Parallelisation of the Fluctuating Finite Element Analysis (FFEA) Research Software

Guanhao Lu

August 19, 2016

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2016
Abstract

This MSc project focuses on the implementation of MPI parallelisation to a novel molecular dynamic simulation software, Fluctuating Finite Element Analysis scientific software, designed by the University of Leeds Computational Biophysics research group. In this report, the detailed benchmarking of software performance is described. The existing OpenMP parallelisation performance is also analysed and some optimisation to the OpenMP code has been implemented. The MPI parallelisation uses a replicated data algorithm to parallel the most time-consuming part in the simulation. The parallelisation uses a derived structure datatype rather than C++ objects to enable the message passing of the proteins mesh surface data.
# Contents

1 Introduction .................................................. 1  
   1.1 Project Goals ........................................... 1  
   1.2 Report Structure ...................................... 2  

2 Background ................................................... 3  
   2.1 Software Overview ..................................... 3  
      2.1.1 Configuration Tool .................................. 3  
      2.1.2 Libraries .......................................... 3  
      2.1.3 Running Simulation .................................. 4  
      2.1.4 FFEA OpenMP Parallelisation ....................... 4  
   2.2 OpenMP Schedules ....................................... 5  
   2.3 MPI Parallelisation ..................................... 5  

3 Methodology .................................................. 6  
   3.1 Configuration Environment .............................. 6  
   3.2 Benchmark .............................................. 7  
      3.2.1 Input File ......................................... 7  
      3.2.2 Benchmarking Tools ................................ 8  
      3.2.3 Serial Benchmarking ................................ 9  
      3.2.4 OpenMP Benchmarking ............................. 10  
   3.3 OpenMP Optimisation .................................... 11  
      3.3.1 Correctness Test .................................... 12  
      3.3.2 OpenMP code structure ............................ 12  
      3.3.3 OpenMP Mode 1: Threads Within Blobs ............. 14  
      3.3.4 OpenMP mode 2: thread per blob .................. 15  
   3.4 MPI Parallelisation ..................................... 16  
      3.4.1 Unstructured Meshes Decomposition ................. 17  
      3.4.2 Data Structure ..................................... 17  
      3.4.3 MPI Parallel Strategy ............................ 17  
      3.4.4 MPI Implementation ............................... 21  

4 Results ....................................................... 26  
   4.1 Benchmark Results ...................................... 26  
      4.1.1 Serial Performance .................................. 26  
      4.1.2 OpenMP Mode 1 Performance ....................... 29
List of Tables

3.1 Proportion of OpenMP parallel regions execution time in Blob::update() and its child functions .................................................. 14
3.2 Data required in MPI communication among processes ............... 20
4.1 Allinea Map benchmarking FFEA function calls time proportion with hinges4-ud-10.ffea on Morar ........................................ 27
4.2 MPI timing with hinges4-ud-30.ffea on ARCHER ....................... 29
4.3 Comparison of performance with and without optimisation flags on Morar 29
4.4 Comparison of serial code and OpenMP mode 1 single thread performance on ARCHER ......................................................... 30
4.5 Speed-up of parallel sections in hinges4-ud-10.ffea on ARCHER .... 31
4.6 Performance of MPI parallelisation code on ARCHER ............... 34
List of Figures

3.1 OpenMP Code Structure ........................................... 13
3.2 Schematics of MPI parallel implementation to FFEA ............. 22
3.3 AoS vs SoA Memory padding ........................................ 24
4.1 Call tree with CrayPat benchmarking information of FFEA ...... 28
4.2 OpenMP mode 1 performance ....................................... 30
4.3 OpenMP mode 2 performance ....................................... 32
4.4 OpenMP mode 2 schedule dynamic 1 performance with hinges4-ud-30.ffd ......................................................... 33
Acknowledgements

I would like to thank my dissertation supervisor Dr. Toni Collis, from EPCC, and Dr. Joanna Leng, from the University of Leeds. The chance of analysing a novel biological simulation software is an interesting experience. The process of benchmarking and design parallel code gave me a good understanding of how high performance computing can help solve biological problems. Both Toni and Joanna have been a constant source of advice and help during this project.
Chapter 1

Introduction

The Fluctuating Finite Element Analysis (FFEA) is a continuum mechanical description of biological macromolecules conformational changes using a finite element algorithm on a mesoscale molecular level [1]. FFEA software package was developed by the University of Leeds Computational Biophysics research group with the aim of providing FFEA simulations both for commercial and scientific usage.

The model is designed to simulate the conformational dynamics of large proteins and other biological molecules including the thermal fluctuations and molecules interactions. Conventional atomistic molecular dynamics simulations provide a clear description of all the atomic position changes. To limit the computational expenses, FFEA uses a coarse-grained simulation to measure proteins dynamics [2]. Instead of reducing the time scale and problem size in the atomistic simulation, FFEA can perform complete biological simulations on the larger size proteins and over longer time scales. The use of coarse-grained data significantly reduces the computational complexity while specifying the length scale to keep the related physical parameters such as elastic and thermal energy in the continuum mesoscale model.

FFEA still takes a long time to simulate hundreds of proteins on the millisecond timescale. It utilises OpenMP parallelism to achieve scaling on shared memory systems.

1.1 Project Goals

The FFEA software was successfully ported to the School of Physics and Astronomy CPLab at the University of Edinburgh and the ARCHER, the UK National Supercomputer Service, systems during project preparation course.

The purpose of this work is to improve the strong and weak scaling of FFEA. Firstly this requires benchmarking to identify the most time-consuming parts of the software. FFEA software contains C++ for main simulation and Python code for visual tools. The C++ simulation code is the computationally expensive part, it requires long execu-
tion time when dealing with a large number of proteins. The development team have added some OpenMP directives in the code to achieve some parallelisation. This project needs a profiling tool that can support C++ parallelism code with OpenMP and MPI. ARCHER provides several benchmarking tools that can therefore be used to profile FFEA. As the FFEA code has also been installed on the CPLab machine, benchmarking of the performance on Morar, the backend supercomputer of the CPLab machines, has also been performed, providing a comparison of two HPC systems. The CPLab machine does not provide any profiling tools suitable for and therefore these were installed during the project.

The original OpenMP code has a limited improvement to the simulation performance. With the benchmarking result, this project expects to optimise the OpenMP parallelisation to improve the shared memory parallelism performance. The next step was to achieve strong scaling by implementing MPI parallelisation. In this case, the code can work on multiple nodes on a distributed memory system.

1.2 Report Structure

The report is split into five chapters. The first part is an overall introduction to FFEA software and project goals. Chapter 2 then detailed the background knowledge of FFEA software package and related parallelisation information. The parallelisation contains the OpenMP parallelisation which is already performed in the code, the MPI parallelisation which this project aim to implement to FFEA.

Chapter 3 then focuses on the methodology of each part of the work. The first section describes the FFEA configuration process on the project aim platform. The second section then introduced the detailed benchmarking process. With the benchmarking result, in the next section the OpenMP code performance is measured and then implemented some optimisation. The last section is the MPI implementation which provide distributed memory parallelisation to the code.

Chapter 4 is the corresponding results collected during this project. This chapter has a corresponding structure to the methodology chapter. It provides the data from the works with an explanation of their meaning.

The conclusion chapter 5 is divided into three sections with the information get from benchmarking, the optimisation results in OpenMP parallelisation and the performance of the implemented MPI parallelism.
Chapter 2

Background

2.1 Software Overview

The FFEA software contains two main parts, the FFEA_runner for simulation which is written in C and C++ code, the FFEA_tools for operation which is written in Python. FFEA offers several kinds of simulation for different biological systems. The simulation runs in a main loop and each iteration represents a time step in the simulation. In this project, the steric simulation model code, in which the development team focused during this project, is been analysed.

2.1.1 Configuration Tool

The software uses CMake to configure the source files and then create makefiles for building the system. There is a main CMake file in the root directory in the FFEA source file which calls other CMake files in each subdirectory of FFEA source file. Building options include the OpenMP mode chosen, calculation precision and optimisation flag should be set during the configuration and assigned with CMake command flags. The generated makefiles also separated in each subdirectory and they are automatic written by CMake. The modification of these makefiles cannot change the software package configuration statement as CMake keeps the configuration information. In this case, any changes that add or delete some files in the software needs to be recorded by updating the CMake files in the corresponding directories rather than changing the makefiles. For example, if a new C++ source file will be compiled in the simulation system, then the name of the new file should be added in the CMakeLists.txt file in the /src directory.

2.1.2 Libraries

The Boost, Eigen and Doxygen library are found during the configuration process and their paths are recorded in the makefiles. The Boost library in charge of the file I/O
and reading input option command to source code. The Eigen is a linear library that calculates and solve the linear approximation in the simulation. The Doxygen library is a documentation generator that can create a latex and a html version documentation.

2.1.3 Running Simulation

After the software is successfully generated in the building directory, the FFEA_runner and FFEA_tools executable files can be found in the /bin directory in the building directory. A test has been added in the building system during the late stage of this project, it checks the simulation system by running test file and printing success result if the software works correctly.

The FFEA_runner executable file then can be used to execute the input files. The input file includes the parameters block and systems block which are specified with angle brackets. The parameters block contains the global system parameters and the system block contains the detailed system information. The detailed information such as the node, surface face and van der Waals data etc. are saved in the corresponding files and input file specify the file name for the FFEA_runner to read the data. Once the software has read all the data in the file, it can then do the simulation and print out the result at each outputted time step to the output files. The output files are classed in 4 types include the measurement, trajectory, stress and force filed files. It used to be two kinds of the measurement files: the world measurement file for overall properties information and the blob measurement files for each blob properties information. The developer finally removed the blob measurement files and only keeps the world measurement. The trajectory file describes the protein structure in each outputted time step. The stress file recorded the thermal stress data and the force field records the force field parameter data. The latter two types of output files are created in the late stage of this project so only the former two are mainly focused here.

2.1.4 FFEA OpenMP Parallelisation

The development team of FFEA software designed two modes of OpenMP parallelism for different protein systems. The OpenMP regions are packed with pre-defined flags, one can choose a parallel region with the configuration flags. The code then uses specified OpenMP code with conditional statement. The parallel mode 1 is for a small number of proteins blobs especially a single large protein. The mode 2 is more suitable for a large number of proteins. The development team then found the second mode improves the code performance effectively but the first one provides limited improvement.
2.2 OpenMP Schedules

FFEA OpenMP parallelisation uses OpenMP for loop directives on its main time-consuming functions. In addition, some regions that have imbalanced iterations calculation use OpenMP schedules clauses. OpenMP schedules decide the assignment of iterations to threads in the OpenMP loop directives. There are four kinds OpenMP loop schedules, each uses a unique assignment strategy. The schedule decide the chunk of iteration size each time a thread can be assigned, it affects the thread load balance so should be carefully chose.

