int i = s; i <= e; i++)
    {
        for (int j = 0; j <= gy + 1; j++)
        {
            U[k][i][j] = temp[k][i][j];
        }
    }
}
```

Figure 4.15: Exchange1 function.

The function exchange1 contains the manual viscous fluid handling with switch function and the corresponding data transfer. And because of the dependency between the data, I first calculate U[0], then U[1], U[2], U[3]. It is worth mentioning that it is not possible to do all the calculations of the local data at the same time as the data is passed due to the excessive dependencies between the data. Some of the local data require a group of U's to be fully updated before it can be calculated. So this non-blocking communication can only improve parallelism to a certain extent. This algorithm cannot achieve fully non-blocking communication. I will roughly show the structure and implementation of my code in Figure 4.15.

In the function exchange2, I mainly implement the calculation and data transfer in the x direction of the grid. To reduce double counting, I compute the vector F separately outside the loop, and the computation of the vector F does not rely on boundary data from other processors. The next step is a two-step differential calculation. Due to the dependency between the data, I had to finish the first step of differencing completely before I could proceed to the second step of differencing. This means that the calculation of the communication time with the overlay is only available for the first differential.

4.3.2 Output

Same as before, I used the txt file output from the program at grid size x * y = 1600 * 400 to plot the pressure distribution and density distribution at y=0.5, as well as the pressure and density contour plots. The output of the non blocking parallel program was compared with the graphs from the serial txt file to confirm that the output of the non blocking parallel program was correct.

After comparison, the non blocking parallel program outputs the same image as the serial program, verifying that the parallel program's calculations are correct.

4.3.3 Timing

Table 4.3 shows the running time and speedup of the non blocking parallel program for grid size x * y = 1600 * 400. And figure 4.16 shows the speedup for this case.

Table 4.3: Time and speedup of the non blocking parallel program.

np	2	4	8	16	32	64
time(s)	2287.077	1296.423	843.649	588.463	448.229	376.866
speedup(time)	1.859	3.279	5.038	7.223	9.483	11.279

From the table and graph, we can see that the use of non-blocking communication has resulted in some improvement in program runtime and speed-up ratios. However, due to the algorithm, non-blocking communication cannot do local data computation at the same time as the message is passed, so the improvement is not significant compared to blocking communication.

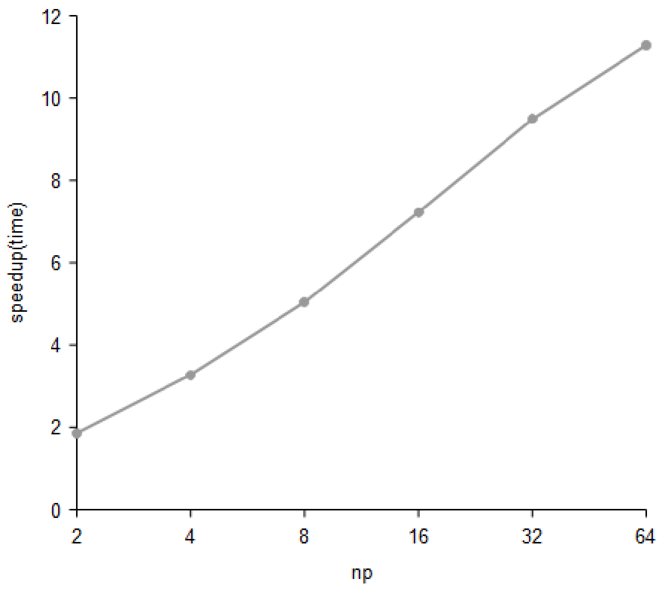


Figure 4.16: Speedup of the non blocking parallel program.

4.3.4 Profiling

Same as blocking communication, I use Score-P and Vampir to profile programs. And to save pages, only the case of grid size x * y = 1600 * 400 at np = 16 will be profiled. The use of Score-P and Vampir is essentially the same as the procedure for blocking communication and will not be described.

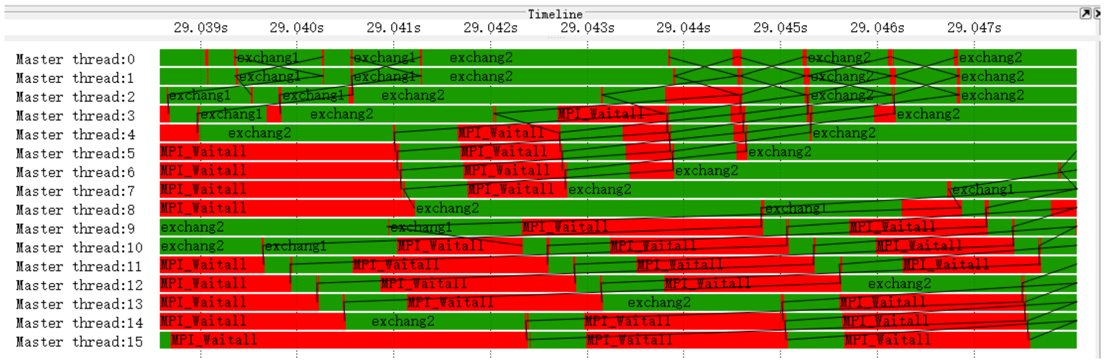


Figure 4.17: Timeline of the no blocking parallel program.

Figure 4.17 shows the timeline of this program. We can see that the time spent sending data and receiving data is very short and most of the time is consumed in the synchronous MPI_Waitall.

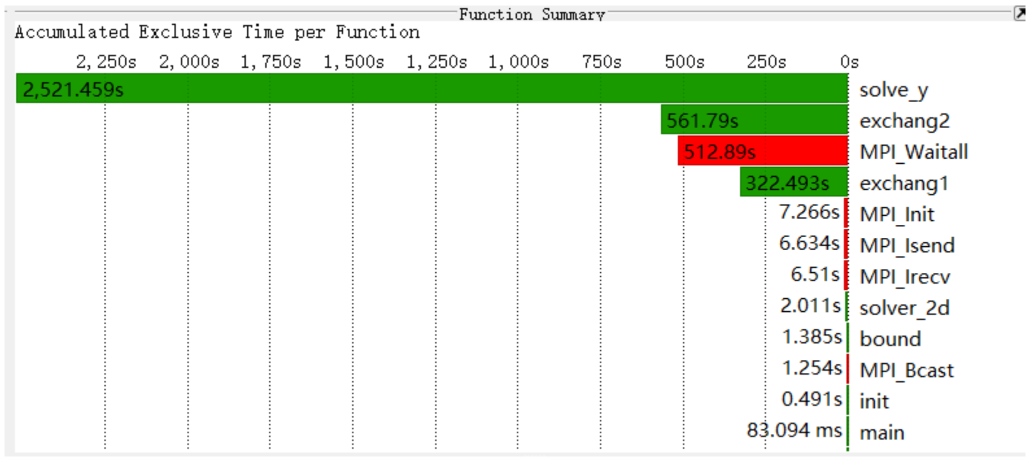


Figure 4.18: Function summary of the no blocking parallel program.

