Hybrid Parallel Computation of OpenFOAM Solver on Multi-Core Cluster Systems

YUAN LIU

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2011

TRITA-ICT-EX-2011:4

Hybrid Parallel Computation of OpenFOAM Solver on Multi-Core Cluster Systems

YUAN LIU

Master's Thesis at Information Communication Technology
Supervisor: Jon-Anders Bäckar (Volvo Technology AB)
Examiner: Mats Brorsson

Abstract

OpenFOAM, an open source industrial Computational Fluid Dynamics (CFD) tool, which contains dozens of simulation application inside. A traditional approach to accelerate physical simulation process is to employ more powerful supercomputer. However, It is bound to expense large amount of hardware resources. In recent years, parallel and distributed computing is becoming an efficient way to solve such computational intensive application.

This thesis pick up the most used compressible reacting solver named dieselFoam as the research target. Through the analysis of code structure in solver equation and native MPI implementation in OpenFOAM, deploy two level parallelism structure on SMP cluster, which use Message Passing Interface (MPI) between each SMP nodes and OpenMP directives inside SMP node. The key idea is making use of feature of threads parallelism, reduce unnecessary MPI communication overhead, thereby achieve performance improvement.

The experiment results demonstrate application speedup by our solution, also in good agreement with the theoretical study. Based on the potential candidates analysis and performance results, we can conclude that the hybrid parallel model is proper for the acceleration of OpenFOAM solver application compare to the native MPI ways. Also through the discussion of the thesis, provides some suggestion about the future improvement area.

Acknowledgement

I would like to thank my advisors Joe-Anders Bäckar and Mats Brorsson for their help and guide during my thesis period. Also, i would like to thank PDC summer school of KTH which arouse my interest of parallel programming. Last but not the least, thank all my friends and my family support for my thesis work.

Contents

Contents v

List of Figures viii

List of Tables x

1 Introduction 1
- 1.1 Problem Statement . . . 1
- 1.2 Project Goals . . . 2
  - 1.2.1 Scope . . . 2
  - 1.2.2 Subject investigated . . . 3
  - 1.2.3 Related Work . . . 3
  - 1.2.4 Contributions . . . 3
  - 1.2.5 Thesis Organization . . . 4

2 DieselFoam 5
- 2.1 Geometry . . . 6
- 2.2 File Structure . . . 6
- 2.3 aachenbomb case directory . . . 7
  - 2.3.1 System directory . . . 7
  - 2.3.2 Constant directory . . . 12
  - 2.3.3 Time directory . . . 12
- 2.4 dieselFoam Solvers and UEqn Object . . . 12
  - 2.4.1 Directory structure . . . 13
  - 2.4.2 Linear Solver . . . 13
  - 2.4.3 PISO algorithm . . . 15
  - 2.4.4 Velocity equation and Navier-Stokes equations . . . 15
- 2.5 Running program . . . 16
- 2.6 Conclusion . . . 16

3 Parallel Computing 19

3.1 Architectures . . . 19
3.1.1 Symmetric multiprocessing (SMP) . . . 20
3.1.2 SMP cluster . . . 20
3.2 Parallel programming models . . . 21
3.2.1 Message Passing Interface (MPI) . . . 21
3.2.2 OpenMP . . . 22
3.2.3 Hybrid parallel model . . . 22
3.2.4 Implementation . . . 24
3.3 Comparison of parallel programming mode . . . 26
3.3.1 Pros and Cons of OpenMP and MPI . . . 26
3.3.2 Trade off between OpenMP and MPI . . . 26
3.3.3 Pros and Cons of hybrid parallel model . . . 28
3.4 Conclusion . . . 29

4 OpenFOAM in parallel **31**
4.1 Running in parallel . . . 31
4.2 Domain decomposition methodology . . . 32
4.3 Simple decomposition . . . 33
4.4 MPI communication description . . . 37
4.4.1 MPI Program structure . . . 37
4.4.2 Environment routines . . . 37
4.4.3 Communication schedule schemes . . . 37
4.4.4 Communication routines . . . 42
4.5 Conclusion . . . 43

5 Prototype Design **45**
5.1 Methodology . . . 45
5.2 Running platforms . . . 46
5.3 Performance profiling . . . 47
5.4 Profiling input . . . 48
5.5 Collaboration diagram for PBICG . . . 48
5.6 Profiling result . . . 48
5.7 Candidate analysis . . . 51
5.8 OpenMP . . . 56
5.8.1 Task parallelism . . . 57
5.8.2 Data parallelism . . . 61
5.9 Conclusion . . . 62

6 Evaluation **63**
6.1 Test case: Jacobi iterative method . . . 63
6.2 Performance results . . . 65

6.3 Discussion 67
6.4 Conclusion 71

7 Summary of conclusion and future work 73

Bibliography 75

List of Figures

2.1 Application structure of OpenFOAM [12] . . . 5
2.2 Geometry of dieselFoam aachenBomb case [7] . . . 6
2.3 Flow chart of dieselFoam solver . . . 8
2.4 Directory structure of OpenFOAM case . . . 9
2.5 Directory structure of OpenFOAM solver . . . 13
2.6 PBICG linear solver function call . . . 14
2.7 PCG linear solver function call . . . 14

3.1 Architecture of symmetric multiprocessing system . . . 20
3.2 Architecture of SMP cluster . . . 21
3.3 Flow chart of hybrid parallel programming model . . . 23
3.4 Execution sequence of hybrid parallel model . . . 25

4.1 Eight points regular mesh . . . 33
4.2 Two processors are distributed in X direction . . . 35
4.3 Two processors are distributed in Y direction . . . 35
4.4 Four processors are distributed in X-Y direction . . . 35
4.5 Two processors are distributed in Z direction . . . 36
4.6 Eight processors are distributed in X-Y-Z direction . . . 36
4.7 Execution procedure of MPI program . . . 38
4.8 Linear communication schemes . . . 39
4.9 When k=0, the node only contain root R, we record it as B0 . . . 39
4.10 When k=1, the binomial tree contains current root R1 and the existing sub-tree B0, we record it as B1 . . . 40
4.11 When k=2, the binomial tree contains current root R2 and the existing sub-trees B0 and B1, we record it as B2 . . . 40
4.12 When k=2, the binomial tree contains current root R3 and the existing sub-trees B0, B1 and B2 . . . 41
4.13 Communication binomial tree for 8 processors . . . 41

5.1 Flow chart of PBICG functions . . . 49

5.2 Collaboration diagram for PBICG solver 50

6.1 Comparison of speedup between Hybrid parallel model and MPI processes on the Jacobi iterative model 66
6.2 Comparison of speedup between OpenMP threads and MPI processes on Preconditioner function 68
6.3 Comparison of speedup between MPI processes and Hybrid parallel model with different map topology on PBICG solver. 69
6.4 Performance speedup of Hybrid parallel model on PBICG solver by 2 MPI processes and 8 OpenMP threads on each. 70

List of Tables

2.1 Keywords listed in fvSchemes [12] 10
2.2 Initial condition of time directory of dieselFoam 12

3.1 Comparison between OpenMP and MPI 27
3.2 Trade off between OpenMP and MPI 27

5.1 Mesh properties of aachenbomb case 48
5.2 Timing results of dieselFoam solver 49
5.3 Timing results of UEqn object 51
5.4 Timing results of PBICG solver 51

6.1 Performance of pure OpenMP implementation in Preconditioner function 66
6.2 Performance of pure MPI implementation in Preconditioner function 66
6.3 Performance of pure MPI implementation in PBICG solver 67
6.4 Performance of hybrid parallel implementation in PBICG solver . 67
6.5 Time scale of UEqn object 71
6.6 Time scale of dieselFoam solver 72

Chapter 1

Introduction

Currently, when industrial companies simulate their Computational Fluid Dynamic (CFD) projects, most of such works carry out on the workstation or supercomputer. However, as the market competition increasing, such experiment platforms expose many shortcomings. First of all, this approach produces computational performance efficiency at high computer resources cost, secondly, it needs to be shorten its execution time one step closer. Therefore, in order to keep up with increasing demand, parallel computing, a new technique is emerged by the development of computer science and numerical computing, which solve fluid numerical simulation at significant efficient way. This thesis presents a hybrid parallel computing method for PBICG solver inside dieselFoam class of OpenFOAM Computational Fluid Dynamic (CFD) tools. In this chapter, section 1.1 mainly introduces the topic of this thesis and the motivation behind them. Section 1.2 presents the goals of this thesis. Section 1.3 is the overview of the remaining chapter.

1.1 Problem Statement

Computational fluid dynamic (CFD) is a subject using numerical methods to solve fluid flow problem. With the increasing development of computer science, numerical computing and fluid mechanics, CFD is becoming one of the most significant methods in industrial area, even used on some basic step of physical analysis. Some impossible problem in the past now can be simulated on workstation or supercomputer nowadays by CFD approach.

OpenFOAM (Open Field Operation and Manipulation) CFD toolbox is an open source Computational Fluid Dynamic application that used to simulate a variety of industrial problem. These problems include complex fluid flows involving chemical reaction, turbulence and heat transfer, solid dynamics, elec-

tromagnetic and the pricing of financial options, etc [12]. It's free of charge can obviously reduce the cost instead of using commercial one. Another feature of OpenFOAM is the core technology based on C++ programming language. By using a set of C++ module, programmer can design CFD solver applications in more flexible and extensible manner.

Volvo Technology AB (VTEC) is an innovation, research and development company in Volvo Group AB. The mission of this company is to develop leading product in existing and future technology areas which high importance to Volvo [3]. CAE department leverage the computer software to aid engineering's task, is interesting in using CFD tools for their daily simulation in variant activities such as automobile, aerospace and boat. Such efforts reduce the computer hardware resources cost and manpower of the simulation process by moving the model simulation environment from real physical and chemistry world to computer based virtual world.

In this thesis, hybrid parallel programming method is taken to accelerate the performance of OpenFOAM solvers execution beyond the traditional achievement. OpenFOAM application primitive is only based on single-level pure MPI model, that each MPI process is executed in each node. In order to improve its performance and reduce overhead caused by MPI communication. One good method is to map multi-level parallelism to OpenFOAM solver, which coarse-grain parallelism is done by message passing (MPI) among nodes and fine-grain is achieved by OpenMP loop-level parallelism inside each node. Moreover, The report compares each parallelism prototype, analysis and benchmark to the others, lay a solid foundation for the future work.

1.2 Project Goals

1.2.1 Scope

The goal of this project is to increase the computing performance of PBICG solver inside dieselFoam class, since OpenFOAM is big open-source software which contains hundreds of Mega Bytes, so we just concentrate on the most interchangeable source code, make sure this prototype has commonality and portability characteristic, can be taken as an exemplary role for future work.

Our prototype approach is adding thread parallelism (OpenMP) directive to the most time consuming loop iteration of the solver function, integrating it to native Message Passing Interface (MPI) implementation. Also comparison is presented with the native parallel approach of OpenFOAM application, discussion about the efficiency of multiple-level parallelism.

1.2.2 Subject investigated

In this thesis, the works are mainly focus on these points:

- Analyzing OpenFOAM programming model, code structure, compiling procedure.
- Studying basic parallel programming model. Investigating native domain-decomposition mechanism and code structure.
- Benchmarks on PBICG solve function of dieselFoam class.
- Discussing the potential source code which can be parallelized, function data structure investigation, employing OpenMP direct to time consuming part, performance analysis on hybrid parallel programming model.

1.2.3 Related Work

Different proposals for parallel computing of CFD solver and hybrid parallel programming work have been done before. Paper [17] exploits parallel computing aspects of multi-physics CFD methods, design and parallelization of an object-oriented molecular dynamics method. The authors of the thesis [11] have exploited parallelism possibility in top level of equation calculation part inside dieselFoam, they have also suggested a few ways to investigate OpenFOAM source code. In [10] and [18], the authors describe and compare pure MPI (Message Passing Interface), pure OpenMP (with distributed shared memory extensions) and Hybrid parallel programming model, also show us the performance impact made by the different nodes topology.

Based on previous work, in this thesis, we introduce the internal MPI implementation structure and present hybrid parallel implementation by adding OpenMP directive. Work with the inner equation calculation part of dieselFoam which go deeper than [11]. Besides, compare OpenMP 3.0 tasking parallelism to [11]'s implementation at the same place. Furthermore, discuss about the potential parallelized code part of dieselFoam.

1.2.4 Contributions

The following aspects can be reached in this project:

- Give a distinct outline of PBICG solver function of dieselFoam solver application that can be a reference to other solver in OpenFOAM toolbox.

- Present potential parts that can be parallelized.
- Propose the prerequisite when the parallelism work should be put in OpenFOAM application.
- Reach a reasonable speedup by adding OpenMP directive on PBICG solve function.

1.2.5 Thesis Organization

This thesis is organized as followed. In Chapter 2, mainly introduce about the structure of dieselFoam solver, which is one of dozens solvers in OpenFOAM application. OpenFOAM programming model, the code structure of dieselFoam and UEqn are also presented in this chapter. Then, in Chapter 3, parallel programming model contains MPI and OpenMP are described, also include but not limited to SMP cluster architecture, language characteristic. Besides, the comparison is mentioned in this chapter between MPI and OpenMP as well as hybrid parallel programming model. Next, Chapter 4 tell us about everything relate to MPI implementation inside OpenFOAM, code structure, algorithm and running mechanism will be included. Subsequently, profiling result and function code structure are analyzed in Chapter 5, various OpenMP directives are demonstrated in this chapter with candidates functions. Prototype result evaluation and performance discussion are carried out in chapter 6. Finally, Chapter 7 is the last chapter of this thesis, concludes the whole article, and gives some recommendations for the future work.

Chapter 2

DieselFoam

OpenFOAM is an open source, free distribution and modification of CFD simulation software. The program structure shows in Figure 2.1 give us some feeling about it. It includes dozens of basic solver applications, such as incompressible flows, compressible flows, combustion, heat transfer and electromagnetic. Moreover, since OpenFOAM use C++ programming language as base language, supported by its libraries you can improve existing solvers by object-orient method, or even create a totally new one. This thesis chooses DieselFoam as research object. It is a compressible reacting solver with Lagrangian evaporating used for diesel spray and combustion [12]. The reason we choose it is that dieselFoam solver has a lot of common ground with majority solvers lay in OpenFOAM application. Besides, the problem size of dieselFoam is large enough for the prototype evaluation. In this chapter, we introduce dieselFoam key equation, file structure, mesh, I/O files, etc. The most important header file related to this thesis called UEqn is explained.

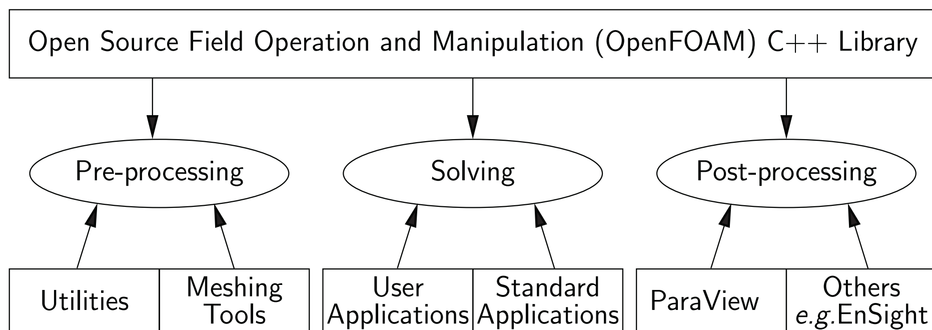

Figure 2.1: Application structure of OpenFOAM [12]

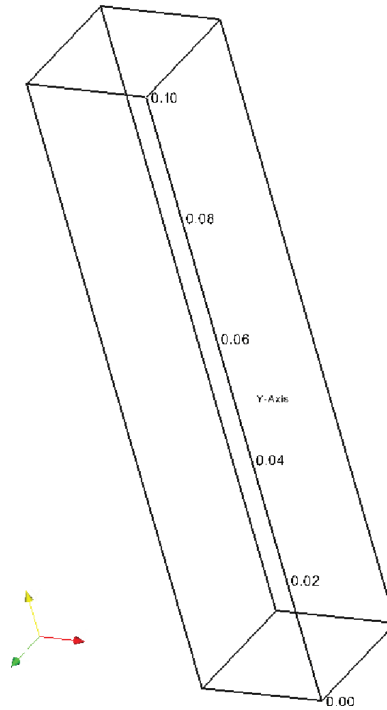

Figure 2.2: Geometry of dieselFoam aachenBomb case [7]

2.1 Geometry

DieselFoam geometry is a block with 0.01 in width and length, 0.1 in height which shows in Figure 2.2. It's fulfilled with air. At the top of this model, one injector there injects n-Heptane (C7H16) into combustion chamber. The discrete droplets there would evaporate and burn. Then gas phase reactions take place there, range from a reaction scheme with 5 species and one reaction up to a reaction scheme involving approximate 300 reactions and 56 species [7].