2.3 MPI Parallelisation

FFEA uses NETGEN source mesh decomposition code to create an unstructured mesh. The mesh decomposition tool usually can perform a parallel decomposition with MPI. However, NETGEN is a serial code that not support MPI parallel computing. The parallelisation strategy in this project is based on the algorithms in Advanced Parallel Programming. A direct linear partition of the mesh data is a quick solution of parallelising the mesh.
Chapter 3

Methodology

3.1 Configuration Environment

Throughout this work project, the development team at the University of Leeds have been updating the code resulting in the compiler and third-party libraries requirement changes.

The development of the FFEA software causes it sometimes cannot work with the specified version of the compiler or libraries on ARCHER as ARCHER provides several version of related modules and some of them are the latest version. It is possible that some unexpected conflict between the compiler and library may exist in the code. Some specific requirements for installing the software on ARCHER need to be illustrated.

ARCHER has three compiler suits: PrgEnv-cray, PrgEnv-gnu and PrgEnv-intel [4]. FFEA readme file suggests using either GCC 4.4 or Intel 13 as the compiler. Testing with the Intel compiler shows FFEA cannot use This compiler on ARCHER as it gives error message of compiling C++ 11 standard code. There is no Intel version 13 compiler on ARCHER so it cannot be tested. Therefore, the GCC compiler is selected in this project.

The default GCC 5.1.0 compilers gives warning messages about using the Eigen library and the development team say the GCC version over GCC 5.0.0 may incompatible with Eigen version under 3.3. Since ARCHER only has a 3.2.1 version Eigen library, it need to switch from GCC 5.1.0 to lower version. The GCC 4.9.2 compiler is then selected in this project.

During the project preparation course, the default Boost library can work well while in the beginning of this project the development team added some new functions that needs a new sub-library in Boost. The software then cannot link to the correct Boost library and failure in the building process. ARCHER has several versions of Boost libraries for each compiler, the code should link to a Boost library under the GNU compiler directory while it always link to the Cray compiler Boost 1.60 even when the library path was manually specified in the configuration command. To avoid the conflict,
compiler wrapper should be used to automatically link to the loaded libraries. Because FFEA uses a mixed C and C++ code it needs to specify two wrapper flags when running CMake, 
\(-\text{DCMAKE\_CXX\_COMPILER}=\text{CC}\) and \(-\text{DCMAKE\_C\_COMPILER}=\text{cc}\). The compiler wrappers bring a new problem as they default use statically linking while FFEA uses a dynamic library. Dynamic library and static library have different source files and FFEA cannot use the static library files. Then to manually set a dynamic linking method, the ARCHER environment parameter \text{CRAYPE\_LINK\_TYPE} should be set to dynamic. In this case FFEA can be successfully installed on ARCHER. As the boost library use a dynamically linking, the boost module should be loaded every time open a new shell. It can also be recorded in the submission script file to directly submit the job on ARCHER.

A Later version of the code then meets a Boost library problem again. The compiler cannot found the library classes included in FFEA when compiling the code. The solution is use a lower version of Boost 1.55 rather than 1.60. Though the problem is solved, the detailed reason can be discussed in the future.

### 3.2 Benchmark

This chapter presents the performance of two OpenMP parallel modes and the serial mode. FFEA provides two modes of OpenMP parallelisation for different kinds of input file. One is for large single protein and the other is for a large number of proteins. The two parallel strategies contain different OpenMP directives, it is possible that they have different performance. The serial mode measures the software performance with no OpenMP overhead as it disabled the usage of OpenMP.

For measuring the scaling performance of the parallel code, Speed-up \(S(P)\) is introduced here:

\[
S(P) = \frac{T(\text{serial})}{T(P)}
\]

Being \(P\) the number of process and \(T\) the code running time.

#### 3.2.1 Input File

The structure of the input file has been introduced in the software overview section. It contains two blocks and describes protein systems with numeric and non-numeric parameters.

The input files used in this project are \textit{hinges4-ud-10.ffeea} and \textit{hinges4-ud-30.ffeea}. The two input files describe two systems with 10 and 30 fibrinogen molecules respectively, the molecules keep fluctuating but their centroid position stay static. The systems use the steric interaction which is a model of the var der Waals (vdW) force interaction in the system. Although the FFEA development team continue to design three other vdw
model, the steric interaction model is mainly focus on their current development. This is not the only vdW model in the software, the FFEA development team have designed four kinds of vdW force models. However, the development team current focus on the design of steric interaction model so the other three models will not be considered in this project. There are more options for other parts of simulation in the system when the project goes and corresponding code been found. It is possible that more benchmark and new parallelisation strategies to other options are introduced in FFEA when there are available testing data.

The input file has a parameter num_steps can set the simulation timescale which is useful to benchmark the scaling performance of parallelism code. For example, In the 10 molecules file the num_steps value is 2e2 which represents 200 iteration of the main simulation code, each iteration is 1 time step. In this project, the default value of num_steps in all the files are 5e2 which means run 500 iterations.

The input file uses a rng_seed parameter to get a random value in the simulation. It used to set to the wall time and use the time value as a random number. The development team suggest to set this parameter to a fixed value and the code can output consistent results. An alternative choice of getting consistent results is set the noise calculation parameter to 0 to disable the noise calculation.

### 3.2.2 Benchmarking Tools

A suitable profiling tool should support C++ parallelism code both with OpenMP and MPI as this project includes MPI parallelisation. In the project preparation report, the CrayPat profiling tool has been chosen to benchmark the code on ARCHER. As the FFEA code has also been installed on the CPLab machine, it is meaningful to benchmark the performance on Morar, the backend supercomputer of the CPLab machines, and then compare the different performance on two supercomputer systems. The CPLab machine does not have any available profiling tools for C++ parallelism code so it needs to be installed during the project. There is no available installation source of CrayPat can be found, so Allinea Map is the one used in this part. The latest version of Allinea Map support OpenMP code benchmarking while ARCHER use an old version that does not support detailed profiling of OpenMP code [3]. Therefore, this project has not provided the comparison of the performance using the same profiling tool on two platforms as the different profiling tool brings different overheads to the code. However, the MPI timing function is introduced to perform a manually profiling and comparison.

The profiling tools usually give benchmarking reports of each function call while the detailed timing in these regions cannot be collected. For more detailed benchmarking data MPI timing function MPI_Wtime() can provide the wall time between any specified section of code. The timing function gets the wall time each time it is called in MPI processes. A useful timing of a piece of code should get from the same process from two MPI_Wtime() function calls.
3.2.3 Serial Benchmarking

In the first part of benchmarking, a serial version FFEA performance is analysed and the most time-consuming functions are located.

FFEA default uses OpenMP parallelism to achieve possible scaling on a shared memory system. The OpenMP regions only work when the compiler finds OpenMP support therefore it can turn off during configuration to avoid the OpenMP overheads. FFEA sets a CMake flag -USE_OPENMP to choose whether use OpenMP or not. The default value is 1 to use OpenMP and in this section it is set to 0 to disable parallelism.

FFEA also provides a group of compiler optimisation flags to improve the performance. In this section the optimisation performance is compared with a non-optimised code. FFEA uses -USE_FAST flag to choose a group of optimisation flags depend on the compiler type and version. For the GCC compiler version under 4.8 and above it provides two groups of optimisation flags. As both the lowest version on Morar and ARCHER are upper 4.8 so they use the same optimisation flags.

Firstly, benchmarking the serial code on Morar with Allinea Map gives an overview of the code performance. Allinea Map provides a trial version with a requested license from its website. Before profiling the code need to be complied with the debug flag, in the Readme file it introduces FFEA debug option as -DCMAKE_BUILD_TYPE=Debug during software configuration. Allinea Map uses a user interface to set the profile program parameters, it allows remote job submission with a qtf type submission file. The submission file uses same grammar with common types of submission script files such as pbs but has some macro flags to get the parameters from the user interface. Therefore, a sge submission file can be renamed with qtf and use it to submit jobs to Morar by Allinea Map.

Allinea Map do not provide a function call tree to help describe the code structure. While profiling FFEA on ARCHER with CrayPat gives more information. CrayPat on ARCHER needs to be loaded with the environment module system before compiling FFEA. In this case, FFEA can be compiled with the profiling tool and then the FFEA_runner executable file needs to be compiled with CrayPat again to get the instrumented executable file. The new executable file now can produce a sampling profiling report after execution. It also provides a further focused tracing experiment by compiling the sampling data again. The tracing experiment then focus on the most time-consuming functions and create a new report for those functions. The text report can be opened in the Cray Apprentice2 visualisation tool to get alternative code information.

The benchmarking result can provide the program running time proportion of each function. The most time-consuming functions can be located in the source file for further analysis. Profiling tools provide functions overall running time but do not support detailed timing of specific piece of code. Therefore, MPI timing function is manually added in these time-consuming functions.

In order to use MPI timing function in FFEA, it needs to find an available MPI standard on the platform. As CMake is used to configure FFEA, it needs to find the MPI standard
The development team then create a CMake flag `-DUSE_MPI` and move all the MPI code in pre-defined flag regions to make the MPI functions an optional choice in FFEA. One can set `-DUSE_MPI=1` in the CMake configuration command to enable FFEA compiles with MPI. In this project, the MPICH standard is used and all C++ source files that include MPI code need to include a `mpi.h` headfile.

```cmake
if (USE_MPI)
    add_definitions(-DUSE_MPI)
    find_package(MPI REQUIRED)
    set(CMAKE_CXX_COMPILE_FLAGS ${CMAKE_CXX_COMPILE_FLAGS} ${MPI_COMPILE_FLAGS})
    set(CMAKE_CXX_LINK_FLAGS ${CMAKE_CXX_LINK_FLAGS} ${MPI_LINK_FLAGS})
    include_directories(${MPI_INCLUDE_PATH})
    set(LIBS_TO_LINK ${LIBS_TO_LINK} ${MPI_LIBRARIES})
endif(USE_MPI)
```

MPI timing has a much lower overhead, the running time of function calls are more reliable than Allinea Map. The software running time is compared both with or without optimisation flags on two platforms.