Figure 4.18 shows the function summary of this program. We can see that the total time for MPI_Isend, MPI_Irecv and MPI_Waitall is 526.034s, which is less than the 551.618s for blocking communication, so there is a performance gain for this non blocking communication parallel program.

4.4 RMA implementation

4.4.1 Implementation

RMA (Remote Memory Access) is a technique for manipulating remote processes and requires hardware support. The RMA flow is shown in Figure 4.19. It consists of three types of operations concerning remote processes: put, get, and update. Changes to receiver memory can only happen when the receiver allows them into a specified receive buffer. It is a way of moving data from one processor to another with a single routine specifying where data coming from and where going to. A complete RMA operation is divided into three main parts as follows.

- Define the memory of the process that can be used for RMA operations: create a memory window.
- Specify data to be moved and where to move it.
- Specify how to know that the data is available.

The code implementation of RMA communication is very similar to that of blocking communication, with only two major inconsistencies, one is the exchange function, as shown in figure 4.20. The other place is the creation of the window, as shown in figure 4.21. And figure 4.22 is the diagram of the RMA MPI code.

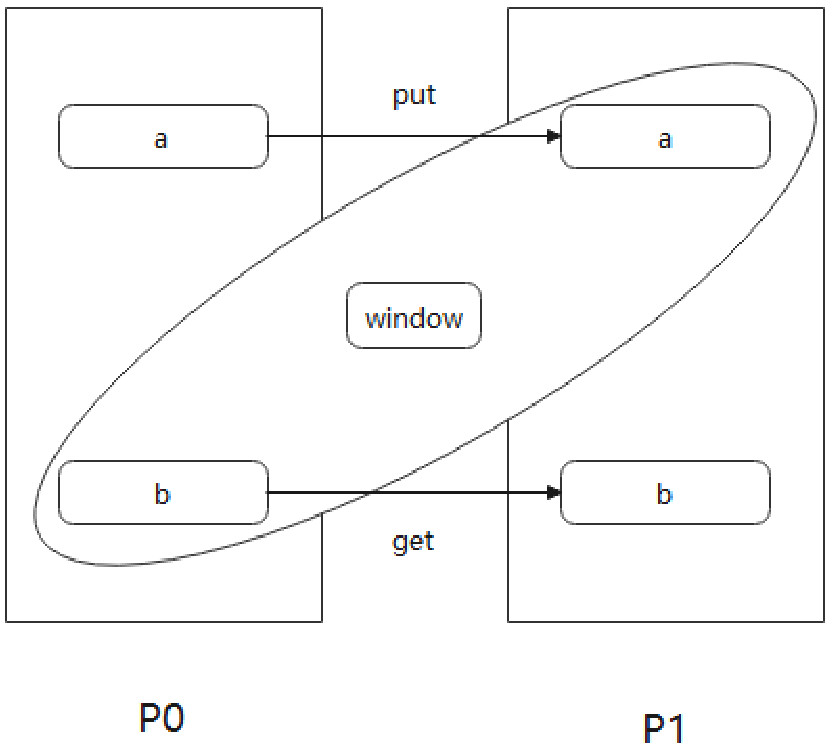
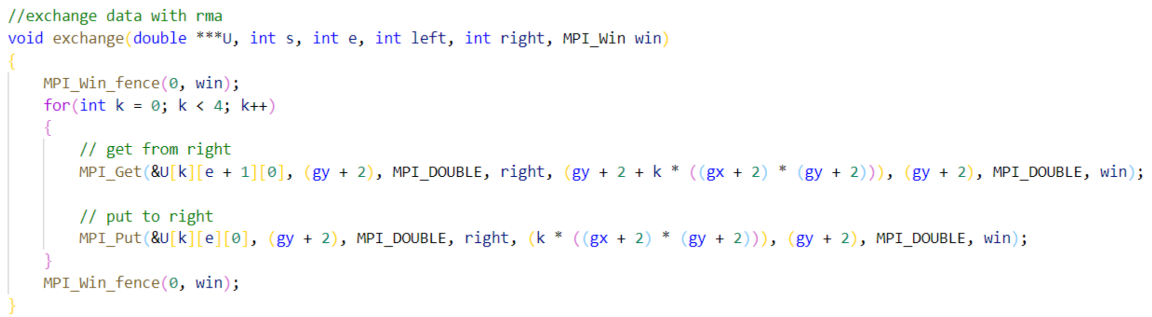
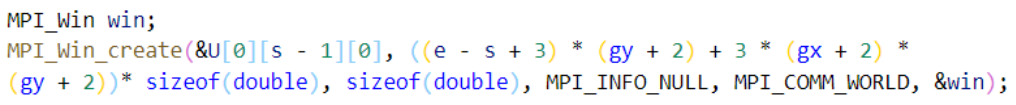


Figure 4.19: RMA code flow.



```
//exchange data with rma
void exchange(double ***U, int s, int e, int left, int right, MPI_Win win)
{
    MPI_Win_fence(0, win);
    for(int k = 0; k < 4; k++)
    {
        // get from right
        MPI_Get(&U[k][e + 1][0], (gy + 2), MPI_DOUBLE, right, (gy + 2 + k * ((gx + 2) * (gy + 2))), (gy + 2), MPI_DOUBLE, win);

        // put to right
        MPI_Put(&U[k][e][0], (gy + 2), MPI_DOUBLE, right, (k * ((gx + 2) * (gy + 2))), (gy + 2), MPI_DOUBLE, win);
    }
    MPI_Win_fence(0, win);
}
```

Figure 4.20: The exchange function of RMA.



```
MPI_Win win;
MPI_Win_create(&U[0][s - 1][0], ((e - s + 3) * (gy + 2) + 3 * (gx + 2) *
(gy + 2))* sizeof(double), sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &win);
```

Figure 4.21: The creation of the window.

