Optimization of a large-scale parallel finite element – finite volume C++ code

Qi Huangfu

August 22, 2008

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2008
Abstract

CSMP is an important software written in C++ for simulations of fluid flow in subsurface systems. Recently, usage of hybrid element meshes and parallelization in CSMP have both shown significant performance improvement. However, the absence of a suitable hybrid element mesh partition tool limits the combination of these two features. This project has filled the gap by developing a customized CSMP hybrid mesh partition program.

The hybrid mesh partition program has been successfully implemented in this project, and tested with several CSMP parallel simulations. The simulation results show that CSMP code scales well and the hybrid mesh partition program can produce good quality partitioned meshes.
## Contents

1 Introduction ......................................................... 1
   1.1 Project Background ........................................... 1
   1.2 CSMP Parallelization Model ................................. 4
   1.3 Hybrid Element Mesh in CSMP ............................... 5
   1.4 Project Motivation and Aim ................................ 7
   1.5 Outline of this Dissertation ............................... 8

2 Porting CSMP ........................................................... 9
   2.1 CSMP Structures and SAMG ................................. 9
   2.2 Porting CSMP to Ness ....................................... 10
     2.2.1 Working environment on Ness ......................... 10
     2.2.2 Encountered Problems and Solutions ................. 11
   2.3 Porting CSMP to ECDF ..................................... 13
     2.3.1 Working environment on ECDF ......................... 13
     2.3.2 Encountered Problems and Solutions ................. 14
   2.4 Test Running .................................................. 17
   2.5 Summary ....................................................... 18

3 Hybrid Element Meshes Partition Program ......................... 19
   3.1 Analysis ...................................................... 19
     3.1.1 Existing Partitioning Programs ........................ 19
     3.1.2 Data Format ............................................... 20
     3.1.3 Existing VSet Partitioning Program ................... 23
   3.2 Design ......................................................... 23
     3.2.1 METIS graph format .................................... 23
     3.2.2 Build nodes graph from PList ......................... 24
   3.3 Implementation ............................................... 25
     3.3.1 2D implementation ..................................... 25
     3.3.2 3D implementation ..................................... 26
   3.4 Integration .................................................... 29
   3.5 Verification .................................................. 31
     3.5.1 VTK file format ........................................ 31
     3.5.2 Verification Program ................................... 32
   3.6 Summary ....................................................... 33
4 Performance Analysis 34
4.1 Encountered Problems ............................................. 34
4.1.1 Poor Scaling on ECDF ........................................ 34
4.1.2 Unstable Scaling on ECDF ................................. 35
4.2 Profiling Data ........................................................... 36
4.2.1 Model fracs2000 with one FE computation .............. 36
4.2.2 Model WellOne with many FE computation loops .......... 38
4.2.3 Model fracs2000 with both FE and FV computation ....... 41
4.3 Summary ................................................................. 44

5 Conclusion 45

A CSMP file structures and makefiles 46
B Brief Instruction for Running CSMP with SAMG 49
C Core Part of the Partition Program 51
D Partition Result Verification Program 54
E WellOne Simulation Bug Explanation 56
F Comparison of the Original Work Plan and the Final Work Process 59
# List of Tables

2.1 ECDF processors information ............................................. 14
2.2 ECDF MPI environment comparison ..................................... 15
2.3 Simulation of ten_fine.1 on Ness ......................................... 17
2.4 Simulation of ten_fine.1 on ECDF ......................................... 17
2.5 More data of ten_fine.1 simulation on ECDF ........................... 18

3.1 PList for the mesh in Figure 3.2 ........................................ 20
3.2 PList for the 3D mesh in Figure 3.4 ...................................... 21
3.3 The `ElementType` const and corresponding element .................. 27

4.1 fracs2000 with one FE computation on Ness ........................... 36
4.2 fracs2000 with one FE computation on ECDF ........................... 37
4.3 WellOne with 24 loops of FE computation on Ness .................... 40
4.4 WellOne with 24 loops of FE computation with original selection criterion on ECDF ................................................. 40
4.5 WellOne with 24 loops of FE computation with revised selection criterion on ECDF ................................................. 41
4.6 fracs2000 with FEFV computation on Ness ............................. 42
4.7 fracs2000 with FEFV computation on ECDF ............................. 43
# List of Figures

1.1 Finite element and finite volume for the same mesh .......................... 1  
1.2 A Real Complex Fractures ....................................................... 2  
1.3 Current CSMP parallelization speed up ......................................... 3  
1.4 CSMP mesh partition steps ....................................................... 6  
1.5 Photograph of a carbonate vein in Jurassic limestone at Kilve .............. 7  
1.6 Seven kinds of element types ................................................... 7  

3.1 Flow chart of the existing partitioning programs ............................... 20  
3.2 A simple 2D mesh for showing PList .......................................... 21  
3.3 Neighbor nodes relationship for all kinds of elements ........................ 22  
3.4 A simple 3D mesh for showing PList .......................................... 22  
3.5 A sample graph and its representation in METIS graph format ............... 24  
3.6 Two possible kind of mesh element in CSMP .................................. 26  
3.7 Mesh Partition Results ........................................................... 28  

4.1 fracs2000 with one FE computation parallel efficiency on Ness ............. 37  
4.2 fracs2000 with one FE computation parallel efficiency on ECDF ........... 38  
4.3 WellOne with 24 loops of FE computation parallel efficiency on Ness .... 40  
4.4 WellOne with 24 loops of FE computation parallel efficiency on ECDF .... 41  
4.5 fracs2000 with one FE computation parallel efficiency on Ness ........... 43  
4.6 fracs2000 with one FE computation parallel efficiency on ECDF ........... 44  

A.1 CSMP file structures ............................................................. 46  
F.1 Original Work Plan ............................................................... 59
Acknowledgements

I would like to thanks my supervisors Dr Fiona Reid and Dr Sebastian Geiger. Without their help, it is impossible to finish this project.

Also thanks the SAMG team who gives free libraries and licence, and ECDF team who gives lots of help on using ECDF.
Chapter 1

Introduction

1.1 Project Background

Simulations of fluid flow in subsurface systems are of high interest in variety of areas, such as hydrocarbon formations research for oil and gas production [1, 3], fractured-geothermal reservoirs simulation in energy production industry [1], nuclear waste repository safety analysis [2], formation of large ore deposits [6]. These subsurface systems share a common challenging characteristic: they all have highly heterogeneous and geometrically complex structures.

To efficiently model these types of systems, a finite element – finite volume (FEFV) method has been widely applied [1-8] recently. A finite element is a cell of the mesh, and a finite volume is a small area surrounding the nodes in the mesh. Figure 1.1 shows a simple mesh, with (a) finite elements and (b) finite volumes highlighted by solid line.

![Finite element and finite volume for the same mesh](image)

The FEFV method is a combination of the finite element method (FE method) and the finite volume method (FV method). FE method is a numerical technique for finding approximate solutions of partial differential equations. The basic idea of the FE method is that the interested variable value of one element, could be computed by the variable values of its neighbors, for example the average of the neighbor values. One typical
variable in subsurface system that could be obtained by the FE method is fluid flow pressure [7]. The FV method is a method based on the concept of flux: flux entering a volume equals to that leaving its adjacent volumes. The FV method is suitable for mass conservative simulation such as the transport of oil or water[7].

One area where FEFV method have become increasingly popular is the simulation of fluid flow transport in geometrically complex fractures. Figure 1.2 shows a real example of such system, which contains thousands of complex fractures. The number of fractures a subsurface simulation could deal with, is a key criterion to evaluate its computational capability, because only realistic representation of fractured rocks which may contain thousands of fractures, will allow us to analyze the flow and the transport behavior adequately.

Figure 1.2: A Real Complex Fractures
Courtesy of Dr. S. Matthai, Imperial College London

There has been much FEFV research focusing on different interests such as [1] which proposes a numerical procedure of finite element finite volume method, [2] which introduces a mixed-dimensional finite volume method, and [3] which involves three-dimensional three-phase fractured reservoir simulation based on the FEFV method. These research are mainly proof of the concept, and thus only a small number of fractures (about 10) are investigated. By contrast, research described in [4] and [5] involves general application level simulation models, and could deal with hundreds of fractures. Particularly, models described in [4] could deal with about 100 fractures, whereas the CSMP experimental research model mentioned in [5] could deal with up to 2000 fractures. The ability to handle large numbers of fractures makes CSMP unique in the
fractured subsurface system simulation area.

CSMP is an application programming interface (API) written in C++ that contains libraries for FEFV operations as well as for constitutive physical and chemical laws. It can work for many different simulations using the FEFV method. As an API, it can help researchers in different areas save time from programming and focus on the physical features in their research. Also, the ability to work with thousands of fractures makes CSMP capable of running potentially large scale simulations. CSMP has been successfully used by many researcher and for varied applications, such as, research of a new FEFV method to model fluid convection driven by temperature and salinity gradients [6], modeling the transport of multiphase fluids in the Earth’s crust [7], and the simulation of water injection in fractured carbonate reservoir [8]. Particularly, the water injection simulation is a real application, in which up to 100 fractures has been successfully dealt with.

CSMP has been continuously developed by many programmers for more than ten years, and in recent improvements, two new features, parallelization [9] and hybrid element meshes [10], were introduced to obtain better performance. According to the previous research on CSMP, the domain partition based parallelization has shown very good speed up with increasing number of processors. As shown in Figure 1.3, the current parallelization gets nearly linear scaling with millions of elements on up to 32 processors. Also, the usage of hybrid element meshes demonstrates the advantage of improving flexibility in capturing complex geometry as well as minimizing number of elements required in the mesh. According to the existing research [10], which solves transient thermal convection, using hybrid element with a particular mesh may achieve as many as 37% nodes reduction and 76% elements reduction.

![Figure 1.3: Current CSMP parallelization speed up](This picture is taken from [9].)
CSMP has recently been parallelized, and it can handle the hybrid meshes in serial simulation. These two features both show performance improvement. The combination of parallelization and using hybrid element meshes in CSMP has been expected to achieve even better performance. However, the absence of a suitable tool for hybrid element mesh partitioning limits further development. Solving this problem is the main target of this project.

The current parallelization model and hybrid element meshes in CSMP together with the motivation and aim of this project are detailed in the following sections.

### 1.2 CSMP Parallelization Model

It is necessary to introduce the serial FEFV method simulation in CSMP before talking about its parallelization. The essential steps of the current CSMP serial work flow described in [11] can be summarized as the following list.

- build mesh from geologic structures
  - build finite element mesh
  - create finite volume for each mesh node
- finite element computation
  - build matrix from nodes
  - solve the nodes matrix
  - interpolate finite element related variables based on the nodes result
- finite volume computation

It is worthwhile to note that the finite element computation is actually on the nodes, and one finite volume is corresponding to one mesh node, so all of the crucial computations take place on the mesh nodes. This influences the parallelization model by determining the mesh partitioning pattern. The current domain partition based parallel CSMP version work flow [9] has several different steps, which are all highlighted with bold font in the following list.

- build mesh from geologic structures
  - build finite element mesh
  - create finite volume for each mesh node
- partition mesh
  - **nodes partitioning**
  - build sub meshes based on nodes partition, each partition a mesh
  - complete sub meshes with halo elements
  - create finite volumes for each sub mesh
- finite element computation
  - build matrix from nodes
  - **solve the nodes matrix in parallel**
  - interpolate finite element related variables based on the nodes result
- **finite volume computation in parallel**
There are three main changes for the parallelization: mesh partition, solving the nodes matrix in parallel and finite volume computation in parallel, which are implemented with different methods. The nodes partition part in mesh partition is implemented by using the METIS[13] mesh partitioning program, and the following mesh building steps are implemented by CSMP. The nodes matrix solving in parallel is carried out by using SAMGp [14], a parallel matrix solver based on domain partition. The parallel finite volume computation is done by native CSMP code.

In the mesh partition step, the METIS applied for nodes partition is a set of serial programs for partitioning graphs and partitioning finite element meshes. CSMP firstly use meshes partitioning program in METIS, to obtain both mesh elements partition and nodes partition. Secondly, CSMP builds sub finite element meshes for each nodes partition, the criterion for such element is that if all nodes of an element belongs to a partition, then the nodes will be assigned to that partition. Thirdly, CSMP complete each sub mesh with its halo elements, the criterion for such halo element is that if and only if part of the nodes of an element belongs to a partition, then it will become a halo element to that partition, and therefore a halo element thus will be in at least two sub meshes. In this halo elements step, halo nodes will also be set for communication, halo nodes are the nodes in the halo elements. Finally, CSMP creates finite volumes for each node in all of the partitioned meshes. Figure 1.4 could help explain these operations.

Currently, the problem with the mesh partition is that METIS can not work with hybrid element mesh partition, which prevent CSMP from obtaining more benefit of combining hybrid element mesh and parallelization. This issue will be detailed in Section 1.3.

SAMGp used in the parallel matrix solving step is a commercial library for solving domain partition based matrix, and its corresponding serial version SAMG [15] is also used in the serial CSMP matrix solving. The commercial SAMG uses a complicated licensing system named FLEXnet to manage its usage. In this project, correctly setting up and running SAMG on the target machine is a crucial step for running CSMP and any further development.

The finite volume computation in parallel has been implemented in CSMP. This part contains no external library, and thus will not be touched during this project.

To sum up, CSMP has been parallelized with domain partition using METIS for the mesh partitioning and SAMGp for solving matrix in parallel, but the parallelization is currently limited to single element meshes.