In the following sections, the compiler optimisation flags are disabled to exclude their influence. The compiler optimisation flags are highly depend on the system environment and code structure. Parallel code may change the serial code structure and the effects of optimisation flags are hard to measure. It is better to turn off the compiler optimisation flags to get more reliable scaling performance of parallel code.

### 3.2.4 OpenMP Benchmarking

FFEA has two modes of OpenMP parallelisation based on the thought of running different kind of systems. With the CMake configuration flag the two modes can be switched and compiled with OpenMP support.

Benchmarking of OpenMP code performance is divided into two parts, the mode 1 benchmarking and the mode 2 benchmarking. These two modes mostly use different OpenMP regions and have different target input file. Mode 1 is especially parallelise a single large proteins system with huge number of nodes, It uses a strategy to parallel the code with OpenMP threads inside each blob. Mode 2 is designed for systems with multiple molecules, it parallelises the code with OpenMP threads on each threads.

Both two modes are benchmarked in this project, on the target platform two versions of code was installed with setting different `-USE_OMP_MODE` flag value during con-
configuration. MPI timings are continuing used here to present the scaling performance of the specified section of code.

In the result section two input files with different proteins blobs and two platform (ARCHER and Morar) will be used to analyse the performance of the two OpenMP parallelisation modes. The input files, hinges-ud-10.ffea describes 10 fibrinogen molecules while the hinges-ud-30.ffea has 30 fibrinogen molecules. Before running any simulation, the hinges-ud-10.ffea file needs to be modified to run 500 time steps of simulation to get a long enough simulation process for getting stable results. This can be done easily with a change in a parameter value: `num_steps=5e2`.

Benchmarking of OpenMP codes should make sure that all threads using a same shared memory node as the threads movement among nodes linking with lower speed interconnection could decrease performance. ARCHER has 24 cores on one node, each node contains 2 NUMA regions connected with high-speed interconnection. Although each core can perform hyper-threading to use 2 threads, it is better to run one thread per core as hyper-threading cannot ensure the same scaling performance of single thread per core. Morar has 16 cores on one node and should not use more than 16 threads during benchmarking too. In addition, the threads should be statically located on the cores with a specified OpenMP environment variable. On ARCHER it is compiler flag `–d` while on Morar it can be set with environment parameters. The submission scripts on ARCHER and Morar shows how these are implemented. To get more stable result without the system influence, the data is collected from the average of 5 execution result.

With the profiling result an overall performance of the OpenMP code can be gathered and further detailed benchmarking can be implemented in the code. Profiling tools perform further benchmarking that shows the threads working status. It gives detailed information of how the OpenMP threads work which helps find the potential inefficient parts in the code.

3.3 OpenMP Optimisation

FFEA two modes of OpenMP parallelisation has quite different performance. It conforms the development teams description: the mode 1 designed for single large protein parallelisation has a limited performance; only when the input file contains a small number of protein blobs, while the mode 2 has better parallel performance especially when a large number of proteins is used in the input file. In the following sections, the performance of the two OpenMP parallelisation modes were tested, then corresponding analysis and optimisation was implemented intend to improve the parallelisation performance.
3.3.1 Correctness Test

OpenMP parallelisation does not report errors in the code, changes to the code may give wrong outputs while no alert provides. It is necessary to use tests to ensure that changes introduced by this project do not break it. As the development team have not released any unit tests for the FFEA software, in this project, a numeric comparison of the output files produced by the program was implemented by a shell script to ensure that the output data are correct.

The FFEA simulation produces random fluctuations in the results that are not consistent in multiple executions of the code. There are two parameters in the input file that cause the inconsistent result. The first one is the random seed `rng_seed` which used to be the wall-time and will enlarge the error. The second one is noise calculation `cal_noise` to calculates the random noise fluctuation. FFEA can get consistent results if both sets `cal_noise` with a constant value and turn off `cal_noise`. However, turn off noise calculation shows a decrease of the total execution time. It is better to keep noise calculation with `cal_noise=1` but replace the rand seed. The rand seed em `rng_seed` value 14 is recommended in the original input file that can replace the time defined random seed.

In order to compare the multiple lines of file differences, a numerical comparison tool with used to help quickly gets the files differences. The output trajectory files and the measurement files are introduced in chapter 2, they contain output data from each outputted time step.

Numdiff is a program that can compare the numerical fields in two kinds of files and can give a statistical report with pre-defined error tolerance \[13\]. It is more functional than `diff` command as it only prints out the different numbers rather than the whole different lines. Besides, it can read the scientific notation so there is no need to use string handle commands to transform the numeric format. With the script file, the multiple output file can be tested in one step. After comparing 10 groups of output files with `-S -q` flag, get the error tolerance value in measurement files is 4e-11, in trajectory files is 9e+2. Then these two values can be set in script file with `-S -q -a` value.

3.3.2 OpenMP code structure

As the most time-consuming functions have been found, the OpenMP code can be easily found and understood. The two modes use different OpenMP codes with pre-defined flags:

```c
#ifdef FFEA_PARALLEL_WITHIN_BLOB
#ifdef FFEA_PARALLEL_PER_BLOB
#ifdef USE_OPENMP
Two modes use own OpenMP directives and bring different speed-up performance. There is a section of code defined with the `USE_OPENMP`, the van der Waal forces solving function `(Vdw_solver())`, which has the same
```
OpenMP code both in mode 1 and mode 2. Except this part, the remaining code in mode 1 parallelises code in the main calculation loop (World::run()) while the mode 2 only parallelises the Blob::update() function. The Figure 3.1 shows how the OpenMP pre-defined flag is placed in the main loop.

If looking into code, two inner loops in the main loop are not parallelised in neither two of the parallelisation modes. This is due to the short execution time so there is no need to parallelise all the loops. The figure has not shown a detailed code structure, Blob::update() function has a more complex parallelisation strategy than described in the flow graph. Because it contains several child functions and all the time-consuming ones should be parallelised, OpenMP regions are placed in these child functions. The FFEA development team have placed all the OpenMP parallel regions on loops which makes the OpenMP code much easier to understand as less dependencies need to be considered. However, this may be the reason for low speed-up as thread imbalance determines the performance. The detailed parallelisation strategy and its performance discussion are explained in next section.
3.3.3 OpenMP Mode 1: Threads Within Blobs

Mode 1 is designed especially for a single large protein while the developer team finds it has no significant speed-up at all. In this case, a simulation of 30 fibrinogen molecules, may have worse speed-up than 10 fibrinogen molecules as there are too many proteins in the system.

One of the noticeable things is that the performance of 1 thread of parallelised code with hinges-ud-30.ffea on ARCHER uses significantly more time than the same serial code profiled. The OpenMP overhead is much more than the profiling tool overhead. Basically, OpenMP overheads come from the OpenMP region, every region such as omp_parallel, omp_for and omp_single, will cause a piece of extra execution time (N) [14]. Blob::update() method contains several child functions and they are called in the for loop (n iterations). In this case, a simple parallel region will be called n times and the overhead will be N*n. Therefore, the nested parallel region should be carefully placed in Blob::update() method since this method will be called hundreds of times in the main loop, especially in large n iterations loops.

<table>
<thead>
<tr>
<th>Parallel regions</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blob::update()</td>
<td>5.91%</td>
</tr>
<tr>
<td>Blob::rotate()</td>
<td>0.13%</td>
</tr>
<tr>
<td>Blob::position</td>
<td>0.17%</td>
</tr>
<tr>
<td>Blob::make_measurements()</td>
<td>0.37%</td>
</tr>
<tr>
<td>Blob::aggregate_forces_and_solve()</td>
<td>1.57%</td>
</tr>
<tr>
<td>Blob::euler_integrate()</td>
<td>1.07%</td>
</tr>
<tr>
<td>SparseMatrixFixedPattern::apply</td>
<td>7%</td>
</tr>
<tr>
<td>NoMassCGSolver::solve()</td>
<td>2.3%</td>
</tr>
</tbody>
</table>

Table 3.1: Proportion of OpenMP parallel regions execution time in Blob::update() and its child functions

The Blob::update() method call tree can be found in CrayPat visual tools. Table 3.1 shows the profiling data of a single thread execution in the Blob::update() child function OpenMP regions. The time proportion presents each of the OpenMP region execution time occupies in FFEA simulation.

With the call tree information the OpenMP region nested situation can be measured. The NoMassCGSolver::solve() function is called in Blob::aggregate_forces_and_solve(), during a 500 time step simulation it has been called 500 times. The NoMassCGSolver::solve() then calls two child function in a loop. These two child functions are parallelised with OpenMP region, vec3_add_to_scaled() and vec3_scale_and_add() in the /src/mat_vec_fns.cpp file. This two function are called in a loop with i_max iterations and then repeated hundreds of time steps. The i_max value in hinges4-ud-30.ffea is 1000 so the total repeated time is 500000. Both the two parallel regions are highly repeated in which case the overhead of running OpenMP directives can be considerable high.
A better solution is to remove the OpenMP region and place them in the outer loop. Although it is still be called from several child functions, the parallel region is not repeated again. However, the data dependence limited the changes in the code.

In other case, some OpenMP directives can even be removed since they are not one of the most time-consuming parts. Considering the overheads, this may actually have better performance than keeping the OpenMP code.

During the tests of remove the two OpenMP regions, the single thread overheads decrease to a common level but also lost its scaling performance. It confirms the reason of OpenMP overheads that discussed above and give useful guiding for further OpenMP parallelisation strategy design. The OpenMP regions should be carefully implemented and try to not place in a nested inner loop. Too much OpenMP regions can significantly increase the execution time so OpenMP code design would better focus on the core part rather than try to parallelise as much as possible.

OpenMP thread imbalance is hard to measure as each small section could introduce some part of the overall imbalance. Profiling tools report each parallel region do have a small overhand but cannot gives information of where the threads are waiting. The way to reduce the threads waiting is testing each region with different schedule clauses. This should start from the most time-consuming functions. In this case it should start from $\text{VdW}\_\text{solver}:\text{solve}()$ function. The remaining parts in mode 1 are dispersed to small sections each with very low proportion. It is hard to find their performance changes with different schedule clauses. Since the $\text{VdW}\_\text{solver}:\text{solve}()$ parallel code also works in mode 2, and mode 2 have imbalance issues too, the testing is left in mode 2 optimisation.

The other tests to improve the performance includes adding the parallel parts with new OpenMP regions to some loops. The actual performance is no significant changes and even increase the overheads.

Therefore, the optimisation is intent to achieve with the mode 2 OpenMP parallelisation.