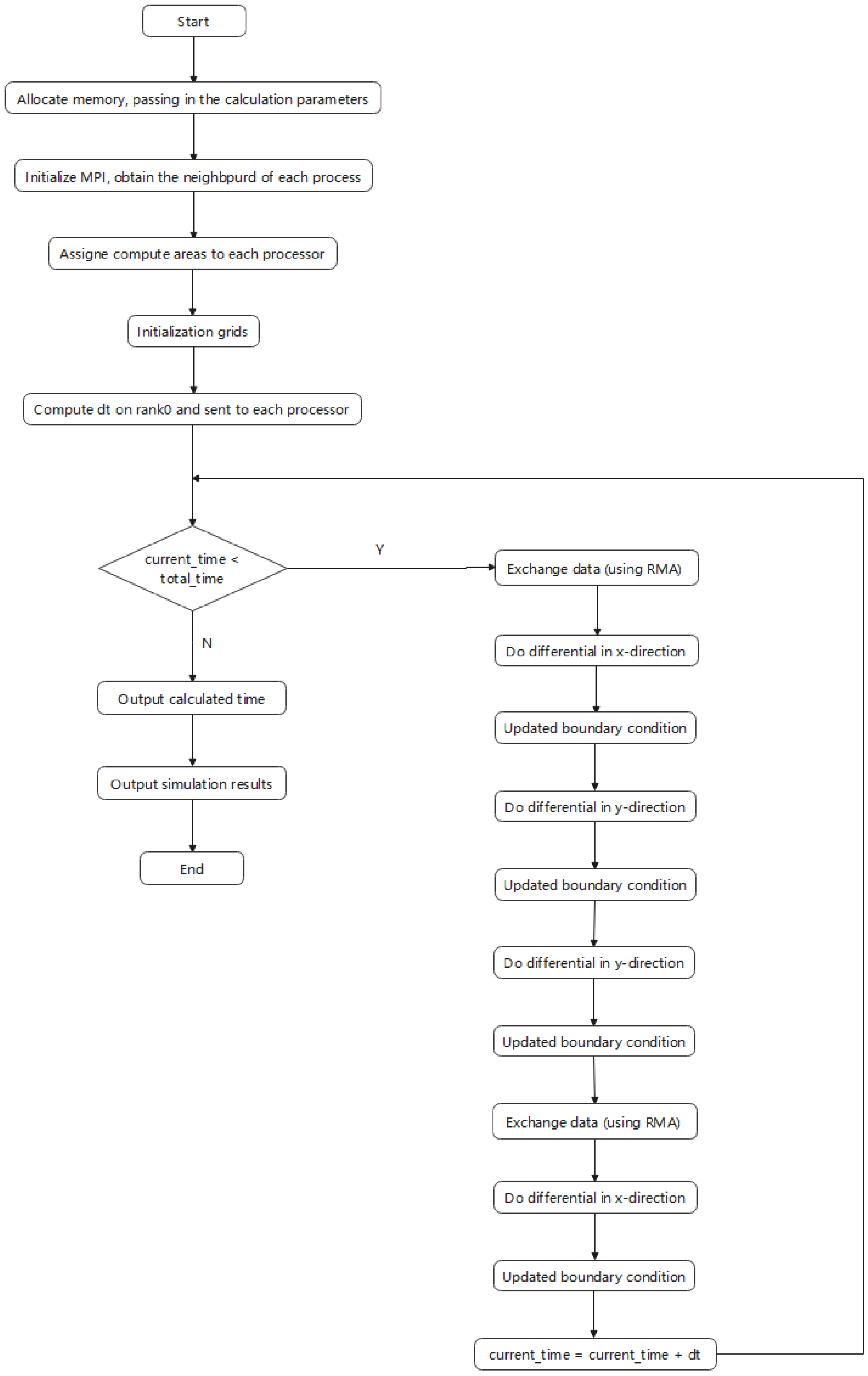

Start
Allocate memory, passing in the calculation parameters
Initialize MPI, obtain the neighbpurd of each process
Assigne compute areas to each processor
Initialization grids
Compute dt on rank0 and sent to each processor
current_time < total_time
Y
N
Exchange data (using RMA)
Do differential in x-direction
Updated boundary condition
Do differential in y-direction
Updated boundary condition
Do differential in y-direction
Updated boundary condition
Exchange data (using RMA)
Do differential in x-direction
Updated boundary condition
current_time = current_time + dt
Output calculated time
Output simulation results
End

Figure 4.22: Diagram of the RMA MPI code.

4.4.2 Output

Same as before, I used the txt file output from the program at grid size x * y = 1600 * 400 to plot the pressure distribution and density distribution at y=0.5, as well as the pressure and density contour plots. The output of the RMA parallel program was compared with the graphs from the serial txt file to confirm that the output of the RMA parallel program was correct.

After comparison, the RMA parallel program outputs the same image as the serial program, verifying that the parallel program's calculations are correct.

4.4.3 Timing

Table 4.4 shows the running time and speedup of the RMA parallel program for grid size x * y = 1600 * 400. And figure 4.23 shows the speedup for this case.

From the table and graph, we can see that the difference in acceleration between using RMA and using non-blocking communication is not significant.

Table 4.4: Time and speedup of the RMA parallel program.

np	2	4	8	16	32	64
time(s)	2287.194	1321.401	846.664	603.762	462.334	380.123
speedup(time)	1.858	3.217	5.020	7.040	9.194	11.182

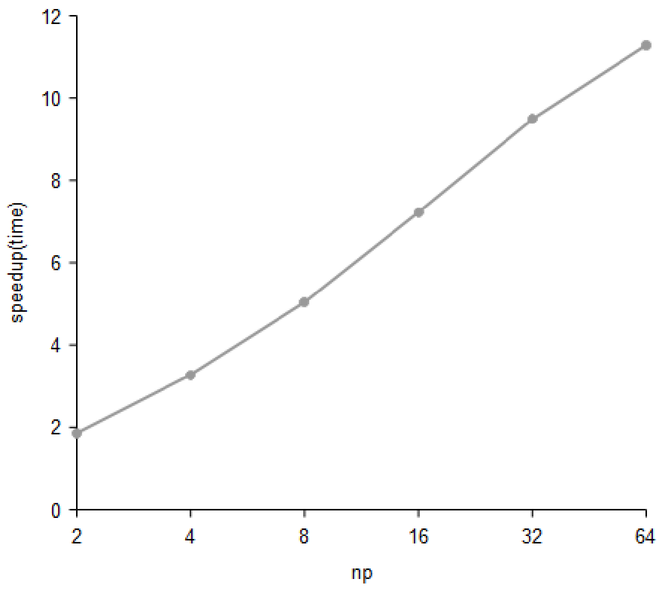


Figure 4.23: Speedup of the RMA parallel program.

5. CUDA introduction

5.1 CUDA overview

The relatively inexpensive and low-maintenance features and advantages of GPU parallel computing, combined with the power of floating-point computing, have led to a focus on GPU programming techniques, and the introduction of the GPGPU (GPU General Purpose Computing) concept has led to an increasing number of researchers using GPUs to perform high-performance computing. Writing GPU programs requires the installation of specific compilation environments and platform tools, mainly: Open GL, Brook+, ATI Stream, CUDA, Open CL, etc (Hiroyasu et al. 2000) [15].

CUDA (Compute Unified Device Architecture), is a computing platform from graphics card manufacturer NVIDIA. CUDA is a general-purpose parallel computing architecture introduced by NVIDIA that enables GPUs to solve complex computational problems. It consists of the CUDA Instruction Set Architecture (ISA) and the GPU's internal parallel computing engine. CUDA programming model makes it very easy for developers to perform complex calculations, significantly reducing development time and increased efficiency.