2.2 File Structure

As a matter of fact, OpenFOAM application consists of solver directory and case directory. In default,

The path of dieselFoam solver is:
`$HOME/OpenFOAM/applications/solvers/combustion/dieselFoam`

And the path of "aachenBomb" case is:
`$HOME/OpenFOAM/tutorials/combustion/aachenBomb`

In computer science, solver application and case files can be treated as executable file and the environment where the executable file running. As shown in Figure 2.3, when the program starts, dieselFoam solver first checks the configuration file (input data) under case directory which pass the arguments to it. Then, each CFD equation will be initialized and calculated by its selected solve function. At last, each runtime iteration generate result displaying and output. The understanding of dieselFoam solver is very important since all equation objects lay down here. Moreover, case directory part still need to be illustrated since it decides which equation would be matched to object solver. According to our project content, UEqn object part will be analyzed and implemented in details in later parts. Everything related to UEqn object will be given in Chapter 5. And the contents of solver and case directories of dieselFoam will be presented later in this chapter.

2.3 aachenbomb case directory

This section present the role of each sub-directory in case directory. Figure 2.4 shows the structure chart of aachenBomb case directory. Usually, the case name can be named as anything, but in practical, it is always assigned to present meaningful aspect. For example, the case name of dieselFoam is called "aachenBomb" which means this product comes from Aachen University. The case directory can be located in anywhere but solver application must be executed under it. By default, types *tut* at the command line can transfer to default case directories path as following:

```
$HOME/OpenFOAM/tutorials
```

2.3.1 System directory

As we seen in Figure 2.4, the basic structure of system directory contains four files inside, which three of them must be presented in order to run correctly. These files are the fvSolution file specifying the chosen solvers for linear equation, the fvSchemes files selects numerical schemes and controlDict file control "time" information of OpenFOAM program. In addition, if the program needs to be running on parallel mode, one more file named decomposeParDict must be provided as well.

- **fvSolution:**: The fvSolution file contains equation linear solvers, selected algorithms, etc. The options can be classified as two types based

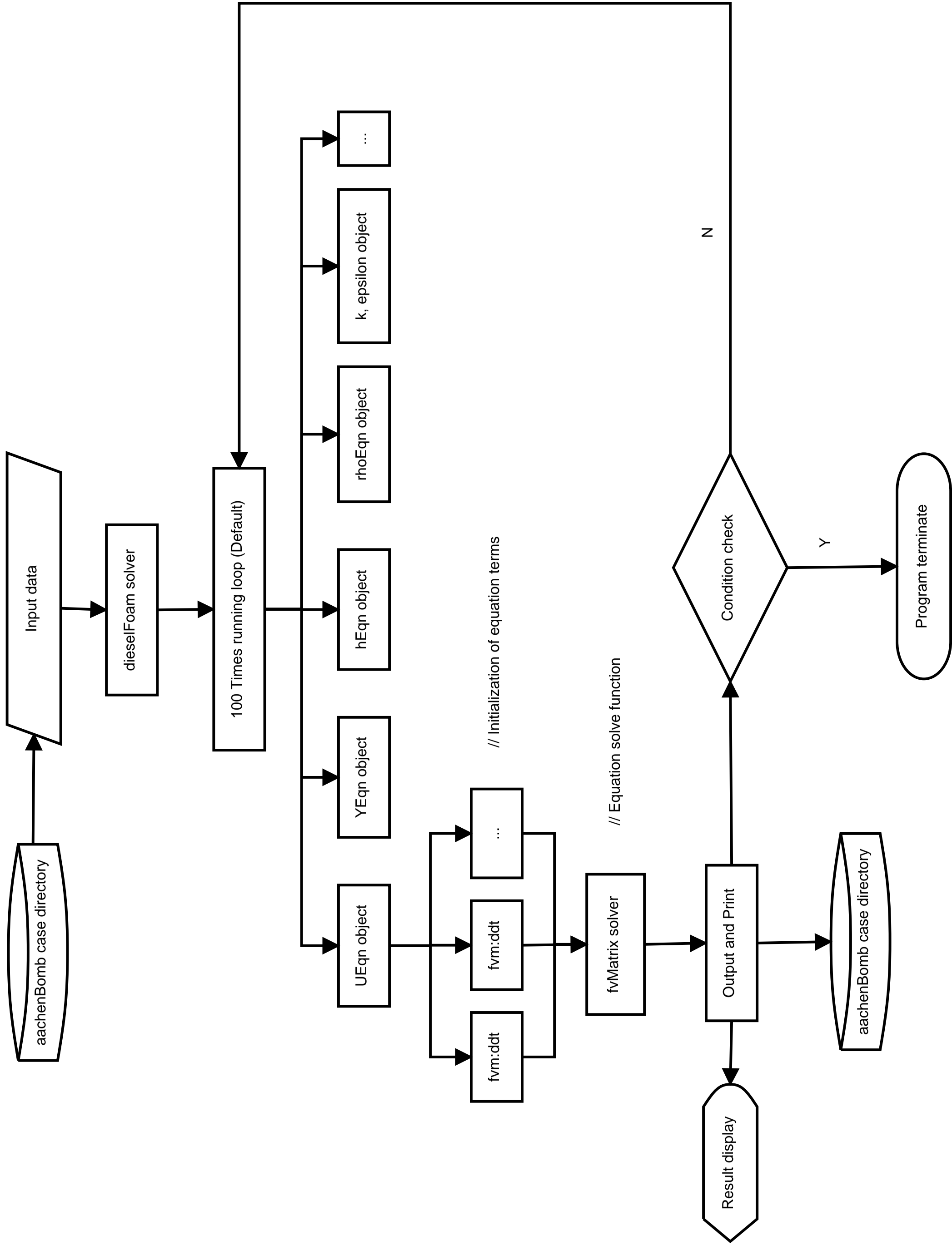

Figure 2.3: Flow chart of dieselFoam solver

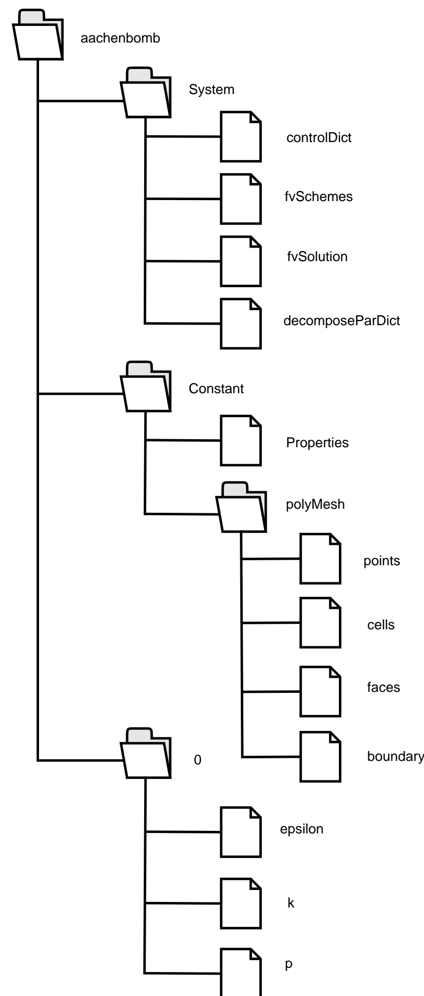

Figure 2.4: Directory structure of OpenFOAM case

on matrices type: asymmetry and symmetry. The first entry showing in Listing 2.1 are linear solvers, which indicates the methods of number-crunching to solve the set of linear equations for numerical equation [12]. linear solvers (PBICG or PCG showing in Listing 2.1) are different from application solvers (dieselFoam solver in Figure 2.3) which only describe by several equations and algorithms to solve specific CFD problem. However, the linear solvers here only in charge of solving particular equations. In the following discussion, we use "solver" instead of "linear solver" in most place, hope this explanation could avoid some ambiguity.

Listing 2.1: The source code of fvSolution

Table 2.1: Keywords listed in fvSchemes [12]

Keyword	Category of mathematical terms
interpolationSchemes	Point-to-point interpolations of values
snGradSchemes	Componenet of gradient normal to a cell face
gradSchemes	Gradient ∇
divSchemes	Divergence $\nabla \bullet$
laplacianSchemes	Laplacian ∇^2
timeScheme	First and second time derivatives $\partial/\partial t$, $\partial^2/\partial^2 t$
fluxRequired	Fields which require the generation of a flux

```
solvers
{
  p
  {
    solver PCG;
    preconditioner DIC;
    tolerance 1e-06;
    relTol 0;
  }

  U
  {
    solver PBiCG;
    preconditioner DILU;
    tolerance 1e-05;
    relTol 0;
  }
}

PISO
{
  nCorrectors 2;
  nNonOrthogonalCorrectors 0;
  pRefCell 0;
  pRefValue 0;
}
```

- **fvSchemes:**The fvSchemes file is used to select numerical schemes for each part of numerical equation which be included in header files such as UEqn.H, for example, gradient ∇ represents gradScemes. Table 2.1 and Listing 2.2 show the keywords list and source code of fvSchemes.

Listing 2.2: The source code of fvSchemes

```
ddtSchemes
{
  default Euler;
}

gradSchemes
{
  default Gauss linear;
  grad(p) Gauss linear;
}

divSchemes
{
  default none;
  div(phi,U) Gauss linear;
}

laplacianSchemes
{
  default none;
  laplacian(nu,U) Gauss linear corrected;
  laplacian((1|A(U)),p) Gauss linear corrected;
}

snGradSchemes
{
  default corrected;
}

fluxRequired
{
  default no;
  p ;
}
```

- **controlDict:** controlDict file contains some configuration information needed by solver when application start running. These data control input, output and the essential programme part, such as the intervals of execution time.
- **decomposeParDict (optional):** decomposeParDict file is used by decomposePar utility in parallel computing, it describe how to break up the mesh according to parameters inside file. We will talk about more on this in Chapter 4.

Table 2.2: Initial condition of time directory of dieselFoam

Variable	Initial conditions
Epsilon	internalField uniform 90.0, wall zeroGradient
K	internalField uniform 1.0, wall zeroGradient
N_2	internalField uniform 0.766, wall zeroGradient
O_2	internalField uniform 0.233, wall zeroGradient
P	internalField uniform 5e+06, wall zeroGradient
U	internalField uniform (0,0,0), wall uniform (0,0,0)

2.3.2 Constant directory

The structure of constant directory of aachenBomb is the same as other constant directory of solver application in OpenFOAM, each file presents different physical variants. Under the directory, some properties files can be found which describing spray, injector, combustion, chemistry and so on. In default, the mesh configuration files are put in the polyMesh subdirectory. It includes properties of the mesh which need to be solved.

2.3.3 Time directory

Initial conditions can be found in time directory. The 0 directory shows in Figure 2.1 is time directory. It is named according to the startTime value in controlDict file. Inside it, each variant such as pressure (p), velocity (U) and epsilon contain the initial values needed by solver application. When program start running, new time directories will be created time by time, results are saved inside each one. Table 2.2 presents initial conditions of 0 directory of dieselFoam.

2.4 dieselFoam Solvers and UEqn Object

In this section, we introduce the directory structure of dieselFoam solver. The explanation focus on software engineering aspect, more physical details can be found in OpenFOAM user guide [12].

Also in this part, velocity equation UEqn object will be illustrated through two aspects: PISO algorithm of dieselFoam is described in section 2.4.3 and Navier-Stock equation is analyzed in section 2.4.4.

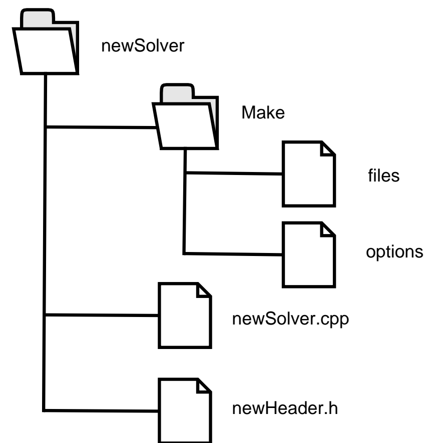

Figure 2.5: Directory structure of OpenFOAM solver

2.4.1 Directory structure

As shown in Figure 2.5, inside the directory of dieselFoam solver, there are main files, header files and Make directory. The name of main files or header files can be anything, but two fixed files must appear inside Make directory, they are "files" and "options".

The two responsibility of "files" file are:

- Show which files need to be compiled.
- Show the type of object files. Files can be compiled as executable file or library.

For "options" file, the intentions are:

- Show the path of included header files.
- Show the path of loading library.

2.4.2 Linear Solver

As we seen in section 2.3.1, Listing 2.1 demonstrates each equation objects (p, U, etc..) with its own specific solver. These are exactly the linear solvers which can be divided into different types due to different purpose. Two most common linear solvers are PBICG and PCG, one is for the asymmetric matrix class and another is for the symmetric matrix. This kind of matrix class is so called lduMatrix in OpenFOAM, more information about this matrix can

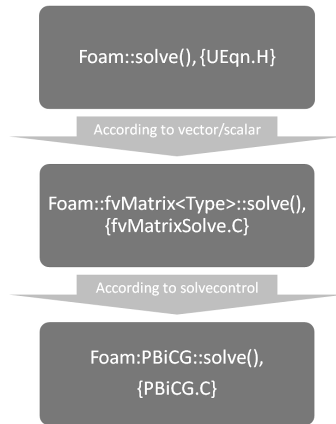

Figure 2.6: PBICG linear solver function call

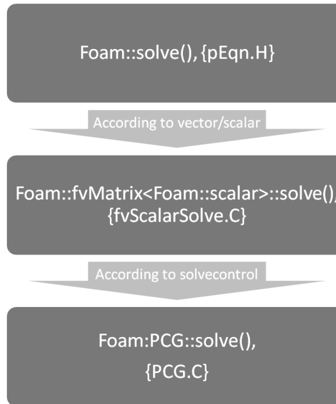

Figure 2.7: PCG linear solver function call

be found in section 5.7. Through fvMatrix interface, each equation object solve function will invoke different linear solver according to the keywords in fvSolution file. Since our research target is UEqn object in dieselFoam, only PBICG will be presented in later. Figure 2.6 and 2.7 shows the different function call between PBICG and PCG.

In dieselFoam, we get variable equation relating to the linear solver as following:

- **PBICG:** hEqn, YEqn, UEqn and k, epsilon.

- **PCG:** rhoEqn, pEqn.

2.4.3 PISO algorithm

In OpenFOAM, two types of method are provided to solve the Navier-Stokes equations, one is SIMPLE algorithm (Semi-Implicit Method for Pressure-Linked Equations), and another is PISO algorithm (Pressure Implicit with Splitting of Operators). The differences between them are as following:

- The momentum corrector is executed more than one in PISO.
- SIMPLE algorithm apply under-relaxation when solve pressure equation.

dieselFoam use PISO algorithm, which the steps of it can be divided into:

1. Set up boundary initial conditions.
2. Solve the discredited momentum equation.
3. Compute mass fluxes.
4. Solve the pressure equation.
5. Correct the mass fluxes.
6. Correct the velocities according to the pressure condition.
7. Update boundary conditions.
8. Repeat from 3 according to the number of times.