### 3.3.4 OpenMP mode 2: thread per blob

Mode 2 parallelisation strategy is much simpler than mode 1 as it parallels each part of inner loops in the main loop. The number of active OpenMP regions are less than mode 1 and shows a reasonable overhead to the code. Running a one thread code on ARCHER uses similar time to a serial code.

From the OpenMP benchmarking result, the most problem here is thread load imbalance which causes the threads waiting for a considerable time. Loading balance can be solved with setting suitable schedule clauses to ensure each thread are assigned iterations in best strategy.

There are 4 clauses static, dynamic, guided and runtime and 3 of them can set the chunk sizes. In this section, the different combinations are compared in the search for the one
with the best performance.

It should be noted that the chunk size may not have the same values here as the input data is uncertain. Different data sizes and the number of iterations can have different performance with same chunk size. Thus, the main tests are concerned with the choice of clauses.

The main loop has 5 inner loops shows in table 4.2 while only Blob::update() and VdW_solver::solve() are the most time-consuming ones. These two parallel regions were been tested.

Before testing different schedule clauses performance, there are some simple optimisation can be done to improve the performance first. The first thing is combine the first loop with second loop. They have same iterations and no data dependency as the first loop is setting all the values to zero. By removing the iterator of second loop, it use the schedule of the first loop.

The second thing is to remove the OpenMP directives on the non time-consuming function as no scaling performance can get from these regions while only the overhead may increase the execution time. loop2 is not time-consuming while it is now combined to loop1 so this is not required. Loop 4 is the one that needs to remove the parallel region.

The tests of different schedule regions intend to find the shortages and try to improve the parallel performance. It is better to a fixed number of threads and compares the different performance. The number of threads should be able to perform most possible improvement to the code. From the timing figure in 4.3 it shows a slow increasing start from 4 threads. The data of hinges4-ud-30.ffe become significantly worse than the other one. Therefore choose number of threads more than 4 should easily see a change in the scaling performance. In the results section tests with 8 threads on Morar are presented and finally achieved useful improvement to the OpenMP mode 2 code performance.

3.4 MPI Parallelisation

Shared memory parallelism has a limited performance to a single node of the Non-uniform memory access (NUMA) structure system which shares cache memory among cores in a node. It cannot guarantee the performance when using multiple nodes as the data transmission is much slower between nodes than within a node. A better solution is implementing MPI parallelism to use more cores on the supercomputer. MPI can achieve distributed memory parallelism and usually have better scalability than OpenMP code. FFEA software uses unstructured tetrahedral mesh in the code, the MPI parallelisation strategies is based on the common mesh decomposition algorithm introduced in the Advanced Parallel Programming.
3.4.1 Unstructured Meshes Decomposition

The unstructured mesh is produced by NETGEN and already available in the code. FFEA uses multiple files to save the mesh structure and read these structure files by *ffea input file. This files describe the system parameter on each mesh angle and face. Usually libraries can perform unstructured mesh decomposition automatically to divide the mesh to each process. In this case, NETGEN is not designed for parallel mesh partition and could not works here [11]. Since the mesh data is already available in the file, considering the complexity of the code a quick solution is use linear partition to directly divide the data into equal parts and send to each process.

3.4.2 Data Structure

FFEA used several basic data to describe the system structure each with detailed physical parameters used in the fluctuation simulation.

A protein system usually contains several protein molecules, each molecular calls a blob in FFEA. The blob data have a index and its central position in the space. Each blob consists a large number of nodes which represent the tetrahedron mesh angles. The node object has a global index in the system, it uses three dimensional coordinate to describe its position. The faces data describe all the faces of the tetrahedron, each face object contains an array of four nodes object, the first three nodes in the array are the three angles of the face and the fourth one is on the opposite direction of the face. The four nodes together become the global index of the face data. The face on the blob boundary are called surface faces, they are connected when the blobs collide to each other. The element data is a matrix that contains ten nodes which positions are inside or on the surface of the element. Element object contains the basic physical parameters in the proteins systems, it has the total parameters values of the nodes.

During the project the face data is mostly used in the MPI parallelisation. Faces object is defined in the /src/Face.cpp source file, it contains the var der Waal forces which to be updated in each time step of process simulation process. It is not easy if a MPI message want to send or receive an object as MPI is designed to send buffers with fixed start memory address and fixed length. The C++ object does not assign continuous memory location therefore another alternative solution is implemented with creating a derived datatype when doing MPI communication.

3.4.3 MPI Parallel Strategy

The simulation process is repeated running in a main loop. The serial code benchmarking divide the main loop in 8 parts as the Table 4.2 shows. It is worth to start
the parallel from the VdW_solver::solve() function call as it takes most time during the main simulation process.

Benchmarking gives a detailed time consumption proportion of each part of calculation. With different input file the proportion would be quite different. The VdW_solver::solve() function is the most time-consuming one. After multiple running of serial FFEA code with two input files, it shows the solve() function average occupies 44.2% and 63.2% in hinges4-ud-10.ffea and hinges4-ud-30.ffea respectively. This means the parallelisation expected to see better performance in protein systems with a large number of molecules.

```cpp
int VdW_solver::solve(int num_blobs) {
    ...
    fieldenergy[i][j] = 0.0;
    for (...) {
        ....
        // LinkedList traverse
        while (llist != NULL) {
            ... do_interaction();
            llist = llist->next;
        }
    }
}
```

The solve() function contains a loop that search linked lists and then do a interaction force calculation between two linked faces do_interaction(). Depending on the simulation system van der Waal model type, the do_interaction() actually call from different inherited classes. The systems analysed in this project uses steric model and call an inherited class Steric_solver. From a detailed timing of each section in this function, the most time-consuming part is the traversing process of linked lists. Therefore, the linked list should be distributed to all the process to reduce its local execution time.

During distributing the data dependence is carefully considered as objects usually contains pointers point to other objects. Each node in the linked list contains a face object, face object then have pointers point to nodes and blobs. In the do_interaction function the face object is updated with the calculated van der Waal forces. The node object parameters and blob object parameters are also used in this function. The parallelisation need to send and receive all the three kinds of objects parameters among processes. Besides, the data communication need to be restricted in the master process doing van der Waall force calculation in each time step. Therefore, the corresponding results can be collected to master process and do the remaining parts of simulation.

A simple description of the parallel strategy is using collective communication to send data to each process, then each process do a local calculation. After all the processes
finished calculation the data can be gathered to master process. This process then re-
peated until the maximum number of simulation time step reaches.

Before the implementation there are some points that need to know first.

**MPI in C++**

The MPI standard *MPICH* is designed both for C and C++ code. Although it contains
the C++ binding in C++ MPI parallel design, the MPI functions are same and no much
special code for C++. The MPI C++ binding uses the standard C function call for-
mat, for example `MPI::Init()`. In C++ code it can also use MPI C function calls
`MPI_Init()`.

However, MPI3 now deprecated C++ binding and C syntax can be used in C++ code.
From the documentation it says MPI C++ binding is deprecated from MPI2.2 [5]. This
means there is no C++ MPI3 functions are available and it is also possible that all the
C++ code will be removed in the future. During the parallel code implementation, it
is been found that the `MPI_Type_create_struct` only have a C version function
with no C++ function available which confirmed the changing. There are special situ-
ations exists, C does not use boolean datatype as a basic one while C has `bool` type.
therefore, there is no corresponding MPI datatype for boolean data in MPI C binding.
The best way is still use MPI C++ datatype `MPI::BOOL` until an alternative way is
presented.

During this project, most case uses the MPI C code to keep the code with good mainte-
nance as MPI C++ binding may be removed.

**Linked List**

Linked List is a special data structure that usually used in C++ computing. It describes
a list that nodes connected by pointers, a node only knows its next node while no global
information is shared among nodes. Each node is created dynamically during the list
creation, therefore it is not continuous on memory.

MPI communication is based on memory access and allocation, it cannot send or re-
ceive non-continuous data neither a data with an unknown length. MPI single-sided
communication is able to create linked list on multiple processes but all the list head is
on the master process, it is not an efficient way to improve the performance here. In-
stead, the lists are created on all processes and each process do it own part of traversing.
The linked list traverse is the most time-consuming part in the var del Waal force calcu-
lation part. By distributing the lists traverse work on multiple processes, it is possible
to improve the performance.

A similar example is sending two dimensional arrays as it is often be treated as contin-
uous datatype while actually it is not continuous. However, it is different from linked
lists as it can allocate continuously and is able to use in MPI [9].
Derived Datatype

MPI Derived datatype allows data with static memory communicate via MPI without the standard MPI datatype limit. The data memory location can be non-continuous only if the displacement is able for measurement. FFEA has defined several kinds of self-defined datatype and this project need to use them during parallel implementation. It is also possible to convert the objects to a derived datatype which could be accessed by MPI functions.

Object datatype cannot be directly transferred to other processes as the object contains dynamically located pointers of its parameters. Sending or receiving the head address of an object cannot get the full content. Therefore, alternative ways for object communications is introduced.