5.2 GPU architecture features

The key to high performance computing is the use of multi-core processors for parallel computing. When we solve a computer program task, the natural idea is to break the task down into a series of smaller tasks and complete each of these smaller tasks. In serial computing, the idea is to have our processor process one computational task at a time, processing one computational task and then computing the next until all the smaller tasks are completed, and then the larger program task is complete [16]. Figure 5.1 below shows the steps of how we can solve the problem using serial programming ideas.

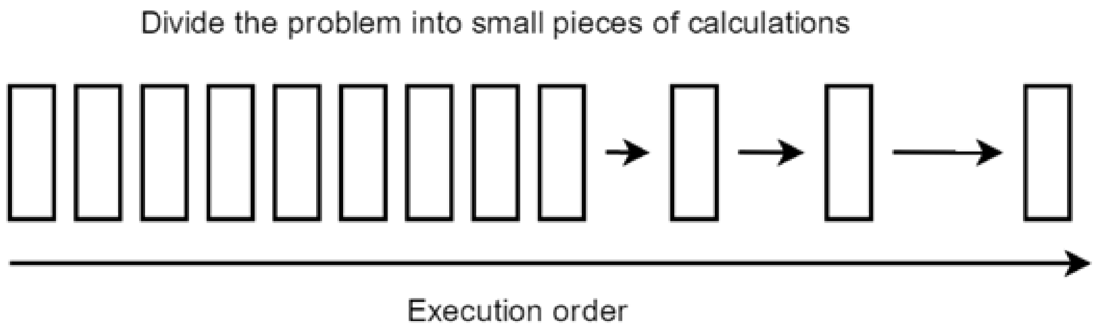


Figure 5.1: Steps of solving the problem using serial programming ideas.

]

To further speed up the computation of a large task, we can assign some independent modules to different processors for simultaneous computation (when there is no data dependency), and then finally combine these results to complete the task once. Figure 5.2 shows the decomposition of a large computational task into smaller tasks and then assigning the independent smaller tasks to different processors for parallel computation, and then finally aggregating the results through a serial program to complete the total computational task this time (Streichert et al. 2005) [17]. Therefore, the key to whether or not a program can perform parallel computing is to analyze how many execution modules the program can be split into, and which of these execution modules are independent and which are strongly dependent and strongly coupled.

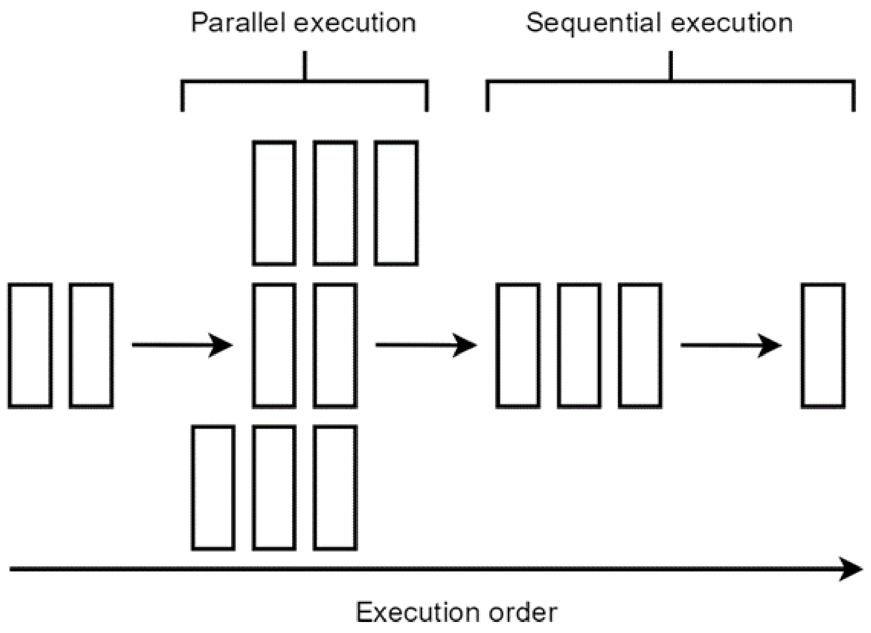


Figure 5.2: Steps of solving the problem using parallel programming ideas.

Next, let's explore the hardware architecture of the CPU and GPU.CPUs are made up of a few cores optimized for sequential serial processing. A GPU, on the other hand, consists of thousands of smaller, more efficient cores that are specifically designed for simultaneous multitasking and can efficiently handle parallel tasks. In other words, although each CPU core is extremely powerful in its own right and very strong in processing tasks, it has few cores and does not perform well in parallel computing; on the contrary, although each core is not very powerful, the GPU has many cores and can handle multiple computing tasks at the same time and does a good job in supporting parallel computing. The different hardware characteristics of the GPU and the CPU determine their application scenarios (Jaimes et al. 2007) [18]. The CPU is the core of computing and control of the computer, while the GPU is mainly used for graphic image processing. The image is presented in the form of a matrix in the computer, and our image processing is the manipulation of various matrices for calculation, and many matrix operations can be done in parallel, which makes the image processing can be done very fast, so the GPU also has the opportunity to make a big impact in the field of graphics images. To summarize the characteristics of both are as follows.

- CPU: good at process control and logic processing, irregular data structures, unpredictable storage structures, single-threaded programs, branch-intensive algorithms.
- GPU: specializes in data-parallel computing, regular data structures, predictable storage patterns.

In today's computer architecture, to perform CUDA parallel computing, the GPU alone cannot complete the computational task and must be used to collaborate with the CPU to complete a high-performance parallel computing task. Generally speaking, the parallel part runs on the GPU and the serial part on the CPU, which is called heterogeneous computing. Specifically, heterogeneous computing means that processors of different architectures collaborate to complete a computational task; the CPU is responsible for the overall program flow, while the GPU is responsible for the specific computational task, and once each thread of the GPU has completed its computational task, we copy the results of the GPU's computation to the CPU to complete the computational task [19].