2.4.4 Velocity equation and Navier-Stokes equations

UEqn object sets up the calculation for the Navier-Stokes equation which used in a lot fluid flow phenomena. Navier-Stokes equations are time-dependent equation [1]. The flux of U and phi are calculated through the previous value of U. According to the priority of math calculation, each part of argument separate by math symbols in UEqn object can be treated simultaneously. In other words, it can be parallelized by task constructs [10]. Chapter 5 will show parallel version of Navier-Stokes equation both in "Section" construct and "Tasking" construct.

Listing 2.3: Navier-Stokes equation

```
fvVectorMatrix UEqn
(
        fvm::ddt(U)
```

```
    +   fvm::div(phi, U)
    +   turbulence->divDevRhoReff(U)
    ==
        rho*g
    +   dieselSpray.momentumSource()
);

If (momentumPredictor)
{
  solve(UEqn == -fvc::gra(p));
}
```

2.5 Running program

After we successfully install OpenFOAM and Third-Party package, we can copy the tutorial directory to other place, then begin to run a simple example.

1. Copy the tutorial directory to run directory by command:
   - `mkdir -p $FOAM_RUN`
   - `cp -r $FOAM_TUTORIALS $FOAM_RUN`
2. Choose simpleFoam as running example:
   - `cd $FOAM_RUN/tutorials/simpleFoam`
3. Running blockMesh which create mesh, by typing:
   - `blockMesh`
4. Execute your selected solver, The command here is simpleFoam:
   - `simpleFoam`
5. By third-part application ParaView, you can view you mesh visually:
   - `paraFoam`

2.6 Conclusion

Nowadays, CFD researchers find that even they increase the hardware resource, but the performance always be narrowed by the endless demand of large fluid application. These problems are caused by:

1. Most CFD programs have large-scale characters in its design and simulation procedure, and these features are growing very quickly.

2. CFD programs always require precise research process in reasonable execution time.

Therefore, even the most excellent computer sometime could not give a satisfied result, or could not show an acceptable simulation in a cost-efficient manner. Although the improvement of numerical algorithm used in program can improve performance to some extent, but many studies indicate that parallelization seem to be the only way can significantly improve efficiency of CFD program.

Next chapter, the introduction of parallel programming with its programming models will be provided.

Chapter 3

Parallel Computing

"Any sufficiently advanced technology is indistinguishable from magic"
-Arthur C. Clarke, 1961 [6]

Parallelism is a pretty old terms which appears everywhere in our daily life. Look around, people perform a lot of activities in parallel manner, building construction, automotive manufacturing, etc. they involve a lot of works done by simultaneously. The most familiar term is assembly line, which means the components of product are performed at the same time by several workers.

The concept of parallel computing in computer science is decomposing the given task into several small ones. Executing or solving them at the same time in order to make the computational process faster. With the emergence of large-scale application, the development of computer system architecture and network technology, the status of parallel programming was becoming more and more important. Up to now, parallel programming is ready and mature for a lot of industrial application acceleration based on previous research outputs.

In this chapter, we will talk about parallel computer architecture in section 3.1. Section 3.2 briefly describes the characters and implementation of three parallel programming models: OpenMP, MPI and hybrid parallel separately. In the last section, comparison of different parallel paradigm will be given.

3.1 Architectures

By development of computer architecture, a number of processors could access one single memory in an efficient way. Hence, the concept of clustered SMP is proposed and becoming more and more popular as parallel architecture. More important, SMP cluster is an excellent platform for hybrid programming model based on previous research [13]. In this section, we briefly present single

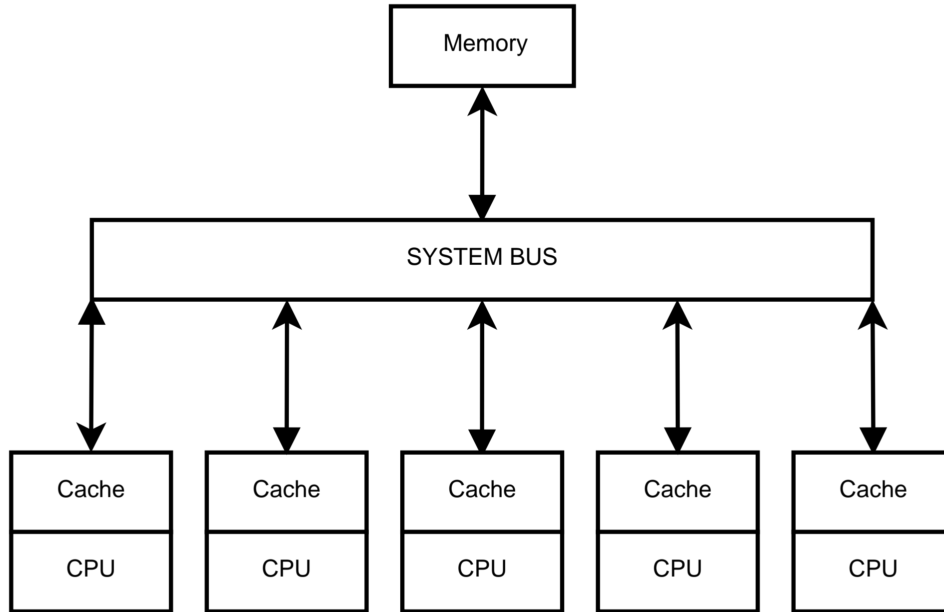

Figure 3.1: Architecture of symmetric multiprocessing system

SMP and clustered SMP architecture.

3.1.1 Symmetric multiprocessing (SMP)

Symmetric multiprocessing system or called shared memory system is a multiprocessor computer which has more than one processor in it and sharing a global memory or other resources. The main bottleneck of SMP system is its scalability: as the number of processors increased, the memory consumption becomes severe. Figure 3.1 shows a shared memory system.

3.1.2 SMP cluster

A hybrid parallel system mode just like a prototype of SMP cluster which represents both shared memory system and distributed memory system. It consists of a number of SMP nodes, and each node contains more than one processors.

Each SMP node is connected by system interconnection, and carries out communication mainly by MPI Message Passing. Within these nodes, OpenMP parallelization could be arranged. This kind of hybrid system is usually more efficient than pure MPI system in previous study. This is examined in this

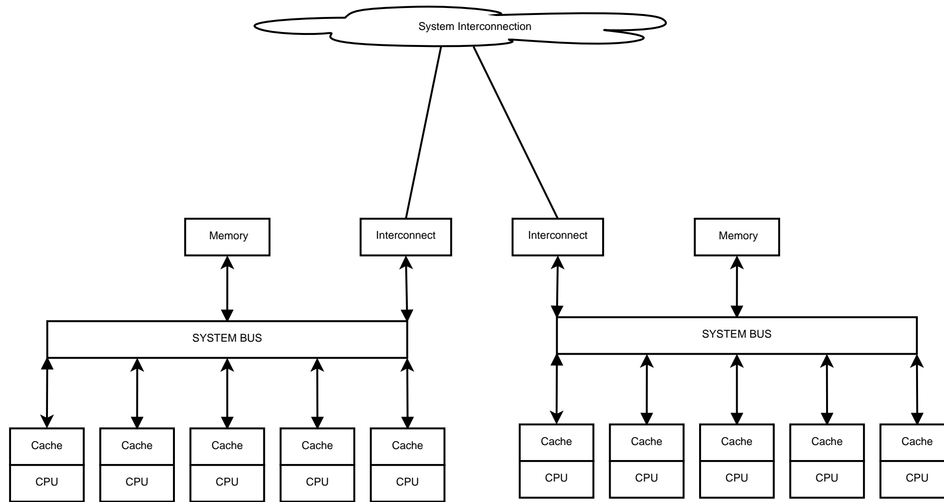

Figure 3.2: Architecture of SMP cluster

project shown in later chapters. Figure 3.2 shows the architecture of a simple SMP cluster.

3.2 Parallel programming models

In order to produce more effective parallelism performance on SMP cluster, hybrid parallel programming concept is proposed, which take advantage of both process level and thread level parallel models. This section presents the characters of MPI and OpenMP programming paradigms in sequence. After that, section 3.2.3 discuss about the feature of hybrid parallel programming model.

3.2.1 Message Passing Interface (MPI)

Message Passing Interface is a library specification for message passing, it becomes API standard in distributed memory model area. MPI implementation have a lot of advantages, one of importance is that MPI can be implemented on almost popular UNIX and Linux system. Furthermore, explicit control mechanism provides better performance and portability. It's very suitable for coarse-grained parallelism implementation. In SPMD (Single Process, Multiple Data) parallel programming, MPI process is allowed to access its partial local memory, and communication through message sending and receiving.

3.2.2 OpenMP

OpenMP is an API for shared memory parallel programming. It is an another parallel programming standard which based on compiler directives for C/C++ or FORTRAN. OpenMP provides a set of directives, library routines and environment variables. It uses high level abstraction by fork-join model, can be implemented on shared memory system.

Recently, OpenMP model can be implemented on cluster of distributed virtual shared memory, but need to be supported by specific compiler, e.g. Intel Cluster OpenMP. In SMP or DSM (Distributed shared memory) system, OpenMP thread is allowed to access the global memory address.

3.2.3 Hybrid parallel model

OpenMP and MPI are two popular approaches in parallel programming area, the features are shortly exhibited as following, more information can be found in Table 3.1:

- OpenMP: threads-level, shared memory, implicit control mechanism, poor scalability, threads synchronization overhead.
- MPI: process level, distributed memory, explicit control mechanism, good scalability, communication overhead.

According to the points above, OpenMP utilize shared memory model, it is difficult to port the code into distributed memory system. Although MPI introduce distributed memory model, but programming models are relatively complicated, load imbalance are also the big problem of MPI program.

Moreover, pure MPI implementation is easily jammed by sending and receiving large number message. In such case, one solution is proposed that, by combination of MPI and OpenMP, move the MPI communication to OpenMP computation, move the unnecessary work to inter-nodes instead of intra-nodes. In additional, multi-level parallelism let us exploit potential parallelism parts as much as possible. This new parallel model is so called hybrid parallel programming.

Hybrid parallel programming can be fully utilized the advantage of MPI and OpenMP models, it allows us to make use of explicit control mechanism of MPI between each node, and fine-grain parallelism inside node by OpenMP threads.

Figure 3.3 shows flow chart of hybrid parallel programming model. One global domain is divided between each node (one process per node), and each part is further divided between threads. As we have seen, hybrid parallel model is naturally related to the architecture of SMP cluster.

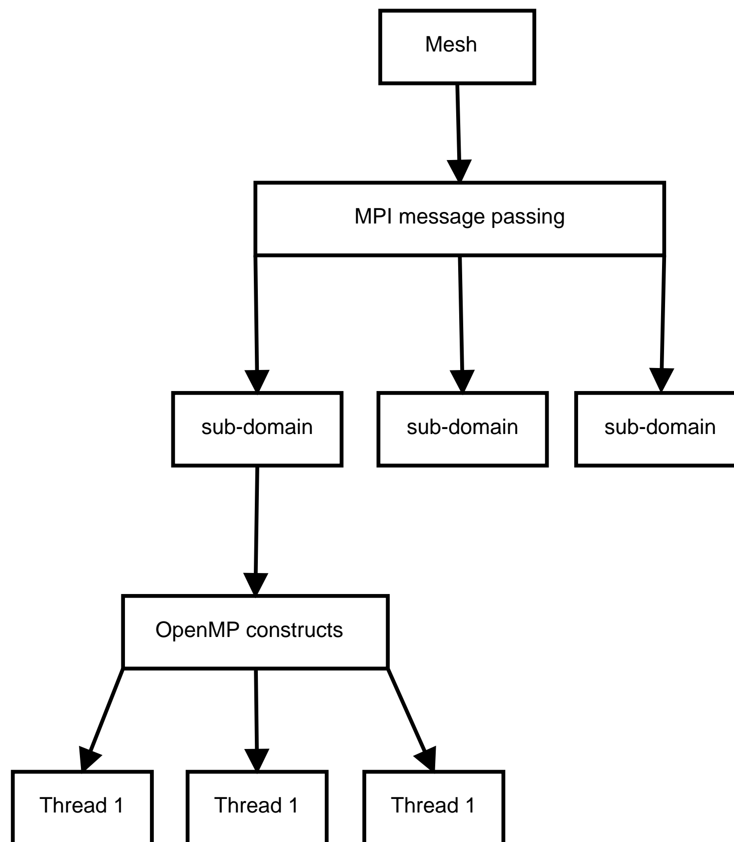

Figure 3.3: Flow chart of hybrid parallel programming model

Hybrid parallel model can be classified as masteronly or overlapping scheme according to time and ways how MPI processes communicate:

Hybrid masteronly (non-overlap communication and computation)

MPI only outside of parallel regions, message passing carry out by single thread or master thread. When procedure goes into communication part, all other thread are sleeping when master thread communicates. Listing 3.1 shows Masteronly schemes pseudo code.

Listing 3.1: Masteronly pseudo code

```
while (conditional statement)
{
  // master thread only
  MPI_Send()
  MPI_Recv()
```

```
  #pragma omp parallel
  for()
}
```

This project work is implemented by hybrid masteronly style. The mesh is divided into different zones by MPI function call, after initialization, boundaries are exchanged by master threads (communication work) and OpenMP loop-level parallelism is occurred in solver function by all threads (computation work).

Overlapping communication and computation

Listing 3.2: Overlapping pseudo code

```
if (thread_rank< N)
{
  // exchange halo data
  MPI_Send()
  MPI_Recv()
}
else
{
// computation part do not need exchange halo data
}

// computation part need exchange halo data
```

Listing 3.2 shows another type of communication scheme. All other threads are computing when master thread or a few threads doing communication.

3.2.4 Implementation

Listing 3.3 is a pseudo code of hybrid parallel mode. This code use standard MPI_INIT and MPI_FINALIZE commands to initialize and finalize parallel program as general MPI program. And inside parallel region, each process spawns a number of threads.

From Figure 3.4, we could also see that each MPI process spawn to several threads under *#pragma omp parallel* directive. Threads parallelism only occurs inside parallel region, the rest of that still run in single thread.