Before the introduction of object communication, it is better to know the object that intends to be transferred in FFEA. The van der Waal force calculation part solves the face interactions simulation. The interacting faces are saved in linked lists and then do the force calculation. In order to distribute the data on all processes, the linked lists and its elements need to be packaged. Table 3.2 shows the main variables that required during the calculation, the variable come from three different classes of objects. As the object name shows all the variables are in the `LinkedListNode<Face> l_i / l_j` objects, these objects are linked lists with Face objects list nodes. The first index and the following x,y,z variables describe the `LinkedListNode` element parameters. The remaining variables describe a face object that corresponding to the node element. The variables contain several self-defined datatypes. The scalar datatype is either float or double depends on compilation options. The `vector3` datatype is a three dimensional vector data, each dimension is a scalar value. These two datatypes are self-defined in FFEA and cannot directly use in MPI.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Data type</th>
<th>Object Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>int</td>
<td><code>LinkedListNode&lt;Face&gt; l_i / l_j</code></td>
</tr>
<tr>
<td>x,y,z</td>
<td>int</td>
<td><code>LinkedListNode&lt;Face&gt; l_i / l_j</code></td>
</tr>
<tr>
<td>normal</td>
<td>vector3</td>
<td><code>Face l_i-&gt;obj</code></td>
</tr>
<tr>
<td>normal</td>
<td>int</td>
<td><code>Face l_i-&gt;obj</code></td>
</tr>
<tr>
<td>daddy_blob&gt;index</td>
<td>int</td>
<td><code>Blob l_i-&gt;obj-&gt;daddy_blob</code></td>
</tr>
<tr>
<td>daddy_blob&gt;index</td>
<td>vector3</td>
<td><code>Face l_i-&gt;obj</code></td>
</tr>
<tr>
<td>kinetically_active[num_blobs]</td>
<td>bool</td>
<td><code>Face l_i-&gt;obj</code></td>
</tr>
<tr>
<td>centroid</td>
<td>vector3</td>
<td><code>Face l_i-&gt;obj</code></td>
</tr>
<tr>
<td>vdw_bb_interaction_flag</td>
<td>bool</td>
<td><code>Face l_i-&gt;obj</code></td>
</tr>
<tr>
<td>vdw_bb_energy[num_blobs]</td>
<td>vector3</td>
<td><code>Face l_i-&gt;obj</code></td>
</tr>
<tr>
<td>vdw_bb_energy[num_blobs]</td>
<td>vector3</td>
<td><code>Face l_i-&gt;obj</code></td>
</tr>
<tr>
<td>force[3]</td>
<td>vector3</td>
<td><code>Face l_i-&gt;obj</code></td>
</tr>
<tr>
<td>vdw_bb_force[num_blobs]</td>
<td>vector3</td>
<td><code>Face l_i-&gt;obj</code></td>
</tr>
<tr>
<td>n-&gt;index</td>
<td>vector3</td>
<td><code>Mesh_node l_i-&gt;obj-&gt;node</code></td>
</tr>
<tr>
<td>n-&gt;pos</td>
<td>vector3</td>
<td><code>Mesh_node l_i-&gt;obj-&gt;node</code></td>
</tr>
</tbody>
</table>

Table 3.2: Data required in MPI communication among processes
Two strategies can be produced to achieve parallel on the above code. The first one is package all the data in the objects, in other words, serialise the object to a fixed length buffer. The second one is creating a new data structure that can set the parameters on memory statically.

The first strategy can be implemented with Boost::MPI library serialisation functions. The Boost library is currently used in FFEA but Boost::MPI is a sub library that not includes in the FFEA complication [6]. By adding this sub library path in CMakeList.txt and re-compile the code, it is able to use the Boost::MPI functions. The library documentation says it could support Plain Old Data (POD) type which means the self-defined datatypes need to be transformed to a standard datatype. It will increase the amount of message buffer and makes the coding more complexed as one vector datatype need to divide into three variables. The serialisation also creates a buffer includes all the data in the object, which is not necessary for parallel VdW_solver::solve() function. It cannot select useful information but only package all the data inside like MPI_Pack does, they perform similar operations in packaging data [12]. With these two reasons, this MPI library is not used here.

The second strategy, which is applied in this project, is create a structure that contains all the variables mentioned above. It is much efficient than serialise the object and do not need any third-party libraries support. MPI can create derived datatypes to send and receive structures. Structure uses static memory assignment in which case each element can be accessed by setting the memory displacement [7]. MPI_Type_create_structure function call does the construction of derived datatype for structure. In the implementation section the detailed process of creating the structure will be explained.

### 3.4.4 MPI Implementation

FFEUA has compiled with MPI during the benchmarking section to enable using timing function MPI_Wtime(). MPI has been added in CMake file as an optional choice of compiling FFEA. The default set is not using MPI, one need to set -USE_MPI=1 to enable MPI parallelisation in FFEA. All MPI code are packaged in a pre-defined flag #ifdef USE_MPI so the original serial code can still work. The implementation tries to avoid modification of original code structure to keeps good maintenance.

Figure 3.2 gives the flow graph of the parallel implementation. It uses replicated data as each process contains a piece of global data. The data is broadcast to all process before doing var der Waal force calculation. Each process then does an equal part of local calculation, the scaling performance comes from this part. After all processes finished local work, the data then be collected and reduced. The force and vdw_bb_force data are return to zero after each time step so there is no need to update the value on all process. A temporary array then gets the reduced force value on master process with MPI_Reduce and then add the force to its local value.

The first step is put all the serial parts code in the MPI rank conditional statement so that only the master process can access these parts. The MPI init and finalise function
are placed in the beginning and the end of the main function in /src/ffea.cpp, therefore all the data including the input options are available in all the processes. With the different input option the code will goes to different conditional sections. There is a simple process communication in the option choice, as the input option gets from input file with Boost library file handler function, it cannot be parallelised on all process which will cause replicated file opening and reading. Therefore the option of FFEA simulation mode is broadcasted to all process from master process. In the conditional statement the main simulation object world is created on all the processes. Only the section of code that need to be parallelised are not concluded in the MPI rank conditional statement. FFEA has a large number of printing statement to explain the current simulation state, it is good for user to understand the what the software is doing but increase the length of MPI code as all the printing and file handle statement need to be serialised.

Once all the serial parts are set, the next step is divide the most time-consuming function and scatter its data to all processes. As the discussion above, in the simulation process the function call VdW_solver::solve() mainly time consuming part is the traverse of the linked lists. The lists describe the faces interaction and each list element, the face, has its nearest faces linked to it. The space position decide which elements are in a same list. In this case the simple linear partition by face index may break lots of lists to several processes nodes. Even the connection relationship is measured to keep
all the lists, it cannot make sure an even partition as some processes may have especially long lists. In this project the lists data is broadcast to all processes so each process has a copy and do its own section of work.

Sending the lists data needs structures to contain the variables with static memory allocation. Then the data can be sent with `MPI_Type_create_structure`. As each surface face is the head of one linked lists, there are large amount of lists need to be converted to structures. Here comes a choice: to create array of structures or create a structure of arrays?

### AoS vs SoA

Array of structures (AoS) and structure of arrays (SoA) are quite different data structure on the memory allocation method [8].

Firstly, the structure has a special memory allocation ways, it has two standards: memory alignment and memory padding. Memory alignment means structure in C and C++ always try to use a standard length of memory for each variable memory allocation. For example, if a structure has a first variable type `char` and second variable type `int`, the second one is obviously using 4 bits memory address while the `char` itself is 1 bit long but the memory displacement between these two variables are 4 bits. The system use a standard length to save the structure variables, which is know as memory alignment. On different systems the standard data length may be different. The memory padding is a sort of empty memory to distribute different group of data. two structures can have 4 bits padding between the last variable of first structure and the head of second structure. The padding and alignment method depends on system and compiler, there is no unify values of the memory displacement.

Although memory alignment and padding cause the empty gap between variables, each element starting address can be get with `(MPI_Aint)offset` function, which measure the memory displacement of each variable with the structure starting address. The really problem comes from using structure of arrays. Figure 3.3 shows the memory allocation of two data structures. The array of structures can keep its primary data structure with padding between neighbour structures. Each variable memory address can still get from `offset` function. However, using structure of arrays will lose the original data structure as the different datatypes will re-sort and organised by same datatypes. It is hard to get a specific variable memory address as different systems could have quite different memory allocation strategies.

Therefore, create the multiple linked lists should use array of structures rather than a structure of arrays. As discussed above, the `offset` functions can get the head address displacement of each structure variable. In addition to each variable, an extra displacement is implemented in the MPI structure datatype creation. This displacement value `MPI_UB` represents the overall size of one structure as the memory padding may cause extra memory space in the data structure [10].
Structure of Faces

Table 3.2 has represented the variables that need to be structured in a structure datatype. The parameters x, y, z are values in linked lists so they are not included in the structure. Here the structure should be a reduced Face object data that only includes the required variables. Self-defined datatypes need to be converted to MPI datatypes before included in the structure type.

The vector3 datatype may not be continuous as it is a structure datatype with three parameters. A safely solution is create a structure datatype for vector3 and then include it in the outer structure type. Vector3 is intend to be replace with array[3] which could do exactly the same thing. The array[3] will be much easier to use when parallelise with MPI but it needs a lot of changes in the code to replace it. Replacing vector3 with arrays needs a vector3 type one-dimension be replaced with a two-dimension array. A lot of functions should do corresponding changes to finish the replacement. Though it should be easier to send a two-dimension array in MPI as it can be treat as a one-dimension array, the modification of vector3 to array is a heavy work.

The scalar is actually a common datatype but it is defined with the precision choice dur-
ing configuration. The scalar can be a single precision float datatype or a double preci-
sion datatype. It can be renamed with the MPI_type_continuous() and chose datatype with predefined flag so other MPI function can use this datatype.

With the derived datatypes for self-defined datatypes, the structure derived datatype can be created as a nested derived datatype. There is one thing that need to be considered as the array of structure cannot use pointer variables to dynamically allocate it memory, which is exactly the way of using structure of arrays. The number of blobs can only get when the system initialised, which means the variables that are arrays with the size of blob numbers can not contain in one structure. The current solution is use a two dimensional array in which the second dimension contain the array variable values.

Now the derived MPI datatype is created and could be used in collective communi-
cation. With the own designed datatype. One derived data size is 288 bits while the original face object is more than 600 bits. With a larger protein system, the face object data size will be even larger than that.

<table>
<thead>
<tr>
<th>double</th>
<th>double</th>
<th>double</th>
<th><em>pad</em></th>
<th>int</th>
<th>int</th>
<th>int</th>
<th><em>pad</em></th>
<th>char</th>
<th>char</th>
<th>char</th>
</tr>
</thead>
</table>

| double | int | char | *pad* | double | int | char | *pad* | double | int | char |

Figure 3.3: AoS vs SoA Memory padding
Linked Lists

Now the face structure is available to create the linked lists. On processes except master one, linked lists are created with a similar structure, the list node element is a face structure rather than a face object. Each list has a index that specifies a unique surface face in the system. The parallel distribution is depended on the surface face index. Situation of uneven division is considered and the remaining faces are distributed to processes with the biggest rank index, since the process 0 have to handle IO it is better to load less work on it.

Collective Communication

The method is broadcast the nodes data and then create the linked lists like the code does when creating build_nearest_lookup in the main loop. The building process increases the overhead, they can be measured with MPI timing. once all the process has its own linked lists, they can perform their own part of VdW_solver::solve function. Master process do the original function, other processes with the copied data do a new function and record the calculation result in their lists.

When all the process finished solve function, use an all reduce function to reduce the values to each process.

Processes communication uses MPI collective functions, the data assignment from local linked lists to the message buffer is an overhead here. As a complex protein system contains a large number of proteins, the assignment of message buffer need to use long iterations loop. Although the structure data already uses only the required variables,

the stage of collect the whole array of structures still is a high consuming process. Though the collective communication overhead is reduced compare to directly sending serialised objects. The data conversion from structure to objects cost much more time and become the main overhead here. To reduce the overhead, the amount of data conversion should be minimized. The array of structures creating and initial value assignment should be completed during the simulation initial part. During the broadcasting and reduction processes only the required data should be sent.