1.3 Hybrid Element Mesh in CSMP

As discussed in the previous Sections 1.1 and 1.2, CSMP has been parallelized with domain partition, and CSMP could handle hybrid elements mesh. Both of these two features offer performance improvement, but the absence of a suitable hybrid element mesh partitioning program limits combination of these two features.
The concept “hybrid mesh” in CSMP refers to both hybrid of elements in a given dimension and hybrid of dimensions. According to previous CSMP research, hybrid elements in a given dimension is introduced to decrease total number of elements and nodes in meshes, and hybrid of dimensions provides an efficient approach to model the heterogeneous geological structure fractures. Figure 1.5 shows an example of same geological structure modeled with both single element mesh and hybrid element mesh discretization. From Figure 1.5, it is clear that the hybrid element mesh in (b) uses less elements to cover larger area than that in (a).

In CSMP, as shown in Figure 1.6, there is one kind of element in 1D, two kinds of elements in 2D and four kinds of elements in 3D. In a 2D simulation, all of these three kinds of 1D and 2D elements may be used, and in a 3D simulation, all of these seven kinds of 1D, 2D and 3D elements may be used.

However, compare to the complicated mesh element types, the requirement of CSMP meshes partition is not exactly the partition of the mesh elements. According to the introduction in Section 1.2, it could be found that in the existing CSMP parallelization model, final mesh elements partition is actually based on the nodes partition, and the
Figure 1.5: Photograph of a carbonate vein in Jurassic limestone at Kilve, same geological structure with different mesh. (a) shows a single element (triangles only) mesh, (b) shows a hybrid element (triangles and quadrilaterals) mesh. These pictures were taken and redrawn from [10].

Figure 1.6: Seven kinds of element types in CSMP used in hybrid elements meshes

To sum up, the hybrid element meshes in CSMP is too complicated to use a general mesh partition program such as METIS mesh partition program, which is different from the mesh partition required by CSMP, is never used.

To sum up, the hybrid element meshes in CSMP is too complicated to use a general mesh partition program such as METIS mesh partition program, and the special structure of the CSMP mesh partition requires a customized partition program based on mesh nodes partitioning, which will be implemented as part of this project.

1.4 Project Motivation and Aim

As mentioned in the Section 1.1, the combination of parallelization and hybrid element mesh in CSMP has been expected to produce better performance, but the absence of suitable partition tool limits the improvement.
Providing a partition program for hybrid element meshes is the main aim of this project. Besides, porting CSMP to working parallel machines and profiling the parallel simulation with partitioned hybrid element mesh are also targets of this project.

The porting task of this project consists of compiling and running exist parallel CSMP code on two parallel machines, Ness [16] and ECDF [17]. This task is a prerequisite for other tasks. The crucial step of this task is correctly setting and running SAMG and SAMGp, which are commercial matrix solver used by CSMP for serial and parallel matrix solving.

About the mesh partition program, according to the discussion in Sections 1.2 and 1.3, it could be found out that the mesh partition in CSMP is actually based on the nodes partition. Two main steps of the mesh partition program are (1) building a nodes graph from CSMP mesh, and (2) partitioning the graph. The nodes graph partition work could be done by the METIS graph partition program.

The profiling task is mainly to profile several CSMP simulations on both Ness and ECDF. This task consists of investigating the quality of the hybrid element mesh partition program, analyzing CSMP scaling features and providing further parallelization improvement suggestion based on the scaling data.

In short, the aims of this project are (1) porting CSMP to Ness and ECDF, (2) hybrid element mesh partition program design and implementation together with (3) profiling several parallel CSMP simulations.

1.5 Outline of this Dissertation

The main body of this dissertation is organized into three chapters according the three aims of this project: porting CSMP, hybrid element mesh partitioning program and performance analysis. Chapter 2 mainly talks about problems encountered in porting CSMP to Ness and ECDF. Analysis of original mesh partition code, design and implementation of the new mesh partition program are introduced in Chapter 3. Chapter 4 consists of profiling problems discussion, lots of profiling data and scaling investigation. In the Chapter 5 conclusion, the dissertation were summarized, and several future development suggestion were given.

There are also six appendixes attached besides the main body of the dissertation. Appendix A contains CSMP file and makefiles structures. Appendix B briefly introduce the porting and running CSMP steps. Appendix C consists core part of the two partition programs in 2D and 3D. A verification program for the partition program is listed in Appendix D. Appendix E contains two detailed explanation from SAMG team of the SAMG bugs encountered when profiling the WellOne simulation. Original work plan and the final work process are compared in Appendix F.
Chapter 2

Porting CSMP

The parallel machines used for this project had been prearranged before the start of the project. They are Ness and ECDF. Ness, compared to ECDF, is a simple and small parallel machine with two identical nodes of 16 processors, whereas ECDF is complicated with more than 100 heterogeneous nodes and thousands of processors. So, the working plan of the porting tasks consists of two phases, firstly porting CSMP to Ness, then to ECDF. While porting CSMP to Ness, biggest problems encountered were default compiler and makefile issues, and the porting work on ECDF was mostly dealing with SAMG license problems. All these problems and solutions are detailed in the following sections.

2.1 CSMP Structures and SAMG

Before talking about the porting work, it is necessary to introduce the main structures of CSMP. CSMP is an API written in C++, the source codes of CSMP are mainly C++ code together with several open source external libraries. There are about 300,000 lines of C++ code in the main part of CSMP. These codes were organized into different folders according to their functions.

The only part of CSMP without source code is an external commercial library named SAMG or SAMGp for parallel version. SAMG is used for solving matrix with algebraic multigrid approaches, and SAMGp is used for solving same matrix in parallel based on domain partition as mentioned in the Chapter 1.

The commercial SAMG library uses the FLEXnet Licensing system [18] to deal with the license management. Regular SAMG is very expensive, but this project is considered as academic research that may help improve the SAMG, so a free temporary license could be obtained for this project. The potential problem, however, about the free SAMG is that if there was something wrong with the library or the license, then it may take a long time to get help from SAMG team, which may delay the porting and subsequent development process.
To avoid this risk, supervisors of this project had helped obtain SAMG/SAMGp for both Ness and ECDF before the start of the project. Nevertheless, the libraries or license obtained in advance may have some compatibility problems, which is up to this project to deal with.

To conclude, CSMP consists of a large amount of source code and a commercial library. Therefore, porting CSMP to a machine includes two main tasks, compile CSMP source code and to get SAMG libraries working for CSMP.

2.2 Porting CSMP to Ness

2.2.1 Working environment on Ness

Ness is a parallel machine in EPCC used for teaching and training purposes. Here is the basic hardware, software information of Ness [16].

Hardware

Ness is a parallel machine consisting of 34 processors, each processor is a 2.6 GHz AMD Opteron (AMD64e) with 2 GB of memory. Two of the processors are used as a front-end for editing jobs and batch system queue operation, the other 32 processors are located in back-end machines and split into two groups, each has 16 processors.

The two back-end nodes can not be used together when running a task, so the maximum processors a task could use is 16. The 16 processors in each same node are connected with each other in a share memory model, which may give good scaling result to a message passing tasks.

Software

The software concerned with porting a code are mainly compilers, and because of the parallelization in CSMP is based on message passing, an MPI environment was also compulsory. On Ness, there are two available C++ compilers, pgcc and gcc. pgcc [19] is a commercial C++ compiler of PGI compilers suite, and it is the recommended and default compiler on Ness. gcc [20] is a well known and widely used mature open source C++ compiler, but it maybe not as good as the commercial compiler on some performance optimization work.

The MPI environment on Ness is MPICH2 Version 1.0.5. It has been built for the different compilers, so there are gcc based MPICH2 as well as pgcc based MPICH2, and the default MPI environment is pgcc based MPICH2.
2.2.2 Encountered Problems and Solutions

Compiler Usage Problem

The compiler usage problems and solution in porting CSMP to Ness includes three phases, (1) using default MPI environment, (2) using default MPI environment with gcc compiler and (3) changing default MPI environment to gcc based environment.

Firstly, porting tasks were carried out directly with the default MPI environment. The MPI C++ compiler, mpicxx, then gave lots of error messages complaining about undefined symbols such as:

```
"../interfaces/skmReadWrite.h", line 11: warning: omission of explicit type is nonstandard ("int" assumed)
   bool skm_C_fwrite( std::FILE* fp, const std::vector<T>& stl_ctner );
```

```
"../interfaces/skmReadWrite.h", line 11: error: qualified name is not allowed
   bool skm_C_fwrite( std::FILE* fp, const std::vector<T>& stl_ctner );
```

```
"../interfaces/skmReadWrite.h", line 11: error: expected a ")
   bool skm_C_fwrite( std::FILE* fp, const std::vector<T>& stl_ctner );
```

These error messages are because there are some compiler related specific code in CSMP, for example there are several lines for the gcc compiler, another several lines for the intel compiler, but no such part for pgcc because it had never been used for CSMP before. One example of such compiler related code is:

```
#ifdef CSP_FOR_INTEL_CPP_COMPILER // this is a switch for intel C++
#include <algorithm> // these lines are include header files
#include <exception> // for intel compilers
#include <stdexcept>
.... 10 lines omitted ....
#include <string>
#include <typeinfo>
#include <utility>
#include <vector>
#include <deque>
#endif
```

To get CSMP compiled with pgcc based mpicxx, a few analogous lines were added to fill the gap. Then the CSMP could be compiled, but still can not run with SAMG. The SAMG libraries was prepared in advance with help of supervisors before the project, and the one for Ness is built with gcc based MPI. Thus, the CSMP compiled with pgcc based MPI can not work with SAMG compiled with gcc based MPI, while asking a new free library based on pgcc from SAMG team is nearly impossible, the only solution is to change working compiler to gcc.

To change the working compiler, the following command was used to substitute mpicxx command:

```
mpicxx -cxx=gcc
```

the `-cxx=gcc` is a switch available in MPICH2 to select compiler used by `mpicxx`.

11
With this gcc based mpicxx, CSMP code can be compiled successfully without any change, but the problem encountered later is that it can not run in parallel in the back-end system. The reason is that the parallel environment in the back-end system is also based on pgcc MPI environment, and the CSMP built with gcc MPI is incompatible with it.

Finally with help from supervisors, the solution of configuring the working MPI environment is simply the command:

```
module switch mpich2/pgi mpich2/gcc
```

to change the MPI environment from default pgcc based to gcc based. Then with the gcc based MPI environment, CSMP could run on Ness smoothly with up to 16 processors.

To sum up, the compiler problem encountered on Ness is the problem of using default MPI which is incompatible with CSMP, and this problem was solved by switching the working MPI environment from pgcc to gcc.

**makefile Problem**

Unfortunately, there is no IDE on Ness, so the makefiles are the only tools to help build the CSMP. The default makefiles in the CSMP source code put compiler and building target information together, and all the makefiles are distributed across about 20 different folders. This structure makes changing compiler or optimization for CSMP hard work: all the makefiles need to be edited if different compiling or optimizing flags are used.

To solve this problem, an approach to separate the working compiler and the build target was used. In this method, all the compiler information and optimization option were put in a single global makefile, named `.config.makefile`. The other makefiles in different folders only contain the building target information locally, and an `include` reference to the `.config.makefile`. So, one possible `.config.makefile` may be:

```makefile
CFLAG1 = -O3 # optimization flag
CFLAG2 = -DAMD_64 -DNDEBUG -DCSP_WITH_SAMG_SOLVER # compiling flag
CC = mpicxx $(CFLAG1) $(CFLAG2) # combine these flags
```

Assume this file located in the root folder for example `/home/user/csmp/`, then a makefile in folder `/home/user/csmp/source_code/parallel` could be:

```makefile
#import global compiler settings
#the 'include' is reference to other files
include ../.config.makefile

#local building information
CSP_MAIN = ../main_library
METIS = ../support_libraries/metis-4.0
CSP_LIBS = ../libraries
INCFLAG = -I$(CSP_MAIN) -I$(METIS)

#compiling rules
.SUFFIXES : .cpp .o
```

12
By using this method, building CSMP with different compiler options or porting CSMP to another machine becomes much easier. This change has been considered to add to CSMP CVS reservoir. A full CSMP source file structure and several entire makefiles are included in Appendix A.

2.3 Porting CSMP to ECDF

While CSMP has been successfully ported to Ness, for porting CSMP to ECDF, some jobs like the makefile problem has been done, and some mistakes such as the incorrect compiler usage could also be avoided. But other problems were encountered in the porting progress, which are mainly SAMG license file problems.

2.3.1 Working environment on ECDF

The ECDF [17] (Edinburgh Compute and Data Facility) is a high-performance cluster of 1456 processors available to hundreds of researchers at the University of Edinburgh. Both the hardware and software environment on ECDF are much more complicated than those on Ness.

Hardware

There are 1456 processors on ECDF, each processor is a 3.00GHz Intel Xeon with 2G memory. These processors are arranged into different nodes for different purposes. Table 2.1 is the processors information.

Each of these nodes has two tcp/ip network interfaces. One is ranged in 192.168.194.0/23, used for inter-process communication, and the other is ranged in 129.215.94.0/23 for data transfer with hard disk. Besides these two networks, there is still another network named the Infiniband Network, specialized for MPI inter-process communication, this network is only available to some of the MPI implementation. Ideally, an MPI application should be built using the Infiniband Network for better performance [17].
Software

There are two compiler suites icc [21] and gcc [20], and many different MPI implementations based on both icc and gcc compiler are available on ECDF. Table 2.2 contains the MPI environment comparison[17] related to this project.

There are lots of reasonable MPI environment choices for running CSMP on ECDF, for example openmpi built with Intel enabling Infiniband network could be both reliable and efficient, while Infinipath MPI with Intel could be the most efficient one. However, it is better to choose a simple and easy one for the initial porting jobs to avoid potential risk, and therefore openmpi built with icc was chosen.