Listing 3.3: Hybrid model pseudo code

```
int main(int argc, char** argv)
{
```

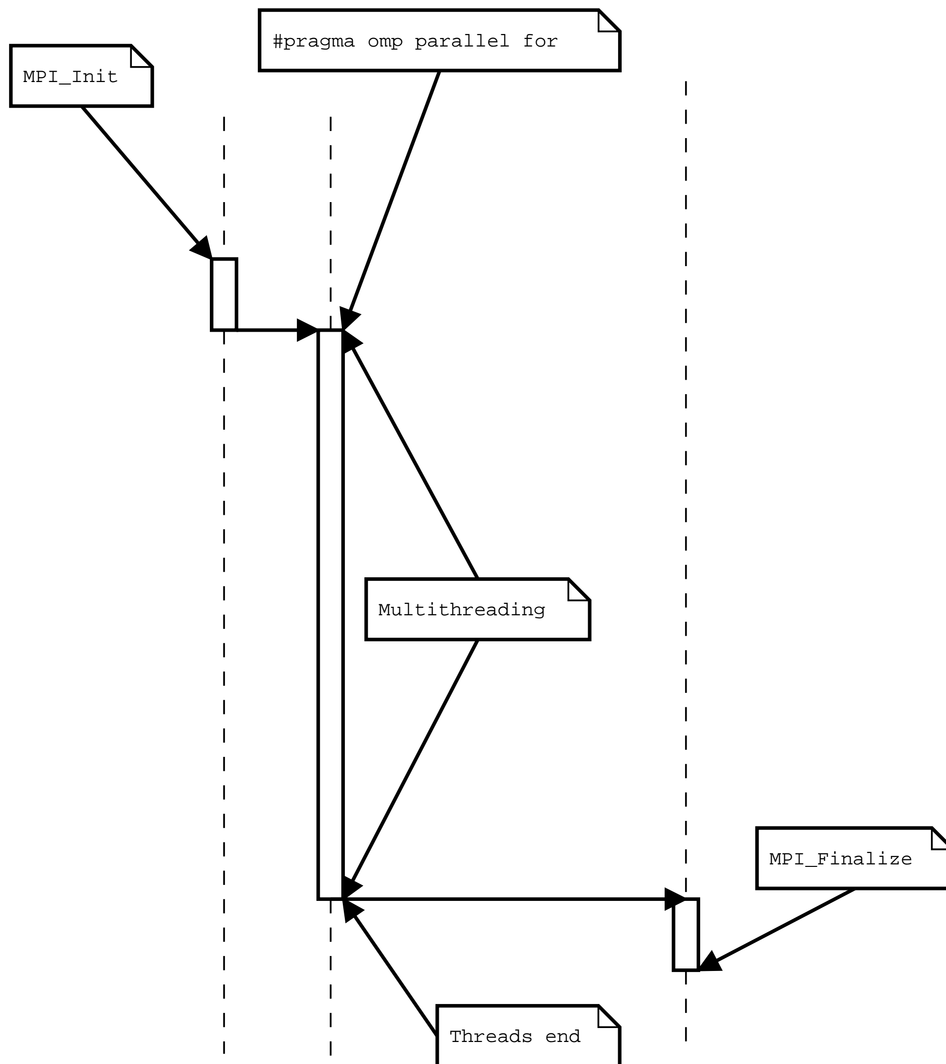

Figure 3.4: Execution sequence of hybrid parallel model

```
    MPI_Init()
    MPI_Comm_rank()
    MPI_Comm_size()

    //MPI communication part
    MPI_Send()
    MPI_Recv()


    omp_set_num_threads(4)
    // each process start four threads

    #pragma omp parallel for
    for (int i=0; i<n; i++)
    {
      //OpenMP computation part
    }

    //... Some work goes here

    MPI_Finalize()
  }
```

3.3 Comparison of parallel programming mode

This chapter describes the different between each parallel programming model. Section 3.4.1 compare MPI and OpenMP paradigm based on their own feature. The pros and cons of hybrid parallelization is given in section 3.4.2.

3.3.1 Pros and Cons of OpenMP and MPI

According to previous analysis on OpenMP and MPI characters, Table 3.1 gives the comparison between OpenMP and MPI based on their pros and cons.

3.3.2 Trade off between OpenMP and MPI

Based on different characters of OpenMP and MPI model talked above, one question might be came up, "which one is better?" We can not give a certain answer, but mainly concern about the problems listing below when we choosing different parallel languages. Table 3.2 give us the trade off between OpenMP and MPI construct, these points also propose some hints when we meets load imbalance or speedup not satisfactory.

Table 3.1: Comparison between OpenMP and MPI

OpenMP	MPI
Shared memory architecture only, which more expensive than distributed memory architecture	Can be used on shared or distributed memory architecture
Incrementally programming, can running sequentially after adding OpenMP directives, less modification compare to the original code	A lot of code structure need to be modified significantly compare to the original one
Easier to debug and implement	Harder to debug and implement
Thread level	Process level
OpenMP fork/join lead to synchronization overhead when loop size are relatively small	Message exchange and date replication lead to communication overhead and memory consumption
Need support by compiler	Highly portable, implementation on most hardware
Both fine-grained and coarse-grained are effective. Mostly used in loop iteration	Fine-grained create communication issues

Table 3.2: Trade off between OpenMP and MPI

OpenMP	MPI
Threads communication through memory access	Processes communication through memory copies
Threads share same global memory address	Processes send/receive messages but not share memory
No data copying, no messages generated	Message exchange

Through such comparison, we found both OpenMP and MPI have their drawbacks:

1. **OpenMP:** Data race conditions, synchronization overhead, performance limited by memory architecture. More OpenMP constructs or threads spawn lead to more synchronization overhead and language directives overhead

2. **MPI:** The biggest problem of MPI is communication overhead, in some case, memory consumption is considerable. More MPI nodes lead to more message communication overhead.

3.3.3 Pros and Cons of hybrid parallel model

As illustrated above, we can easily find that hybrid parallel model could be omitted disadvantages of MPI by adding OpenMP directives, which reduce:

1. **Load imbalance and scalability problem** Two big advantage of pure MPI and OpenMP implementation are load imbalance and poor scalability. By adopt hybrid parallel mode, better granularity can be achieved. MPI process only responsible for the inter-node communication, intra-node computation parallelism are instead by OpenMP threads. Bottleneck caused by MPI can be further parallelized by OpenMP threads.

2. **Memory consumption** Data replication in pure MPI always restrict by global memory management. Hybrid parallel model perform computational parallelism within each node which improving performance.

3. **Communication overhead** Hybrid parallel model reduce number of message between SMP nodes or within SMP node.

Although in some situations where hybrid parallel mode is more efficient than pure MPI, but disadvantage still exist which caused by topology mismatch problem. These shared-blocks originally need to be communicated between SMP nodes, one solution of it is allocating them to the same node as much as possible.

In addition, the OpenMP parts in hybrid parallel model produce more overhead than pure OpenMP. Because of all OpenMP threads have to flush cache synchronously, the locality problem will get worse. When masteronly schema is chosen, all other threads are idling during master thread communicates. These problems also should be considered when choosing hybrid parallel model.

3.4 Conclusion

In order to get better performance, hybrid programming model of OpenFOAM application on SMP cluster is investigated in this thesis. This method based on combination of coarse-grain parallelism and fine-grain parallelism. Domain decomposition is achieved by MPI message passing between inter nodes and time consuming loop iterations are parallelized by OpenMP threads.

Compare to hybrid parallel model, another popular programming model implemented on SMP cluster is pure MPI implementation. Some previous research shows that MPI perform better under some situations [10] [18]. Since the native parallel programming model of OpenFOAM is done by pure MPI implementation, the analysis is necessary and will be presented in next chapter

At last, we will compare these two implementations in evaluation chapter.

Chapter 4

OpenFOAM in parallel

In previous chapter, we already know that parallel programming model in OpenFOAM are done by MPI message passing. Therefore, coming next is the contents of OpenFOAM MPI principle. The aim of this chapter is to gain a clear idea of native MPI code structure and decomposition methodology.

Mesh is divided into numerous sub domains by MPI domain decomposition. Each part will be assigned to one or more SMP nodes. In section 4.1, the description of MPI classes will be given. In the rest of chapter, the methodology of mesh decomposition will be analyzed.

4.1 Running in parallel

The methodology of parallel computing in OpenFOAM is done by domain decomposition where geometry is divided into sub domains, each processor get its own pieces to work.

Before running application in parallel mode, we have to do some configuration work as below:

1. Copy decomposeParDict file into system directory which can be found in pitzdailtExpInlet case.

2. Edit these entries of decomposeParDict file.

   - **numberOfSubdomains** - indicate how many pieces shall be divided, this number have to equal to the number of available nodes.
   - **method** - method of decomposition are simple, hierarchical, scotch, metis and manually, the simple geometric decomposition approrach will be introduced in section 4.3.

- **distributed** - OpenFOAM allow user to distribute data if only local disks are used, such as Beowulf cluster system.
- **root** - since path of source code may differ between each nodes, the path must be specified.

3. Use decomposePar utility by command:
   - decompose -case [casename]
4. A set of directories have been created automatically under case directory. It named *processorN* which N is from 0 to the number of nodes minus one.
5. Create machine file which specifying machines you want to use.

Listing 4.1: Machine file code

```
# machine file

linux01
linux02
linux03
```

6. Finally, an application can be running in parallel by typing such mpirun command.
   - mpirun -hostfile <machinefile> -np <nProcs> <solvername> <root> <case> <otherArgu> -parallel

4.2 Domain decomposition methodology

The decomposition methods can be selected in decomposeParDict where have four types: simple, hierarchical, metis and manual.

Simple decomposition: Spilt the domain into sub-domain only by direction. For example, the simpleCoeeffs entry showed in Listing 4.2 means 2 pieces in x and y direction, 1 in z direction. We will illustrate this more in next section.

Listing 4.2: Simple decomposition code

```
simpleCoeffs
{
```

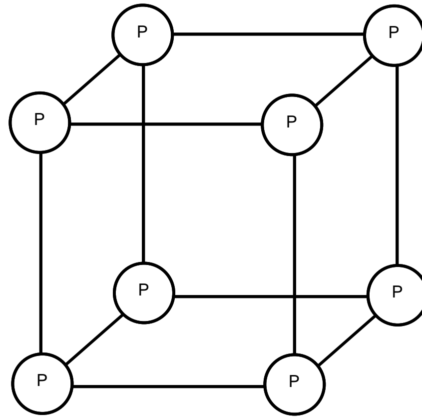

Figure 4.1: Eight points regular mesh

```
    n     (2 2 1);
    delta     0.001;
}
```

Hierarchical decomposition: Same as simple decomposition, except user can specify the order of decomposition. Code order xyz means the split work begin with x direction, end with z direction. This is important for processor distribution.

METIS decomposition: METIS decomposition do decomposition automatically, this approach no longer require user input. One optional keyword is processorWeights which specify different performance factor between nodes. According to this input, such kind of approach gives optimal solution which contains minimal number of boundaries [12].

Manual decomposition: Manually specify the data location to each node.

4.3 Simple decomposition

The aim of "Simple" decomposition method is to divide the domain with minimal effort but still can lead to good enough decomposition result. The technique behind it is simple and very suitable as the basis of this thesis project.

Consider one scenario shown in Figure 4.1. We get a mesh have 8 points, which 2 points in x, y, z direction separately. In order to simplify this example,

we also make the available node have the exactly same number with mesh points in x, y, z direction.

Simple decomposition algorithm use two data structure, one called pointIndices and another is finalDecomp.

- **pointIndices**: A list contains the index of points. Initial index value pointIndices[i] is assigned by position value i.
- **finalDecomp**: A list contains the index of processor.

The decomposition algorithm consists of six steps:

1. Reorder the pointIndices according to the sorting of x-axis decomposition. Each pointIndices[i] will get the value of index of node according to small to large order, in another words, it begins from the smallest x-axis.
2. After step one, the nodes in x direction is well formed. Then map the sorted nodes to the corresponding processors in finalDecomp list. Since processor 0 and processor 1 are available in x direction, so each processor gets 4 points. This step is shown in Figure 4.2.
3. Repeat step 1 and 2 for y-axis. This step is shown in Figure 4.3.
4. Combine x and y axis, we can get x-y axis results shown in Figure 4.4. This result can be got from following equation 4.1:

$$MESH_{XY} = MESH_X + N_{procx} \times MESH_Y \tag{4.1}$$

5. Repeat step 1 and 2 for z-axis. This step is shown in Figure 4.5.
6. Combine x, y and z axis, we can get x-y-z axis results shown in Figure 4.6. This result can be got from following equation 4.2:

$$MESH_{XYZ} = MESH_{XY} + N_{procx} \times N_{procy} \times MESH_Z \tag{4.2}$$

This simple example only consider equal amount scenario, but in practical, we always meet non-equal amount node with the number of processors. Consider if we have 9 points, the processors available still the same as previous example. Like that, in such situation, each processor in x or y or z direction still get 4 points. But the processor 1 in x-axis have to get one more point to process. If Hierarchical Decomposition specify yxz order, then the processor 1

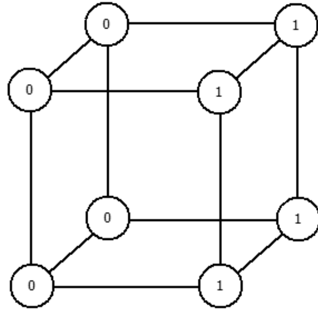

Figure 4.2: Two processors are distributed in X direction

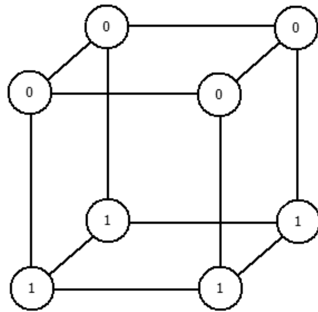

Figure 4.3: Two processors are distributed in Y direction

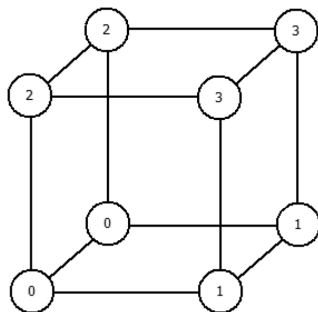

Figure 4.4: Four processors are distributed in X-Y direction

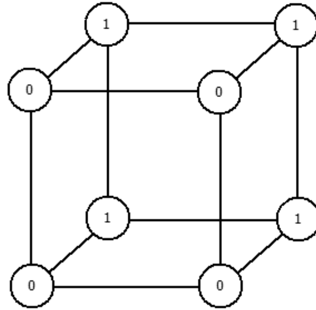

Figure 4.5: Two processors are distributed in Z direction

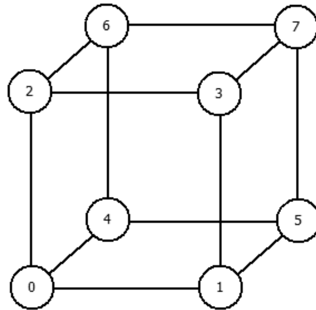

Figure 4.6: Eight processors are distributed in X-Y-Z direction

in y-axis get the work. The result can be calculated from following Equation 4.3:

$$PointsperProcessor = N_{points} mod N_{procs} \tag{4.3}$$

After points are distributed to corresponding processor, the boundaries between each sub domain are created. So compared with running in serial, there are at least one more iterations should be carried out when solving meshes, this is why you will see different iteration times between sequential running and parallel running.

Although this approach ignores the communication minimization between each processor, but it make the basis clear. Since the aim of this project works are investigating the possibility of hybrid parallel programming in OpenFOAM and comparing pure MPI implementation with hybrid implementation. Simple decomposition is enough to satisfy the background conditions.

4.4 MPI communication description

The goal of this section is to present MPI usage in OpenFOAM. The contents that are presented here focus on MPI routines, MPI collective communication and the structure of MPI call, give some briefly knowledge about MPI constructs of OpenFOAM.

OpenFOAM framework implement MPI message passing by Pstream shared library which contains two sub classes named IPstream and OPstream.

4.4.1 MPI Program structure

Figure 4.7 shows us the execution process of MPI program, the parallel code goes between initialization and termination of MPI environment.

4.4.2 Environment routines

MPI environment routines are used to initialize and terminate MPI parallel running. Some important routines are described as following:

- **MPI_Init**: Initialize the MPI environment.
- **MPI_Comm_size**: Determine the number of processes.
- **MPI_Comm_rank**: Determine the rank of processes within group.

When OpenFOAM program start, it first checks its command argument. If the program running in parallel, Three routines mentioned above are called by Foam::Pstream::init which referenced from main function.

- **MPI_Abort**: Terminate all MPI process when error is happened.
- **MPI_Finalize**: Terminate the MPI enviroment, last MPI routine call.

Foam::Pstream::exit include above routines which are referenced by main function and Foam::handler function call them when main program terminate.

4.4.3 Communication schedule schemes

According to result, the number of processor is compared with system variable called nProcsSimpleSum. Then, a communication scheme will be determined from the candidates introducing below. nProcsSimpleSum is OpenFOAM system optimization switches which can be found in controlDict file, the default number is 16.

