UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA

Ederson Augusto Grein

A PARALLEL COMPUTING APPROACH APPLIED TO PETROLEUM RESERVOIR SIMULATION

Florianópolis
2015

Ederson Augusto Grein

A PARALLEL COMPUTING APPROACH APPLIED TO PETROLEUM RESERVOIR SIMULATION

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Mecânica da Universidade Federal de Santa Catarina para a obtenção do grau de Mestre em Engenharia Mecânica.

Orientador: Clovis Raimundo Maliska, Ph.D.
Coorientador: Kamy Sepehrnoori, Ph.D.

Florianópolis
2015

Ederson Augusto Grein

A PARALLEL COMPUTING APPROACH APPLIED TO PETROLEUM RESERVOIR SIMULATION

Esta dissertação foi julgada aprovada para a obtenção do título de "Mestre em Engenharia Mecânica", e aprovada em sua forma final pelo Curso de Pós-graduação em Engenharia Mecânica.

Florianópolis, 07 de Agosto de 2015

Armando Albertazzi Gonçalves Jr, Dr. Eng.
Coordenador

Clovis Raimundo Maliska, Ph.D.
Orientador

K. Sepehrnoori
Kamy Sepehrnoori, Ph.D.
Coorientador

Banca examinadora:

António Fábio Carvalho da Silva, Dr. Eng.

Fernando Sandro Velasco Hurtado, Dr. Eng.

Mário Antônio Ribeiro Dantas, Ph.D.

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

Grein, Ederson Augusto, 1989 -
A Parallel Computing Approach Applied to Petroleum Reservoir Simulation / Ederson Augusto Grein - 2015.
148 f. : il. color. ;

Orientador: Clovis Raimundo Maliska, PhD.
Coorientador: Kamy Sepehrnoori, PhD.
Dissertação (mestrado) - Universidade Federal de Santa Catarina, Curso de Engenharia Mecânica, 2015.

1. Simulação numérica de reservatórios de petróleo. 2. Computação Paralela. 3. UTCHEM. 4. *Element-based Finite Volume Method.* I. Maliska, Clovis Raimundo. II. Universidade Federal de Santa Catarina. Curso de Engenharia Mecânica

À Taisa, por todo
carinho e apoio.

Acknowledgements

I would like to express my gratitude to professor Clovis Maliska. Working under his supervision at the SINMEC laboratory was fundamental for me to acquire the necessary experience required to develop this study. I am also thankful to him for making the necessary arrangements for me to develop part of this study at the University of Texas at Austin.

I am truly thankful to professor Kamy Sepehrnoori for accepting me as a collaborator in one of the most recognized centers of Petroleum Engineering in the world. Furthermore, he supervised my work closely, scheduling weekly meetings to know my developments and giving appropriated feedbacks, which helped me reaching my aims. With him and Mojtaba Ghasemi Doroh I have learned the basic concepts of parallel computing applied to reservoir simulation. I also would like to express my gratitude to professor Sepehrnoori and to Center of Petroleum & Geosystems Engineering (CPGE) for the financial support during the period I was in the United States.

I am also grateful to Tatiane Schveitzer and Axel Dihlmann for continuously helping me and giving support to the execution of this study. My gratitude also goes to the SINMEC laboratory, to the Programa de Formação de Recursos from the Agência Nacional do Petróleo, and to POSMEC for providing the necessary financial support and infrastructure for this study.

I am heartily thankful to Taisa Pacheco for giving me support and for helping me to review this text. My gratitude also goes to Henrique Pacheco, who helped me with the code and with the execution of some tests, and to my colleagues Giovani Cerbato, Victor Magri, Gustavo Ribeiro, Hamid Lashgari, and Aboulghasem Nia, for all the discussions about this

study and the support they gave me.

And of course I am deeply thankful to my beloved parents Aliomar Grein and Marlene Joseli Moreira Grein. Without their support and guidance it would be impossible to fulfil my dreams.

"What we observe is not nature itself, but nature exposed to our method of questioning."
Werner Heisenberg

Contents

List of Figures **v**

List of Tables **ix**

List of Symbols **xi**

Resumo **xiii**

Abstract **xv**

1 Introduction **1**
1.1 Preliminaries 1
1.2 Objectives 3
1.3 Organization of the study 4

2 Parallel Computing **5**
2.1 Introduction 5
2.2 Algorithm classification 7
2.2.1 Serial algorithms 7
2.2.2 Parallel algorithms 8
2.2.3 Serial-parallel algorithms 8
2.2.4 Nonserial-parallel algorithms 8
2.2.5 Regular iterative algorithms 8
2.3 Parallel architectures 9
2.3.1 Shared-memory architecture 9
2.3.2 Distributed-memory architecture 11
2.3.3 Hybrid-memory architecture 11

2.4 Application Programming Interfaces 12
2.4.1 OpenMP 12
2.4.2 MPI 13
2.5 Parallel efficiency and theoretical limits 14

3 UTCHEM Reservoir Simulator 19
3.1 Introduction 19
3.2 Mathematical formulation 20
3.3 Grids 23

4 The Element-based Finite Volume Method 25
4.1 Introduction 25
4.2 Grid entities 26
4.3 Numerical scheme 28
4.4 Discretization of a conservation equation 30

5 The Proposed Approach 33
5.1 UTCHEM 33
5.1.1 Domain decomposition 34
5.1.2 Inactive grid blocks 36
5.1.3 IPARS framework 37
5.2 EFVLib 39
5.2.1 Graph partitioning 41
5.2.2 Ghost nodes 44
5.2.3 Assembling in parallel a system of linear equations . 46
5.2.4 Solving in parallel a system of linear equations 50
5.2.5 Wells and boundaries 54
5.2.6 The code 56

6 Experimental Environment and Results 61
6.1 Case 1 62
6.2 Case 2 67
6.3 Case 3 73
6.4 Case 4 78
6.5 Case 5 84
6.6 Case 6 94