The high overhead of array assignment was solved by move the interaction calculation to master process. With this method all the required data on processes except master one is face centroid and kinetic state, they are the condition of creating linked lists. All the interaction faces are send by local processes with the face global index, master process then get the face object in the lookup array to do the calculation. However, this solution cause some problems when processes get no interactions and master cannot get the ending message to terminate the process.
Chapter 4

Results

4.1 Benchmark Results

4.1.1 Serial Performance

Allinea Map provides a summarise information of the system resource usage on its visualisation tool when running FFEA program. The profiling result with the input file hinges4-ud-30.ffea on Morar shows the file I/O occupy 0.2%, the floating-point execution occupy 2.5%. Therefore, the I/O functions do not have much influence to the code performance and can be excluded during parallel code performance analysis and parallel strategy design. The floating-point execution time is surprisingly lower than expected in FFEA, in the MPI implementation section this will be mentioned and further analysed. The libraries linking also can be ignored with a very low percentage of 1.2%. Actually in FFEA the only use of boost is handling the file I/O and input commands. The Eigen library is used in linear approximation functions but occupy a very small proportion.

The data collected in this section have not use average value from multiple runs as no accurate data is required in this section. Allinea Map has a detailed function calls time proportion table that can be exported from it. The table contains the detailed time proportion of function itself consumption and child function consumption.

Table 4.1 shows the total time proportion over 5% functions collected from the Allinea Map exported data, the time proportion includes the proportion of function call itself, its child function calls and the overall time consumption. A more completed table is provided in the appendix.

It is obvious that the VdW_solver::solve() function takes the most execution time with the total time proportion of 45.20%. The function contains most calculation in itself rather than its child functions. The second one is Blob::update() which occupies 41.40% of total execution time. Different from the VdW_solver::solve() function, this function
Table 4.1: Allinea Map benchmarking FFEA function calls time proportion with hinges4-ud-10.ffea on Morar

<table>
<thead>
<tr>
<th>Function</th>
<th>Total</th>
<th>Self</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>100.00%</td>
<td>&lt;0.1%</td>
<td>100.00%</td>
</tr>
<tr>
<td>World::run()</td>
<td>94.30%</td>
<td>&lt;0.1%</td>
<td>94.30%</td>
</tr>
<tr>
<td>VdW_solver::solve()</td>
<td>45.20%</td>
<td>41.60%</td>
<td>3.60%</td>
</tr>
<tr>
<td>Blob::update()</td>
<td>41.40%</td>
<td>0.50%</td>
<td>40.90%</td>
</tr>
<tr>
<td>Blob::aggregate_forces_and_solve()</td>
<td>29.50%</td>
<td>3.30%</td>
<td>26.20%</td>
</tr>
<tr>
<td>NoMassCGSolver::solve()</td>
<td>24.50%</td>
<td>&lt;0.1%</td>
<td>24.50%</td>
</tr>
<tr>
<td>NoMassCGSolver::get_alpha_denominator()</td>
<td>10.30%</td>
<td>0.60%</td>
<td>9.70%</td>
</tr>
<tr>
<td>SparseMatrixFixedPattern::apply()</td>
<td>9.70%</td>
<td>9.70%</td>
<td></td>
</tr>
<tr>
<td>SparseMatrixFixedPattern::build()</td>
<td>7.40%</td>
<td>1.30%</td>
<td>6.10%</td>
</tr>
<tr>
<td>sparse_entry_sources::sum_all_sources()</td>
<td>6.10%</td>
<td>6.10%</td>
<td></td>
</tr>
<tr>
<td>World::init()</td>
<td>5.70%</td>
<td>&lt;0.1%</td>
<td>5.70%</td>
</tr>
<tr>
<td>World::read_and_build_system()</td>
<td>5.50%</td>
<td>&lt;0.1%</td>
<td>5.50%</td>
</tr>
<tr>
<td>Blob::init()</td>
<td>5.40%</td>
<td>0.20%</td>
<td>5.20%</td>
</tr>
<tr>
<td>NoMassCGSolver::get_alpha_denominator()</td>
<td>9.70%</td>
<td>9.70%</td>
<td></td>
</tr>
<tr>
<td>SparseMatrixFixedPattern::build()</td>
<td>6.10%</td>
<td>6.10%</td>
<td></td>
</tr>
<tr>
<td>sparse_entry_sources::sum_all_sources()</td>
<td>5.70%</td>
<td>&lt;0.1%</td>
<td>5.70%</td>
</tr>
<tr>
<td>World::init()</td>
<td>5.50%</td>
<td>&lt;0.1%</td>
<td>5.50%</td>
</tr>
<tr>
<td>Blob::init()</td>
<td>5.40%</td>
<td>0.20%</td>
<td>5.20%</td>
</tr>
</tbody>
</table>

CrayPat visualisation tool provides a function call tree with each function running time proportion. The call tree gives an overview of the code structure and the function running time proportions. The figure 4.1 shows a main part of the FFEA function call tree with the input file hinges4-ud-30.ffea running on ARCHER.

With the profiling information and function call tree the most-timing functions can be located in the code and also find their function calls in the main loop.

The MPI timing then gives a more detailed timing of the code. The main function is in the ffea.cpp file which will create a world object that defined in World.cpp. The most time consuming functions are called in the World::run() function. It contains a loop that each iteration is a single simulation time step. The main loop contains several inner loops and function calls of the most time-consuming functions: VdW_solver::solve() and Blob::update(). The table 4.2 shows the MPI timing of hinges4-ud-30.ffea on ARCHER.

The whole code is separated and timed in 3 parts: the initialise part, the main loop and the finalise part. The initialise part and finalise part occupy short execution time, thus in this project firstly focus on the main loop parallelisation. The VdW_solver::solve() and Blob::update() function calls occupy the most execution time, which conforms to the profiling tools results. while the other loops also occupy some execution time in the main loop. Therefore, these inner loops should also be parallelised for better performance. These loops should be parallelised with OpenMP directives and they are confirmed when looking into the code. The code has not entered the last loop after the conditional statement. This loop is a quick kinetic energy calculation, it is not included.
Figure 4.1: Call tree with CrayPat benchmarking information of FFEA
<table>
<thead>
<tr>
<th>Section parts</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialisation (World::init())</td>
<td>2.796</td>
</tr>
<tr>
<td>loop1 (zero the force across all blobs)</td>
<td>17.995</td>
</tr>
<tr>
<td>loop2 (modify blob position)</td>
<td>0.417</td>
</tr>
<tr>
<td>loop3 (centroid blobs &amp; reset face)</td>
<td>6.857</td>
</tr>
<tr>
<td>function call (VdW_solve::solve)</td>
<td>120.753</td>
</tr>
<tr>
<td>loop4 (zero node forces)</td>
<td>0.088</td>
</tr>
<tr>
<td>function call (apply_string)</td>
<td>8.491</td>
</tr>
<tr>
<td>function call (Blob::update)</td>
<td>51.837</td>
</tr>
<tr>
<td>loop5 (kinetic blob state change)</td>
<td>0</td>
</tr>
<tr>
<td>Finalisation (delete world)</td>
<td>0.001</td>
</tr>
</tbody>
</table>

Table 4.2: MPI timing with hinges4-ud-30.flea on ARCHER

in the current simulation system and is not considered in this project.

There is one thing that different from the benchmarking result in Allinea Map. The Blob::update() function occupies much less time than the Allinea Map benchmarking result. The timing of different files on both Morar and ARCHER shows different input files the Blob::update() function occupies different proportions. It is presented in the OpenMP benchmarking section that the Blob::update() function in the input file with more proteins has a lower proportion.

<table>
<thead>
<tr>
<th>File names</th>
<th>Optimised time(s)</th>
<th>Non-optimisation time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hinges4-ud-10.flea</td>
<td>167.857</td>
<td>235.649</td>
</tr>
<tr>
<td>hinges4-ud-30.flea</td>
<td>547.673</td>
<td>824.496</td>
</tr>
</tbody>
</table>

Table 4.3: Comparison of performance with and without optimisation flags on Morar

In addition, the optimisation flags performance has been tested on Morar with different input files. Table 4.3 shows the optimisation flags can save a plenty of execution time on Morar. It is an efficient way to improve the software performance. Although it is not used in this project, it is worth to set the default option of using the optimisation flags true when releasing to public as they can provide significant improvement to the software performance.

### 4.1.2 OpenMP Mode 1 Performance

The mode 1 benchmarking shows a significant overhead of OpenMP regions. From the timing of FFEA mode 1 single thread performance, it has a clearly extension compared to the serial code.

Table 4.4 compares the two input file execution time on ARCHER, hinges4-ud-30.flea used to running approximately 212 seconds while the OpenMP mode 1 overhead increase the execution time by more than 30 seconds. It also sees different running time
File names | Serial Time (s) | OpenMP mode 1 thread time(s)
---|---|---
hinges4-ud-10.ffea | 69.758 | 72.508
Hinges4-ud-30.ffea | 212.102 | 245.825

Table 4.4: Comparison of serial code and OpenMP mode 1 single thread performance on ARCHER

Figure 4.2: OpenMP mode 1 performance

extension percentage with different input file. Hinges4-ud-10.ffea has a less increase time when running OpenMP mode 1 code. The OpenMP overhead when running 30 molecules is worth to analyse and try to reduce it in the following sections.

The next step the code scaling performance is analysed on two platform with different input files. Hinges4-ud-10.ffea and hinges4-ud-30.ffea has different number of molecules, the timing result shows the 10 molecules system has a better speed-up performance than the 30 molecules system. Table 4.2 is the two input files performance on ARCHER and Morar. It is clear different system do not cause much difference to the scaling performance while the input data do the majority. Although FFEA has not showed good scaling with both two input files, the result confirm the mode 1 parallelisation descriptions that it is designed to parallel a single protein system. Too much molecules in a system can not perform enough scaling with OpenMP mode 1. With the profiling report in CrayPat a basic threads usage information is presented. The benchmarking result of running hinges4-ud-10.ffea shows 36.2% overheads comes from OpenMP function calls while about 18% comes from thread barrier waiting. This means the possible reason of the exceed execution time is too much OpenMP region calls during the software execution. The threads imbalance could also explain part of
the reason that causes low scaling performance.