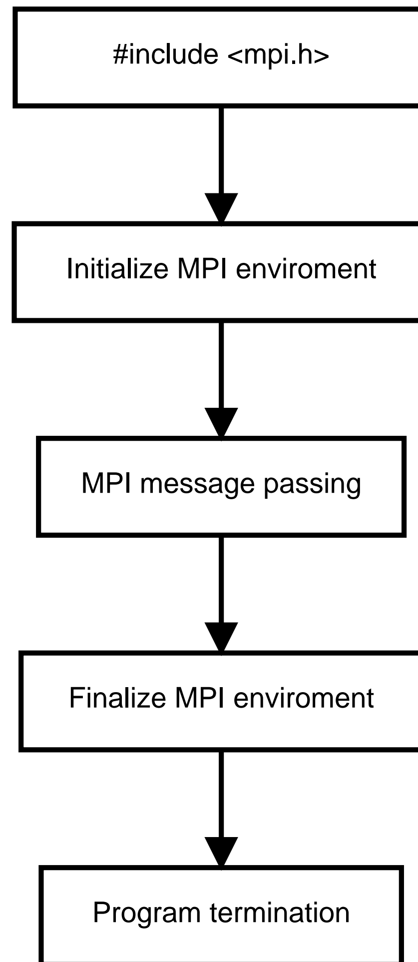

Figure 4.7: Execution procedure of MPI program

These two communication schemes candidates embedded in OpenFOAM are:

Linear Communication

Linear Communication is also called master-slave communication or self scheduling. Figure 4.8 shows the structure of linear communication scheme.

The idea is that each slave process sends up its own sub domain value to master process. Then master process performs operation on gathered data, after that, broadcast the data to each slave.

The slaves have no any below successor, they only communicate up with master process, and the co-ordinate work is done by master process.

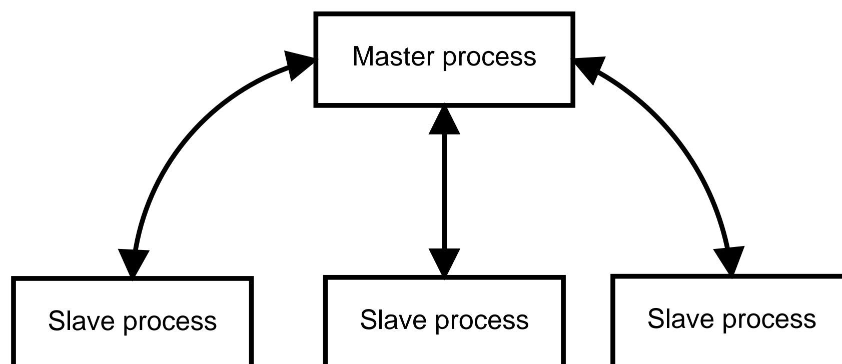

Figure 4.8: Linear communication schemes

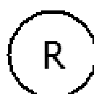

Figure 4.9: When k=0, the node only contain root R, we record it as B0

Binomials Tree Communication

When nodes involving in communication bigger than 16, scatter and gather operations turn to use binomial tree-based algorithm. A binomial tree algorithm is defined as following [16]:

- If k=0, Bk = B0= R. (A binomial tree of order 0 is a single node R)
- If k>0, Bk = R, B0, B1 ... Bk-1. (The binomial tree of order k>0 consist of the node R, and k binomial sub-trees, B0, B1 ... Bk-1)
- Order k has 2k nodes which order is k

For example, the communication involve in 8 processes, therefore we will have 3 orders correspondingly, each order k binomial tree has a node with order k-1 binomial tree, the order 3 binomial tree is connected to other binomial tree. The construction process of its unique structure is showed in Figure 4.9 to Figure 4.12.

According to this algorithm, the communication binomial tree for all 8 processes can be drawn as below in Figure 4.8,

Hence, we could easily get the communication schedule as follow steps,

- Step one

  Node 0 receive data from node 1
  Node 2 receive data from node 3

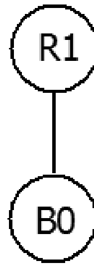

Figure 4.10: When k=1, the binomial tree contains current root R1 and the existing sub-tree B0, we record it as B1

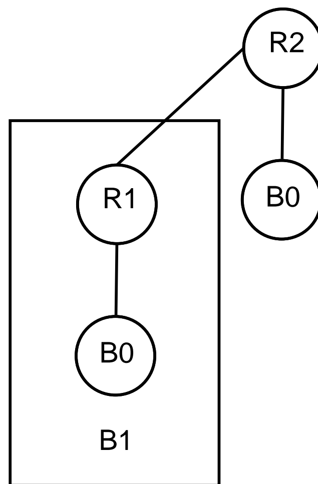

Figure 4.11: When k=2, the binomial tree contains current root R2 and the existing sub-trees B0 and B1, we record it as B2

Node 4 receive data from node 5
Node 6 receive data from node 7

- Step two

  Node 0 receive data from node 2
  Node 4 receive data from node 6

- Step three

  Node 0 receive data from node 4

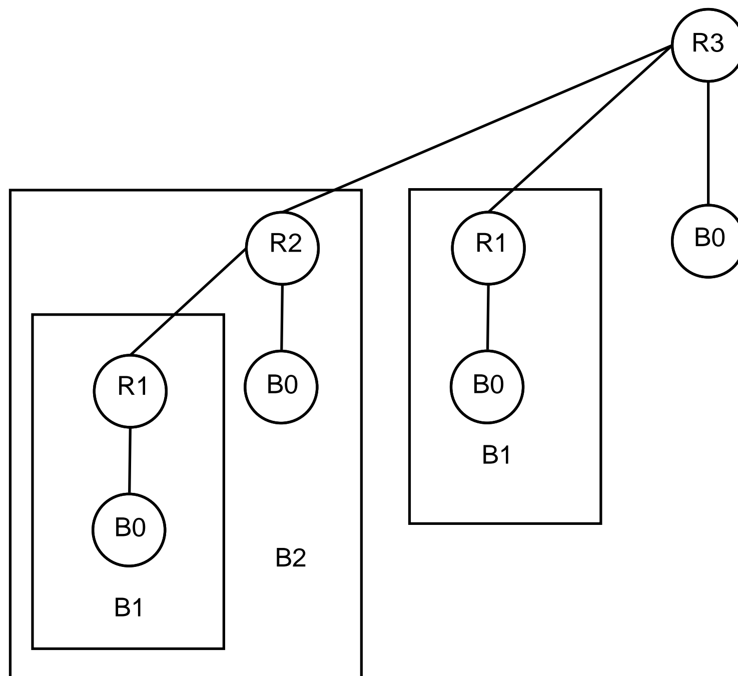

Figure 4.12: When k=2, the binomial tree contains current root R3 and the existing sub-trees B0, B1 and B2

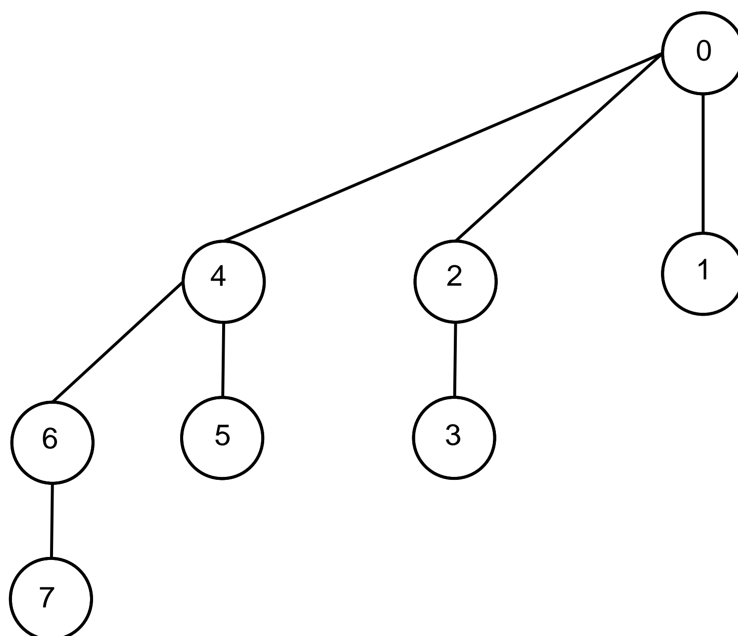

Figure 4.13: Communication binomial tree for 8 processors

It proves that, since the data communication in each step only involve peer-to-peer scheme which reduce the communication overhead. The measurement work shows tree-based communication scheme is more efficiency than linear-based scheme when the numbers of nodes increasing, also when the number of message is large but size relatively small, the quality of communication can benefit from binomial tree algorithm [15].

4.4.4 Communication routines

Collective communications involve all processes which are specified in domain decomposition. The types of operations in OpenFOAM are:

- **Data movement** - scatter and gather
- **Reduction** - Based on data movement, master processor not only collects all information from all other processors, but also use given operations compute on the data (min, max, add, minus, etc).

Pstream library have three types of data movement:

1. Pstream::gather and Pstream::scatter

   These classes only consider the standard data movement. One processor gathers data from all other processors according to communication schedule. The gathered data will be performed calculation according to user-specified operator.

   After all data are calculated, each process uses "scatter" to send data to downstairs neighbors.

   The gathered data here is only single value. Besides, operation work is done inside gather function.

2. Pstrem:: gatherList and Pstream::scatterList

   Same as above, but gathered data is a list which filled by element from each processor. This element consists by processor ID and its value. No operation performs here. Only initiators or master hold the full filled list, then use scatterList to broadcast list to each processor.

   The elements inside list are ordered that current value goes first and all other leaves next.

3. Pstream::combineGather and Pstream::combineScatter

> Same as standard gather and scatter, but data type could be diversity, and more operators are allowed to use, such as binary operator and assignment operator.
>
> There are variants of combineGather and combineScatter which can be performed on list and map structure.

There are two types of reduction for each gather/scatter type, since real reduction work is carried out inside Pstream::gather, therefore, it just call Pstream::gather and Pstream::scatter in sequence.

4.5 Conclusion

Native MPI message passing model used in OpenFOAM is definitely a good solution. But in fact, most experiments prove that MPI implementations require a large amount of "halo" message communication between each SMP nodes. It seems not the most efficient parallelism approach for SMP cluster. However, shared memory parallelism such as OpenMP provide pretty nice solution on inter node parallelism theoretically. Then we have an idea that, by integrate OpenMP thread parallelism construct to native MPI model, replace the extra message exchange by shared memory access, the application may provide better performance than before.

Next, we will talk about hybrid parallel prototype of dieselFOAM. The contents in next chapter include time profiling and candidates analysis.

Chapter 5

Prototype Design

The parallelism approach through top-down scheme, starts from the top level of the whole program, and then parallelizes each sub-domains of the mesh. The structure of prototype is that coarse grain parallelism between different sub-domains by MPI and fine grain loop-level parallelism in solver functions.

In previous chapters we have already known the basic structure of OpenFOAM program, parallel programming method and domain decomposition methodology in OpenFOAM. Now it is time to identify the potential places in program which allow us to accelerate it, after that, applying hybrid techniques on these codes to get faster execution time of physical simulation.

This chapter will focus on profiling of dieselFoam, by means of the input and output, the result and candidates will be discussed in the following sections.

5.1 Methodology

This section is an overview of next two chapters, it present the methodology that how the prototype can be implemented and analyzed. These stages are designed to machine independent, which can be concluded as following:

1. Sequential implementation

   Parallel programming usually starts from correct sequential implementation, since this step is already done, what we should do here is record the correct result in order to examine parallel implementation.

2. Profiling

   The profiling work is to identify bottleneck of the application, find out which part consuming large time and where is the computational intensive part. Following steps will be spread out based on this.

3. Task-parallelism part

   Some functions in program have a number of obvious concurrent stages. These parts can be easily carried out by OpenMP task parallelism. Identify the sequence of execution, reconstruct the program logic if necessary.

4. Data-parallelism part

   Based on evaluation result, we could parallelize the most time-consuming loop iteration, it is possible to parallelism without algorithm changing, but sometime we have to introduce another algorithm to adopt it, such as divide-conquer algorithm. More important, data-racing problem should be tested before any practical coding.

5. Ideal model analysis

   Analyze the results of application execution based on theoretical knowledge, most prerequisite environment is ideal in such phenomena, such as infinite number of processors and ignore communication latency. This step mainly concentrate on theory study.

6. Prototype simulation

   The parallelized code will run on SMP cluster. Analysis based on result comparison will be provided. It shows the overhead of communication and threads synchronization. This step also includes some suggestion to the future work.

5.2 Running platforms

Since the hardware architecture greatly affected the result of test, such as different function have different execution time on different platform. So we have to choose hierarchical hardware running application which best fit to the programming model.

The hardware which used to profile dieselFOAM solver are SMP cluster which contains two SMP nodes. Each node equips dual socket quad core Intel Xeon(R) CPU with 2.33GHz.

These SMP nodes have OpenFOAM 1.6 installation on them. The OpenMPI which default included in OpenFOAM is used in this project. Other configuration work is same as we presented in chapter 4.

5.3 Performance profiling

The purpose of performance profiling is to find bottlenecks in execution time of dieselFoam. From profile summary, we can get the exact code place where CPU spent most time on it. Also some analyze effort should be put on these code in order to make sure correct result even after OpenMP optimization.

There are a lot of profiling methods exist, the most common ones can be seen as following:

- **Instrumentation:** the instrumentation profiling mainly adds some profiling code into application source code, collects information about detailed timing of function call. The measurement criteria can divided into CPU time and wall clock time, which the later one consist of CPU time, I/O time and communication channel delay.

  We calculate the `wall time (elapsed time) = calendar_time (end) - calendar_time (begin)`, the get ten datum per one task, compute the average value.

  The reason not use the CPU time is that mesh calculation will cost considerable time on I/O consuming, so we want inspect the total time of execution not just the CPU time, since other factors are critical as well.

  Other approaches such as compiler instrumentation profiling, binary instrumentation profiling also very common to use. The advantage of source-level instrumentation profiling is easy to use, the compiler-level one need compiler support, sometime have to recompile the whole source code, such as "`gcc -pg ...`" for gprof.

- **Sampling:** Sampling give us statistical samples of application performance, it has the same purpose of instrumentation profiling but does not require any instrumentation. It does not interfere the program running, so sometime lower overhead than the other.

  So sampling profiling capture information at specified time by sampling, but instrumentation profiling add extra code to catch every function call in the application.

  Here we select instrumentation profiling which fits our testing scenarios.

Table 5.1: Mesh properties of aachenbomb case

Properties	Value
Points	178164
Faces	514181
Internal faces	494419
Cells	168100

5.4 Profiling input

The I/O files of aachenbomb case were introduced previously in chapter 2. Here we use the default input parameters of aachenbomb case to carry out our profiling test, Table 5.1 shows the information about each properties of mesh, application will be executed 100 times according to the value in controlDict file. And the process of PBICG solve class can be seen in Figure 5.2.

5.5 Collaboration diagram for PBICG

As we talked in section 2.4.2. UEqn solve function invokes PBICG solver via fvMatrix interface. Some functions in PBICG solve are organized in computational intensive manner. And other parts are coding in class intensive manner. PBICG is a very complex class containing a variety of elements inside. We only draw some meaningful function here and will analyze these candidates in section 5.7. The Figure 5.1 is the collaboration diagram for PBICG solver.

5.6 Profiling result

After time profiling, we get the total time for each runtime loop consume 21s. and detailed results for each equation part and functions in UEqn object are showing in tables as following. By analysis of PBICG solver class. we found the Amul (Tmul), normFactor and Precondition are the three most time consuming functions. Apart form this, The initiation functions such as fvm::ddt shown in Figure 2.3 also take about 0.4s which are very considerable. Table 5.2 shows the time plot of each part in dieselFoam solver. Table 5.3 shows the time plot of each part in dieselFoam solver. And Table 5.4 shows the Timing results of each time consuming function in PBICG solver. The *etc* part is consist by variable assignment, object creation, object initialization and so on. They are relatively small alone, we suppose they don't need parallelism, or need more deep investigation.