7 Conclusions **101**
7.1 Summary 101
7.2 Conclusions 103
7.3 Suggestions for future studies 104

Bibliography **107**

Appendix A UTCHEM's Input File **111**

Appendix B Example of code using EFVLib in parallel **115**

List of Figures

1.1 Frequency in MHz of several processors along the years [22] 2

2.1 Common architecture of sequential computers (adapted from [23]) 6
2.2 Ilustration of a shared memory parallel computer. 10
2.3 Ilustration of a shared memory parallel computer. 11
2.4 Ilustration of a shared memory parallel computer. 12
2.5 Theoretical speedup according to Amdahl's Law. 16
2.6 Theoretical speedup according to Gustafson-Barsis's Law. 17

4.1 Discretization of an hypothetical reservoir using a unstructured grid (adapted from [17]) 26
4.2 Main entities of a triangular element 27
4.3 Control volume creation 28
4.4 Mapping into a transformed space 28
4.5 Stencil of p 32

5.1 Example of a grid division 35
5.2 Example of ghost cells from a 6x8x3 grid 36
5.3 Loop modification when type 2 arrays are presented 36
5.4 Example of inactive grid blocks been used to better describe a reservoir domain 37
5.5 Organization of UTCHEMP 38
5.6 Simulator workflow 39
5.7 Graph in **(b)** is the graph of grid **(a)** 41

5.8 Unstructured grid divided into two subdomains 45
5.9 Subdomains extended 45
5.10 Local indexes of two subdomains 46
5.11 PETSc indexes of two subdomains 49
5.12 Matrix assembled in parallel 50
5.13 Boundaries and well 55
5.14 Boundaries and well 55

6.1 Water saturation after 800 simulation days of case 1 63
6.2 Average aqueous phase pressure of case 1 64
6.3 Average water saturation of case 1 64
6.4 Overall oil production rate of case 1 65
6.5 Total computational time according to the number of processors of case 1 66
6.6 Global speedup of case 1 66
6.7 Contribution on the total computational time of case 1 . . 68
6.8 Temperature after 510 simulation days of case 2 69
6.9 Average aqueous phase pressure of case 2 70
6.10 Average water saturation of case 2 70
6.11 Overall oil production rate of case 2 71
6.12 Computational time according to the number of processors of case 2 71
6.13 Global speedup of case 2 72
6.14 Contribution on the total computational time of case 2 . . 73
6.15 Case 3 aqueous phase saturation after 600 simulation days 74
6.16 Average aqueous phase pressure of case 3 75
6.17 Average water saturation of case 3 75
6.18 Overall oil production rate of case 3 76
6.19 Computational time according to the number of processors of case 3 76
6.20 Global speedup of case 3 77
6.21 Contribution on the total computational time of case 3 . . 78
6.22 Microemulsion phase saturation after 350 simulation days of case 4 79
6.23 Average aqueous phase pressure of case 4 80
6.24 Average water saturation of case 4 80
6.25 Overall oil production rate of case 4 81

6.26 Computational time according to the number of processors of case 4 82
6.27 Global speedup of case 4 82
6.28 Contribution on the total computational time of case 4 . . 83
6.29 Microemulsion phase saturation after 355 simulation days using inactive cells of case 4 84
6.30 Illustration of the grid and wells of case 5 85
6.31 Saturation field after 170 days of case 5 86
6.32 Grid division of case 5 using different number of processors 87
6.33 Average pressure of case 5 88
6.34 Average water saturation of case 5 89
6.35 Overall oil production rate of case 5 89
6.36 Computational time according to the number of processors of case 5 90
6.37 Global speedup of case 5 90
6.38 Contribution on the total computational time of case 5 . . 91
6.39 Pressure computation speedup of case 5 91
6.40 Water saturation computation speedup of case 5 92
6.41 Computational time of some grid operations of case 5 . . 92
6.42 Maximum ratio between the surplus number of nodes and the ideal number of nodes of case 5 93
6.43 Average ratio between the number of ghost nodes and the number of local nodes of case 5 94
6.44 Grid and wells of case 6 95
6.45 Computational time according to the number of processors of case 6 96
6.46 Global speedup of case 6 96
6.47 Contribution on the total computational time of case 6 . . 97
6.48 Pressure computation speedup of case 6 98
6.49 Water saturation computation speedup of case 6 98
6.50 Computational time of some grid operations of case 6 . . 99
6.51 Maximum ratio between the surplus number of nodes and the ideal number of nodes of case 6 100
6.52 Average ratio between the number of ghost nodes and the number local nodes of case 6 100

List of Tables

6.1 Case 1 properties 63
6.2 Computational times in seconds of case 1 67
6.3 Case 2 properties 69
6.4 Computational times in seconds of case 2 72
6.5 Case 3 properties 74
6.6 Computational times in seconds of case 3 77
6.7 Case 4 properties 79
6.8 Computational times in seconds of case 4 83
6.9 Case 5 properties 85
6.10 Case 6 properties 95

List of Symbols

(ξ, η, ζ)	Local coordinates
(x, y, z)	Global coordinates
$\bar{Q}$	Production/injection rate
Γ	Diffusivity coefficient
γ	Specific gravity
λ	Mobility
$\mathbb{K}$	Medium absolute permeability
$\mathbf{u}$	Fluid velocity
$\mathcal{N}$	Shape function
WI	Well index
μ	Viscosity
Ω_i	Grid element
ϕ	Porosity
Π	Well pressure
ρ	density
$\tilde{C}$	Overall concentration
φ	General function defined in the domain of interest

$\vec{D}$	Dispersive flux
$\vec{u}$	Superficial velocity
C	Concentration
C^o	Compressibility
C_r	Rock compressibility
C_t	Total compressibility
E	Parallel efficiency
h	Vertical depth
J	Jacobian matrix
K	Preconditioner matrix
k	Relative permeability
n_p	Number of phases
n_{vc}	Number of volume-occupying phases
P	Pressure
Q	Source term
r	Reaction rate
S	Speedup
S_l	Saturation of phase l
T	Computational time