2.3.2 Encountered Problems and Solutions

The work of porting CSMP to ECDF was carried out after the porting tasks on Ness, so lots of similar problems have been solved. The problems encountered in this progress are mainly concerning the SAMG license.

FLEXnet Licensing system

To help discuss these issues, it is necessary to introduce the model of SAMG license system, i.e. FLEXnet Licensing system. Here is a part of typical license file content with signature field substituted by XXXX for the reason of confidentiality.

```
1 SERVER frontend01 00145edd0532
2 FEATURE samg FHGSCAI 1.0 01-jun-2009 2 HOSTID="INTERNET=192.168.*.* INTERNET=129.215.*.*" SIGN="XXXX XXXX"
3 FEATURE samgp FHGSCAI 1.0 01-jun-2009 64 HOSTID="INTERNET=192.168.*.* INTERNET=129.215.*.*" SIGN="XXXX XXXX"
```

Line 1 contains the information of the machine which is used to start the daemon, frontend01 is hostname of the machine, and 00145edd0532 is the hostid of the machine. Command

```
hostname
```

will return the hostname of a machine, and command
```
./lmutil lmhostid
```

<table>
<thead>
<tr>
<th>Name (hostname)</th>
<th>Type</th>
<th>CPU Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>eddie</td>
<td>Front-end for root user</td>
<td>Unknown</td>
</tr>
<tr>
<td>frontend01</td>
<td>Front-end 1 for all users</td>
<td>4 processors</td>
</tr>
<tr>
<td>frontend02</td>
<td>Front-end 1 for all users</td>
<td>4 processors</td>
</tr>
<tr>
<td>node001</td>
<td>Back-end with 4 processors</td>
<td>4 processors on each node</td>
</tr>
<tr>
<td>node128</td>
<td>Back-end with 8 processors</td>
<td>8 processors on each node</td>
</tr>
</tbody>
</table>

Table 2.1: ECDF processors information
gives hostid. hostname is a common command in any Linux system, whereas lmutil is a program provided by FLEXnet Licensing system.

Line 2 contains lots of information of using the license: 01-jun-2009 is the date it will expire, number 2 after the date field is the total number of process allowed using the serial SAMG at the same time, HOSTID="INTERNET=192.168.*.* INTERNET=129.215.*.*" is the ip address of the nodes which are allowed to use the SAMG, and SIGN="XXXX XXXX" is signature for this line.

Line 3 has the same format as Line 2. The difference is that Line 2 is for SAMG license, whereas Line 3 is for SAMGp license, so much more processes are allowed.

The SAMG license daemon server can only start from the machine with the same hostname and hostid as specified in the first line of the license file, and then the daemon will only allow a machine whose ip address is in the range of HOSTID in the second line. The hostid of first line and second line has same name, but totally different meaning. The one in the first line is some characteristic unique digits which identify a daemon machine, while the one in the second line is the allowed ip address range.

**SAMG license problems encountered on ECDF**

In this project, problems with both Line 1 and Line 2 were encountered when porting and using SAMG.

In the SAMG obtained in advance, the hostid in line 1 is 001a64066cd9 instead of 00145edd0532. This problem was identified when the program complained:

```
10:46:26 (FHGSCAI) Server started on frontend01 for: samg
10:46:26 (FHGSCAI) samgp
10:46:26 (FHGSCAI) Wrong hostid on SERVER line for license file:
10:46:26 (FHGSCAI) pet.hw.ac.ECDF.lic
10:46:26 (FHGSCAI) SERVER line says 001a64066cd9, hostid is 00145edd0532
10:46:26 (FHGSCAI) Invalid hostid on SERVER line
10:46:26 (FHGSCAI) Disabling 2 licenses from feature samg(0014 D843 7395 16F0 )
10:46:26 (FHGSCAI) Disabling 64 licenses from feature samgp(0044 F513 697D 3513 )
10:46:26 (lmgrd) FHGSCAI using TCP-port 46087
```

in the daemon startup process. This was then solved with a new license file from SAMG team.

<table>
<thead>
<tr>
<th>MPI Name and Version</th>
<th>Tcp/ip</th>
<th>Infiniband</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPICH2 1.0.6</td>
<td>Yes</td>
<td>No</td>
<td>Work with less than 32 processors</td>
</tr>
<tr>
<td>OpenMPI 1.2.5</td>
<td>Yes</td>
<td>Yes</td>
<td>most reliable on ECDF</td>
</tr>
<tr>
<td>LAM/MPI 7.1.4</td>
<td>Yes</td>
<td>No</td>
<td>most efficient on ECDF</td>
</tr>
<tr>
<td>Infinopath MPI 2.1</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: ECDF MPI environment comparison
The second problem is about the HOSTID field in the line 2. This problem is complicated because each node has two tcp/ip interfaces, i.e. two ip addresses. In the early stages, this project used easy tools to obtain ip address of a machines, either by running command

```
hostname -i
```
on a node, or simply pinging that node

```
ping nodexxx
```
Take node002 for example, the first command gives:

```
192.168.194.129
```

whereas the second gives:

```
PING node002.beowulf.cluster (192.168.194.129) 56(84) bytes of data.
64 bytes from node002.beowulf.cluster(192.168.194.129):icmp_seq=0 ttl=64 time=0.109 ms
64 bytes from node002.beowulf.cluster(192.168.194.129):icmp_seq=1 ttl=64 time=0.143 ms
--- node002.beowulf.cluster ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.109/0.126/0.143/0.017 ms, pipe 2
```

They both indicate the ip address of node002 is 192.168.194.129.

However, all of these command will only give one ip address in range 192.168.194.0/23. And this was also the one used in the Line 2 HOSTID field of SAMG license files at the beginning. When running the simulation with SAMG, it complained that it had an invalid hostid. The error message from the simulation is:

```
Rank 0: calling SAMGP now........
*** ERROR: samg_main: error occured at position pos 2
*** ERROR: samg_main: licence error - licences expired or too many procs
```

and the SAMG license server also report:

```
14:46:48 (FHGSCAI) DENIED: "samg" s0789844@node081 (Invalid host. (-9,333))
```

With the confusing hostid issues mentioned in the FLEXnet Licensing system section, it takes quite a lot time to find out that the invalid hostid refers to wrong ip addresses in the second line. Then, with complete ip information from ECDF support service, this problem was resolved by obtaining another new license file from SAMG team.

**Other Problems**

Besides the SAMG license problem, several other problems including incompatible SAMG library, compiler usage were also encountered, they are similar to the problem experienced on Ness, so they are not not discussed in detail here.
2.4 Test Running

To verify the success of the porting work, a parallel test case was given by the CSMP team. It is a simulation of a steady-state fluid pressure distribution \(^1\), which involves one FE method computation. This test case comes with a mesh named `ten_fine.1`. The `ten_fine.1` is a 2D triangular mesh with 505915 nodes and 1009632 elements.

The simulation involves input, finite element computation and output stages. The finite element computation is the one of highest interest in which SAMGp was used to solve matrix in parallel. So the timing work is focused on the finite element computation and the SAMGp usage.

On Ness, the timing results of `ten_fine.1` are shown in Table 2.3. The FE time is the total time related to finite element computation, the SAMGp time is a part of FE time when SAMGp was used to solve matrix in parallel, and the CSMP time is another part of FE time besides SAMGp calling, running time of CSMP native code.

<table>
<thead>
<tr>
<th>Processors</th>
<th>FE time</th>
<th>SAMGp time</th>
<th>CSMP time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>21.7318</td>
<td>15.896</td>
<td>5.8358</td>
</tr>
<tr>
<td>4</td>
<td>13.4526</td>
<td>10.457</td>
<td>2.9956</td>
</tr>
<tr>
<td>8</td>
<td>6.50431</td>
<td>5.020</td>
<td>1.48431</td>
</tr>
<tr>
<td>16</td>
<td>4.43186</td>
<td>3.609</td>
<td>0.82286</td>
</tr>
</tbody>
</table>

Table 2.3: Simulation of `ten_fine.1` on Ness

This simulation is just used for testing the parallel runs. So the serial simulation was not given. From the data, it could be found that the scaling behavior of CSMP native code is better than SAMGp. And the drawback of SAMGp leads to the total FE computation scaling problem.

<table>
<thead>
<tr>
<th>Processors</th>
<th>FE time</th>
<th>SAMGp time</th>
<th>CSMP time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>72.5844</td>
<td>8.312</td>
<td>64.2724</td>
</tr>
<tr>
<td>4</td>
<td>69.2334</td>
<td>6.469</td>
<td>62.7644</td>
</tr>
<tr>
<td>8</td>
<td>64.7601</td>
<td>3.281</td>
<td>61.4791</td>
</tr>
<tr>
<td>16</td>
<td>62.5307</td>
<td>1.875</td>
<td>60.6557</td>
</tr>
<tr>
<td>32</td>
<td>64.7955</td>
<td>1.422</td>
<td>63.3735</td>
</tr>
<tr>
<td>64</td>
<td>69.0423</td>
<td>5.516</td>
<td>63.5263</td>
</tr>
</tbody>
</table>

Table 2.4: Simulation of `ten_fine.1` on ECDF

For ECDF the data is in Table 2.4. The scaling behavior on ECDF with more than 16 processors is quite unstable, so the best data was selected here. It seems that SAMGp part scales much better than CSMP native code, but which is not, and actually CSMP uses 60 more seconds to check out SAMGp licenses, this issue will be discussed in Chapter 4.

\(^1\)Details of the physical features of this simulation can be found in CSMP user’s guide [12]

17
The data above was selected, the actual scaling behavior is quite unstable. Table 2.5 contains more details of scaling data, each number of processors was profiled 5 times. The scaling behavior for 2, 4 and 16 processors is good, but not for 8, 32 and 64 processors.

<table>
<thead>
<tr>
<th>Run</th>
<th>Processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>72.5844</td>
<td>69.2098</td>
<td>64.743</td>
<td>62.5307</td>
<td>64.7896</td>
<td>69.0423</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>72.616</td>
<td>69.2334</td>
<td>64.7601</td>
<td>62.5855</td>
<td>64.7955</td>
<td>69.3335</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>72.6548</td>
<td>69.2417</td>
<td>64.7619</td>
<td>62.739</td>
<td><strong>67.7809</strong></td>
<td>69.5325</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>72.6637</td>
<td>69.3438</td>
<td>64.861</td>
<td>62.739</td>
<td><strong>67.8518</strong></td>
<td>69.685</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>72.7163</td>
<td>69.5246</td>
<td><strong>66.276</strong></td>
<td>62.8678</td>
<td><strong>67.8861</strong></td>
<td><strong>75.3088</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: More data of ten_fine.1 simulation on ECDF

### 2.5 Summary

This chapter discussed the CSMP porting work and presented some initial scaling data. Firstly the structure CSMP and SAMG was introduced, and then for each machine, the working environment and main problems encountered in the porting process were detailed. Three major porting problems discussed are (1) compiler usage problem solved by switching the MPI environment, (2) makefile issues solved by working out a makefile suite with separated compiler and building target information, (3) SAMG license file problem solved with help from both ECDF support group and SAMG team.

The next Section will talk about the implementation of the hybrid element mesh partition program.

---

2The timing data seems have little difference, but recall that there is a 60 seconds for checking out SAMGp license.
Chapter 3

Hybrid Element Meshes Partition Program

This Chapter describes the programming work of the hybrid element mesh partitioning program for CSMP. It starts from analyzing the existing single element meshes partitioning code in CSMP, sorting out the data format and locating any reusable code. Then, based on the existing code analysis, the design and implementation of the new partitioning program are introduced. This is followed by the discussion of the integration work of all the partitioning related programs. Finally the partitioning program verification work is also discussed.

3.1 Analysis

3.1.1 Existing Partitioning Programs

Currently, there are two programs in CSMP used for the single element mesh partitioning. One program mesh2vset is used to convert various kinds of mesh to a uniform format named VSet, and then to save the VSet data to an external file for example, mesh.vset. The other program vsetpartition reads in the VSet file mesh.vset, partitions the mesh in the VSet, works out partitioned VSet and then saves the partitioned VSet to a set of files:

mesh_1.vset, mesh_2.vset ... mesh_n.vset

one file for each processor. These partitioned VSet files will be the input files for the parallel simulation. The flow chart in Figure 3.1 shows these relationships.

The mesh2vset program was implemented before this project. It can deal with all kinds of mesh, and therefore is not the key point of this project. This project concentrates on the second part of the flow chart in Figure 3.1: VSet partition program. The remaining parts of this section talk about the existing VSet date format and details of existing partitioning program.
3.1.2 Data Format

VSet data type in CSMP is used to represent the mesh. A VSet object contains nodes position, element information, element relationship, boundary nodes information. A VSet data file is the external storage format of an instance of the VSet object in the program, these two can be converted to each other with existing programs, so they are treated as the same in this project. The VSet object related source file, which consists of more than 2000 lines of C++ code, is too big to analyze completely. In the complex structures of VSet, the most important data member for mesh partitioning is PList, which contains node information for each element in the mesh, and it is the only data member used in the original single element mesh partitioning program.

The PList is a two dimensional array which is implemented by nesting a deque\(^1\) and a vector of int in C++:

\[
\text{std::deque<\text{std::vector<int}> }> \text{plist;}
\]

The first dimension index of PList denotes the ID of a mesh element, the second dimension index denotes the node position of an element, and the content in the PList is the node ID. Figure 3.2 shows a simple 2 elements mesh with 5 nodes.