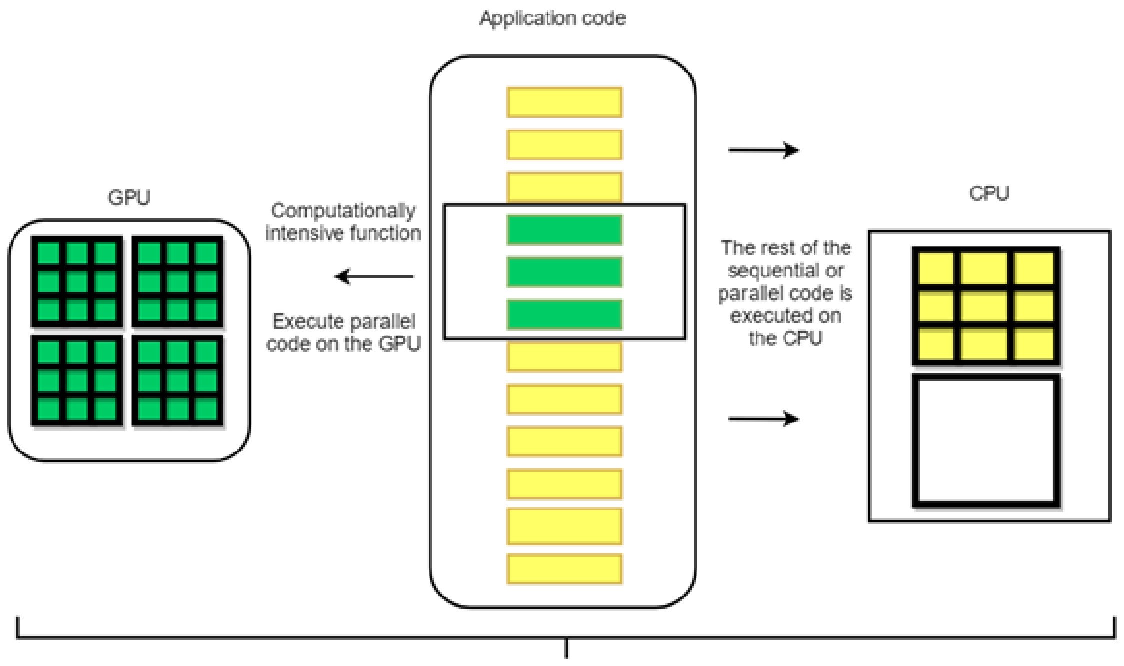

Application code
GPU
Computationally intensive function
Execute parallel code on the GPU
CPU
The rest of the sequential or parallel code is executed on the CPU

Figure 5.3: How GPU acceleration works.

So the overall division of labour in the application using GPU acceleration is that the GPU is responsible for the computationally intensive code (about 5% of the code) and the CPU is responsible for the remaining serial code, as shown in Figure 5.3.

5.3 CUDA thread model

In the following, we describe the thread organization of CUDA. First of all, we all know that threads are the most basic unit of program execution and that CUDA's parallel computing is achieved by the parallel execution of thousands of threads. Figure 5.4 illustrates the different levels of the GPU's structure. CUDA's thread model, from small to large, is summarized as follows.

- Thread: Threads, the basic unit of parallelism.
- Thread Block: A block of threads, a group of threads that cooperate. They allow synchronization with each other, can exchange data quickly through shared memory and can be organized in 1, 2, or 3 dimensions.
- Grid: A group of thread blocks, which can be organized in 1 or 2 dimensions and share global memory.

The Kernel is the core program that executes on the GPU and this kernel function is run on a certain Grid. Each block and each thread has its ID, and we find the corresponding thread and thread block by the corresponding index. All threads started by a kernel are called a grid, and threads on the same grid share the same global memory space (Meunier et al. 2000) [20].

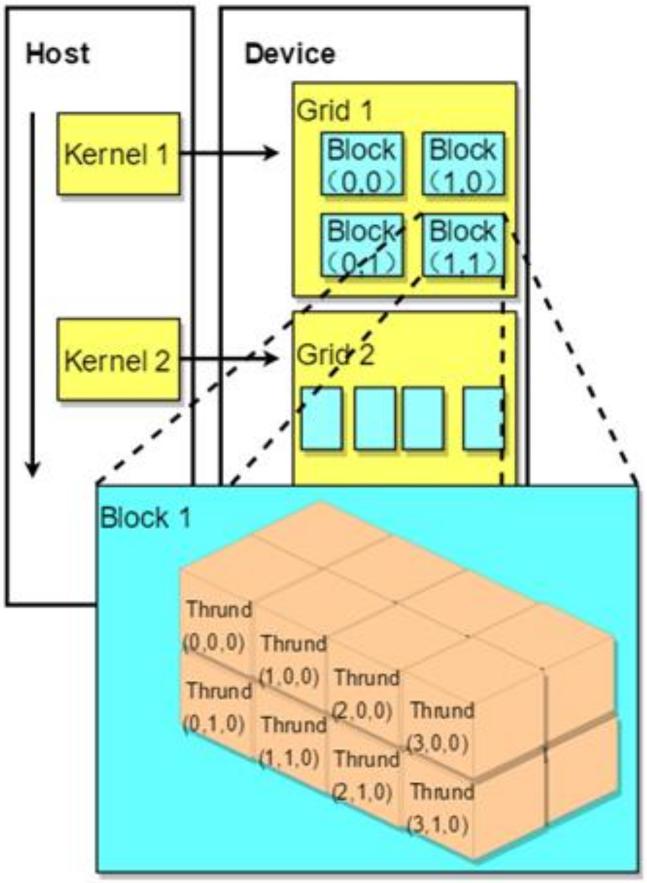


Figure 5.4: Different levels of the GPU's structure.

From a hardware point of view, the streaming processor is the most basic processing unit, and the final specific instructions and tasks are processed on the sps, which perform parallel computing, i.e. many sps at the same time. Multiple sps plus other resources make up a streaming multiprocessor [21]. Figure 5.5 compares CUDA's threaded model from a hardware perspective and a software perspective.

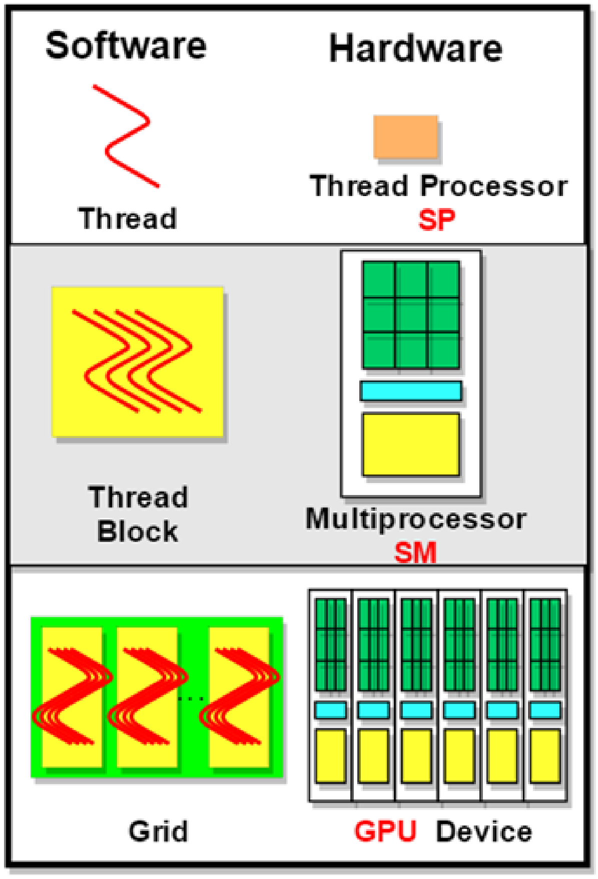


Figure 5.5: Hardware and Software CUDA thread model.