Resumo

A simulação numérica é uma ferramenta de extrema importância à indústria do petróleo e gás. A partir dela, podem-se prever os cenários de produção de um dado reservatório de petróleo e, com base nos dados obtidos, traçar melhores estratégias de explotação. Entretanto, para que os resultados advindos da simulação sejam fidedignos, é fundamental o emprego de modelos físicos fiéis e de uma boa caracterização geométrica do reservatório. Isso tende a introduzir elevada carga computacional e, consequentemente, a obtenção da solução do modelo numérico correspondente pode demandar um excessivo tempo de simulação. É evidente que a redução desse tempo interessa profundamente à engenharia de reservatórios. Dentre as técnicas de melhoria de *performance*, uma das mais promissoras é a aplicação da computação paralela. Nessa técnica, a carga computacional é dividida entre diversos processadores. Idealmente, a car-ga computacional é dividida de maneira igualitária e, assim, se N é o núme-ro de processadores, o tempo computacional seria N vezes menor. No presente estudo, a computação paralela foi aplicada ao simulador de reservatórios UTCHEM e à biblioteca EFVLib. UTCHEM é um simulador químico-composicional desenvolvido pela *The University of Texas at Austin*. A EFVLib, por sua vez, é uma biblioteca desenvolvida pelo laboratório SINMEC – laboratório ligado ao Departamento de Engenharia Mecânica da Universidade Federal de Santa Catarina – cujo intuito é prover suporte à aplicação do Método dos Volumes Finitos Baseado em Elementos. Em ambos os casos a metodologia de paralalelização é baseada na decomposição de domínio.

Abstract

Numerical simulation is an extremely relevant tool to the oil and gas industry. It makes feasible the procedure of predicting the production scenery in a given reservoir and design more advantageous exploit strategies from its results. However, in order to obtain reliability from the numerical results, it is essential to employ reliable numerical models and an accurate geometrical characterization of the reservoir. This leads to a high computational load and consequently the achievement of the solution of the corresponding numerical method may require an exceedingly large simulation time. Seemingly, reducing this time is an accomplishment of great interest to the reservoir engineering. Among the techniques of boosting performance, parallel computing is one of the most promising ones. In this technique, the computational load is split throughout the set of processors. In the most ideal situation, this computational load is split in an egalitarian way, in such a way that if N is the number of processors then the computational time is N times smaller. In this study, parallel computing was applied to two distinct numerical simulators: UTCHEM and EFVLib. UTCHEM is a compositional reservoir simulator developed at The University of Texas at Austin. EFVLib, by its turn, is a computational library developed at SINMEC – a laboratory at the Mechanical Enginering Department of The Federal University of Santa Catarina – with the aim of supporting the Element-based Finite Volume Method employment. The parallelization process were based on the domain decomposition on the both cases formerly described.

CHAPTER

1

Introduction

1.1 Preliminaries

Large-scale reservoir simulations may demand high computational performance and a huge amount of computer memory in order to be feasible its use in reservoir engineering. A lot of effort has been made to improve the computation performance. Better linear system solvers, multiscale methods and other technics has proven to be very important for such improvement. However, when a code is optimized, it becomes an arduous job to achieve a great time reduction by simply enhancing the algorithms. Thus, it is quite relevant to consider the hardware architecture in order to make reservoir simulators faster.

One of the main parameters that interferes on the hardware performance is the CPU frequency. As the frequency increases, more operations can be performed and consequently the algorithm runs faster. Figure 1.1 shows the frequency of several processors along the years. It is clear that the current processor architecture is saturating and hence no significant performance gain may happen unless a radical architecture change is introduced. Because of this, computer manufacturers adopted the strategy of using several simple processors instead of a single and probably

complex one. Each processor works only on a part of the problem and the overall computational performance is given by the time consumed by the most loaded processor.Employing a large number of processors diminishes the problem size on which each processor works. Ideally, if N processors are used, the code will run N times faster.

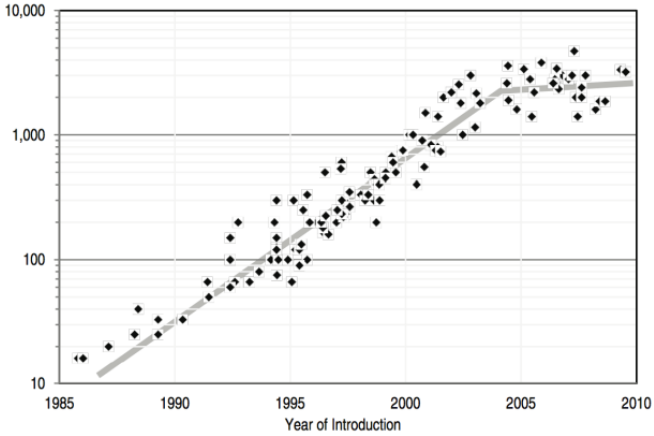


Figure 1.1 – Frequency in MHz of several processors along the years [22]

Parallel computer's relevant features go beyond having more than one processor, forasmuch as the amount of available memory is usually much bigger. CPU clusters own a memory module for each node that is contained by them. Consequently, a greater amount of memory becomes available if a larger number of nodes is employed. This is of great interest since a substantial number of simulation cases is unfeasible not due to a running time constraint, but because their models are so complex that it is not even possible to load them.

One should note, however, that using a parallel computer usually is not enough to take the advantages of parallel computing. In fact, a code originally conceived for serial computers may run even slower in a parallel computer since only a single – and probably simple – processor is allocated. It is also necessary to program what operation each processor should execute. Furthermore, an egalitarian division of the operations optimizes the code performance. Otherwise, the overloaded processor dictates the overall performance.

The main purpose of this study is the parallelization of two distinct simulation codes: UTCHEM and EFVLib. The University of Texas Chemical Compositional Simulator (UTCHEM) is a three-dimensional, multicomponent, multiphase, compositional, variable temperature, finite-difference reservoir simulator developed at The University of Texas at Austin. It can be used to simulate the enhanced recovery of oil and the enhanced remediation of aquifers. Some of its features are modeling of capillary pressures, three-phase relative permeabilities, dispersion, diffusion, adsorption, chemical reactions, and non-equilibrium mass transfer between phases [2]. UTCHEM is used worldwide and its parallelization will directly benefit those who intend to run large and realistic reservoir simulation cases with this simulator.