A possible corresponding PList is

<table>
<thead>
<tr>
<th></th>
<th>Position 0</th>
<th>Position 1</th>
<th>Position 2</th>
<th>Position 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element 0</td>
<td>Node 0</td>
<td>Node 1</td>
<td>Node 2</td>
<td>N/A</td>
</tr>
<tr>
<td>Element 1</td>
<td>Node 1</td>
<td>Node 3</td>
<td>Node 4</td>
<td>Node 2</td>
</tr>
</tbody>
</table>

Table 3.1: PList for the mesh in Figure 3.2

The sequence of the nodes in element 0 is not important, because there are only three nodes, to any node, the other two are always neighbors. However, in element 1, sequence 1, 3, 4, 2 is not the same as 1, 2, 3, 4, the correct sequence is a cyclical neighbor

---

\(^1\)In C++, deque is a vector that support quick backwards access.
relationship. In three dimension meshes, the node sequence becomes more important and complicated, but they have a fixed style for each kind of elements. For example in a hexahedron element, nodes in Position 0 and Position 1, 3, 4 are neighbors, while nodes in Position 0 and Position 2, 5, 6, 7 are not. The neighbor nodes relationship for all kinds of elements related to this project is shown in Figure 3.3.

Therefore, for example, to store the mesh containing 2 elements with 9 nodes in Figure 3.4, a possible PList could be

<table>
<thead>
<tr>
<th></th>
<th>Pos 0</th>
<th>Pos 1</th>
<th>Pos 2</th>
<th>Pos 3</th>
<th>Pos 4</th>
<th>Pos 5</th>
<th>Pos 6</th>
<th>Pos 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elemt1</td>
<td>Node13</td>
<td>Node15</td>
<td>Node10</td>
<td>Node54</td>
<td>Node91</td>
<td>Node72</td>
<td>Node34</td>
<td>Node67</td>
</tr>
<tr>
<td>Elemt2</td>
<td>Node72</td>
<td>Node15</td>
<td>Node10</td>
<td>Node34</td>
<td>Node84</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3.2: PList for the 3D mesh in Figure 3.4

The work of building the nodes graph based on the PList will be detailed in the design and implementation part.
Figure 3.3: Neighbor nodes relationship for all kinds of elements

Figure 3.4: A simple 3D mesh for showing PList
3.1.3 Existing VSet Partitioning Program

In the existing code of CSMP, the VSet partitioning program `vsetpartition` is the most important one to this project, because the main aim of this project is to enable hybrid mesh partitioning in the VSet partitioning program.

The work flow of the current VSet partitioning program is mainly based on the concept of the FEFV method, which has been mentioned in Chapter 1. The key point of the FEFV method in CSMP is that all of the crucial computations and potential communications are carried out on nodes, so the mesh partition is actually a partition of nodes. Based on these concepts, the work flow of the current VSet partitioning program `vsetpartition` is:

- 1. Build a METIS format mesh based on the PList
- 2. Partition the mesh with METIS routines, get element and node partitions
- 3. Build finite element partitions based on node partitions
- 4. Complete finite element partitions with halo elements
- 5. Set outer halo nodes for each partition
- 6. Get VSet partitions based on finite element partitions and halo nodes

For the hybrid elements mesh partitioning program, concerning the six steps talked above, step 1 needs to be changed to build a nodes graph in METIS format, step 2 will be substituted with the METIS graph partitioning routine, steps 3, 4, 5 and 6 are all reusable with minor changes. From this point of view, the design and implementation work will be mainly building the METIS format graph based on PList, and this work is detailed in the Section 3.2 and 3.3.

3.2 Design

In the Chapter 1, the FEFV mesh partitioning was introduced. FEFV mesh partitioning consists of firstly partitioning the nodes, and then building finite element partitioning and setting halo nodes based on the nodes partitioning. Therefore, with the detailed analysis work talked described Section 3.1, the programming tasks of this project are: to build a METIS format nodes graph based on PList and then to partition it with the METIS graph partitioning routine. Other partitioning related programming work such as partitioning the elements based on the node partitioning have been done before and could be reused. The following section firstly introduces the METIS graph format, and then talks about design work on building such a graph from PList.

3.2.1 METIS graph format

The METIS graph format is basically a nodes adjacency relationship. Take Figure 3.5 for example, it is a graph with 7 nodes, the related nodes adjacency relationship is listed on the right side.
To store this adjacency relationship effectively, a compressed storage format named compressed sparse row (CSR) was used. CSR format includes two arrays, one named Data which stores the nodes relationship in sequence, and the other one named Index which stores the start and end index of the neighbor nodes in the first array for each node. With this format, the graph in Figure 3.5 is:

```
Data = 1 2 4 0 2 3 0 1 3 4 1 2 5 6 0 2 5 3 4 6 3 5
Index = 0 3 6 10 14 17 19 21
```

### 3.2.2 Build nodes graph from PList

Based on the format of the PList and METIS graph described above, the workflow of building nodes graph from PList is straightforward and involves the following steps.

1. Build an empty set $S$ for each node $N$ to store the adjacency nodes.

2. Loop over all the elements in PList, for each node $N$ in a element, add its neighbor nodes into set $S_N$.

3. Copy all the $S$ sets in sequence into the Data array, and build the Index array at the same time.

In step 1, the set is used to avoid duplicating neighbors that may be introduced by different elements. Step 3 is straightforward and simply involves copying data from one space to another. Step 2 is the hardest one because it involves dealing with different types of elements, and each kind of element will have a different node sequence style mentioned above. These are detailed in the implementation part in Section 3.3.
3.3 Implementation

3.3.1 2D implementation

Firstly the hybrid element mesh partitioning program was implemented in 2D.

In a 2D mesh, there are three possible mesh elements: line, triangular and quadrilateral, respectively with 2, 3, 4 nodes per element. Particularly, these nodes in the element are stored in a circular shape. For example, in a quadrilateral mesh element, if the four nodes in PList are a, b, c and d, then the neighbor relationship is:

a: d, b  
b: a, c  
c: b, d  
d: c, a

The first 2D implementation uses this feature to help build the nodes graph.

prepare a neighbor relationship set $S$ for each node $N$
for each elements $E$ in PList
for each No. i node $N$ in element $E$
add Node No. i-1 and No. i+1 nodes to set $S_N$

Suppose there are $M$ nodes in a 2D mesh element, it could be 2, 3 or 4. For the No. i node in the $M$ nodes, if its neighbor i-1 is smaller than 0, then it will be set to the last one No. M-1, or if i+1 is bigger than $M$, then it will be set to No. 0.

The set $S$ was implements with C++ standard container set, and all the sets $S$ for different nodes are put in a vector by the sequence of the nodes.

```
std::vector<std::set<int> > node_relation_sets;
```

The benefit of using the set data type, is that it could avoid inserting duplicated nodes.

With node_relation_sets, work flow about building the METIS format graph is:

prepare two empty vectors Data and Index
add the current size of Data (=0) to the vector Index
for each node neighbor relation set $S$ in the vector node_relation_sets
add all the nodes in S to the vector Data
add the current size of Data to the vector Index

In C++ the Data and Index vector can be implemented with standard container vector, that is:
The benefit of using the vector data type here, is that it could grows automatically when inserting new element. Both the C++ container set and vector usage in this program saves much coding effort.

### 3.3.2 3D implementation

To convert the 3D meshes into METIS graphs is more complicated as talked in the Section 3.1. Compare to 2D meshes, there is no simple circular neighbor relationship in the PList, but for each kind of element, the number of nodes and the sequence of the nodes is fixed as shown in Figure 3.3. Also in VSet there is another vector ElementType which contains the element type for each element in the PList. Using these two, it is relatively straightforward to create 3D implementation of the graph building program.

To deal with the complicated 3D mesh neighbor relationship for up to seven kinds of elements, this project uses an approach that hard codes all the nodes neighbor relationship for each kind of element. Take the Quadrilateral mesh elements in Figure 3.6 for example,

![Figure 3.6: Two possible kind of mesh element in CSMP](image)

In the quadrilateral, the neighbor nodes for node 0 are node 1 and node 3, for node 1 they are node 0 and node 2, for node 2 they are node 1 and node 3, and for node 3 they are node 0 and node 2. These relationships could be expressed as number pairs, where the first item is source node, and the second item of a pair is the neighbor node of the first one, so for the Quadrilateral element, the nodes and neighbor nodes pairs are 0:1, 0:3, 1:0, 1:2, 2:1, 2:3, 3:0 and 3:2. Similarly, for a Prism element, the number pairs are: 0:1, 0:2, 0:3, 1:0, 1:2, 1:4, 2:0, 2:1, 2:5, 3:0, 3:4, 3:5, 4:1, 4:3, 4:5, 5:2, 5:3 and 5:4.

To store these number pairs, a two dimensional array is a good choice. In the implementation of this project, all the pairs to different element types are stored in the same array one after the other, with a second array containing index to access these pairs. This idea is similar to CSR used in METIS graph storage mentioned in Section 3.2.1. Therefore, the neighbor pairs and its index are:

```c++
int neighbor_pairs[][2] = {
  0, 1, 1, 0, // line, 2 pairs
  // other kinds of elements
};
```
In the VSet there is an array named ElementType which contains element type information for each element in the PList. An ElementType is a const int value with a name, the corresponding relationship between ElementType name and the mesh element type is shown in Table 3.3.

<table>
<thead>
<tr>
<th>ElementType Name</th>
<th>element</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISOPARAMETRIC_LINEAR_BAR</td>
<td>line</td>
</tr>
<tr>
<td>ISOPARAMETRIC_LINEAR_TRIANGLE</td>
<td>triangle</td>
</tr>
<tr>
<td>ISOPARAMETRIC_LINEAR_QUADRILATERAL</td>
<td>quadrilateral</td>
</tr>
<tr>
<td>ISOPARAMETRIC_LINEAR_TETRAHEDRON</td>
<td>tetrahedron</td>
</tr>
<tr>
<td>ISOPARAMETRIC_LINEAR_PRISM</td>
<td>prism</td>
</tr>
<tr>
<td>ISOPARAMETRIC_LINEAR_PYRAMID</td>
<td>pyramid</td>
</tr>
<tr>
<td>ISOPARAMETRIC_LINEAR_HEXAHEDRON</td>
<td>hexahedron</td>
</tr>
</tbody>
</table>

Table 3.3: The ElementType const and corresponding element

To match an ElementType to an index in neighbor_pair_index, this project uses the switch / case selection method:

```c
switch(vset.ElementType(index_element))
{
   // 1D element
   case ISOPARAMETRIC_LINEAR_BAR: index_table=0; break;
   // 2D elements
   case ISOPARAMETRIC_LINEAR_TRIANGLE: index_table=1; break;
   case ISOPARAMETRIC_LINEAR_QUADRILATERAL: index_table=2; break;
   // 3D elements
   case ISOPARAMETRIC_LINEAR_TETRAHEDRON: index_table=3; break;
   case ISOPARAMETER_LINEAR_PRISM: index_table=4; break;
   case ISOPARAMETER_LINEAR_PYRAMID: index_table=5; break;
   case ISOPARAMETER_LINEAR_HEXAHEDRON: index_table=6; break;
}
```

With all of these hard coding data mentioned above, the work flow of building METIS format graph based on the PList is:

prepare a neighbor relationship set $S$ for each node $N$

for each elements $E$ in $PList$

get ElementType $T$ for $E$ and index for $T$ in neighbor_pair_index

get the index range $R$ from neighbor_pair_index for neighbor_pairs

for all the node neighbor pairs $P(P_1, P_2)$ in the range $R$ of neighbor_pairs

add node at position $P_2$ to $S_N$ of node $N$ at position $P_1$
The other steps about element partitioning are the same as the 2D implementation. After the 3D implementation, it is found that 2D situation is totally covered in the 3D hybrid element meshes partitioning program. So, the 2D one was then expected to be replaced by the 3D implementation in this project. However, in the 2D regular i.e. triangular mesh, ElementType is not defined, because it does not need the ElementType. One possible solution is change the existing CSMP code to enable ElementType in triangular mesh for using the 3D implementation, the other one is keeping the 2D implementation. In this project the the latter one was selected, because this solution involves less changes to the existing code, and it is simpler than the 3D implementation. The core parts of both 2D and 3D implementations are included as Appendix C.

Figure 3.7 shows the 2, 4 and 8 processors partitioning results to the hybrid element mesh fracs2000, which is used for most of the performance work discussed in Chapter 4.

![Mesh Partition Results](image)

Figure 3.7: Mesh Partition Results
(a) is the original mesh, the dimension of the mesh is 1000m x 1000m x 200m.
(b), (c) and (d) are 2, 4 and 8 parts partitioning of the mesh.
3.4 Integration

The integration work of this project is initially combination of the mesh2vset and the new VSet partition program. This combination could produce partitioned VSet files directly from mesh, and save hard disk space from avoiding outputting the serial VSet file. Although this task essentially involves copying and pasting work, it is important to ensure the result partitioned VSet are still the same.

In addition to the combining work, during this project, it is found that the mesh partitioning work is a very heavy task for a front end machine, because it may take more than 2G memory, 100% CPU usage for up to 10 minutes or even more. So, it worth to adjusting the program to enable it runs in a back end machine smoothly.

The existing partitioning programs mesh2vset and vsetpartition were designed as interactive programs which take commands and arguments from screen when they are running. Take the mesh2vset program for example, (the bold font represents sample input from user.)

```
$ mesh2vset
Please enter the full name of the finite element mesh
ten.1

Please enter the full name for the output VSet file to which the mesh will be saved
(excluding *.vset extension)
test

Please enter the full name of the CSMP physical variables file
(excluding *.txt extension)
parallel_example1

Is the input finite element mesh in 3D (n = 2D)? [y/n] n

Do you want to configure your model using a *-configuration file? [y/n] n
```

This is hard to use on a back end machine. To fill the gap, the command line option is used in this project to substitute the interactive ones. So the above interactive choice in the integrated 2d program mesh2vsetp2d will become:

```
mesh2vsetp2d -i ten.1 -o test -t 1 -v parallel_example1.txt -p 16
```

in which, i, o, v and p refers to input file, output file, variable file and number of partitions respectively. Option t is used to select mesh types. There are three mesh types in 2D, hybrid element mesh for t=1, triangular mesh for t=2, triangular mesh with 1D line element for t=3. In 3D meshes, the hybrid element mesh type is the only type currently, so there is no option t in the 3D integrated program. Also there is one more option c, the appearance of which indicates that there is a *-configuration file.