5.4 CUDA memory model

The memory model in CUDA is divided into the following levels:

- Each thread uses its own registers.
- Each thread has its own local memory.
- Each thread block has its own shared memory, which is shared by all threads in the block.
- Each grid has its own global memory, which can be used by threads in different thread blocks.

- Each grid has its own constant memory and texture memory, which can be used by threads in different thread blocks.

The speed at which threads can access these types of memory is (register > local memory >shared memory > global memory) [22]. Figure 5.6 represents the layers in the computer architecture where these memories reside.

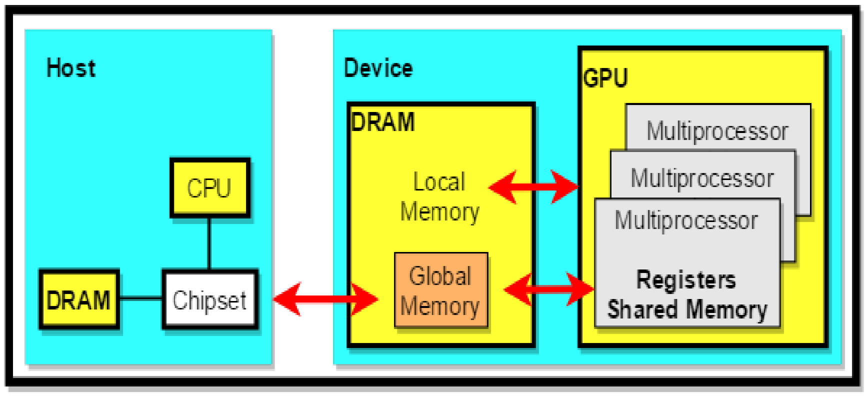


Figure 5.6: Different levels of memory exist in the computer architecture.

6. CUDA implementation

6.1 Hardware of the test system

Table 6.1 lists the hardware configuration of the test system on cuda01. All tests on CUDA I have done on this hardware platform.

Table 6.1: Hardware configuration on cuda01.

cluster	cuda01
CPU	Intel Core i5-3570K 4 cores 4 threads
GPU	NVIDIA RTX 2080 super 8GB
RAM	16 GB DDR3 1600MHz
Operating System	Ubuntu 18.04.5

6.2 Implementation

The main functions implemented in the CUDA code are the same as the serial version, except for the addition of memory copies and data transfers between the CPU and GPU, which are not described here, see the main.cu file for details. For memory management, I used a unified memory model and compared the performance with the classical memory management model.

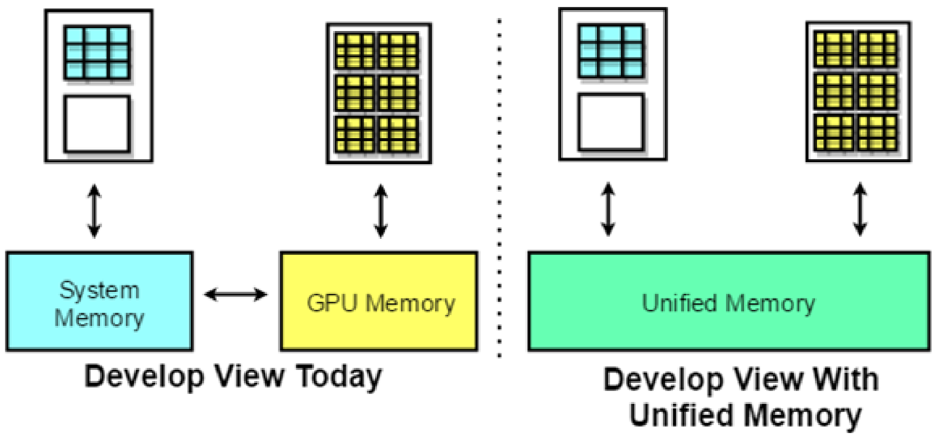
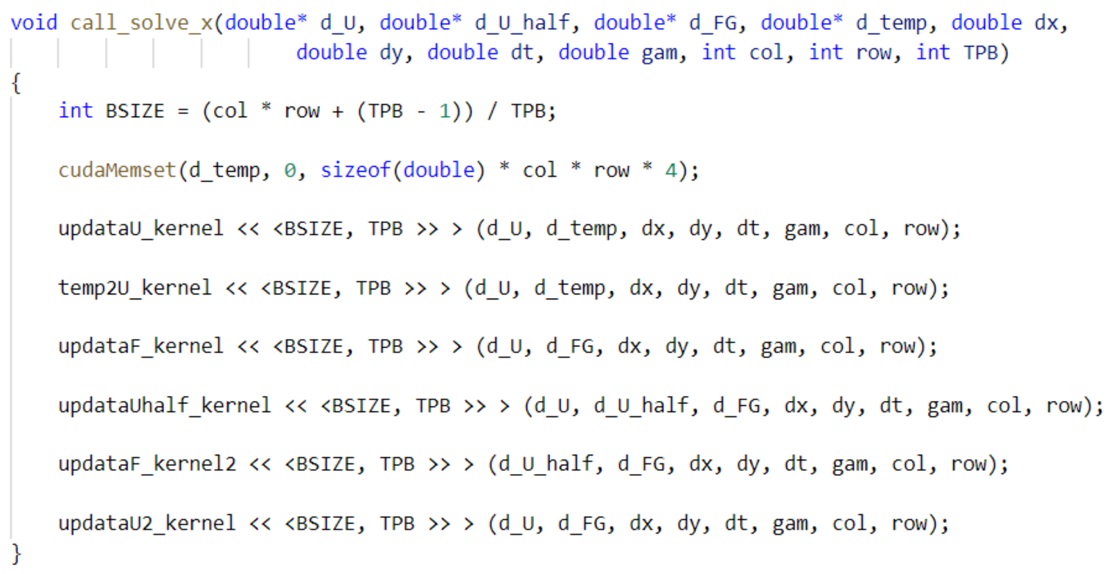


Figure 6.1: A simple illustration of unified memory.

NVIDIA introduced a unified memory model in CUDA 6 by creating a managed memory pool that is shared between the CPU and the GPU, bridging the gap between the CPU and the GPU [23]. both CPUs and GPUs can access managed memory using a single pointer. Figure 6.1 is a simple illustration of unified memory. The unified memory model reduces the cost of learning to program on CUDA platforms and makes it easy to port existing code to the GPU. By migrating data on-demand between the CPU and GPU, the unified memory model meets the performance needs of local data on the GPU, while also providing easy-to-use global shared data. The intricate details of this functionality are hidden by the CUDA driver and runtime to ensure that application code is easier to write. The key to migration is to get the full bandwidth from each processor [24]. It is worth noting that techniques such as stream can still be used while using the unified memory model, without affecting any CUDA functionality for advanced users.