The MPI timing of each section of code then gives more information about the parallel performance of each section. Table 4.5 presents the maximum speed-up achieved on ARCHER, the VdW_Solver::solve() function actually perform a good scaling while Blob::update() can nearly improve its performance.

<table>
<thead>
<tr>
<th>Function calls</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>VdW_solver::solve()</td>
<td>x11.91</td>
</tr>
<tr>
<td>Blob::update()</td>
<td>x2.9</td>
</tr>
</tbody>
</table>

Table 4.5: Speed-up of parallel sections in hinges4-ud-10.flea on ARCHER

The maximum speed-up get with OpenMP mode 1 is in simulation hinges4-ud-10.flea on Morar with x3.368 speed-up, it is not a efficient way of parallelisation as 16 threads should expect to have better scaling performance. Since no available input data can perfrom a single proteins molecular system simulation, the best scaling performance of this parallelisation strategy is unable to measure. However, during the optimisation section the OpenMP code that works in this mode was further analysed and explanation of low scaling performance is provided.

### 4.1.3 OpenMP Mode 2 Performance

Mode 2 shows a better situation both on the OpenMP overheads and the scaling performance. The mode 2 single thread running time do not show significantly exceeded than the serial code. Actually, the average data of single thread running time is similar to the serial code.

From Figure 4.3 it can be seen a better than mode 1 scaling performance is achieved here. Different platforms still shows slight difference that can be ignored here. The different scaling performance still comes from input files as hinges4-ud-10.flea have better speed-up than henges4-ud-30.flea on both two platforms. The best scaling achieved here is x6.349 with hinges4-ud-10.flea on ARCHER using 24 threads.

With MPI timing of each part function, the scaling of VdW_Solver::solve() is almost the same in mode 1 benchmarking result, it is reasonable as this part use the same parallel code. The bad scaling part is still the Blob::update() as it achieve highest x4.81 speed-up with hinges4-ud-10.flea on ARCHER. It is better than in mode 1 but still not get enough scaling.

CrayPat gives a reason of its limited speed-up for 20.6% on OpenMP barrier waiting. It means threads imbalance is affecting the code performance for further scaling. It is possible to improve mode 2 scaling by balanced threads assignment.

The other reason of limited scaling performance is also because not enough parallelism in the code. Benchmarking with Allinea Map gives the information that 72.5% of total execution time is in parallel while it also includes the huge overheads in it.
4.2 OpenMP Optimisation Performance

In this section, the testing of using different OpenMP schedule clauses are presented in the modified code with mode 2 on Morar.

The threads number is set with 8 to compare different clauses performance with same scaling condition. Tests are based on the most time-consuming part \texttt{VdW\_solver::solve()} and the \texttt{Blob::update()} loop with four schedule clauses, static, dynamic, runtime and guided. Within which the runtime schedule is a system auto choice from one of the other three schedule clauses. For a piece of code with unknown structure runtime is a good choice as it leaves the scheduling to compiler rather than manually set. The original setting of schedules are both runtime. However, compiler cannot ensure the best choice of scheduling so it is possible get better performance than using runtime schedule.

The tests is based on the hinges4-ud-30.flea with 8 threads. Chunk size can specify the iterations each time a thread can get, each schedule clauses are tested with 1,2,4,8,16 chunk sizes. The data is collected from 5 times repeated running and use the average here. The results show quite different execution time with different schedules clauses. Figure 4.4 shows the performance compare the best schedule choice with the dynamic chunk size 1 schedule for both \texttt{VdW\_solver::solve()} and \texttt{Blob::update()} and the original one. The new speed-up gets from the dynamic schedule increases from x4.47 to x5.1. For the \texttt{Blob::update()} function itself the speed-up increase from x3.41 to x4.85 and solve() function for x8.14 to x10.56. The scheduling do improve the
Figure 4.4: OpenMP mode 2 schedule dynamic 1 performance with hinges4-ud-30.ffe

performance to hinges4-ud-30.ffe but whether it works on different molecules systems still need further analysis.

In addition, as VdW::solve() function has not use any schedule clauses before the test, it shows more performance improvement. The tests shows runtime schedule also produce a good scaling performance that create a x10.14 speed-up to the code, which is similar to dynamic 2 schedule performance. It seems without enough information about different schedule performance the runtime schedule is the best choice as it uses the compiler choice of best schedule method and can ensure a satisfactory performance.

### 4.3 MPI Performance

The MPI parallelisation was successfully implemented in FFEA with a replicated data parallelisation strategy. However, the high overhead of array assignment limited speed-up of the var der Waal force calculation. The serial code performance becomes a slight different with the original one as the development team has kept updating new functions and this MPI implementation use a newer version of code.

The performance is measured on ARCHER with 5 times repeated running result. As table 4.6 shows very limited improvement is got from the current parallel code. The timing result shows in va der Waal calculation function the overhead increase when using
more than 4 processes. It becomes more serious in the simulation with 30 molecules input file, the execution time in the parallel part quickly increasing when use 6 processes.

<table>
<thead>
<tr>
<th>process</th>
<th>Total time</th>
<th>Speed-up</th>
<th>VdW time</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>55.468</td>
<td>1</td>
<td>33.935</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>41.664</td>
<td>1.331</td>
<td>20.651</td>
<td>1.643</td>
</tr>
<tr>
<td>4</td>
<td>37.832</td>
<td>1.466</td>
<td>15.101</td>
<td>2.247</td>
</tr>
<tr>
<td>6</td>
<td>43.336</td>
<td>1.279</td>
<td>20.565</td>
<td>1.650</td>
</tr>
<tr>
<td>8</td>
<td>44.806</td>
<td>1.237</td>
<td>18.7161</td>
<td>1.813</td>
</tr>
</tbody>
</table>

hinges4-ud-30.ffea

<table>
<thead>
<tr>
<th>process</th>
<th>Total time</th>
<th>Speed-up</th>
<th>VdW time</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>187.435</td>
<td>1</td>
<td>121.732</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>157.951</td>
<td>1.186</td>
<td>89.057</td>
<td>1.367</td>
</tr>
<tr>
<td>4</td>
<td>161.954</td>
<td>1.157</td>
<td>86.090</td>
<td>1.414</td>
</tr>
<tr>
<td>6</td>
<td>204.078</td>
<td>0.918</td>
<td>116.463</td>
<td>1.045</td>
</tr>
<tr>
<td>8</td>
<td>219.514</td>
<td>0.854</td>
<td>120.002</td>
<td>1.014</td>
</tr>
</tbody>
</table>

Table 4.6: Performance of MPI parallelisation code on ARCHER

A simple solution to reduce the overhead come from array assignment is combine the interaction calculation work on master process. The face var der Waal interaction needs most part of data in the current collective communication parts while with moving it back to master process, the only data need in other processes are face centroid and kinetic state. These two data is for linked list creation, processes then only need to create and traverse their local linked lists and send interacting face pairs index to master process. Master process will keep waiting for new interaction face pair messages until all the processes send a finish message tells it all the linked list traverses finish. Master process can get the face objects with their index from the surface lookup buffer and then do the interaction force calculation.

The face pairs need to use a global index that corresponding to the lookup surface face buffer index on master process. Faces objects does not use global index in the Face class while it has a local index in its blob. To get the global index the face structure created in this project then need to add new parameter global_index and assign its value during initialising array of structures.

Therefore, the remaining collective communications and array assignments are the centroid and kinetic state. The forces are not required on processes except master processes. By using 20 cores on ARCHER a speed-up of x12.65 was achieved. However the problem comes when some processes cannot create the linked lists. During the tests it was found that the code cannot always create the linked list on other processes. There is also a problem that some systems do not have any interaction calculation such as the hinges4-ud-10.ffea, in this case master will keep waiting a non-exist message of interacting face_pairs and the code cannot terminate. Possible reasons of this problem could be the other data that could change the centroid and kinetic state when using multi-
ple processes divided the surface array the data cannot be added. After figure out the reason why some systems do not have face interacting a solution is expected to choice whether to receive interacting messages or not on master process. After this, the expected scaling performance could increase to an acceptable range. If a master-worker pattern is introduced as master process only do the interaction force calculation with the received messages while worker processes each do a equal part of linked list traverses and sending interacting face pairs, the parallel code can perform a better balanced working process and is able to achieve higher speed-up.
Chapter 5

Conclusion

The main goal of this project was to parallel FFEA software on distributed memory system. During this project, the benchmarking was successfully implemented and collected useful profiling information about the code improvement. The OpenMP code performance was measured and some optimisation is implemented. The MPI code was successfully implemented but need further improvement to get more speed-up.

5.1 Benchmarking

The benchmarking process helps understand the code structure and find the potential bottleneck in the code. Two profiling tools provide complementary information that one can not achieve. MPI timing gives a detailed information of execution time in specific regions. The benchmarking result gives a good introduction to the code structure with which the following work become much easier. OpenMP code benchmarking helps find the shortages in the two OpenMP parallel strategies.

With the benefits of benchmarking tools, the complex FFEA system structure becomes much simpler. The benchmarking results show FFEA bottleneck is on van der Waal force calculation process, the Vdw_solver::solve() function in the source code. The traverses of linked lists in this function cause the long execution time here. Besides, blob state update function Blob::update() also occupies a smaller part of execution time in the simulation. Benchmarking also helps to find the potential issues that limit the OpenMP parallel performance.

5.2 OpenMP Optimisation

Benchmarking result shows the original OpenMP mode 1 code has a significant overhead and threads imbalances. The overheads basically come from nested OpenMP
parallel regions. The overhead of each OpenMP directive such OpenMP for loop is
enlarged to a significant level when it is placed in a long iteration loop. Due to data
dependence the parallel region in child functions cannot move to the outer loop. Di-
rectly removing the nested OpenMP regions do reduce the overhead but also disabled
part of the current parallelisation. The input data limited further analysis as the mode 1
parallelisation was designed for a single large protein while in this project the smallest
protein system contains 10 proteins blobs. The benchmarking result shows it has better
performance on a system with less proteins. The imbalance problem can be improved
with schedule clauses as the OpenMP for loop main have bad iteration assignment. The
improvement with testing different schedules is done in mode 2 optimisation.

OpenMP mode 2 has a better performance with both two input files. The benchmarking
then gives information about the thread imbalance among processes. To improve the
load balance, the tests of using different schedules and different chunk sizes gives a
better solution to use schedules. The imbalance of OpenMP threads has been founded
VdW::solve() and Blob::update() function, they are reduced to perform a better
scaling. The mode 2 optimisation has achieved scaling improvement on Morar when
simulation 10 molecules systems in Blob::update() function scaling from x3.41 to x4.85
and VdW::solve() function from x8.14 to x10.56.