The second simulation code intended to be parallelized is a library (EFVLib) in which the Element-based Finite Volume Method (EbFVM) [17] is implemented. It was developed at SINMEC, a CFD laboratory from The Federal University of Santa Catarina, at Florianópolis. This library supports two and three-dimensional hybrid unstructured grids. In 2D, the grid is composed by triangles and quadrangles and in 3D by hexahedrons, tetrahedrons, prisms, and pyramids. EFVLib handles grid's operations, geometry, and topology in a user-friendly way. Also, it provides a convenient environment to develop numerical methods in fluid mechanics and heat transfer.

The motivation for parallelizing two softwares instead of just one is that different aspect of the parallel computing will be contemplated. UTCHEM is a huge code with a lot of physical models, but the grid is structured, which simplifies the parallelization. EFVLib, on the other hand, is a much smaller code and does not require the validation against several physical cases, since those cases are implemented by the final user. The grid however is unstructured and the calculation of fluxes is much more complex.

1.2 Objectives

The main objectives of this study are

- Parallelize UTCHEM;
- Parallelize EFVLib.

These objectives may be split into the following secondary objectives:

- Define an efficient methodology to divide the computational load among the available processing units;
- Search and make usable external resources that may help to execute in parallel some of the required simulation steps;
- Since Linux is the usual operation system of cluster of CPU's and EFVLib was implemented in Windows, adapt EFVLib to a Linux platform;
- Validate both UTCHEM and EFVLib parallel versions against benchmark cases;
- Evaluate the performance gains achieved using parallel processing.

1.3 Organization of the study

In chapter 2 has some basic concepts about parallel computing, including software and hardware matters. Chapter 3 introduces UT-CHEM's main features and its mathematical formulation. In chapter 4 the Element-based Finite Volume Method (EbFVM) is discussed. In chapter 5 it is briefly described the domain decomposition methodology used to distribute the computational load among processors. It is also described the inactive grid blocks methodology and the new UTCHEM's input file format. Besides, it also gives some highlights of the EFVLib code itself and presents some other details. Chapter 6 contains the results achieved and its analysis.

CHAPTER

2

Parallel Computing

2.1 Introduction

It is quite hard to rigorously define what is parallel computing. Such area includes aspects related to algorithms, applications, programming languages, operating system, and computer architecture. All of these aspects must be specialized in order to provide support for computations that involve more than one processor [21]. The motivation to use more than one processor comes from the fact that nowadays it is worthless trying to improve computer performance adopting a single processor. Such processor would consume too much power and a very sophisticated refrigeration system would be required to dissipate the heat generated. It is much more practical to exploit several simple processors to attain the same desired performance [10]. In fact, as the trend of parallel computers spreads, launching a single processor in these newer computers makes serial algorithms run even slower than they would in a serial machine.

Seemingly, the main motivation of parallel computing is to speed up computations. With the nowadays computer technologies, it is quite easy to acquire and store huge amounts of data. However, process that data is something that, if performed by a single processor, would require a

prohibitively long time [23]. In this context, the basic idea of parallel computing is to split the job among several processors. Each one of them works concurrently only in a small segment of the problem. Besides, they communicate between themselves in order to achieve the expected result.

The natural way to analyze a parallel algorithm is comparing it to the best approach that solves the same problem in a sequential scenario. Single-processor computers usually adopt the architecture illustrated in Figure 2.1. Throughout its calculations, only a single sequence of instructions and data can be processed at a time by its sole process unit. The control unit, in its turn, is aware of the operation that must be executed and by this knowledge it sets the operands. Furthermore, the control unit recognizes the available variables, although it cannot inform its values. Hence, to enable a complete loading of the desired data into the processor's internal registers the memory access creates a path between the memory and the processor. Such design is commonly referred to as Random Access Machine [23].

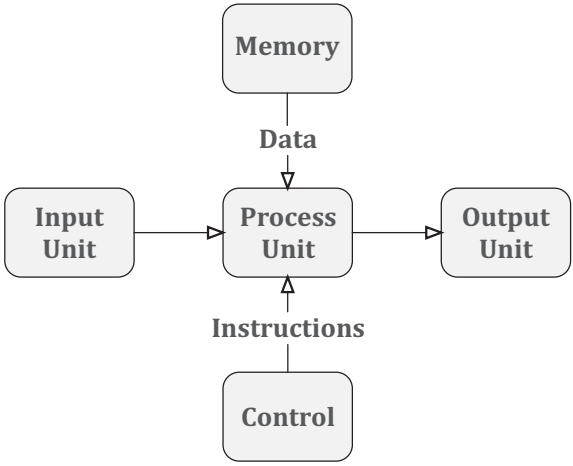


Figure 2.1 – Common architecture of sequential computers (adapted from [23])

When more than one processor is addressed to execute a job, the way it might be performed may differ greatly from the traditional concept of sequential computing. From the algorithm standpoint, the problem is divided into subproblems, each one solved concurrently by a single

processor. Those processors might communicate between themselves in order to yield the final result. However, there are important architecture issues that might interfere in the way that the program is parallelized, such as the arrangement of memory and processors, the possibility of executing different tasks in distinct processors, the processors' communication system, and the processors' operation (synchronously or asynchronously) [23]. The following two sections are intended to discuss the algorithms' task dependency and parallel architectures, respectively. In particular, labelling the algorithms according to its task dependencies is quite convenient, once that will be useful to recognize which parts of UTCHEM and EFVLib may be efficiently parallelized.

2.2 Algorithm classification

The algorithm classification adopted here – the same as in [10] – is based on task dependencies. According to this classification, there are five types of algorithms:

1. Serial algorithms;
2. Parallel algorithms;
3. Serial-parallel algorithms;
4. Nonserial-parallel algorithms;
5. Regular iterative algorithms.

The following subsections expose briefly the types mentioned above.

2.2.1 Serial algorithms

Serial algorithms are those that require a sequential execution of their tasks. Due to the data dependency, launching a new task depends on finalization of the previous one. Hence, a synchronously running is not possible and thus no gain is obtained by exploiting several processor unities.

2.2.2 Parallel algorithms

Opposed to the previous type, parallel algorithms have tasks that share no data dependency between each other. Thus the tasks are independent at such degree that they can be executed concurrently by several processor unities. Moreover, the overall performance of parallel algorithms are limited by the overloaded processor.