Due to the dependencies between data, each calculation unit forms a separate kernel. The kernel split for the computation of the grid x direction (the y direction is essentially the same as the x-direction and will not be described too much) is shown in figure 6.2.



```
void call_solve_x(double* d_U, double* d_U_half, double* d_FG, double* d_temp, double dx,
                  double dy, double dt, double gam, int col, int row, int TPB)
{
    int BSIZE = (col * row + (TPB - 1)) / TPB;

    cudaMemset(d_temp, 0, sizeof(double) * col * row * 4);

    updataU_kernel << <BSIZE, TPB >> > (d_U, d_temp, dx, dy, dt, gam, col, row);

    temp2U_kernel << <BSIZE, TPB >> > (d_U, d_temp, dx, dy, dt, gam, col, row);

    updataF_kernel << <BSIZE, TPB >> > (d_U, d_FG, dx, dy, dt, gam, col, row);

    updataUhalf_kernel << <BSIZE, TPB >> > (d_U, d_U_half, d_FG, dx, dy, dt, gam, col, row);

    updataF_kernel2 << <BSIZE, TPB >> > (d_U_half, d_FG, dx, dy, dt, gam, col, row);

    updataU2_kernel << <BSIZE, TPB >> > (d_U, d_FG, dx, dy, dt, gam, col, row);
}
```

Figure 6.2: The kernel split for the computation of the grid x direction.

6.3 Output

As before, I used the txt file output from the program at grid size x * y = 1600 * 400 to plot the pressure distribution and density distribution at y=0.5, as well as the pressure and density contour plots. The output of the CUDA version was compared with the graphs from the serial txt file to confirm that the output of the parallel program was correct.

After comparison, the CUDA version outputs the same image as the serial program, verifying that the parallel program's calculations are correct.

6.4 Timing

Table 6.2 shows the runtime and speedup of different memory scheduling methods with different TPB for grid size = 1600 * 400. And in Figure 6.3, we show the speedup or the case. From the diagram, we can see that the use of unified memory can improve program performance. Because the unified memory model meets the performance requirements of local data on the GPU, it also provides easy-to-use global shared data, exploiting the principle of data locality to improve program performance. Maximum speedup is close to 240 times.

Table 6.2: CUDA program runtime and speedup.

TPB		32	64	128	256	512	1024
classic memory	time(s)	53	35	36	38	44	41
	speedup(time)	80.201	121.447	118.073	111.859	96.605	103.674
unified memory	time(s)	19	19	19	19	19	21
	speedup(time)	223.718	223.718	223.718	223.718	223.718	202.411

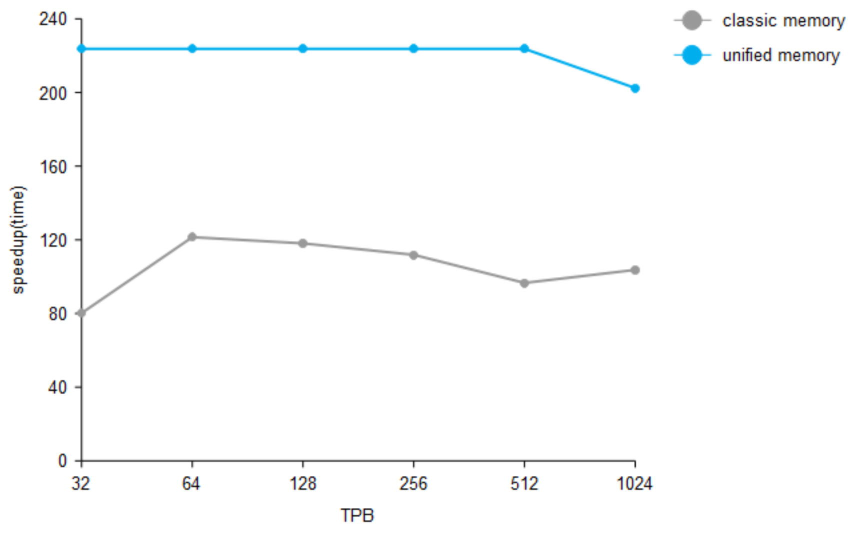
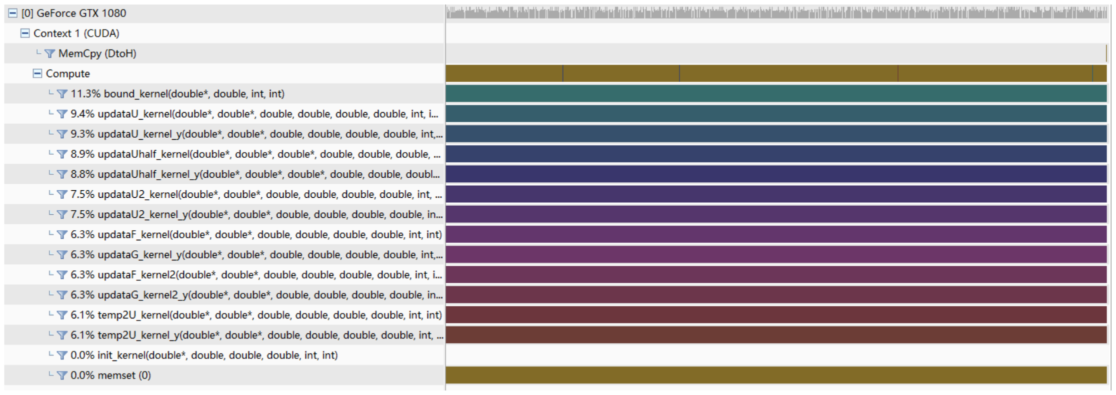


Figure 6.3: CUDA program speedup.

6.5 Profiling

The NVIDIA Visual Profiler is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. It is a GUI that provides information about the performance of kernels that are run on a GPU. I use this tool to profile the CUDA program at grid size = 1600 * 400. Figure 6.4 indicates that most of the time was spent on computation.



```
[0] GeForce GTX 1080
  Context 1 (CUDA)
    MemCpy (DtoH)
    Compute
      11.3% bound_kernel(double*, double, int, int)
      9.4% updataU_kernel(double*, double*, double, double, double, double, int, i...
      9.3% updataU_kernel_y(double*, double*, double, double, double, double, int,...
      8.9% updataUhalf_kernel(double*, double*, double*, double, double, double, ...
      8.8% updataUhalf_kernel_y(double*, double*, double*, double, double, doubl...
      7.5% updataU2_kernel(double*, double*, double, double, double, double, int, ...
      7.5% updataU2_kernel_y(double*, double*, double, double, double, double, in...
      6.3% updataF_kernel(double*, double*, double, double, double, double, int, int)
      6.3% updataG_kernel_y(double*, double*, double, double, double, double, int,...
      6.3% updataF_kernel2(double*, double*, double, double, double, double, int, i...