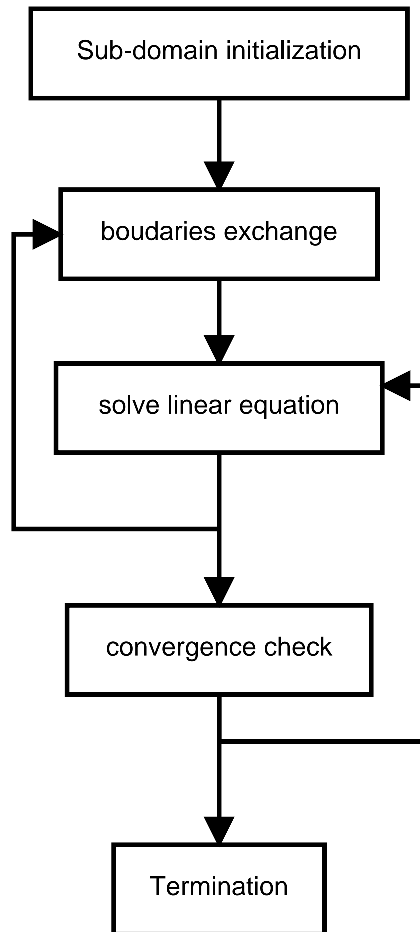

Figure 5.1: Flow chart of PBICG functions

Table 5.2: Timing results of dieselFoam solver

Object name	Time consuming(∼)
UEqn	2.8s
pEqn × 2	6s
hEqn	2.35s
YEqn	2.47s
Turbulence (k, epsilon)	2.51s
rhoEqn	1.3s
etc.	3.4s

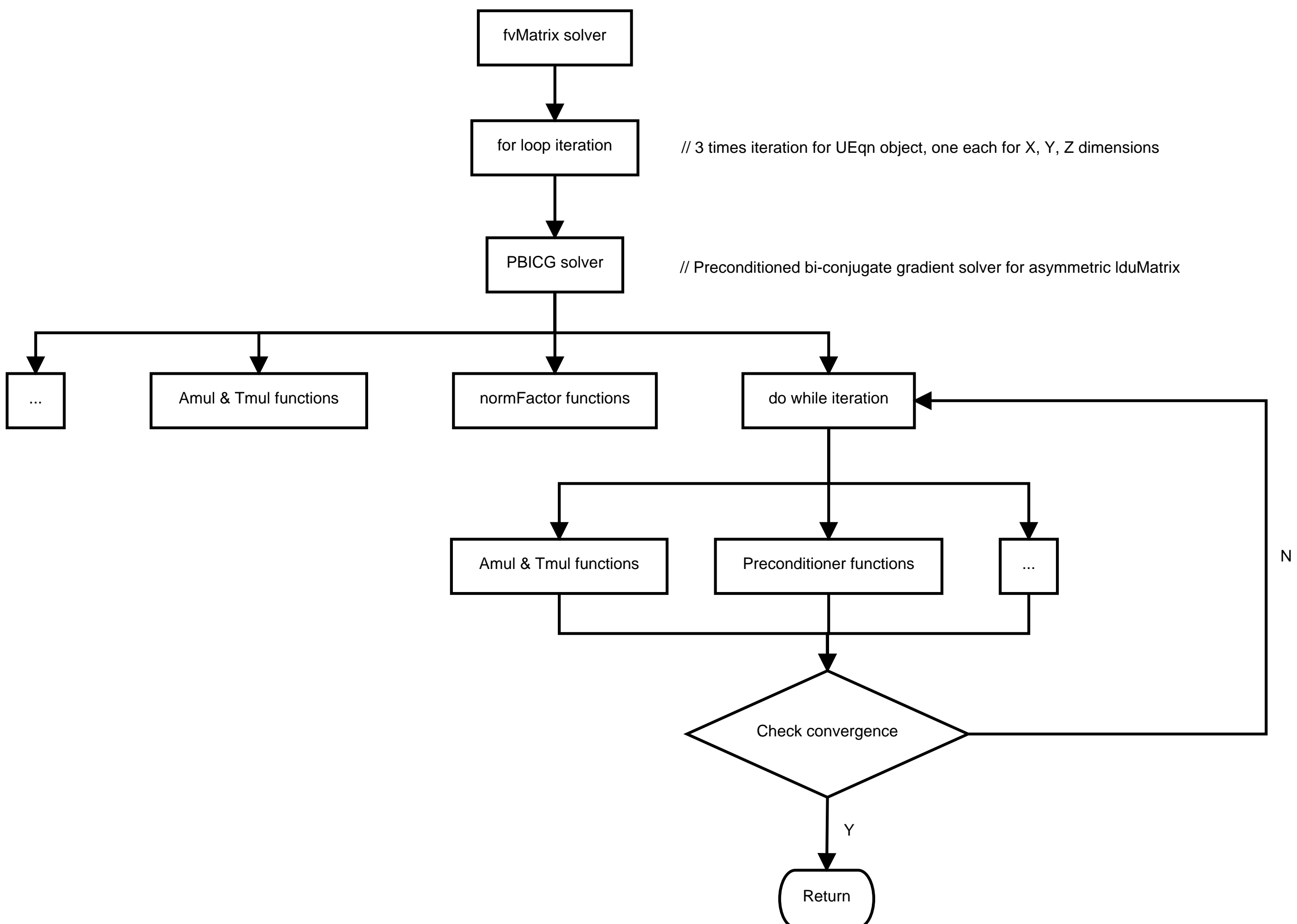

Figure 5.2: Collaboration diagram for PBICG solver

Table 5.3: Timing results of UEqn object

Object name	Time consuming (∼)
fvm::ddt	0.38s
fvm::div	0.43s
fvc::grad	0.61s
solve	0.6s
etc.	0.83s

Table 5.4: Timing results of PBICG solver

Function name	Time consuming (∼)
Amul(Tmul) × 9	0.2s
normFactor × 3	0.06s
Precondition × 9	0.3s
etc	0.04s

5.7 Candidate analysis

This profile summary shows us the potential function which worth to be accelerated. But before we adding OpenMP directive on these functions, data structure must be analyzed in order to prevent undesirable problem happened, such as data-racing.

As we described in previous section, these potential functions are worth to exploit in data level parallelism since they are in computational intensive manner. Next we pick up two typical functions out of above and initiation function fvm::ddt, presenting the detailed illustration:

- **Amul(Tmul)**

Listing 5.1: Amul function code

```
register const label nCells = diag().size();
for (register label cell=0; cell<nCells; cell++)
{
  TpsiPtr[cell] = diagPtr[cell]*psiPtr[cell];
}

register const label nFaces = upper().size();
for(register label face=0; face<nFaces; face++)
{
  TpsiPtr[uPtr[face]] += upperPtr[face]*psiPtr[lptr[
      face]];
```

```
    TpisPtr[lptr[face]] += lowerPtr[face]*psiPtr[uptr[
        face]];
}
```

The Amul function is a matrix multiplication function inside PBICG solver. It includes communication and computational part. Communicational part is carried out by MPI message passing that exchange face boundary information between each neighbor nodes. According to profiling test, we found its computational part consume a lot of time to perform calculation. Even after MPI decomposition, each sub-domain still left big amount according to the properties of mesh. Therefore, frequency memory accesses may results in low performance in its loop iteration. From Listing 5.1, we see Amul mainly consist by two for-loops, these iterations are good candidates where parallelism can be achieved.

In order to analyze its data structure, find out if it have any data dependence problem. So some description about Matrix and Mesh in OpenFOAM are necessary presented here.

In OpenFOAM, there are two concepts which make us confused, one is Mesh, and the other is Matrix. Domain decomposition divides the problem space into several sub-domains, which are meshes.

In OpenFOAM, mesh can be divided by points, faces and cells:

1. Points:consist by x,y,z coordinate.
2. Faces:consist by points.
3. Cells:consist by faces. Part of mesh.

And matrix is a scalar storage format in OpenFOAM, which consists by three parts:

1. Diagonal list(diagPtr in Amul function): diagonal coefficient.
2. Upper list(upperPtr in Amul function): upper triangle value.
3. Lower list(lowerPtr in Amul function): lower triangle value.

This is why the matrix related to dieselFoam called lduMatrix which means lower, diagonal and upper matrix storage class.

Every cell only has one diagonal coefficient, so the number of diagonal list is equal to the number of cells in mesh. The value of upper triangle and lower list is the one which influences another, for example:

Consider we have a 3×3 mesh:

$$\begin{pmatrix} 1 & 2 & 3 \\ 6 & 5 & 4 \\ 7 & 8 & 9 \end{pmatrix}$$

From mesh above, we see cell 1 influence cell 2 and cell 6, cell 2 influence cell 5, cell 1, and cell 3, etc.

Hence, we can obtain a matrix contains coefficients like below:

$$\begin{pmatrix} D & N & - & - & - & N & - & - & - \\ O & D & N & - & N & - & - & - & - \\ - & O & D & N & - & - & - & - & - \\ - & - & O & D & N & - & - & - & N \\ - & O & - & O & D & N & - & N & - \\ O & - & - & - & O & D & N & - & - \\ - & - & - & - & - & O & D & N & - \\ - & - & - & - & O & - & O & D & N \\ - & - & - & O & - & - & - & O & D \end{pmatrix}$$

The positions of elements are symmetric. And positions of upper list and lower list have the mutual but opposite value.

Each list in matrix is actually like a dictionary, all the value are stored in its own database. These databases are isolated with dieselFoam main program. As we see in Amul code, these places are instead by placeholder. After application running, OpenFOAM retrieve the element of the matrix according to the entries number.

For example, we have a 4×4 sparse matrix as Equation 5.1 below,

$$M = \begin{pmatrix} 1 & 0 & 5 & 0 \\ 0 & 2 & 0 & 6 \\ 7 & 0 & 3 & 0 \\ 0 & 8 & 0 & 4 \end{pmatrix} \tag{5.1}$$

So in OpenFOAM, we have corresponding placeholder matrix as Equation 5.2 below, d, u and l represent diagonal list, upper list and lower list separately.

$$\begin{pmatrix} d(0) & - & u(0) & - \\ - & d(1) & - & u(1) \\ l(0) & - & d(2) & - \\ - & l(1) & - & d(4) \end{pmatrix} \tag{5.2}$$

It's very easy to understand, diagonal list contains particular elements (1, 2, 3, 4, 5...). And for off-diagonal list, for example, the upper list, which OpenFOAM introduce two lists called UpperAddr (uptr in Amul function) and LowerAddr (lptr in Amul function) to locate the elements. These elements in upper list are located by UpperAddr as the column coordinate and the LowerAddr as the row coordinate. In contrast, elements in lower list are located by UpperAddr as the row coordinate and the LowerAddr as the column coordinate.

Therefore, we have elements (5, 6) in upper list and elements (7, 8) in lower list in this example. And the position is,

$$u(0) = [2, 0] \quad u(1) = [3, 1] \quad l(0) = [0, 2] \quad l(1) = [1, 3]$$

Thence, we have coordination lists which are UpperAddr = (2, 3) and LowerAddr = (0, 1).

Now we can go further to investigate into for-loop in Amul, as we said before, these two for-loop take matrix operations like Equation 5.3 below.

$$\begin{pmatrix} 1 & 0 & 5 & 0 \\ 0 & 2 & 0 & 6 \\ 7 & 0 & 3 & 0 \\ 0 & 8 & 0 & 4 \end{pmatrix} \times \begin{pmatrix} b1 \\ b2 \\ b3 \\ b4 \end{pmatrix} = \begin{pmatrix} c1 \\ c2 \\ c3 \\ c4 \end{pmatrix} \tag{5.3}$$

This is typical discretization process AB=C in OpenFOAM [12], where

- A is the ldumatrix contains the coefficient
- B is the data we need to solve
- C is the discretization source

Therefore, when we unroll these iterations, fill it with list elements. We can get each iteration results in:

1. TpsiPtr [2] += upperlist [0] × psiPtr [0]
   TpsiPtr [0] += lowerlist [0] × psiPtr [2]

2. TpsiPtr [3] += upperlist [1] × psiPtr [1]
   TpsiPtr [1] += lowerlist [1] × psiPtr [3]

It seems as if no data dependence between them from the example above. But when go further about this example, the problems are arising.

Consider if we have a another 4×4 sparse matrix as previous example but with a little change, Equation 5.4 shows this matirx.

$$M = \begin{pmatrix} 1 & 0 & 5 & 0 \\ 0 & 2 & 6 & 0 \\ 7 & 8 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix} \tag{5.4}$$

Then we get its coordination lists as UpperAddr = (2, 2) and LowerAddr = (0, 1). Now after we unroll each iteration again, we get the results in:

1. TpsiPtr [2] += upperlist [0] × psiPtr [0]
   TpsiPtr [0] += lowerlist [0] × psiPtr [2]

2. TpsiPtr [2] += upperlist [1] × psiPtr [1]
   TpsiPtr [1] += lowerlist [1] × psiPtr [2]

From the above equations, we can find these iterations cannot execute at the same time since iteration 0 and iteration 1 both write data into TpsiPtr[2].

Iteration 0 and iteration 1 have obviously data-racing problem, of course sometime we will meet good situation, but still cannot guarantee this problem not happened. In a word, Amul function cannot be performed OpenMP directive on it, must be executed in order.

- **Precondition**

Listing 5.2: Precondition function code

```
For (register label cell = 0; cell<nCells; cell++)
{
  wAPtr[cell] = rDPtr[cell]*rAPtr[cell];
}

register label sface;

for (register label face=0; face<nFaces;face++)
{
  sface = losortPtr[face];
  wAPtr[sface] -=
```

```
      rDPtr[uPtr[sface]]*lowerPtr[sface]*rAPtr[lPtr[
         sface]];}
}

for (nface=nFacesM1; nface>=0; nface--)
{
  wAPtr[nface] -=
    rDPtr[lPtr[nface]]*upperPtr[nface]*rAPtr[uPtr[nface
       ]];
}
```

The structure of precondition function is very similar with Amul function. From the source code we can see, except the first straightforward for-loop block, the rest are still performing matrix face update calculation. Instead of face multiplication, precondition function return preconditioned form of residual. These two function have the common ground is that all the data structure related to lPtr and uPtr. According to the analysis in Amul section, all the lists are independent to each other, and constant. Besides, in the second loop, sface variant means face of shape, each element in the losortPtr list is unique. So from these, we can conclude that Precondition function can be parallelized by OpenMP for-loop construct.

- **Fvm::ddt**

  Fvm::ddt is one of discrete schemes in OpenFOAM equation category, some other schemes such like fvm:div and fvm:grad do the same thing which you can specify them in fvSchemes file. The fvm::ddt and fvm::div are also the most time consuming functions in Navier-Stokes equation of our project target.

  After code anaylsis, we found such functions are difficult to be parallelized, since all the functions are built in class intensive manner. Also there are no obvious for-loop iteration, concurrent execution or tree-like traversal parts can be found.

5.8 OpenMP

In shared memory programming model, OpenMP use fork-join model to parallel execute each partial sub-domains. This purpose can be achieved by either task parallelism or data parallelism. After previous analysis of dieselFoam source code, the time consuming part can be parallelized mainly appear in two places:

- At top level of Ueqn where Navier-stoke equation carries out [11].
- Some meaningful function inside linear solver (PBICG solver) which mentioned last section.

This section will discuss these two parts separately according to different parallel model.