2.2.3 Serial-parallel algorithms

Serial-parallel algorithms may have their tasks grouped in stages. Each stage can have its tasks executed in parallel, but the stages themselves must be executed sequentially. It is clear that if there is only one stage, then the algorithm is parallel. On the other hand, if each stage has a single task, the algorithm is serial.

2.2.4 Nonserial-parallel algorithms

A nonserial-parallel algorithm (NSPA) cannot be put in the above classes because its workflow follows no pattern at all. According to [10], an NSPA graph is characterized by two types of constructs: nodes, which corresponds to the algorithm tasks, and directed edges, which describes the direction of the data flow among the nodes. The graph provides important information, such as the work – amount of work to complete the algorithm –, the depth – maximum path length between any input node and any output node –, and the degree of parallelism – maximum number of nodes that can be processed in parallel.

2.2.5 Regular iterative algorithms

Regular iterative algorithms are designed by a fixed pattern. Identifying this pattern can be an arduous job, though. Such algorithms are the most difficult to be parallelized, but they deserve special attention due to their ample presence in fields such signal, image and video processing, linear algebra, and numerical simulation.

2.3 Parallel architectures

The approach used to make an algorithm parallel is strictly related to the multiprocessing architecture. It is crucial to choose a processor architecture that is sufficiently qualified to perform the algorithm's instructions assuring reliability. Furthermore, the processors must communicate between themselves using some kind of interconnection network that, if not fast enough, might be the bottleneck for the software performance. Thus, if the interconnection network is known to have poor quality, reducing data exchange between processors is a pivotal aim for the algorithm design. Besides the processor architecture, the memory system may also be taken into account. The memory modules might be shared among the processors so that all of them are able to access the global data. On the other hand, it is also possible to dedicate a memory module to each processor. In this case, each processor has only a part of the global data and as a consequence data needs to be communicated between memory modules via an interconnection network. If some data is updated in a processor unit, all of the other ones must be informed somehow in order to have the correct values [10].

Regarding multiprocessing architectures, this study will focus in three of the main types:

- Shared-memory;
- Distributed-memory;
- Hybrid-memory.

The above classification relies on how the memory resources are available to the processors. In the following sections the types above will be briefly described.

2.3.1 Shared-memory architecture

In a shared-memory architecture, all processors share a common memory address space and implicitly communicate between themselves via memory. Usually they have local caches and are interconnected not only with each other, but also with the common memory through an interconnection (a bus, for example) [21]. In general such architectures are

symmetric, which means that all processing units are equal, nevertheless they also might be asymmetric.

In Figure 2.2 a symmetric shared-memory architecture is schematized. Despite the fact that in that figure all processors are connected to a single memory module, there might be several memory modules. Employing a bank of memories increases the overall computational performance since only one processor may access a given memory at a given time. That is, a bank of memories allows the simultaneously execution of several read/write memory operations [10].

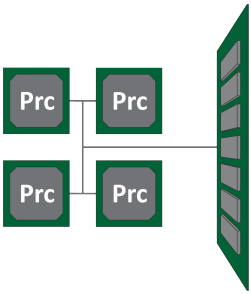


Figure 2.2 – Ilustration of a shared memory parallel computer.

According to [10], programming for shared-memory multiprocessors is not difficult, since all memory read operations are hidden from the programmer and hence can be coded as a serial code. Memory write operations, on the other hand, might require locking the data access until a thread has finished the work on it. It is necessary to identify the critical code sections and synchronize the processors in order to assure data integrity. Libraries based on OpenMP directives – discussed later – are commonly used to handle synchronization and other related operations.

The main disadvantage regarding shared-memory architectures is their incapability of scaling to a large number of processors [7]. Generally the bus-based systems are limited to at most 32 processors, while those based on crossbar switch can achieve as many as 128 processors. However, in the latter case the switch cost increases with the square of the number of processors. If a computer with a exceedingly large number of processors were to be built, this incremental cost would make its construction unfeasible.

2.3.2 Distributed-memory architecture

Scalability limitations of shared memory systems led the development of distributed-memory parallel computers. Instead of a huge global memory, each processor owned by that machine is connected to a smaller local memory, so that the memory access can be done faster than a shared-memory computer would be able to perform it [7]. This structure is also classified as non-uniform – Non-uniform Memory Access (NUMA) –, since it depends on which memory a given processor attempts to access [10].

Distributed-memory computers resort to an interconnection network in order to provide an adequate communication among the processors, as illustrated in Figure 2.3. Data are sent from a processor memory module to another via a message passing (MP) mechanism. The Massage Passing Interface (MPI) may be used as a language-independent message protocol [7]. Aiming the improvement of the overall computational performance, data should be carefully placed in the memory modules in order to lessen the number of messages exchanged between the processors themselves.

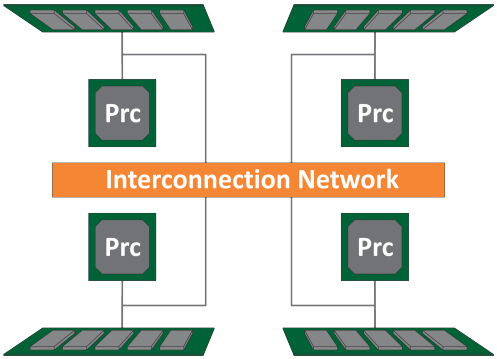


Figure 2.3 – Ilustration of a shared memory parallel computer.

2.3.3 Hybrid-memory architecture

Hybrid-memory architecture combines paradigms from shared and distributed memory. In massively parallel processing this architecture is of-

ten termed *SMP cluster*. Its structure is similar to a distributed-memory's arrangement, although in the hybrid case each node is a shared-memory system. This configuration takes advantage of both memory architectures, since it allows not only high parallel efficiency within a node but also scaling the program to a large number of processors.