      6.3% updataG_kernel2_y(double*, double*, double, double, double, double, in...
      6.1% temp2U_kernel(double*, double*, double, double, double, double, int, int)
      6.1% temp2U_kernel_y(double*, double*, double, double, double, double, int, ...
      0.0% init_kernel(double*, double, double, double, int, int)
      0.0% memset (0)
```

Figure 6.4: NVIDIA visual profiler output of CUDA calculation.

7. Conclusions

The fluid simulation algorithms and parallel solutions mentioned in this paper can be applied to a wide range of scenarios, such as the shape of high-rise buildings, the aerodynamic layout of cars, and even advanced fans, blowers, etc. The parallel solutions can also be applied to other computationally intensive algorithms (such as atmospheric prediction, earthquake simulation, etc.) to improve model performance.

The performance is optimal when using MPI with non-blocking communication. A speedup of over 11 times was obtained with 64 processors. It is worth noting that the speedup increases more and more slowly with the number of processors, as MPI has a not insignificant overhead. With CUDA, a speedup of over 223 times was obtained with the addition of unified memory. So in terms of acceleration performance, I think CUDA is more promising.

In terms of coding difficulty, CUDA is less difficult than MPI because there is no need for data exchange and algorithm splitting between each processor. Therefore, in terms of generalisability, CUDA is more easily accepted. I believe that CUDA will have a good future in the field of fluid simulation.

Through this project, I have integrated my professional knowledge, gained a deeper understanding of parallel computing and bottleneck profiling and performance optimisation, and improved my programming skills. This process has also led me to think about technologies related to parallel computing.

In the next step of my research, I plan to use a three-level hybrid programming model of MPI + OpenMP + CUDA to schedule multiple GPUs for parallelism. Use MPI to map subtasks to each node and use OpenMP for parallelism in each node. Think of the GPU as a co-processor of the CPU, so that the CPU threads run in the GPU. In this way, coarse-grained parallelism between nodes and fine-grained parallelism within nodes is achieved to further increase the speed of the program.

Reference

[1] Peng J, et al. "Effect of vibration excitation on shock reflection." Acta Aeronautica Et Astronautica Sinica 38.8 (2017): 38-48.

[2] L. Zhang, “Appendix C Numerical solution and calculation program for two-dimensional oblique shock wave reflection on flat rigid walls.” Computational Fluid dynamics tutorial, Beijing, Higher Education Press, 2010.11, pp. 287-301.

[3] A. Attiya and Ahmed Kishk, “Modal analysis of a two-dimensional dielectric grating slab excited by an obliquely incident plane wave.” Progress in Electromagnetics Research, pp. 221-243, 2006.

[4] J. Li, “Anisotropy of excitons in two-dimensional perovskite crystals.” ACS nano, pp. 14, no. 2. 2156-2161, 2020.

[5] Z. Chow, “Hydrodynamic modelling of two-dimensional watershed flow.” Journal of the Hydraulics Division, pp. 99.11. 2023-2040, 1973.

[6] “MPI Standard Document.” MPI 3.1 specification, https://www.mpiforum.org/docs/ mpi-3.1/mpi31-report.pdf, June 2015, p. 108.

[7] Coello C A. Evolutionary multi-objective optimization: A historical view of the field[J]. Computational Intelligence Magazine, IEEE, 2006, 1(1): 28-36.

[8] Corne D W, Knowles J D, Oates M J. The Pareto envelope-based selection algorithm for multi-objective optimization[C]//Parallel Problem Solving from Nature PPSN VI. Springer Berlin Heidelberg, 2000: 839-848.

[9] Corne D W, Jerram N R, Knowles J D, et al. PESA-II: Region-based selection in evolutionary multi-objective optimization[C]//Proceedings of the Genetic and Evolutionary Computation Conference GECCO’2001. 2001.

[10] Du Zhihui, “Parallel programming technique for high performance computing-MPI parallel programming.” Tsinghua University Press, pp. 7-8, 2001.

[11] Wanai Li, Jianhua Pan, Yu-Xin Ren. "The discontinuous Galerkin spectral element methods for compressible flows on two-dimensional mixed grids." Journal of Computational Physics, 2018.

[12] Roman Trobec, Boštjan Slivnik, Patricio Bulić, Borut Robič. "Introduction to Parallel Computing." Springer Science and Business Media LLC, 2018.

[13] Cheng Hua Li, Laurence T, Yang, Man Lin. "Parallel Training of An Improved Neural Network for Text Categorization." International Journal of Parallel Programming, 2013.

[14] "Algorithms and Architectures for Parallel Processing." Springer Science and Business Media LLC, 2016.

[15] Hiroyasu T, Miki M, Watanabe S. The new model of parallel genetic algorithm in multi-objective optimization problems divided range multi-objective genetic algorithm[C]//

Evolutionary Computation, 2000. Proceedings of the 2000 Congress on. IEEE,2000, 1: 333-340.

[16] Xiao N, Armstrong M P. A specialized island model and its application in multi-objective optimization[C]//Genetic and evolutionary computation-GECCO 2003. Springer Berlin Heidelberg, 2003: 1530-1540.

[17] Streichert F, Ulmer H, Zell A. Parallelization of multi-objective evolutionary algorithms using clustering algorithms[C]//Evolutionary multi-criterion optimization. Springer Berlin Heidelberg, 2005: 92-107.

[18] Jaimes A L, Coello C A. MRMOGA: A new parallel multi-objective evolutionary algorithm based on the use of multiple resolutions[J]. Concurrency and Computation: Practice and Experience, 2007, 19(4): 397-441.

[19] Stanley T J, Mudge T N. A Parallel Genetic Algorithm for Multi-objective Microprocessor Design[C]//ICGA. 1995: 597-604.

[20] Yanyan Xu, Hui Chen, Reinhard Klette, Jiaju Liu, Tobi Vaudrey. "Chapter 19 Belief Propagation Implementation Using CUDA on an NVIDIA GTX 280." Springer Science and Business Media LLC, 2009.

[21] Jones B R, Crossley W A, Lyrintzis A S. Aerodynamic and aeroacoustic optimization of rotorcraft airfoils via a parallel genetic algorithm[J]. Journal of Aircraft, 2000, 37(6): 1088-1096.

[22] Meunier H, Talbi E G, Reininger P. A multi-objective genetic algorithm for radio network optimization[C]//Evolutionary Computation, 2000. Proceedings of the 2000 Congress on. IEEE, 2000, 1: 317-324.

[23] De Toro F, Ortega J, Fernández J, et al. PSFGA: A parallel genetic algorithm for multi-objective optimization[C]//Parallel, Distributed and Network-based Processing, 2002. Proceedings. 10th Euromicro Workshop on. IEEE, 2002: 384-391.

[24] Pospíchal P, Schwarz J, Jaros J. Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu[J]. 2010.