5.8.1 Task parallelism

Task parallelism, which is also called function parallelism, is a non-iterative work sharing parallelization form. It performs the different independent operation at the same time. OpenMP section construct is one of them, it divide code block which can be executed concurrently among all threads. All the works are determined by compiler, such as how to divide the code block, how many threads involved and how many sections for one single thread. Also it's possible that one thread join more than one section. The syntax is:

Listing 5.3: The syntax of OpenMP section construct

```
#pragma omp sections
{

  #pragma omp section

     code_block

  #pragma omp section   newline

      code_block
}
```

The best place can be performed by OpenMP section construct is Navier-Stokes equation. After investigation in Chapter 2, we already know each term of equation can be performed in parallel. The thesis work [11] is based on it, which divide each statement into one section. Through some minor modification, the source code looks like this:

Listing 5.4: Parallelism by OpenMP section construct

```
#pragma omp parallel sections
{
  #pragma omp section
  {
    fvScalaMatrix *T1 = new fvScalaMatrix(fvm::ddt(rho,Yi)()
       );
```

```
    }

    #pragma omp section
    {
      fvScalaMatrix *T2 = new fvScalaMatrix(mvConvection->
          fvmDiv(phi, Yi));
    }

    #pragma omp section
    {
      fvScalaMatrix *T3 = new fvScalaMatrix(fvm::laplacian(
          turbulence->muEff(),Yi)());
    }

  ...

  solve(*T1 + *T2 == *T3);
```

It insert each term of equation into a general matrix object, let these object function call can be executed concurrently, then after all object values are returned, call solve subroutine.

Based on the result of that thesis, we can go further to re-implement this part by OpenMP 3.0 tasking construct.

Tasking directive specify explicit tasks, these tasks will be executed by the encountering thread. A task pool with the data environment is created by one single thread of the team, and other threads can retrieve works from it. All the process is optimized by compiler. Another advantage of tasking construct is highly compostable. It can be nested inside other task or work-sharing. For instance, in some other solver applications, solver equation may exist inside for-loop iteration, tasking construct may take more efficient way than section construct under such situation.

By tasking construct, parallelization result gives us almost same structure as section in this example. Because of we cannot parallel the outer loop of Navier-Stokes equation by tasking construct due to time dependent feature:

Listing 5.5: Parallelism by OpenMP tasking construct

```
   #pragma omp parallel
  {
    #pragma omp single nowait {

    //...Some object initialization goes here

    fvScalaMatrix *T1 = null, *T2=null, *T3=null;

    #pragma omp task
```

```
    {
      *T1 = new fvScalaMatrix(fvm::ddt(rho,Yi)());
    }

    #pragma omp task
    {
      *T2 = new fvScalaMatrix(mvConvection->fvmDiv(phi, Yi));
    }

    #pragma omp task
    {
      *T3 = new fvScalaMatrix(fvm::laplacian(turbulence->muEff
         (),Yi)());
    }

    #pragma omp taskwait

    //...

    solve(*T1 + *T2 == *T3);
       }
  }
```

However, if linear solver equation exists in loop iteration without data dependence, we can nest tasking construct in another tasking construct. Usually, most loop iterations of solver equation in OpenFOAM only occurs small times, it is very suitable to implement it by adding tasking construct.

For example, if we have code snippet likes below:

Listing 5.6: Equation example pseudo code

```
for (int id=0; id<=Number; id++)
{
  U = fvm:laplacian(id,p);

  fvScalarMatrix Eqn
  (
    fvm::ddt(p)
  + fvm::div(id, p)
  ==
    rho
  + U
  );

  if(U.predictor())
  {
    Eqn.solve();
  }
}
```

Suppose variant p is a constant value which reading from input file. In other words, there is no data dependence between iterations. In addition, we make predictor function cannot be executed before initialization of U object finished. Hence, we need one more condition to control the data flow, the best choice of such case is using tasking construct showing in Listing 5.7.

Listing 5.7: Parallelism version of Equation example

```
int main()
{
  #pragma omp single nowait
  {
    for (int id=0; id<=Number; id++)
    {
      #pragma omp task firstprivate(id)
      Eqn(id);
    }
  }
}

void Eqn(int id)
{
  fvScalaMatrix *T1 = null, *T2=null, *U=null;

  #pragma omp task
  {
    *T1 = new fvScalaMatrix(fvm::ddt(p)());
  }

  #pragma omp task
  {
    *T2 = new fvScalaMatrix(fvm::div(id, p)());
  }

  #pragma omp task if(0)
  {
    #pragma omp task
    {
      *U = new fvScalaMatrix(fvm::laplacian(rho, p)());
    }
  }
  flag = U.predictor();

  #pragma omp taskwait

  if (flag)
  {
```

```
    Eqn.solve(*T1 + *T2 == *U + rho);
  }
}
```

In OpenMP 2.5, sometime we can do the same thing as tasking construct by nested parallel region, but usually do not perform well due to tree imbalance, synchronization overhead and so on [4]. But the OpenMP 3.0 tasking construct avoid these problem since there is only one parallel region exist. One task can create more tasks in the same team, and any threads can join in any tasks.

5.8.2 Data parallelism

The data parallelism is the most common and familiar OpenMP construct: the for-loop directive. Compare to task parallelism, it perform the same operation but with different data. The for-loop directive specifies the iteration should be executed in parallel by all threads. The syntax is:

Listing 5.8: The syntax of OpenMP for construct

```
#pragma omp for

for loop
```

After discussion in section 5.5, we get the conclusion that precondition function is a good object can be parallelized. Without data-racing problem, these for-loop can be parallelized in straightforward manner. After adopt with for-loop construct, we can get the accelerated code as below:

Listing 5.9: Parallelism version of Precondition function

```
#pragma omp parallel private(cell,face,sface,nface)
{
  #pragma omp for
  For (cell = 0; cell<nCells; cell++)
  {
    wAPtr[cell] = rDPtr[cell]*rAPtr[cell];
  }

  #pragma omp for
  for (register label face=0; face<nFaces;face++)
  {
    sface = losortPtr[face];
    wAPtr[sface] -=
       rDPtr[uPtr[sface]]*lowerPtr[sface]*rAPtr[lPtr[sface
          ]];}
```

```
    }

    #pragma omp for
    for (nface=nFacesM1; nface>=0; nface--)
    {
      wAPtr[nface] -=
        rDPtr[lPtr[nface]]*upperPtr[nface]*rAPtr[uPtr[nface]];
    }
  }
```

Besides, if we get the loop iteration has nested loop, one thing should be mentioned here is that, if for construct is used in the inner loop, a increasing overhead is made since language construct produce large overhead in every outer loop, also with synchronization overhead. So the efficient way is deploy loop construct at outermost loop.

5.9 Conclusion

After experiment profiling and prototype implementation, we found majority time occurs in linear equation calculation. In most programs, loop iteration in these equations take up a large proportion. After carefully investigate about read-write dependence, we could easily add work-sharing construct on them in order to get higher performance. Besides, we can also leverage OpenMP 3.0 tasking character, which can be used to parallelize some situations almost impossible or producing bad performance in OpenMP 2.5. By combined work-sharing and tasking construct, we can get better performance when the number of elements or loops unbalance.

Chapter 6

Evaluation

In the previous chapters, we have discussed about function analysis and prototype implementation. This chapter continues to discuss about performance analysis. First in section 6.1, Jacobi iterative method case study will be presented on the Linux SMP cluster, which is the same platform conduct prototype performance study. Then, PBICG solver prototype performance results will be given. by Last but not the least, in the section 6.3, conclusion of comparison between pure OpenMP, MPI and hybrid parallel model will be discussed. All the performance study is conducted on a small SMP cluster, each SMP node contains two-socket quad core Intel Xeon CPU, The compiler environment is GCC 4.3 and OpenMPI.

6.1 Test case: Jacobi iterative method

Jacobi iterative method is widely used to solve linear equation in many areas. In Jacobi iterative method, the new value is the mean value of old ones of each loop. Because of its locality feature, good parallelism performance can be achieved on it. Hence, it becomes a classic example of parallel computing.

In Jacobi iterative method, after the domain decomposition, except data communication of the boundary pieces between the neighbor nodes. Within each sub-domain, the computation can be performed independently. Meanwhile, with the increase of computational degree, the ratio of the cost of communication part to computational part will reduce, it go further to improve the efficiency. From this it can be seen that Jacobi iterative method share the same feature with our project.

Hence, in order to take advantage of SMP cluster to get higher performance, MPI message passing is used to accelerate top process level, then use OpenMP threads construct parallelize most time-consuming iteration part

where occurs at two level loop iteration.

First, we test with pure MPI model, map MPI process running on each processor. And then we deploy hybrid parallel mode on it which enable one MPI process and four OpenMP threads on each SMP node. There are a lot of methods used to measure performance of parallel program, each one have its own benchmark aspect. Two common used criteria are presented as below.

- Speedup

  According to the simplest term, the criteria to investigate parallel program is the time reduction. Hence, we use the straightforward measure which the ratio of execution time in serial running to the execution time in parallel running on cluster. The speedup equation is given as Equation 6.1:

  $$S = T_S \div T_P \tag{6.1}$$

  Where Ts is execution time in serial running and Tp is the execution time on cluster. Therefore, the Speedup also indicates the scalability of parallel program when numbers of processors are increasing. The ideal model is that the program could be accelerated in Speedup=N when N processors are used. This is so called linear speedup.

  But in practice, we always meet Speedup less than N because of a variety of problems such as load imbalance, communication overhead, synchronization overhead and memory consumption.

  The Amdahl's Law tells us if you want to enhance a fraction f of a program by a speedup s, which (1-f) is the fraction that could be parallelized [14]. Then the maximum speedup can be achieved as Equation 6.2:

  $$S(f,n) = 1 \div ((1-f) \div n + f) \tag{6.2}$$

  Moreover, the speedup of overall application is given as Equation 6.3, where s is speedup caused by (1-f) fraction:

  $$S = 1 \div ((1-f) \div s + f) \tag{6.3}$$

  Amdahl's Law describe two important factors:

  - When n tends to infinite, then the speedup bound to 1/f
  - In another word, if the 1-f is small enough, then the performance of parallelized program closes to the original one.

Because of those two features above, Amdahl's Law obeys the performance in reality. When proportion of S is increasing, the accelerated performance you get would much lesser than the price you cost. It seems like when the portion of code could be parallelized only have five percentages, the maximum acceleration is five no matter how many processor you used. This is the reason that the applicability of parallel computing is low, it is very useful when the numbers of processors relative small, but as the scale increasing, the benefits begin falling.

- Efficiency

  The efficiency of parallel program is the ratio of time which is used to take computational work by N processors. Which Ts is the execution time of sequential program, and Tp is the execution time on cluster. The Equation 6.4 is given as following:

$$E = T_S \div (n \times T_P) \tag{6.4}$$

or

$$E = S \div n[2] \tag{6.5}$$

  According to this equation, the value of E states the portion of processors performing actual computational work.

The results of case study can be seen in Figure 6.1 which shows speedup of two different programming model. As we see, the performance of hybrid parallel mode apparently higher than pure MPI model.

Leverage by hybrid programming model, Jacobi iterative model obtains higher ratio of computational to communication than before. Hence, as the efficiency equation previously mentioned, lower communication overhead leads to higher efficiency.

6.2 Performance results

The prototype performance of PBICG solver will be tested as the follow way:

1. OpenMP only (Preconditioner function on single SMP node)

   Results show in Table 6.1

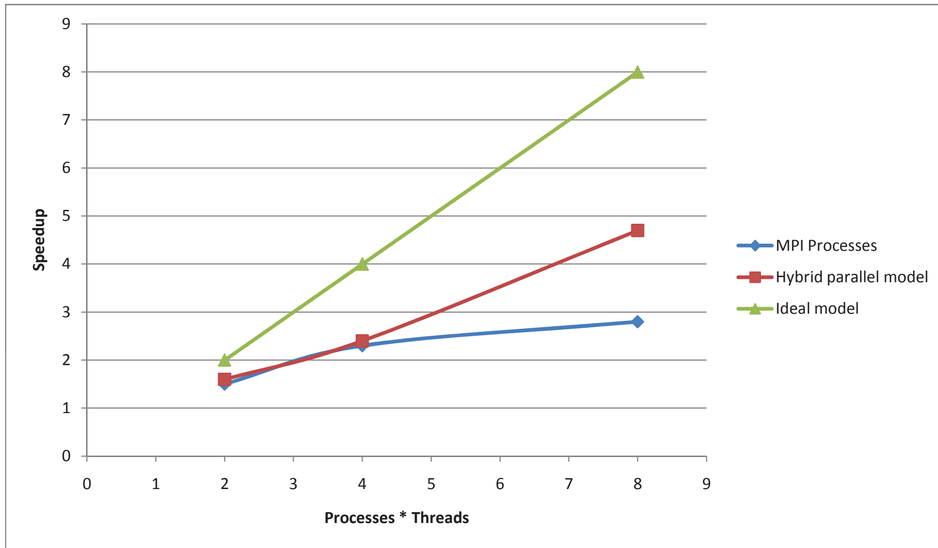

Figure 6.1: Comparison of speedup between Hybrid parallel model and MPI processes on the Jacobi iterative model

Table 6.1: Performance of pure OpenMP implementation in Preconditioner function

OpenMP Threads	MPI Processes	Execution Time [s]	Speedup [%]	Efficiency [%]
-	-	0.034	-	-
1	-	0.038	-	-
2	-	0.022	155	77.2
4	-	0.012	283	70.8
8	-	0.008	425	53.1

Table 6.2: Performance of pure MPI implementation in Preconditioner function

OpenMP Threads	MPI Processes	Execution Time [s]	Speedup [%]	Efficiency [%]
-	-	0.034	-	-
-	2	0.020	170	85
-	4	0.014	243	60.7
-	8	0.011	310	38.6

Table 6.3: Performance of pure MPI implementation in PBICG solver

OpenMP Threads	MPI Processes	Execution Time [s]	Speedup [%]	Efficiency [%]
-	-	0.21	-	-
-	2	0.142	148	73.9
-	4	0.088	237	59.6
-	8	0.058	362	45.2

Table 6.4: Performance of hybrid parallel implementation in PBICG solver

OpenMP Threads	MPI Processes	Execution Time [s]	Speedup [%]	Efficiency [%]
-	-	0.21	-	-
1	1	0.23	91.3	-
1	2	0.148	142	70.9
2	2	0.078	269	67.3
4	2	0.045	465	58.3
2	4	0.047	453	55.9
8	2	0.028	752	46.9

2. MPI only (Preconditioner function on single SMP node)

   Results show in Table 6.2

3. MPI only (PBICG solver including normFactor and Precondition function on SMP cluster)

   Results show in Table 6.3

4. Hybrid parallel model, OpenMP and MPI (PBICG solver including normFactor and Precondition function on SMP cluster)

   Results show in Table 6.4

6.3 Discussion

According to the benchmark we got from evaluation, two discussions will be presented between OpenMP and MPI, MPI and Hybrid parallel model. Since OpenMP implementation is assessed on single computational node, MPI and hybrid parallel model are tested on Linux cluster. It's more meaningful to compare them respectively.

- **OpenMP versus MPI on single SMP node**

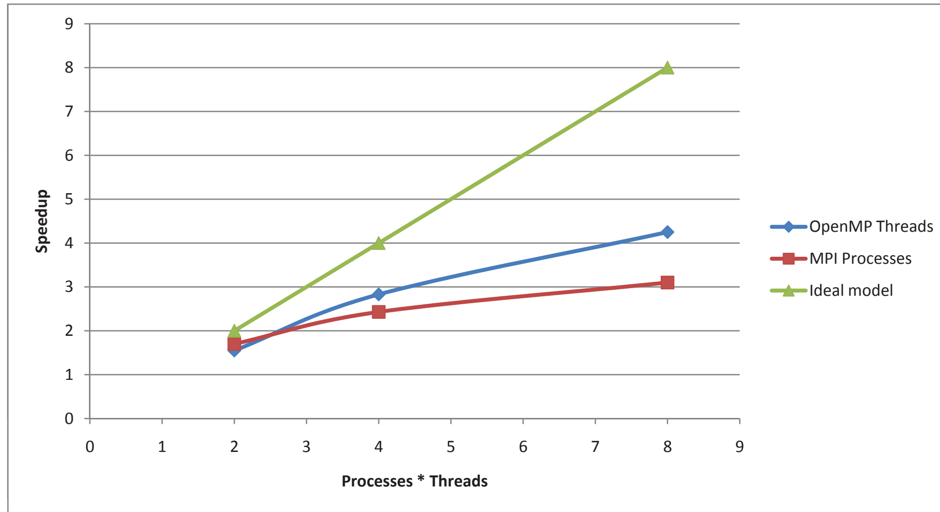

Figure 6.2: Comparison of speedup between OpenMP threads and MPI processes on Preconditioner function

Figure 6.2 shows us the speedup of OpenMP and MPI implementation for Preconditioner function. This numerical calculation performance test conducted on a single SMP node. For OpenMP part, we create eight threads on a single node to conduct comparison. The aim of this comparison is to investigate, for a given computational function part, such as Preconditioner, which approach brings better solution.