If the command mesh2vsetp2d is issued with no command line arguments, or insufficient command line arguments, then it will output an error message to help users to complete the command, for example, if there is no command argument, then the output will be:

```
29
```
Please indicate input file by ‘-i input_filename’
Please indicate output file by ‘-o output_filename’
Please indicate physical variables by ‘-v variable_filename’
Please indicate mesh type file by ‘-t 1/2/3’
  1 for ICEM Mesh (hybrid)
  2 for Triangular Mesh
  3 for Triangular Mesh with 1D line element
Has Configuration? : 0
Please indicate required partitions by ‘-p NUMBER’

whereas if the partition number option \texttt{p} is missing, for example, the command:

\texttt{./mesh2vsetp2d -i ten.1 -o test -v parallel_example1.txt -t 1}

will output:

<table>
<thead>
<tr>
<th>Input Mesh File</th>
<th>ten.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Parallel File</td>
<td>test</td>
</tr>
<tr>
<td>Variable File</td>
<td>parallel_example1.txt</td>
</tr>
<tr>
<td>Input Mesh Type</td>
<td>ICEM Mesh (hybrid)</td>
</tr>
<tr>
<td>Has Configuration?</td>
<td>0</td>
</tr>
<tr>
<td>Please indicate required partitions by ‘-p NUMBER’</td>
<td></td>
</tr>
</tbody>
</table>

with other argument output for double checking, and the instruction of the \texttt{p} option.

If the command is correctly executed with appropriate arguments, e.g:

\texttt{./mesh2vsetp2d -i ten.1 -o test -v parallel_example1.txt -t 1 -p 16}

then the output will be:

<table>
<thead>
<tr>
<th>Input Mesh File</th>
<th>ten.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Parallel File</td>
<td>test</td>
</tr>
<tr>
<td>Variable File</td>
<td>parallel_example1.txt</td>
</tr>
<tr>
<td>Input Mesh Type</td>
<td>ICEM Mesh (hybrid)</td>
</tr>
<tr>
<td>Has Configuration?</td>
<td>0</td>
</tr>
<tr>
<td>Partitions</td>
<td>16</td>
</tr>
<tr>
<td>You can now add ‘-r’ to issue a real run.</td>
<td></td>
</tr>
</tbody>
</table>

which tell us that the command line usage is correct, and indicates that we should add a \texttt{-r} to this command will start a real running.

Once all of this information is correct for the user, then just copying the command, adding the \texttt{-r} option, and then issuing it in the back end machine will finish the partitioning. That is:

\texttt{./mesh2vsetp2d -i ten.1 -o test -v parallel_example1.txt -t 1 -p 16 -r}

This user interface design is based on the idea of combining both the benefit of a interactive program and a command line program. In this integrated program, the interactive method in the existing CSMP program was kept in another form, and the command line usage makes the program easier to run as a background task.
3.5 Verification

To make sure the serial and parallel (i.e. partitioned) simulations give the same result, a program for verification the result was implemented. The verification work is based on the CSMP simulation result, which is in VTK [22] format. This section starts with an introduction to the VTK file format used in CSMP, and then details the verification program in python.

3.5.1 VTK file format

VTK is used to represent the result variable on mesh nodes or mesh elements, such as the pressure, velocity and flux. Some of these variables are scalar variables such as pressure, while the others are vector variables such as velocity. In a VTK file, there are typically several parts:

1. The first part is the file version and identifier. This part contains the single line:
   
   # vtk DataFile Version x.x
   
   In the VTK file generated by CSMP, this line is:
   
   # vtk DataFile Version 2.0

2. The second part is the header. The header is a string terminated by character \n. In VTK from CSMP, an example of this line is:
   
   Finite-element dataset (CSP3D5.0): ...

3. The next part is the file format. The file format could be either ASCII or binary. CSMP uses ASCII format.

4. The fourth part is the mesh structure. In CSMP the mesh structures include three parts: nodes coordinates, elements consists of nodes and element types. An example to describe CSMP mesh in VTK format is:

```
# The '#' notation is used for comment, and all the comments are added by hand
DATASET UNSTRUCTURED_GRID
POINTS 223705 float  # 223705 nodes, each by 3 float numbers
696.428 113.117 382.404  # node 0 at (696.428, 113.117, 382.404)
695.516 112.758 385.7  # node 1
685.943 113.801 383.985 # node 2
973.976 63.8565 554.947 # node 3
... ...
CELLS 1012383 5278558  # 1012383 elements by 5278558 numbers
3 0 1 2  # element 0 by 3 nodes (0, 1, 2)
3 3 4 5  # element 1
3 6 7 8  # element 2
3 9 10 11 # element 3
... ...
CELL_TYPES 1012383  # 1012383 element types, each by a number
5  # element 1 is a triangular element
5
5
... ...
```
5. The final part is the nodes variables. In CSMP, the variables could be scalar variable or vector variable. An example is:

```
POINT_DATA 223705  # variables for 223705 nodes
SCALARS fluid_pressure float  # scalar, float data
LOOKUP_TABLE default
3.082962e+06 3.082879e+06 3.135370e+06 1.904158e+05
1.690411e+05 1.984888e+05 2.178235e+05
2.163964e+05 9.640473e+05 1.108601e+06 1.080251e+06
8.889974e+05 8.252742e+05 5.80569e+06
5.754034e+06 5.598953e+06 5.715960e+06
5.744869e+06 5.686571e+06
... ...
```

3.5.2 Verification Program

Based on the VTK file format introduced in Section 3.5.1, the idea of the verification program is straightforward: compare variable value of node one by one. The more detailed description of this idea is: firstly use dictionary $D$ to record the serial node coordinate – value relation, in which the key is node coordinate, the value is node variable value; then loop over all of the parallel VTK files and check node value in parallel files with the value in dictionary $D$.

The dictionary data type is also known as associative memory. Dictionaries consist of some key – value pairs. All keys in a dictionary are unique, and values can be accessed by key. The advantage of dictionary is the ability of quick access to the data. A typical dictionary is:

```
tel = {'Alice': 4098, 'Bob': 4139}
```

and a dictionary for CSMP serial VTK result looks like:

```
pv = {
    (696.428, 113.117, 382.404) : 3.082962e+06,
    (695.516, 112.758, 385.7) : 3.082879e+06,
    (685.943, 113.801, 383.985) : 3.135370e+06,
    ...
}
```

The key in dictionary $pv$ is the coordinate of points, and the value is scalar pressure variable. These data were taken from lists in Section 3.5.1.

To implement the verification, this project use python. The reasons include: the VTK file in CSMP is in ASCII format, python is good at ASCII string operation, python has good object oriented structures such as the dictionary, and using python is relatively straightforward and flexible.

The entire python verification program including comments is listed in Appendix D.

By using this program on the simulations carried out in this project, it was found that the difference between the serial result and the parallel result is really tiny (one or two number out of hundreds of thousands numbers), mainly rounding error, and thus the correctness of the partitioning program is verified.
3.6 Summary

This chapter focuses on the hybrid element mesh partitioning program. Five steps of the work on the partition program are discussed: analysis of the existing mesh partitioning code, design of the node graph building algorithm, implementation of the node graph building code for 2D and 3D meshes respectively, integration work for combining all the partitioning related programs, and the verification program to ensure the correctness of the partition program relative to the serial version.

The next chapter details the profiling work of several parallelized hybrid element mesh simulations based on the partition program introduced in this chapter.
Chapter 4

Performance Analysis

This chapter includes discussion of encountered problems in profiling, profiling data and analysis of three different simulations on both Ness and ECDF.

4.1 Encountered Problems

During the profiling stage, several problems were encountered. Such as poor scaling of small simulations on ECDF, unstable scaling on ECDF, and failure of using SAMGp with certain simulations.

The scaling problems observed on ECDF are common to all of the CSMP simulations on ECDF, so they are detailed in this section. The failure of using SAMGp happens when running the simulation WellOne, which will be detailed in the Section 4.2.2.

4.1.1 Poor Scaling on ECDF

As mentioned in Section 2.4, the scaling behavior shown in Table 2.4 is quite poor on ECDF, the simulation time of model ten_fine.1 on 1 processor up to 64 processors are all about 70 seconds. While the SAMGp reported that it uses less than 10 seconds, it seems that it is the problem of CSMP code, that leads to this poor scaling issues. However, CSMP native code scales much better on Ness as shown in Table 2.3, so it may not be the CSMP code.

The source of this problem was identified when we examined at the SAMGp license server log information. Take this piece of ECDF SAMGp usage information for example,

15:18:28 (lmgrd) FHGSCAI using TCP-port 57922
21:18:28 (lmgrd) TIMESTAMP 8/5/2008
21:31:28 (FHGSCAI) OUT: "samg" s0789844@node069
21:31:59 (FHGSCAI) OUT: "samgp" s0789844@node069 (64 licenses)
SAMGp license usage involves two steps to start, firstly check out 1 SAMG license, and then check out n SAMGp licenses where n is the number of processors to be used. It is clear that on ECDF, the SAMG checking out phase always takes 30 seconds. Also, it is reasonable to conjecture that SAMGp checking out takes another 30 seconds. Together they take 1 minute to initialize the SAMGp for usage. For the 1 processor task, it takes 30 seconds to initialize the SAMG for usage, because there is no SAMGp with 1 processor task.

Further data that could confirm the conjecture is that, if we run the same FE method computation twice, the second time running should use about 60 seconds less time, as the license already have been checked out.

To the ten_fine.1 simulation, this result is not satisfied. However, the real simulations involve much more FE computation loops or FV computation, which takes much bigger portion of the total time. Some FV computations of serial simulations in CSMP even takes about 12 hours to finish a loop, and thus the 1 minute checking out time is reasonable to be ignored in such case.

4.1.2 Unstable Scaling on ECDF

As mentioned in Section 2.4, the scaling behavior shown in Table 2.5 is quite unstable. This issue is considered as unsuitable usage of ECDF back end nodes, and therefore asked ECDF team for help and explanation.

The response is that there are two options for the qsub command to get a better node configuration.

One is to use openmpi-smp to substitute openmpi which ensures the simulation only uses the nodes with 4 processors, and uses all of the 4 processors. The other option for qsub is -q ecdf@infinihosts which tells the scheduler to allocate the job to the nodes which run parallel jobs only.

So using both -pe openmpi-smp and -q ecdf@infinihosts were expected to provide the best profiling behavior. However, the latter qsub option -q ecdf@infinihosts make the jobs in the queue wait too much long (e.g. more than 24 hours for a 64 processors job) to run. What's more as the name of this option "infinihosts" shows, this option is suitable for the jobs using the Infiniband network, which is not used for this project. Therefore, only the option -pe
openmpi-smp was used in the following simulations.

4.2 Profiling Data

4.2.1 Model fracs2000 with one FE computation

The model fracs2000 involves solving steady-state fluid pressure, on the mesh fracs2000, with 1 FE computation. The fracs2000 model is a 3D mesh, with 223705 nodes and 1012383 hybrid elements.

For the simulation on ECDF, as discussed in Section 4.1.1, where it takes 1 minute to check out both SAMG/SAMGp license, the overall scaling will be very poor, because there is only one FE computation loop. To investigate the real computational ability of the CSMP and the quality of the partition program in this project, one method is to run the FE computation repeatedly many times and focusing on that part. The other method is get another simulation involving more computation i.e. more nodes and more loops. Actually both methods have been applied. In the fracs2000 run with one FE computation simulation, the FE computation was run 20 more times after the first run, and the profiling focuses on this part. About another simulation involving more computation, the model WellOne solving transient fluid pressure was given, which will be detailed in Section 4.2.2.

Scaling on Ness

<table>
<thead>
<tr>
<th>Proc.</th>
<th>FE time</th>
<th>SAMG time</th>
<th>CSMP time</th>
<th>FE effi.</th>
<th>SAMG effi.</th>
<th>CSMP effi.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44.440</td>
<td>22.547</td>
<td>21.893</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>20.048</td>
<td>9.750</td>
<td>10.298</td>
<td>1.108</td>
<td>1.156</td>
<td>1.063</td>
</tr>
<tr>
<td>4</td>
<td>10.864</td>
<td>5.516</td>
<td>5.348</td>
<td>1.023</td>
<td>1.022</td>
<td>1.023</td>
</tr>
<tr>
<td>8</td>
<td>5.836</td>
<td>3.078</td>
<td>2.758</td>
<td>0.952</td>
<td>0.916</td>
<td>0.992</td>
</tr>
<tr>
<td>16</td>
<td>4.348</td>
<td>2.797</td>
<td>1.551</td>
<td>0.639</td>
<td>0.504</td>
<td>0.882</td>
</tr>
</tbody>
</table>

Table 4.1: fracs2000 with one FE computation on Ness
FE time consists of SAMG time and CSMP time.

Table 4.1 shows the profiling data of this model on Ness, with the original FE method timing. Both CSMP native and SAMG code scale well from 1 to 8 processors. Some even have more than 1 efficiency, that could be because of the distribute of the data which leads to less memory on each processors, and therefore, all the data could be load in to the main memory other than use swap which may heavily slow down the simulation. With 16 processors, the running time is smaller than the 8 processors simulation, but the efficiency is much worse. This is a common problem when run a small simulation on a increasing number of processors, as the amount of the spent in communications starts to dominate compared with the time spent in computation. However,
the quality of the partition program can not be ignored. While it is hard to get a perfect partition, the different load balance on different processors can not be avoided.