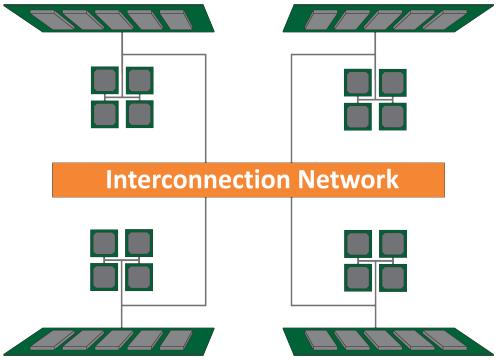


Figure 2.4 – Ilustration of a shared memory parallel computer.

2.4 Application Programming Interfaces

An Application Programming Interface (API) is a set of routines, protocols, and tools that assist the development of software applications. For parallel softwares, the most common APIs are the Open Multi Processing (OpenMP) and the Message Passing Interface (MPI) [8]. The former is used in shared-memory parallel computers; the latter, in both shared and distruted-memory machines. Besides, those APIs can be used together in hybrid memory computers, as discussed in the previous section. In the following subsections each one of them explained.

2.4.1 OpenMP

OpenMP is an API employed in shared-memory computers. It is composed by a set of compiler directives and environmental variables. Additionally, it also includes a runtime library, which may be used to implement the desired parallelism. OpenMP is suitable to parallelize sequential programs implemented in C, C++ or Fortran [21].

OpenMP is considered a high level API. This means that the programmer does not need to worry about too many technical details, such as data decomposition and flow control, which are left to the compiler. The programmer only needs to use OpenMP directives to indicate what portions of the code must be executed in parallel. If a compiler does not support OpenMP, then the directives will be interpreted as comments and thus ignored. Hence, the application is simultaneously sequential and parallel.

OpenMP does not require a whole parallelized code. That is, it allows an incrementally parallelization. One may progressively parallelize a sequential code until the desired performance is achieved. It is also worth noting that the collection of OpenMP directives is relatively small, which means that the programmer does not need to learn a whole new language in order to use them [21].

2.4.2 MPI

The Message Passing Interface (MPI) is proposed as a standard specification for message-passing. Nowadays MPI is the leading programming language employed in highly scalable programs [21]. It treats communications among processors explicitly and for this reason may be used in both shared and distributed-memory parallel computers.

In the message-passing model, the processors communicate between each other by sending messages through a two-sided operation: a processor must send a message, while another one needs to recieve it. A MPI message consist of two parts: the envelope and the message body [20]. The envolpe has four parts:

- **Source:** the processor that sends the message;
- **Destination:** the processor that receives the message;
- **Communicator:** the group of processors to which both source and destination processors belong;
- **Tag:** marker used to distinguish between different message types.

The message body, by its turn, is composed of the three following parts:

- **Buffer:** the data to be sent;
- **Datatype:** type of the message data;

- **Count:** number of items in the buffer.

Almost all message-passing operations can be done using send/recieve operations (point-to-point operations). However, some operations that involve all processors. This kind of operation is common to such a degree that MPI provides routines (collective communication routines) to execute them. Some of these routines are [20]:

- Barrier synchronization;
- Broadcast from one processor to all the other ones;
- Global reduction operations, such as *max*, *min*, and *sum*;
- Gather data from all processor to a single one;
- Scatter data from a single processor to all the other ones.

2.5 Parallel efficiency and theoretical limits

One of the most common metric to measure the benefits of parallel computing is the *speedup*. The speedup is simply defined as the ratio of the times required to run a program with a single and multiple parallel processors. If T_1 and T_N denotes the time consumed by one and by N processors, respectively, then the speedup achieved is

$$S_N = \frac{T_1}{T_N}. \tag{2.1}$$

Ideally, if the code is fully parallelized, there is no overloaded processor. Furthermore, in the ideal case the communication time between processors and memory are negligible, the speedup is linear [10], which means

$$S_N = N. \tag{2.2}$$

In general, however, the speedup is sub-linear, since the conditions above rarely are respected. There are situations, on the other hand, in which the speedup is greater than the linear speedup (super-linear speedup). This might happen, for example, if searching operations are important or due to hardware issues [8].

Another common metric used for parallel computing is the parallel efficiency. It is defined as the speedup divided by the number of processors [21]:

$$E_N = \frac{S_N}{N}. \tag{2.3}$$

The parallel efficiency may be used to measure the scalability of a program. Given a chosen level of efficiency and a number of processors, it apprises the size of the problem to be solved.

There are theoretical limits for the speedup. According to Amdahl's Law, an algorithm is composed by a parallizable fraction f and by a serial fraction $1-f$. Considering that the execution of the parallizable fraction is N times faster when N processors are used, the time needed for the code execution is

$$T_N = f\frac{T_1}{N} + (1-f)T_1. \tag{2.4}$$

The theoretical speedup is thus

$$S_N = \frac{T_1}{T_N} = \frac{1}{f/N + (1-f)} \tag{2.5}$$

and the maximum speedup according to Amdahl's Law is

$$S_{\max} = \lim_{N\to\infty} S_N = \frac{1}{1-f}. \tag{2.6}$$

Figure 2.5 displays some speedup's curves and its behavior due to variations in the number of processors and parallelizable fractions.

Another popular law for the theoretical speedup is the Gustafson-Barsis's. It states that the parallelism increases as the problem size increases [10]. This law differs from the Amdahl's Law, in which the parallel fraction is constant. In the Gustafson-Barsis's formula, the time of execution in parallel is taken as the reference. The time for execution in serial is thus

$$T_1 = (1-f)T_N + fNT_N, \tag{2.7}$$

which gives the theoretical speedup

$$S_N = 1 + f(N-1). \tag{2.8}$$

Figure 2.6 exhibits the theoretical speedup for the same parallel fractions

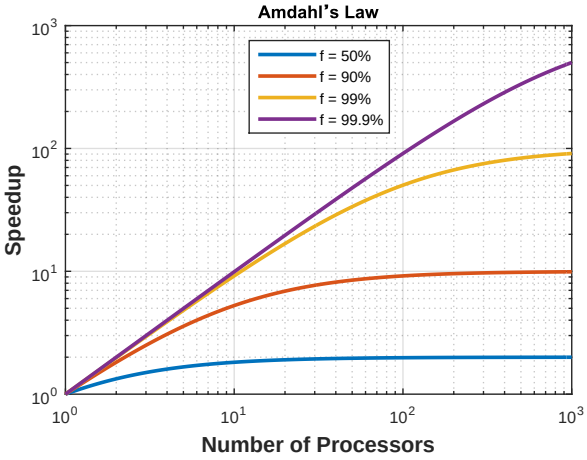


Figure 2.5 – Theoretical speedup according to Amdahl's Law.

used in Figure 2.5. It is important to point out that the Gustafson-Barsis's Law is much less pessimistic than Amdahl's.