5.3 MPI Implementation

MPI code was successfully implemented in FFEA with a replicated data algorithm. The
var der Waal force calculation part is equally divided and placed on each process. With
the collective communication, the data is able to transmit between master and other
processes. To solve the problem of MPI message passing with C++ objects, a struc-
ture is created and replace the object during MPI communication. It largely reduced
the communication message size than using object serialise method. The structure uses
a derived datatype to describe its memory location for MPI message passing. During
the simulation an initial array of structures is created with the system initial data, each
time step processes create new linked lists with data from master process to simulate
the molecules face interaction and calculate a local force data. When all the processes
finished the calculation in current simulation time step, the data is collected and up-
dated by MPI collective function. The collective communication and data conversion
becomes the main overhead in the current parallel code. With the testing on ARCHER
a maximum speed-up of x1.466 on 4 processes is achieved. By move the interaction
calculation to master process, the overhead overhead of assign parameters value to each
processes is reduced to a more reasonable range. With tests on ARCHER it can parallel
on 20 processes with a speed-up of x12.65 was achieved. However, this solution has
some limits as when the processes number changes the linked lists on some processes
cannot be created. This will cause the master process waiting and cannot not terminate.
There is also a situation that some proteins systems do not have any face interaction.
All of these causes the code cannot get a correct result.
5.4 Future Work

This project MPI parallelisation only focus on most time-consuming part, the var der Waal solve function. By implement more MPI parallelism to FFEA it can get a better scaling performance.

The current MPI parallelisation uses replicated data, which cause a high overhead to FFEA. In the following works it should be improved to use a data partition method with each process using a part of local data. This may requires data exchange between processes so it needs careful design of partition method that minimum message passing will be used.

During project preparation the ParMETIS is introduced as a parallel mesh decomposition tool, it allows a topological mapping of the divided meshes onto processes. However, due to the different mesh format, it cannot directly read the mesh data in FFEA. The mesh decomposition work should start with format translate to enable ParMETIS using mesh data.

OpenMP code can be further applied to FFEA to achieve more parallel parts in it, with sufficient OpenMP and good MPI parallelisation strategy a hybrid code of both MPI and OpenMP parallelisation is expected to achieve more scaling performance.
Appendix A

FFEA software package configuration on ARCHER

System compiler and third-party libraries requirement:
GCC (>=4.4 & <5.0)
Boost(<1.60)
Eigen (>=3.2.1)
Doxygen (>= 1.8)

Configuration and Building

```bash
mkdir install
cd install
module swap PrgEnv−cray Prgenv−gnu
module load boost/1.60
module load CMake
module load eigen
export BOOST_ROOT=$BOOST_DIR
export EIGEN3_HOME=/work/y07/y07/cse/eigen/3.2.1
export CRAYPE_LINK_TYPE=dynamic
CMake ../ffea −DCMAKE_INSTALL_PREFIX=../install \
−DCMAKE_CXX_COMPILER=CC \
−DCMAKE_C_COMPILER=cc −DUSE_OPENMP=0
make
make install
```

1. Serial code
−DUSE_OPENMP=0
2. OpenMP Mode 1
−DUSE_OMP_MODE=1
3. OpenMP Mode 2
   - DUSE_OMP_MODE=2
4. MPI
   - DUSE_MPI=1
Appendix B

Profiling reports

B.1 CrayPat Benchmarking Report

hinges4-ud-30.fea running FFEA with OpenMP mode 1

<table>
<thead>
<tr>
<th>Profile by Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samp%</td>
</tr>
<tr>
<td>Samp</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>100.0%</td>
</tr>
<tr>
<td>63.8%</td>
</tr>
<tr>
<td>33.7%</td>
</tr>
<tr>
<td>5.5%</td>
</tr>
<tr>
<td>3.4%</td>
</tr>
<tr>
<td>2.6%</td>
</tr>
<tr>
<td>2.6%</td>
</tr>
<tr>
<td>2.4%</td>
</tr>
<tr>
<td>1.1%</td>
</tr>
<tr>
<td>1.1%</td>
</tr>
</tbody>
</table>
B.2 Allinea Map function calls proportion report

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Time (ms)</th>
<th>Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Face::zero_force</td>
<td>117.0</td>
<td>48.1%</td>
</tr>
<tr>
<td>ETC</td>
<td>3,938.0</td>
<td>36.2%</td>
</tr>
<tr>
<td>gomp_team_barrier_wait_end</td>
<td>2,241.0</td>
<td>20.6%</td>
</tr>
<tr>
<td>GOMP_parallel</td>
<td>1,463.0</td>
<td>13.4%</td>
</tr>
<tr>
<td>libc.so.6</td>
<td>230.0</td>
<td>2.1%</td>
</tr>
<tr>
<td>==LO_MEMORY==</td>
<td>225.5</td>
<td>66.2%</td>
</tr>
<tr>
<td>Depth</td>
<td>Start</td>
<td>Total</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>0</td>
<td>0.04</td>
<td>0.06</td>
</tr>
<tr>
<td>0</td>
<td>0.06</td>
<td>0.08</td>
</tr>
<tr>
<td>0</td>
<td>0.08</td>
<td>0.10</td>
</tr>
<tr>
<td>0</td>
<td>0.10</td>
<td>0.12</td>
</tr>
<tr>
<td>0</td>
<td>0.12</td>
<td>0.14</td>
</tr>
<tr>
<td>0</td>
<td>0.14</td>
<td>0.16</td>
</tr>
<tr>
<td>0</td>
<td>0.16</td>
<td>0.18</td>
</tr>
<tr>
<td>0</td>
<td>0.18</td>
<td>0.20</td>
</tr>
<tr>
<td>0</td>
<td>0.20</td>
<td>0.22</td>
</tr>
<tr>
<td>0</td>
<td>0.22</td>
<td>0.24</td>
</tr>
<tr>
<td>0</td>
<td>0.24</td>
<td>0.26</td>
</tr>
<tr>
<td>0</td>
<td>0.26</td>
<td>0.28</td>
</tr>
<tr>
<td>0</td>
<td>0.28</td>
<td>0.30</td>
</tr>
<tr>
<td>0</td>
<td>0.30</td>
<td>0.32</td>
</tr>
<tr>
<td>0</td>
<td>0.32</td>
<td>0.34</td>
</tr>
<tr>
<td>0</td>
<td>0.34</td>
<td>0.36</td>
</tr>
<tr>
<td>0</td>
<td>0.36</td>
<td>0.38</td>
</tr>
<tr>
<td>0</td>
<td>0.38</td>
<td>0.40</td>
</tr>
<tr>
<td>0</td>
<td>0.40</td>
<td>0.42</td>
</tr>
<tr>
<td>0</td>
<td>0.42</td>
<td>0.44</td>
</tr>
<tr>
<td>0</td>
<td>0.44</td>
<td>0.46</td>
</tr>
<tr>
<td>0</td>
<td>0.46</td>
<td>0.48</td>
</tr>
<tr>
<td>0</td>
<td>0.48</td>
<td>0.50</td>
</tr>
<tr>
<td>0</td>
<td>0.50</td>
<td>0.52</td>
</tr>
<tr>
<td>0</td>
<td>0.52</td>
<td>0.54</td>
</tr>
<tr>
<td>0</td>
<td>0.54</td>
<td>0.56</td>
</tr>
<tr>
<td>0</td>
<td>0.56</td>
<td>0.58</td>
</tr>
<tr>
<td>0</td>
<td>0.58</td>
<td>0.60</td>
</tr>
<tr>
<td>0</td>
<td>0.60</td>
<td>0.62</td>
</tr>
<tr>
<td>0</td>
<td>0.62</td>
<td>0.64</td>
</tr>
<tr>
<td>0</td>
<td>0.64</td>
<td>0.66</td>
</tr>
<tr>
<td>0</td>
<td>0.66</td>
<td>0.68</td>
</tr>
<tr>
<td>0</td>
<td>0.68</td>
<td>0.70</td>
</tr>
<tr>
<td>0</td>
<td>0.70</td>
<td>0.72</td>
</tr>
<tr>
<td>0</td>
<td>0.72</td>
<td>0.74</td>
</tr>
<tr>
<td>0</td>
<td>0.74</td>
<td>0.76</td>
</tr>
<tr>
<td>0</td>
<td>0.76</td>
<td>0.78</td>
</tr>
<tr>
<td>0</td>
<td>0.78</td>
<td>0.80</td>
</tr>
<tr>
<td>0</td>
<td>0.80</td>
<td>0.82</td>
</tr>
<tr>
<td>0</td>
<td>0.82</td>
<td>0.84</td>
</tr>
<tr>
<td>0</td>
<td>0.84</td>
<td>0.86</td>
</tr>
<tr>
<td>0</td>
<td>0.86</td>
<td>0.88</td>
</tr>
<tr>
<td>0</td>
<td>0.88</td>
<td>0.90</td>
</tr>
<tr>
<td>0</td>
<td>0.90</td>
<td>0.92</td>
</tr>
<tr>
<td>0</td>
<td>0.92</td>
<td>0.94</td>
</tr>
<tr>
<td>0</td>
<td>0.94</td>
<td>0.96</td>
</tr>
<tr>
<td>0</td>
<td>0.96</td>
<td>0.98</td>
</tr>
</tbody>
</table>

43
Appendix C

Correctness Test script file and tool

test.sh
#!/bin/sh

# Compare the internal measurement files
for file in *blob*.out;
do
  echo "−−−−−−"$file"−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

  a=$file
  # the file that use to compare
  b=diff3/$file
  # get the error tolerant value
  ../../../numdiff_tool/bin/numdiff -S -q $a $b
  # test the files with tolerant value
  ../../../numdiff_tool/bin/numdiff -S -q -a 4e-11 $a $b
done

# compare the trajectory file
echo "−−−−−−"$file"−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

  a=$file
  # the file that use to compare
  b=diff3/$file
  # get the error tolerant value
  ../../../numdiff_tool/bin/numdiff -S -q $a $b
  # test the files with tolerant value
  ../../../numdiff_tool/bin/numdiff -S -q -a 9e+2 $a $b
done
NumDiff tool
Source file: http://savannah.nongnu.org/download/numdiff
Installation:
./configure --prefix=$HOME
make
make install
Bibliography