Figure 4.1 shows the efficiency chart of this simulation.

![Efficiency Chart](image)

**Figure 4.1: fracs2000 with one FE computation parallel efficiency on Ness**

### Scaling on ECDF

<table>
<thead>
<tr>
<th>Proc.</th>
<th>FE time</th>
<th>SAMG time</th>
<th>CSMP time</th>
<th>FE effi.</th>
<th>SAMG effi.</th>
<th>CSMP effi.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>610.859</td>
<td>185.938</td>
<td>424.921</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>403.467</td>
<td>179.929</td>
<td>223.538</td>
<td>0.757</td>
<td>0.517</td>
<td>0.950</td>
</tr>
<tr>
<td>4</td>
<td>223.392</td>
<td>98.824</td>
<td>124.568</td>
<td>0.684</td>
<td>0.470</td>
<td>0.853</td>
</tr>
<tr>
<td>8</td>
<td>111.557</td>
<td>48.558</td>
<td>62.999</td>
<td>0.684</td>
<td>0.479</td>
<td>0.843</td>
</tr>
<tr>
<td>16</td>
<td>57.148</td>
<td>26.428</td>
<td>30.720</td>
<td>0.668</td>
<td>0.440</td>
<td>0.864</td>
</tr>
<tr>
<td>32</td>
<td>41.762</td>
<td>24.571</td>
<td>17.191</td>
<td>0.457</td>
<td>0.236</td>
<td>0.772</td>
</tr>
<tr>
<td>64</td>
<td>50.453</td>
<td>40.788</td>
<td>9.665</td>
<td>0.189</td>
<td>0.071</td>
<td>0.687</td>
</tr>
</tbody>
</table>

**Table 4.2: fracs2000 with one FE computation on ECDF**

FE time consists of SAMG time and CSMP time.

Table 4.2 shows the profiling data of this model on ECDF, which focuses on the 20 runs of FE method. It could be found from the data that SAMG scales well up to 16 processors, but becomes much worse with 32 and 64 processors. It is the SAMG scaling problem which influence the over all FE computation efficiency. However, it is worth noticing that the CSMP native code has very good scaling behavior, mainly because there is no parallel communication in that part. Compared to the Ness scaling data for 1 to 16 processors, it could be found that on Ness, the communication is much faster than that on ECDF, because Ness is a machine based on share variable model, the communication is actually happening inside the node, and thus very quick.

Figure 4.2 shows the efficiency chart of this simulation.
4.2.2 Model WellOne with many FE computation loops

As mentioned in Section 4.2.1, this simulation involves solving transient fluid pressure, with 240 loops of the FE computation, which corresponds to 240 hours i.e. 10 days in the real world. The WellOne model consists of 840293 nodes and 5224098 elements. The original 240 loop iterations were decreased to 24 loops to effectively investigate the scaling behavior of the simulation within a reasonable runtime.

Encountered Problems

This simulation encountered some problems when running and on Ness and ECDF. When running on Ness, it is found that the simulation can run with 1, 2, 4 or 8 processors, but with 16 processors, it always failed. The information listed below from the standard output indicate this is caused by not having enough C-variables or F-variables.

```
AMG setup information
=======================
Approach: variable-based. Details:
  coarsening - based on full matrix A
  interpolation - SW-pattern of full matrix A
  - weights based on entries of A

Galerkin: truncation:   no
block-diagonal:        no
variant:              standard
Coordinates available: no, dim = 0
Optimization level:   0

creating level # 2 using STANDARD COARSENING, #rows 0.536E+05 nhal 0.290E+04

take_external in local_pcol F
pcol(1):  gratio= 0.00 FF/XF/FC= 53568/ 0/ 0
```

Figure 4.2: fracs2000 with one FE computation parallel efficiency on ECDF
C-variables and F-variables are a concept of the algebraic multi grid (AMG) matrix solver. The AMG method solves matrices by three steps: firstly it selects a subset of the original matrix named the coarse matrix, then solves the coarse small matrix, finally interpolates the result of the other part of the original matrix by the solution to the coarse matrix. The subset of the original matrix apart from the coarse matrix is the fine matrix. In the AMG method, the coarse matrix and fine matrix are usually called coarse grid and fine grid.

The selection of the nodes in the coarse grid and fine grid is the crucial step in the AMG method. The C-variables and F-variables problems are actually problems in such selecting process, where C-variables represent the selected nodes in the coarse grid, and F-variables represent the selected nodes in the fine grid. Many methods has been applied to resolve this problem, mainly changing the SAMG variables according to the SAMG/SAMGp manual [15, 14], but unfortunately, with only one success by changing the nodes graph partition program from a Kway method to a Recursive method. However the Recursive method still cannot work with 32 or 64 processors.

Later on, this problem was resolved with help from SAMG team. Their explanation is that it has been affirmed as a bug of the current SAMG/SAMGp release. The problem is that when the current SAMGp works on some the domain partitioned matrices, it works on each matrix respectively. If one of the matrices is in the format that fulfil is the SAMG default coarse/fine grid nodes selection criterion, it will stop, which causes a deadlock to the whole process. A remedy to this problem, according to the comment form SAMG team, is to change the coarse/fine grid nodes selection criterion. The side effect of the change, is that there will be more nodes in the coarse grid, i.e. the coarse matrix is bigger, so it takes more time. Therefore, there will be not a satisfied scaling data. The new version of SAMG/SAMGp with the bug fixed, will be available in late September 2008, after this project has finished.

In addition, there is one more problem on ECDF, that is the simulation can not start because lack of memory. It seems unreasonable since the serial version, which definitely use the most memory, could run. An in-depth investigation found that, the openmpi on ECDF only allows 2.0 G memory for each running processors, while the serial version does not have such limit (actually the limit for the serial run is the available memory in a node). So, the profiling work on ECDF only involves 1, 8, 16, 32 and 64 processors.

Two set of data are included in this report, one is under the original SAMG coarse/fine grid nodes selection criterion, up to 16 processors, and the other one is under the changed coarse/fine grid nodes selection criterion, from 1 to 64 processors.

1The original explanation is much more technical and related too much to the AMG solver concept, and SAMG features, which both beyond the scope of this project. Thus a less technical one was rewritten for this report. The original explanation from SAMG team for this problem can be found in Appendix E.
Scaling on Ness

<table>
<thead>
<tr>
<th>Proc</th>
<th>FE time</th>
<th>SAMG time</th>
<th>CSMP time</th>
<th>FE effi.</th>
<th>SAMG effi.</th>
<th>CSMP effi.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3309.000</td>
<td>1131.815</td>
<td>2177.185</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>1643.280</td>
<td>541.674</td>
<td>1101.606</td>
<td>1.007</td>
<td>1.045</td>
<td>0.988</td>
</tr>
<tr>
<td>4</td>
<td>887.126</td>
<td>345.237</td>
<td>541.889</td>
<td>0.933</td>
<td>0.820</td>
<td>1.004</td>
</tr>
<tr>
<td>8</td>
<td>458.785</td>
<td>180.470</td>
<td>278.315</td>
<td>0.902</td>
<td>0.784</td>
<td>0.978</td>
</tr>
<tr>
<td>16</td>
<td>322.756</td>
<td>170.922</td>
<td>151.834</td>
<td>0.641</td>
<td>0.414</td>
<td>0.896</td>
</tr>
</tbody>
</table>

Table 4.3: WellOne with 24 loops of FE computation on Ness
FE time consists of SAMG time and CSMP time.

Table 4.3 shows the profiling data of this model on Ness, which is the 24 loops of FE method timing, with original grid nodes selection criterion. Figure 4.3 shows the efficiency chart of this simulation.

![Figure 4.3: WellOne with 24 loops of FE computation parallel efficiency on Ness](image)

Scaling on ECDF

<table>
<thead>
<tr>
<th>Proc</th>
<th>FE time</th>
<th>SAMG time</th>
<th>CSMP time</th>
<th>FE effi.</th>
<th>SAMG effi.</th>
<th>CSMP effi.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2085.000</td>
<td>497.364</td>
<td>1587.636</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>8</td>
<td>441.528</td>
<td>239.389</td>
<td>202.139</td>
<td>0.590</td>
<td>0.260</td>
<td>0.982</td>
</tr>
<tr>
<td>16</td>
<td>256.944</td>
<td>155.405</td>
<td>101.539</td>
<td>0.507</td>
<td>0.200</td>
<td>0.977</td>
</tr>
</tbody>
</table>

Table 4.4: WellOne with 24 loops of FE computation with original selection criterion on ECDF
FE time consists of SAMG time and CSMP time.

40
<table>
<thead>
<tr>
<th>Proc.</th>
<th>FE time</th>
<th>SAMG time</th>
<th>CSMP time</th>
<th>FE effi.</th>
<th>SAMG effi.</th>
<th>CSMP effi.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2433.000</td>
<td>871.141</td>
<td>1561.859</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>8</td>
<td>492.308</td>
<td>290.781</td>
<td>201.527</td>
<td>0.618</td>
<td>0.374</td>
<td>0.969</td>
</tr>
<tr>
<td>16</td>
<td>350.804</td>
<td>243.921</td>
<td>106.883</td>
<td>0.433</td>
<td>0.223</td>
<td>0.913</td>
</tr>
<tr>
<td>32</td>
<td>323.131</td>
<td>269.845</td>
<td>53.286</td>
<td>0.235</td>
<td>0.101</td>
<td>0.916</td>
</tr>
<tr>
<td>64</td>
<td>576.338</td>
<td>536.984</td>
<td>39.354</td>
<td>0.066</td>
<td>0.025</td>
<td>0.620</td>
</tr>
</tbody>
</table>

Table 4.5: WellOne with 24 loops of FE computation with revised selection criterion on ECDF

FE time consists of SAMG time and CSMP time.

Table 4.4 shows the profiling data of this model on ECDF, which is the 24 loops of FE method timing, with original grid nodes selection criterion for 1, 8, 16 processors.

Table 4.5 shows the profiling data of this model on ECDF, which is the 24 loops of FE method timing, with revised grid nodes selection criterion for 1, 8, 16, 32, 64 processors. By comparing the 1, 8 and 16 processors performance under the two different grid nodes selection criterions, it could be found that, the original one offers much better scaling behavior.

Figure 4.4 shows the efficiency chart of this simulation.

![Efficiency Chart](image)

Figure 4.4: WellOne with 24 loops of FE computation parallel efficiency on ECDF

### 4.2.3 Model fracs2000 with both FE and FV computation

The previous two simulations only involve FE computations. They have both shown that, though the SAMG/SAMGp did not scales well, the native CSMP code still exhibits good scaling behavior. However, the overall performance of the FE computation is still not satisfied. The simulation with FV computation, is expected to work much better, since it does not involve matrix solving.
The simulation with both FE and FV computation for this project firstly solves the steady-state fluid pressure by FE method as mentioned in Section 4.2.1, then simulates the transport of the fluid flow by the FV method, which takes much more time than the FE method. The mesh used for this simulation is still the mesh fracs2000, which is also used in the Model fracs2000 with one FE computation as described in Section 4.2.1, it has 223705 nodes and 1012383 hybrid elements.

The original given FV computation of the simulation represents 100 loops of 1000 minutes (i.e. about 60 days in total) fluid flow transport in the real world. In the serial simulation, each FV computation loop takes about 10 hours to finish, and therefore the whole simulation will takes 1000 hours or about 40 days. To profiling this entirely would overload the Ness and ECDF too much, and also exceed the running time limit on both machine: Ness for 12 hours, ECDF for 2 days. So, both number of loops and the simulation time in each loop were decreased to achieve efficient profiling. On Ness, The 1000 minutes each loop were changed to 5 minutes, and the 100 loops were changed to 5 loops, so the overall simulation is 4000 times smaller. On ECDF, The 1000 minutes each loop were changed to 100 minutes, and the 100 loops were changed to 10 loops, so the overall simulation is 100 times smaller. ECDF was used to carry out longer simulation, because it supports up to 2 days running tasks.

### Scaling on Ness

<table>
<thead>
<tr>
<th>Proc.</th>
<th>FE time</th>
<th>FV time</th>
<th>Overall time</th>
<th>FE effi.</th>
<th>FV effi.</th>
<th>Overall effi.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>41.990</td>
<td>1737.000</td>
<td>1778.990</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>20.833</td>
<td>857.371</td>
<td>878.204</td>
<td>1.008</td>
<td>1.013</td>
<td>1.013</td>
</tr>
<tr>
<td>4</td>
<td>11.058</td>
<td>444.215</td>
<td>455.273</td>
<td>0.949</td>
<td>0.978</td>
<td>0.977</td>
</tr>
<tr>
<td>8</td>
<td>6.206</td>
<td>268.768</td>
<td>274.974</td>
<td>0.846</td>
<td>0.808</td>
<td>0.809</td>
</tr>
<tr>
<td>16</td>
<td>4.072</td>
<td>157.288</td>
<td>161.360</td>
<td>0.644</td>
<td>0.690</td>
<td>0.689</td>
</tr>
</tbody>
</table>

Table 4.6: fracs2000 with FEFV computation on Ness
Overall time consists of FE time and FV time.

Table 4.6 shows the profiling data of this model on Ness, which focus on the FV method. The data shows that both of the FE computation and the FV computation scale well on Ness up to 8 processors. For 16 processors, the bad scaling of the FV method should be the communication problem. However, the FV method scales good on ECDF, as shown in Table 4.7, so the problem might be that CSMP native FV code does not suitable for either the MPI environment or the network for communication on Ness. Figure 4.5 shows the efficiency chart of this simulation.