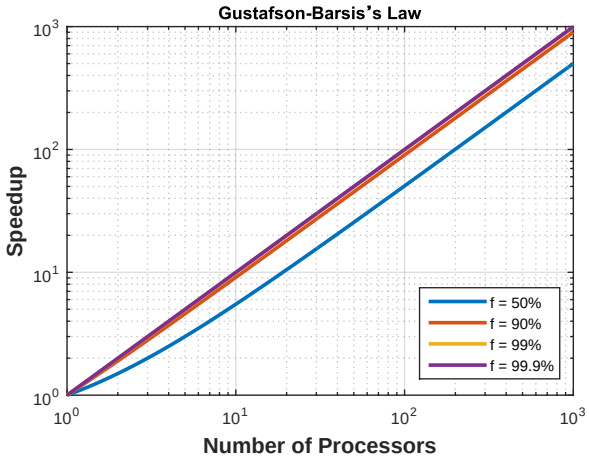


Figure 2.6 – Theoretical speedup according to Gustafson-Barsis's Law.

CHAPTER

3

UTCHEM Reservoir Simulator

3.1 Introduction

UTCHEM is a three-dimensional, multicomponent, multiphase, compositional, variable temperature, finite-difference reservoir simulator developed at The University of Texas at Austin. It can be used to simulate enhanced recovery of oil and enhanced remediation of aquifers. Some of its features are the modeling of capillary pressures, three-phase relative permeabilities, dispersion, diffusion, adsorption, chemical reactions, and non-equilibrium mass transfer between phases [2]. Moreover, for ground water its features are:

- NAPL spill and migration in both saturated and unsaturated zones;
- Partitioning interwell test in both saturated and unsaturated zones of aquifers;
- Remediation using surfactant/cosolvent/polymer;
- Remediation using surfactant/foam;
- Remediation using cosolvents;

- Bioremediation;
- Geochemical reactions.

For oil reservoirs:

- Waterflooding
- Single well, partitioning interwell, and single well wettability tracer tests;
- Polymer flooding;
- Profile control using gel;
- Surfactant flooding;
- High pH alkaline flooding;
- Microbial EOR;
- Surfactant/foam and ASP/foam EOR.

In UTCHEM, the mass and energy equations are solved for an arbitrary number of chemical components. There may be up to four fluid phases – air, water, oil, and microemulsion – besides an arbitrary number of solid minerals. The transport equations are discretized using a finite difference scheme and are solved via an IMPEC method, which solves the aqueous phase pressure equation implicitly and the concentration equations explicitly. In the following section the UTCHEM mathematical model will be briefly described following the guidelines presented in [12].

3.2 Mathematical formulation

The main equations to be solved are the aqueous phase pressure, the concentration, and the energy equations. Moreover, the aqueous phase pressure equation is the only one solved implicitly. It is obtained by summing all the concentration equations of the volume-occupying components. The pressure of the other phases is computed adding the capillary pressures between the other phases. The knowledge of the pressure field is a sufficient requirement in order to solve all the other equations.

The mass equation of a component κ in a porous media is given by

$$\frac{\partial}{\partial t}\left(\phi\,\tilde{C}_\kappa\rho_\kappa\right)+\nabla\cdot\left[\sum_{l=1}^{n_p}\rho_\kappa\left(C_{\kappa,l}\,\vec{u}_l-\vec{D}_{\kappa,l}\right)\right]=\bar{R}_\kappa, \tag{3.1}$$

where $\tilde{C}_\kappa$ is the concentration of component κ over all phases, including the adsorbed ones. Mathematically,

$$\tilde{C}_\kappa=\left(1-\sum_{c=1}^{n_{vc}}\hat{C}_c\right)\sum_{l=1}^{n_p}S_l\,C_{l\kappa}+\hat{C}_\kappa, \tag{3.2}$$

where n_p is the number of phases; n_{vc} is the number of volume-occupying phases, S_l is the saturation of phase l, and $C_{l\kappa}$ is the concentration of component κ in phase l, and $\hat{C}$ is the adsorbed concentration. In Equation (3.1), ϕ is the media porosity. It is assumed that it changes linearly with the pressure according to the expression

$$\phi=\phi_0(1+C_r(P-P_0)), \tag{3.3}$$

where ϕ_0 is the referential porosity, evaluated at pressure P_0, and C_r is the rock compressibility, which is supposed to be constant.

Still referencing Equation (3.1), ρ_κ represents the ratio between the density of the pure component κ at reservoir conditions and its density at standard conditions. It is modelled according to the expression

$$\rho_\kappa=1+C_\kappa^o(P-P_{\text{std}}), \tag{3.4}$$

where C_κ^o is the component compressibility, also supposed constant. $\vec{D}_{\kappa,l}$ is the dispersive flux, which is assumed to have a Fickian form, while $\vec{u}_l$ is the superficial velocity. It is assumed that the superficial velocity obeys Darcy's Law:

$$\vec{u}_l=-\lambda_l\mathbb{K}\cdot(\nabla P_l-\gamma_l\nabla h). \tag{3.5}$$

λ_l is the phase mobility, given by the expression,

$$\lambda_l=\frac{k_{r,l}}{\mu_l}, \tag{3.6}$$

where $k_{r,l}$ is the phase relative mobility and μ is the phase viscosity, $\mathbb{K}$

is the medium absolute permeability, P_l is the phase pressure, γ_l is the phase specific gravity, and h is the vertical depth.