Therefore, to the specific function part, the MPI implementation gives good speed-up to four processes while OpenMP implementation almost acts the same to four threads. But after that, OpenMP scale better than MPI from four to eight. We believe this is due to OpenMP threads synchronization produce less overhead than MPI communication overhead. Regarding to the complex geometry, as we described previously, shared-block between each neighbor sub-domains have to exchange by processes, unless we allocate the shared block as many as possible on the same process. In such computational intensive partial program, OpenMP construct gives better solution.

However, there is one thing should be mentioned, this example are only for a partial function, it is not the common phenomenon for all OpenFOAM classes since most time consuming part in such function is two complex loop iteration. If we take a look at the whole dieselFoam program running in parallel, even only the UEqn part, the native MPI implementation produce very good speedup to eight nodes when submit

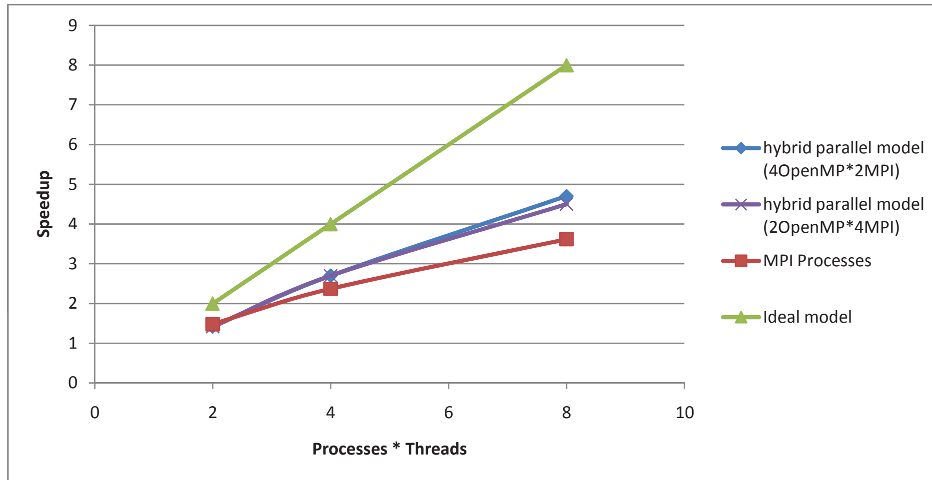

Figure 6.3: Comparison of speedup between MPI processes and Hybrid parallel model with different map topology on PBICG solver.

job to Volvo HPC cluster. When the serial fraction increasing, the performance by OpenMP construct begin failing. This is due to OpenMP limited by the ratio between serial part and potential parallelized part. And if the test object contains a big amount of data set and complex tree-based function call, instead of our test results, OpenMP under such circumstance gives considerable overhead create by synchronization and memory consumption [5] [19], HPC cluster improve the MPI performance by cache effect.

- **MPI versus Hybrid parallel model on SMP cluster**

  Figure 6.3 and 6.4 shows the project performance and speedup comparison separately. As Figure 6.3 shows, at first MPI implementation and hybrid parallel mode all perform nearly the same result, but after four processes, the MPI implementation begin slow down because of message exchange produce more communication overhead there. According to this figure, we can expect that, when more computational node involved, the situation getting worse since more "halos" message will be generated.

  Therefore, after we add OpenMP directive to the most time consuming part, compared to pure MPI implementation, the performance of PBICG solver is significantly improved by hybrid parallel model which contains eight MPI processes and two OpenMP threads in each process shown in Figure 6.4. We believe this is because better parallelism granularity

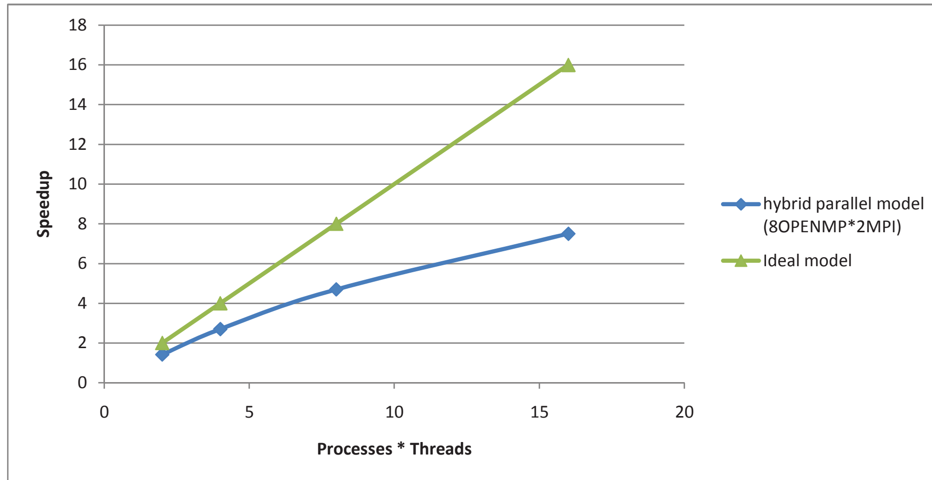

Figure 6.4: Performance speedup of Hybrid parallel model on PBICG solver by 2 MPI processes and 8 OpenMP threads on each.

is exploited after communications overhead reduction. Load imbalance problem is also reduced by adding one more level parallelism.

Moreover, we test PBICG solver with another mapping solution, with one MPI per SMP node, and four OpenMP threads per each MPI process. The results can be seen in Figure 6.3,

Although the different between each other is very tiny, but depends on theoretical knowledge, compared with one MPI process per SMP node, several MPI processes created per SMP node produce unnecessary intra node communication [10] [9], Since MPI implementation require data copying for intra node communication, it waste memory bandwidth and reduce performance even if intra-node bandwidth is higher than inter-node one. In other words, the number between MPI process and OpenMP thread depends on overlapping of computation and communication. In our case, fewer threads cause more intra communication overhead from shared domain exchange.

But meanwhile, we should notice that, more threads also cause more overhead on thread creation, destruction and synchronization. These created threads should involve in large amount of computational work, otherwise, it may reduce the performance. Such approach is only suitable for the computational intensive program.

Table 6.5: Time scale of UEqn object

Object name	Proportion (%)
fvm::ddt	13
fvm::div	15
fvc::grad	21
solve	22
etc.	29

6.4 Conclusion

Based on our performance speedup results, compare to two MPI processes deploy on two SMP cluster nodes, hybrid parallelism adding 8 OpenMP threads on each node increase 604% speedup on PBICG solve function. According to time scale of UEqn object in Table 6.5 and Amdahl's law, when the original time is presumed as 1, we could predict the running time for whole UEqn object as Equation 6.6:

$$\frac{0.13}{1}+\frac{0.15}{1}+\frac{0.21}{1}+\frac{0.22}{6.04}+\frac{0.29}{1}\approx 0.8 \tag{6.6}$$

Therefore the overall speedup for UEqn object is 1/0.8=1.25.

And if we consider [11]'s results, the speedup of UEqn object will be:

$$\frac{0.13}{1.2}+\frac{0.15}{1.2}+\frac{0.21}{1.2}+\frac{0.22}{6.04}+\frac{0.29}{1}\approx 0.74 \tag{6.7}$$

$$S_{UEqn}=\frac{1}{0.74}=1.35 \tag{6.8}$$

On the basis of results above, we take further analysis on dieselFoam application. Assume the performance of all other Eqn object (including PCG solver) could be increased in the same way. We find the the running time is:

$$\frac{0.14}{1.35}+\frac{0.29}{1.35}+\frac{0.11}{1.35}+\frac{0.12}{1.35}+\frac{0.12}{1.35}+\frac{0.06}{1.35}+\frac{0.16}{1}\approx 0.78 \tag{6.9}$$

$$S_{dieselFoam}=\frac{1}{0.78}=1.28 \tag{6.10}$$

If the assumption is true, comparing to the pure MPI implementation, dieselFoam program running on hybrid parallelism could be sped up by 1.28. But this results is based on we could do the same extent as PBICG in UEqn object, more work should be advanced in order to prove this is established.

Table 6.6: Time scale of dieselFoam solver

Object name	Proportion(%)
UEqn	14
pEqn $\times$ 2	29
hEqn	11
YEqn	12
Turbulence (k, epsilon)	12
rhoEqn	6
etc.	16

As we presented before, dieselFoam involves 100 times loop iteration in default. Besides, the running time is increasing as the iteration getting large, our experiment results are obtained by first ten times loop, in such way, the hybrid parallel model at least reduce the running time of dieselFoam program by 25%.

Indeed, this results still not yet significantly reduce the total running time, we believe this is caused by the program lack of fully parallelism. As above, there are still remain large proportion divide by 1 or almost equal to 1. According to Amdahl's Law in Equation 6.2 and 6.3, the performance you get is direct proportion to the ratio of potential parallelism code part. If the serial part proportion keeps at very high level, we could not get satisfying results by any means. Moreover, one things we finding here is that, Amdahl's Law equation points out diminishing returns problem. When we continually add more processors to same problem area, the return will fall until 1/f is reached. This is so called embarrassingly parallel problem, parallel programming only gives satisfactory answer for the case which involve small size of processors and high proportion of (1-f).

Next chapter, the summary of the whole project and the future work suggestion again to the leftover problem will be given.

Chapter 7

Summary of conclusion and future work

This thesis describes a hybrid parallel programming approach for OpenFOAM solver based on its native MPI implementation. The method deploy two-level hybrid parallel model on a linear equation solver called PBICG inside dieselFOAM. The entire mesh is divided into several sub-domains through domain decomposition. Each part with its specific geometric information and initial value is assigned to each SMP nodes separately. One communication schemes will be selected to all SMP cluster nodes, the communication scheduling and convergence criteria is handled by master node. During this time, each node exchanges the boundary/shared block by MPI message passing, within each node, the specific computation part is accelerated by OpenMP construct.

After time profiling, we use OpenMP threads replace some part of unnecessary MPI communication on linear solver equation code, which produce less overhead than message passing. Adopt with hybrid parallel programming, we get more speedup than native MPI implementation for PBICG solver. We expected that when the problem sizes get larger, the overhead of threads creation will take up much less proportion which results in better efficiency.

Threads parallelism is a straightforward approach to accelerate application, easily add OpenMP directive to the most time consuming loop iteration, but many evaluation results show that there is always misunderstand about what compiler really does to a given OpenMP statement. So it's important to analyze data structure and algorithm before any substantial coding work. The developer has to ensure the understanding of parallel execution is the same as compiler does in order to get the correct results.

We believe that, when a physical geometry and numerical algorithms is given, the efficiency of execution is mainly depends on the numbers of processors, CPU frequency, memory size, bus bandwidth, network latency and the quality of domain decomposition. Some relative work [8] proves that the best solution depends on the ratio of number of sub-domain to number of proces-

sor. In other words, that is how you decompose the domain to each node. More sub-domains lead to communication overhead, and reduce the efficiency of OpenMP construct. Although fewer sub-domains solve the communication overhead problem, but in such case, program cannot fully utilize the effect of cluster, it will also create more computation overhead. Moreover, some variety of problems like load imbalance, synchronization overhead also need to be considered. Based on this point, how to choose the optimal number between MPI processes and OpenMP threads, also known as mapping strategy, is also a important study in the future.

Hence, based on this thesis work, more code analysis should be carried out in order to find best domain decomposition approach and mapping topology between application and hardware. From the conclusion of previous chapter, we could find large part of dieselFoam class/function still remaining radix as one or close to one. They are all can be next research target. In additional, some optimization including cache improvement, OpenMP loop scheduling also need be taken into account in the future, in such case, better load balance could be obtained.

At present, hybrid parallel model is being a hot topic on accelerate CFD problem. This thesis work illustrate the PBICG solver which original design by only MPI and scale up to eight processes could perform excellent acceleration result by adding OpenMP construct. Also propose a reasonable approach on how to parallelize OpenFOAM application. For the purpose to gain more speedup of running time, more work should be conducted for the proposal suggesting above. In such ways, more performance speedup could be achieved.

Bibliography

[1] *http://universe-review.ca/R13-10-NSeqs.htm*.

[2] *http://www.clear.rice.edu/comp422/homework/parallelefficiency.html*.

[3] *http://www.volvo.com*.

[4] N. Duran A. Hoeflinger J. Yuan Lin Massaioli F. Teruel X. Unnikrishnan P. Guansong Zhang Ayguade, E. Copty. The design of openmp tasks. *Parallel and Distributed Systems*, 20:404–418, March 2009.

[5] J. M. Bull. Measuring synchronisation and scheduling overheads in openmp. *In Proceedings of First European Workshop on OpenMP (1999)*, pages 99–105, 1999.

[6] Larry Snyder Calvin Lin. *Principles of Parallel Programming*. Addison Wesley, 2008.

[7] Per Carlsson. *Tutorial dieselFoam*. Energy Technology Center, Lulea University of Technology, February 2009.

[8] William D. Gropp David E. Keyes. A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation. *SIAM Journal on Scientific and Statistical Computing*, 8:162–202, March 1987.

[9] Daniel Etiemble Franck Cappello. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. *Supercomputing, ACM/IEEE 2000 Conference*, page 12, Nov 2000.

[10] Gabriele Jost Georg Hager and Rolf Rabenseifner. Communication characteristics and hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. *2009 Parallel, Distributed and Network-based Processing*, pages 427–436, 2009.

[11] Wuilbert lopez Gustaf Hallberg, Per Hallgren. Parallelisering av algoritmer for flerkarniga processorer. Master's thesis, Chalmers University of Technology, 2010.

[12] Mattijs Janssens Andy Heather Sergio Ferraris Graham Macpherson Helene Blanchonnet Henry Weller, Chris Greenshields and Jenya Collings. *OpenFOAM, The Open Source CFD Toolbox - User Guide*. OpenCFD, 1.7.1 edition, August 2010.

[13] Gabriele Jost and Ferhat F. Hatay Haoqiang Jin, Dieter an Mey. Comparing the openmp, mpi, and hybrid programming paradigm on an smp cluster. *NAS Technical Report*, NAS-03-019:10, November 2003.

[14] Michael R. Marty Mark D. Hill. Amdahl's law in the multicore era. *IEEE Computer*, pages 33–38, July 2008.

[15] T. Bosilca G. Fagg G.E. Gabriel E. Dongarra J.J. Pjesivac-Grbovic, J. Angskun. Performance analysis of mpi collective operations. *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, 4:8, April 2005.

[16] Bruno R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, 1998.

[17] George N. Barakos Rene Steijl and Ken J. Badcock. Parallellisation of cfd methods for multi-physics problems. *European Conference on Computational Fluid Dynamics*, ECCOMAS CFD 2006:16, 2006.

[18] Gabriele Jost Rolf Rabenseifner, Georg Hager. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. *2009 Parallel, Distributed and Network-based Processing*, pages 427–436, 2009.

[19] Guido Nieken Rudolf Berrendorf. Performance characteristics for openmp constructs on different parallel computer architectures. *EWOMP'99-First European Workshop on OpenMP*, 12:1261–1273, October 2000.

TRITA-ICT-EX-2011:4

www.kth.se