### Scaling on ECDF

Table 4.7 shows the profiling data of this model on ECDF, which focuses on the FV method. It could be found that the FE part scales worse, and it is because of the SAMG
Table 4.7: fracs2000 with FEFV computation on ECDF

<table>
<thead>
<tr>
<th>Proc.</th>
<th>FE time</th>
<th>FV time</th>
<th>Overall time</th>
<th>FE effi.</th>
<th>FV effi.</th>
<th>Overall effi.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>57.380</td>
<td>38930.000</td>
<td>38987.380</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>73.558</td>
<td>18000.400</td>
<td>18073.958</td>
<td>0.390</td>
<td>1.081</td>
<td>1.079</td>
</tr>
<tr>
<td>4</td>
<td>69.454</td>
<td>11307.500</td>
<td>11376.954</td>
<td>0.207</td>
<td>0.861</td>
<td>0.857</td>
</tr>
<tr>
<td>8</td>
<td>62.195</td>
<td>5538.150</td>
<td>5600.345</td>
<td>0.115</td>
<td>0.879</td>
<td>0.870</td>
</tr>
<tr>
<td>16</td>
<td>62.456</td>
<td>2693.040</td>
<td>2755.496</td>
<td>0.057</td>
<td>0.903</td>
<td>0.884</td>
</tr>
<tr>
<td>32</td>
<td>68.202</td>
<td>1342.730</td>
<td>1410.932</td>
<td>0.026</td>
<td>0.906</td>
<td>0.864</td>
</tr>
<tr>
<td>64</td>
<td>85.343</td>
<td>652.260</td>
<td>737.603</td>
<td>0.011</td>
<td>0.933</td>
<td>0.826</td>
</tr>
</tbody>
</table>

Overall time consists of FE time and FV time.

license check out issue as mentioned before, which is essentially an artifact related to the FLEXnet server. The FV computation shows very good performance scaling with up to 64 processors. However, it is worth noticing that in the FV computation, for 4, 8, 16, 32 and 64 processors simulation, the more the processors number, the better the scaling. This may be because of the better and better use of the memory. The overall time, and over all efficiency is influenced by the FE computation. However, recall that we are working with a reduced version (fewer loops and time steps) of the original simulation. The original simulation, with 100 times more FV computation, is expected to have nearly the same scaling behavior as the purely FV scaling behavior with the decreased simulation.

Moreover, the performance of the FV computation part, also illustrates that the mesh partitioning program produced by this project has good quality, because only balanced partitions could produce such good scaling behavior.

Figure 4.6 shows the efficiency chart of this simulation.
4.3 Summary

This Chapter has discussed several problems encountered when running the simulations for profiling, and scaling behavior of the three different simulations. The first simulation for steady-state fluid pressure computation, involves only one FE element step and scales well on Ness, but the overall scaling on ECDF is poor because of the SAMG license checking out problem. The second simulation for transient fluid pressure computation, has many more FE loops than the first one, and therefore it is expected have good scaling on both machines. However, it failed to work fully because of SAMGp bug, there is a remedy to the bug, but it takes extra time which then influences the performance. Though the first two simulation are not perfect for the profiling purpose, the good scaling behavior of CSMP native code can not be ignored, the native CSMP code scales well in both simulations on both machine. The third simulation involves both one FE step for steady-state fluid pressure computation, and many FV loops for fluid flow transport computations. The FV method takes much more time than the FE method, and thus the best one to investigate the computational capability and scaling behavior of CSMP, and the quality of the mesh partitioning program. The analysis in Section 4.2.3 shows that the simulation with the FEFV method has good scaling with up to 64 processors.
Chapter 5

Conclusion

This report details the work involve in designing and testing the hybrid element partition program used in CSMP for FEFV method computations. The porting work is the prerequisite of all other tasks. During the porting progress, several problems were identified and solved. The hybrid element partition program work is the core part of this project, and in this report, all of the analysis, design, implementation, integration and verification work on the partition program have been detailed. The performance work aims at investigating the scaling behavior of the parallelized CSMP code with hybrid element meshes, as well as examining the quality of the partition program implemented by this project. The profiling results, especially the result of the fracs2000 model with both FE and FV computation, shows that the CSMP code scales well and the partitioning program could produce balanced mesh partitioning.

Naturally, a small project like this cannot solve all the encountered problems, some of which maybe even bigger or harder than this project, but it is worth briefly mentioning these issues for further development or improvement. (1) In the porting CSMP phase, one untouched interest is to use the Infiniband network on ECDF. The reason is that it need the SAMG team to compile the SAMG libraries on ECDF to enable the specific Infiniband network support on ECDF. This work is worth doing, if ECDF will be used as a main machine for CSMP simulations in future. (2) In the partition program, there are two METIS graph partition programs used for partitioning CSMP node graphs. They have both been used for different partition numbers as the METIS documentation suggested, but there are several arguments which may influence the partition were never investigated. This may be of interest to some future researcher, if the quality of the partitioned mesh is very crucial. (3) Also the partitioning program could consider to be parallelized and even integrated to the main simulation in future. (4) In the performance analysis part, one remaining problem is using the new version SAMG to investigate the real scaling behavior of the WellOne model.
Appendix A

CSMP file structures and makefiles

This Appendix consists of the CSMP file structures and several sample makefiles introduced by this project as mentioned in Section 2.2.2. The file structures of CSMP are:

![Figure A.1: CSMP file structures](image)

Most of the source file folders are not detailed. Only the files in folder `source_code\generic_element_centered_finite_volumes`...
are fully listed.

The makefiles highlighted by bold font in line 2, 3, 5, 12 and 28 of Figure A.1 are listed bellow respectively:

--- .config.makefile in line 2 ---

CFLAG1 = -O3
CFLAG2 = -DAMD_64 -DDEBUG -DSAMG_WITH_SAMG_SOLVER -DSAMG_LCASE_USCORE
CC = mpicxx $(CFLAG1) $(CFLAG2)

--- parallel.makefile in line 3 ---

default:
  (cd support_libraries && make -f parallel.makefile)
  (cd source_code && make -f parallel.makefile)

--- parallel.makefile in line 5 ---

default:
  (cd algorithms && make -f parallel.makefile)
  (cd applied_math && make -f parallel.makefile)
  (cd computational_geometry && make -f parallel.makefile)
  (cd eos && make -f parallel.makefile)
  (cd finite_elements && make -f parallel.makefile)
  (cd generic_element_centered_finite_volumes && make -f parallel.makefile)
  (cd generic_node_centered_finite_volumes && make -f parallel.makefile)
  (cd interfaces && make -f parallel.makefile)
  (cd interrelations && make -f parallel.makefile)
  (cd main_library && make -f parallel.makefile)
  (cd parallel && make -f parallel.makefile)
  (cd pde_operators && make -f parallel.makefile)
  (cd solver && make -f parallel.makefile)
  (cd triangular_finite_volumes && make -f parallel.makefile)
  (cd two_phase_flow && make -f parallel.makefile)
  (cd visitors && make -f parallel.makefile)

--- parallel.makefile in line 12 ---

#import global compiler settings
#the 'include' is reference to other files
include ../..../.config.makefile

#local building information
CSP_MATH = ../applied_math
CSP_INTF = ../interfaces
CSP_SOLV = ../solver
CSP_MAIN = ../main_library
CSP_GEOM = ../computational_geometry
CSP_FEM = ../finite_elements
CSP_TWO = ../two_phase_flow
CSP_PDE = ../pde_operators
CSP_PAR = ../parallel
CSP = ./
CSP_LIBS = ../../libraries

INCFLAG = -I$(CSP_FEM) -I$(CSP_MATH) -I$(CSP_MATH) -I$(CSP_INTF) -I$(CSP_SOLV)
  -I$(CSP_MAIN) -I$(CSP_GEOM) -I$(CSP_TWO) -I$(CSP_PDE) -I$(CSP_PAR)

#compiling rules
.SUFFIXES : .cpp .o
.cpp.o :
  $(CC) -c $(CFLAGS) $(INCFLAG) $<

#building targets
OBJS = ElementCenteredFiniteVolumeTransport.o
      ThetaLimitedAdvect.o

# linking target
default : libgecfv.a
libgecfv.a : $(OBJS)
ar cr libgecfv.a $(OBJS)

47
mv libgecfv.a $CSP_LIBS

--- parallel.makefile in line 28 ---

default:
    (cd f2c && make -f parallel.makefile)
    (cd jpeg && make -f parallel.makefile)
    (cd meschach && make -f parallel.makefile)

In addition to these makefiles, there are other makefiles all named parallel.makefile located in different folders as list in Figure A.1.

The command
    make -f parallel.makefile
in the CSMP root folder will automatically build the whole parallel version CSMP.
Appendix B

Brief Instruction for Running CSMP with SAMG

To successfully run the CSMP, there are several key steps. This is based on the situation on ECDF, which may be further used for other CSMP simulation.

(1) Obtain libraries and license from SAMG team. The process to obtain the SAMG and SAMGp libraries and license from SAMG team involves mainly finding out the target MPI implementation, hardware information. Two important points in this phase are the hostname of the running machine, and the ip address range, especially if there is more than one network interface to a node. The hostname must be obtained from the machine from which the license server will start. The ip address range should be specified completely.

(2) Start the license server FLEXnet. After getting the license file from SAMG team, the license server FLEXnet could then be started with the license file. The standard command to start the server is:

```
./lmgrd -c pet.hw.ac.ECDF.lic -l samg.log
```

the argument pet.hw.ac.ECDF.lic of -c option is license file, and the argument samg.log of -l option is the logging file. Normally the `pet.hw.ac.ECDF.lic` could be omitted if there is only one license file in the current working directory.

(3) Use SAMG in CSMP. To use SAMG in a CSMP simulation, the position of SAMG must be provided. On ECDF the method to indicate the SAMG license position is:

```
export LM_LICENSE_FILE=57922@frontend01
```

In the `57922@frontend01`, `frontend01` is the hostname, on which the server starts, the `57922` is the port number which is opened on the host frontend01, for the license check out. The hostname is one thing needed to give to SAMG team, and the port number could be found from the samg.log file. The following list is part of the SAMG logging file samg.log. The port for the LM_LICENSE_FILE is highlighted by the bold font.
The port will be changed each time the command lmgrd is issued.

It is also worth remembering that using such form of LM_LICENSE_FILE, e.g:

```
export LM_LICENSE_FILE=57922@frontend01
```
costs 30 seconds to check out the SAMG license, and 60 seconds to check out the SAMGp license.
Appendix C

Core Part of the Partition Program

2D Implementation of the Partition Program

```cpp
void mesh_partition2D()
{
    // 0. copy some values out for further usage
    int n_nodes = vset.Vertices();
    int n_elements = vset.Elements();

    // 1. build node graph.
    // use this "node_edges_sets" to build a 2 dimensional array
    // that contains all the neighbor nodes to one node. [Edge relationship]
    // So, the shape of the "node_edges_sets" is "n_nodes" * "neighbor nodes of current nodes".
    std::vector<int> node_graph_index;
    std::vector<int> node_graph_data;
    std::vector<std::set<int> > node_edges_sets;
    node_edges_sets.resize(n_nodes);

    // 1.1. Now, loop over the "plist" (in the vset), rip the edge relationship from "nodes of element" data,
    // and store these in the "node_edges_sets"
    for(std::deque<std::vector<int> >::const_iterator plist_element_iter; plist_element_iter < vset.PlistEnd(); plist_element_iter++)
    {
        int nodes_in_element = plist_element_iter->size();
        for (int index_node = 0; index_node < nodes_in_element; index_node++)
        {
            int index_next = (index_node+1) % nodes_in_element; // neighbor node
            int node_id = (*plist_element_iter)[index_node];
            int next_id = (*plist_element_iter)[index_next];
            node_edges_sets[node_id-1].insert(next_id);
            node_edges_sets[next_id-1].insert(node_id);
        }
    }

    // 1.2. transform these data to the CSR format and save to "node_graph_index" and "node_graph_data"
    node_graph_index.resize(0); // clean these datas
    node_graph_data.resize(0);
    for(int i=0 ; i<node_edges_sets.size() ; i++)
    {
        // save the index of the CSR graph
        node_graph_index.push_back(node_graph_data.size()+1);
        // save the data of the CSR graph
        node_graph_data.insert(node_graph_data.end(), node_edges_sets[i].begin(), node_edges_sets[i].end());
    }
}
```
void mesh_partition3D(const VSet<csp_float,3>& vset, const int n_parts, std::vector<std::vector<stl_index> >& partitioned_elements, std::vector<std::set<std::pair<stl_index,int> > >& outerhalos)
{
    // 0. copy some values out for further usage
    int n_nodes = vset.Vertices();
    int n_elements = vset.Elements();
    int n_edges = vset.Edges();

    // 1. build node graph.
    // use this "node_edges_sets" to build a 2 dimensional array
    // that contains all the neighbor nodes to one nodes. (Edge relationship)
    std::vector<int> node_graph_index;
    std::vector<int> node_graph_data;
    std::vector<std::set<int> > node_edges_sets;
    node_edges_sets.resize(n_nodes);

    // 1.1. Now, loop over the "plist" (in the vset), rip the edge relationship
    // from "nodes of element" data, and store these in the "node_edges_sets"
    std::deque<std::vector<stl_index> >::const_iterator plist_element_iter;

    // 2. get the nodes in the mesh partitioned. Sort out all the arguments for graph partition program.
    int nn = n_nodes; // 1 : number of nodes
    int* xadj = &node_graph_index[0]; // 2 : nodes graph index
    int* adjncy = &node_graph_data[0]; // 3 : nodes graph
    int wgtflag = 0; // 4 : argument 4 & 5 are used only in a weighted graph.
    int pnumflag = 1; // 5 : Fortran numbering
    int np = n_parts; // 6 : number of partition
    int options[10] = {0}; // 7 : some option for graph algorithm optimization
    int edgecut = 0; // 8 : return how many edges were cut

    // it is suggest by the METIS manual to use METIS_PartGraphRecursive with less than 8 partitions,
    // and METIS_PartGraphKway for more than 8 partition.
    if(n_parts<8)
    METIS_PartGraphRecursive(&nn,xadj,adjncy,NULL,NULL,&wgtflag,&pnumflag,&np,options,&edgecut,npartition);
    else
    METIS_PartGraphKway(&nn,xadj,adjncy,NULL,NULL,&wgtflag,&pnumflag,&np,options,&edgecut,npartition);

    // now, the nodes graph has been partitioned.
    // the other code is not the work of this project.
    // other code for partition mesh based on the nodes partitioning.
    ... ...
}
int index_element = 1;
int index_table;
for(plist_element_iter = vset.PlistBegin(); plist_element_iter < vset.PlistEnd(); plist_element_iter++)
{
    switch(vset.ElementType(index_element)) {
    case ISOPARAMETRIC_LINEAR_BAR: index_table=0; break;
    case ISOPARAMETRIC_LINEAR_TRIANGLE: index_table=1; break;
    case ISOPARAMETRIC_LINEAR_QUADRILATERAL: index_table=2; break;
    case ISOPARAMETRIC_LINEAR_TETRAHEDRON: index_table=3; break;
    case ISOPARAMETRIC_LINEAR_PRISM: index_table=4; break;
    case ISOPARAMETRIC_LINEAR_PYRAMID: index_table=5; break;
    case ISOPARAMETRIC_LINEAR_HEXAHEDRON: index_table=6; break;
    }
    for(int i=edge_table_index[index_table]; i< edge_table_index[index_table+1]; i++)
        node_edges_sets[(plist_element_iter)[type_edge_table[i][0]] - 1].insert((plist_element_iter)[type_edge_table[i][1]]);
    index_element ++;
}

// the following part is the same as the 2D implementation
// 1.2. transform these data to the CSR format and save to "node_graph_index" and "node_graph_data"
node_graph_index.resize(0); // clean these datas
node_graph_data.resize(0);
for(int i=0 ; i<node_edges_sets.size() ; i++)
{
    // save the index of the CSR graph
    node_graph_index.push_back(node_graph_data.size()+1);
    // save the data of the CSR graph
    node_graph_data.insert(node_graph_data.end(), node_edges_sets[i].begin(), node_edges_sets[i].end());
}
node_graph_index.push_back(node_graph_data.size()+1); // finish the index of the CSR graph

///////////////////////////////////////////////////////////////////////////////////
// 2. get the nodes in the mesh partitioned. Sort out all the arguments for graph partition program.
int nn = n_nodes; // 1 : number of nodes
int* xadj = &node_graph_index[0]; // 2 : nodes graph index
int* adjncy = &node_graph_data[0]; // 3 : nodes graph
int* gnumflag = &n_nodes; // 4 : argument 4 & 5 are used only in a weighted graph.
int* options[10] = &n_parts; // 8 : number of partition
int* edgecut = &edgecut; // 10: return how many edges were cut
std::vector<int> node_partition;
node_partition.resize(n_nodes);
int* npartition = &node_partition[0]; // 11: store partitioning of the nodes

// it is suggest by the METIS manual to use METIS_PartGraphRecursive with less than 8 partitions,
// and METIS_PartGraphKway for more than 8 partition.
if(n_parts<8)
    METIS_PartGraphRecursive(&nn,xadj,adjncy,NULL,NULL,&wgtflag,&pnumflag,&np,options,&edgecut,npartition);
else
    METIS_PartGraphKway(&nn,xadj,adjncy,NULL,NULL,&wgtflag,&pnumflag,&np,options,&edgecut,npartition);

// now, the graph has been partitioned.
// the other code is not the work of this project.
... ...
}
Appendix D

Partition Result Verification Program

```python
#!/usr/bin/python
import sys, re

def usage(argc):
    if argc == 3: print "Error : There should be more than 1 parallel vtk files."
    if argc == 2: print "Error : Please also tell me the parallel vtk files."
    print "Usage :", sys.argv[0]," <serial-vtk-filename> <parallel-vtk-filenames>"
    print "Sample ":, 'vtkutil ../serial/fluid_pressure0.vtk fluid_pressure_*'

class VtkResults:
    def __init__(self):
        self.vtk_type = 'UNKNOWN'
        self.point_value_pairs = {}  
        self.diff = 0.0
        self.ssum = 0.0

    def processVtkFile(self, vtkfile, result_type):
        print 'processing',result_type,'file :', vtkfile
        points = [] ; values = [] # result to be stored

        input = open(vtkfile, 'r')
        lines = input.readlines()
        input.close()

        for k in range( len(lines)-1, -1, -1): # do some clean job
            lines[k] = lines[k].strip() # remove surrounding spaces
            if lines[k]=='' : del lines[k] # delete the empty line

        del lines[0:4] # delete first 4 lines

        number = int(lines[0].split(' ')[1]) # POINTS <number> <type>
        for k in range(1,number+1):  # read in and store all coordinates
            x, y, z = lines[k].split(' ') # POINTS <number> <type>
            points.append((float(x), float(y), float(z)))

        del lines[0:number+1] # delete the POINTS part

        number = int(lines[0].split(' ')[1]) # CELLS <number> <size>
        del lines[0:number+1] # delete the CELLS part

        number = int(lines[0].split(' ')[1]) # CELL_TYPES <number>
        del lines[0:number+1] # delete the CELL_TYPES part

        number = int(lines[0].split(' ')[1]) # POINT_DATA <number>
```

this_vtk_type = lines[1].split(' ')[0] # <SCALARS/VECTORS> <name> <type>

if self.vtk_type == 'UNKNOWN':
    self.vtk_type = this_vtk_type
elif self.vtk_type != this_vtk_type:
    print 'ERROR: All files should have the same point data type.'
    sys.exit(1)

if self.vtk_type == 'SCALARS':
    i = 0 # POINT_DATA, SCALARS/VECTORS,
    k = 3 # LOOKUP_TABLE takes 3 lines
    while i < number :
        for v in lines[k].split(' '):
            values.append(float(v))
            i = i + 1
        k = k + 1 # this 'k' belongs to while loop.
    elif self.vtk_type == 'VECTORS': # there's no LOOKUP_TABLE for VECTORS
        for k in range(2, number+2):
            v1, v2, v3 = lines[k].split(' ')
            values.append((float(v1), float(v2), float(v3)))

del lines

number = len(points)
assert number == len(values)
if result_type == 'serial': # build up the serial data dictionary
    for i in range (number):
        self.point_value_pairs[points[i]] = values[i]
elif result_type == 'parallel': # compare and record the difference
    if self.vtk_type == 'SCALARS':
        for i in range (number):
            self.ssum = self.ssum + abs(self.point_value_pairs[points[i]] - values[i])
            self.diff = self.diff + abs(self.point_value_pairs[points[i]] - values[i])
    elif self.vtk_type == 'VECTORS':
        for i in range (number):
            for j in range(3):
                self.ssum = self.ssum + abs(self.point_value_pairs[points[i]][j] - values[i][j])
                self.diff = self.diff + abs(self.point_value_pairs[points[i]][j] - values[i][j])

# end of process vtk file

def showDiff(self):
    print 'The difference is :', self.diff/self.ssum

# end of class VtkResult

def main():
    argc = len(sys.argv)
    if(argc<=3):
        usage(argc)
        sys.exit(0)

    vtk_result = VtkResults()
    vtk_result.processVtkFile(sys.argv[1], 'serial')
    for parallel_vtk in sys.argv[2:]:
        vtk_result.processVtkFile(parallel_vtk, 'parallel')
    vtk_result.showDiff()

if __name__ == '__main__': # entrance of the program
    main()
Appendix E

WellOne Simulation Bug Explanation

The email from Arno Krechel of SAMG team for resolving the bugs

The matrices show a situation which has not been considered in the previous versions of the code. However, I have found a simple workaround.

Large parts of the matrices are locally diagonal dominant according to SAMG’s default criterion. Therefore it may happen for some processors and for some partitionings that you are using, that all rows of the matrix on the corresponding processor are locally diagonal dominant. SAMGp then decides not to coarsen this part of the domain at all. In this way the code enters into the exceptional case (which has not been considered before as mentioned above) that there are processors which are becoming idle already with the second level. For these processors it looks as if performing a one level method, whereas there are still some processors proceeding to coarser levels. The code then unfortunately deadlocks or aborts.

The remedy that I am proposing is to change the SAMG internal parameter ecg(*). The default value for ecg according to the user manual is ecg=21.25 which means that the value for strong local diagonal dominance in defining strong couplings is ecg1=1.0d-2. If e.g. you use ecg1=1.0d-9 instead (-> ecg=91.25, see chap 5.2.2 and 7.1.1 of the user manual), the criterion for local diagonal dominance is much more restrictive and many more rows will be considered in the coarsening process. Therefore, all the processors will proceed to coarser grids (well - at least for your test cases that I checked today:

/16processors_partitioning1/level
/16processors_partitioning2/level
/32processors_partitioning1/level
/32processors_partitioning2/level
/samg_dumped_matrix_16p/level

In the next releases which can be expected end of September/beginning of October this problems will be avoided. We’ll send you an update of the SAMGp libraries with the next release as I am hesitating to send you some intermediate versions now.

Anyway, I am awfully sorry for the inconveniences that has been caused by this gap in
The implementation of SAMGp!

The email from Klaus Stueben of SAMG team for performance remark

Since your matrices seem to have very special properties, I want to say a few more words regarding their impact on our SAMG algorithm so that you understand why and how performance may depend on the parameter ecg. At the same time, I hope that I can learn something about why your matrices are the way they are.

Remarks on the performance as a function of ecg

Most rows of your matrices are strongly diagonally dominant. In the last test case, for instance, there are 840,293 rows with 591,845 of them being diagonally dominant (around 70%), most of them even very (!) strongly. Clearly, linear systems, which are globally strongly diagonally dominant do not require any multigrid approach for their solution; they can very efficiently be solved by any standard numerical method. This is different in your test case since there are still quite some rows of a different nature.

Anyway, by default (using the standard ecg value), SAMG tries to take advantage of the special properties of your matrices and attempts to "isolate" the variables corresponding to the most strongly dominant rows. That is, SAMG does not put these variables on the coarser levels but rather resolves them just by the smoothing process on the finest level. As a consequence, there is a very rapid coarsening, coarse levels have a very low number of variables left on them, and most variables strictly remain on the finest level only. (Arno has explained why this exceptional situation has led to a bug in your version of SAMGp.)

Now, if you change ecg as explained by Arno (eg, to ecg=91.25), SAMG does no longer detect the strongly diagonally dominant rows in your last test case (at least not on the finest level) but rather treats them like any other row. As a consequence, coarsening becomes much slower (in fact, it is now more typical to what SAMG would do in more standard applications!), and much more work has to be done on coarser levels, which explains why the computational time for the 1-proc case as well as the required memory significantly increases compared to the default ecg-setting. This is without any benefit in terms of speed of convergence.

Clearly, this difference also has an impact both on the parallel performance and the scalability. In particular, the less work is done on coarser levels, the better the scalability can be. This roughly explains the different performances you measured for ecg=21.25 and ecg=91.25.

Where do the special properties come from?

All of the above is not typical at all for standard elliptic applications where weak diagonal dominance would be typical. So the question arises where these strongly diagonal rows come from. Can this easily be explained?

For us, understanding a bit more the origin of such special properties, may help to tune
our solvers to such situations.

**General remark on parallel scalability**

The parallel scalability of SAMGp depends on many aspects. Naturally, it degrades with increasing influence from the coarser levels (where communication increasingly dominates computation). Consequently, the bigger a problem, the more processors may be used and still get reasonable performance. By our experience, very roughly, something like 50,000 variables per processor should still permit reasonable performance. Below this, scalability may soon reach its limits. In your problem, this means that the use of more than 16 processors might be questionable (and, in fact, is).

Besides on the problem details, to some extent, scalability also depends on the chosen SAMG settings such as cycle type and type of coarsening. Also the following (hidden) SAMGp parameters may influence scalability to some substantial extent: ipara_coarsen, icomm_flag and minimal_msg_len. Question: Did you explicitly change the setting of any of these variables?

Hope this helps a bit in understanding.
Appendix F

Comparison of the Original Work Plan and the Final Work Process

Figure F.1 shows the original work plan of this project.

Compare to the original work plan, there are many difference in the real work process. The porting work on Ness is roughly finished on time, but the porting work on ECDF problems finally takes about 8 weeks, because of the incorrect SAMG libraries and license. The coding work started a little late due to the delay of the porting work, but it finished in two weeks. The profiling work takes much more time than expected while there was a bug in SAMG identified. The biggest change to the non writing work is the optimization part. The optimization was expected to contain both the hybrid element partition and further optimization of the CSMP native parallel code, and finally only the
hybrid element partition program was finished. The optimization of the CSMP native parallel code was not carried out, mainly because of two reasons: one is that the porting / profiling phase takes too much time, the other one is that there is not too much parallel code, which has been well developed.

The writing related works, were adjusted according to the delay of the porting, coding or profiling work.
Bibliography


[16] Introduction to the EPCC HPC Service, Ness (Version 1.00) 2008-03-18 http://www2.epcc.ed.ac.uk/ness/documentation/ness/index.html