The source term $\bar{R}_\kappa$ in Equation (3.1) is composed by a the rate of injection/production of component κ plus the rate of consumption/production of κ in chemical reactions. It is expressed by

$$\bar{R}_\kappa = \phi \sum_{l=1}^{n_p} S_l r_{\kappa l} + (1-\phi) r_{\kappa s} + \bar{Q}_\kappa. \tag{3.7}$$

$r_{\kappa l}$ and $r_{\kappa s}$ are the reaction rates in phase l and in the solid phase. $\bar{Q}_\kappa$, on the other hand, is the rate of production/injection of κ due to neighbouring wells.

Summing Equation (3.1) for each volume-occupying component the component concentrations cancel out. The resulting equation is, then:

$$\phi_{\text{ref}} C_t \frac{\partial P_1}{\partial t} + \nabla\cdot(\lambda_{Tc}\mathbb{K}\cdot\nabla P_1) = \nabla\cdot\left[\sum_{l=1}^{n_p} \lambda_{lc}\mathbb{K}\cdot(\nabla P_{cp1} - \gamma_l \nabla h)\right] + \sum_{c=1}^{n_{vc}} \bar{Q}_c. \tag{3.8}$$

In the above equation λ_{lc} includes the correction for fluid compressibility. It is given by

$$\lambda_{lc} = \frac{k_{rl}}{\mu_l} \sum_{\kappa=1}^{n_{vc}} \rho_\kappa C_{\kappa l}. \tag{3.9}$$

The total relative mobility by its turn is given by

$$\lambda_{Tc} = \sum_{l=1}^{n_p} \lambda_{lc}. \tag{3.10}$$

Finally, the total compressibility C_t is the volume-weighted sum of the rock (C_r) and component (C_κ^o) compressibilities:

$$C_t = C_r + \sum_{\kappa=1}^{n_{vc}} C_\kappa^o \tilde{C}_\kappa \tag{3.11}$$

The main equations to be solved are the (3.1) and the (3.8). Equation (3.8) is solved implicitly, which implies that a linear system must be solved in order to obtain the aqueous phase pressure field. From those calculations arises the solution of Equation (3.1) for each component, solution

from which the concentration fields are obtained. The knowledge of the pressure and concentrations enables determining all the other variables. It's noteworthy that a discourse about physical and chemical models implemented in UTCHEM would be an out-of-scope topic for this study. Hence, it'll not be included in this description. For a complete description of UTCHEM and its features it is recommended to consult [2].

3.3 Grids

Despite UTCHEM having a version supporting unstructured grids, structured grids are still the most used kind of grid in this simulator. The UTCHEM's version received for this work had originally three types of coordinate system: Cartesian, radial, and curvilinear. The radial system was implemented to study the flow near the wellbore. The flow is assumed to be plenty radial and thus the domain is divided only in the radial direction r and in the vertical direction z. This type of coordinate system does not take advantage of the parallelization of the code because the division of the grid described in Chapter 5 is in the y direction, which is not associated to any of the radial system axes. The curvilinear coordinate system implemented in UTCHEM is basically a two dimensional curvilinear grid in the $x-z$ plane extruded in the direction y. Besides these three types of grids, an implementation of a corner-point grid was moved from another version of UTCHEM to the version of this work and it was further parallelized.

CHAPTER

4

The Element-based Finite Volume Method

4.1 Introduction

Finite Volume Methods (FVM) are numerical methods used for solving differential equations derived from the application of conservation laws. The customary approach to deduce those equations consists in applying a balance of a given property to a control volume. Then, assuming that its size tends to zero leads to a differential equation. Instead of using the differential equations directly, the Finite Volume Method takes a step back and utilize the prior balance equations. This procedure assures the method's conservative character.

The Element-based Finite Volume Method (EbFVM) is a Finite Volume Method that applies some Finite Element Method (FEM) concepts in order to provide to provide more geometrical flexibility to the Finite Volume Method. It allows applying the control volume approach to a unstructured grid. This study intends to use triangular and quadrangular

elements in two-dimensional grids and tetrahedron, hexahedron, square-based pyramid, and triangular based prism elements in three-di-mensional grids. These grids are unstructured, which means that there is no pre-defined rule to associate an element with its neighbors [16]. Figure 4.1 shows how such grids may provide a good discretization of geometrically complex reservoirs.

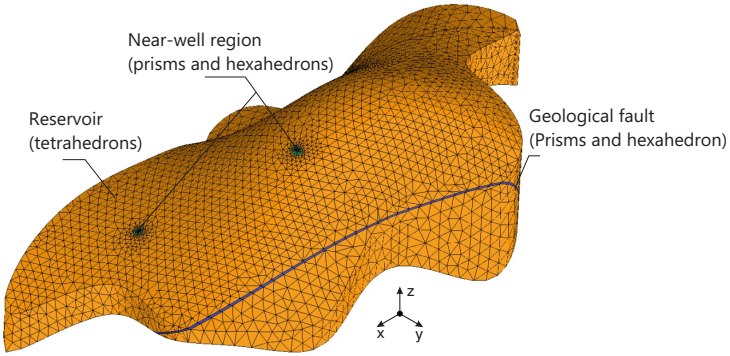


Figure 4.1 – Discretization of an hypothetical reservoir using a unstructured grid (adapted from [17])

4.2 Grid entities

A grid is a collection of geometrical entities Ω_i such that for a domain Ω

$$\bigcup_i \Omega_i = \Omega. \tag{4.1}$$

Each Ω_i has a nonempty interior, but the interior of the intersection of Ω_i and Ω_j, with $i \neq j$, is empty [11]. The entities Ω_i are called the elements of the grid [16]. Here, it is considered that in two-dimensional domains the elements may be triangles or quadrilaterals. In three-dimen-sional domains, on the other hand, there may be tetrahedra, hexahedra, square-based pyramids, or triangle based prisms. If a grid has elements of different types, the grid is called hybrid. One may note that this broadens the

scope of Cartesian grids, in which only rectangles and parallelepipeds are used.

The contour of the elements is formed by entities here called facets. Facets are edges in 2D elements and surfaces in 3D elements. If a facet is not at the boundary of the domain, then always exist two elements sharing that facet. Hence, it is not possible partial contact between facets, that is, the grid is conformal. A more precise definition of conformal grids may be found in [11].

The term face are let to geometrical entities that delimit the control volumes. In 2D they are the segments that connect the edges's midpoints to the element centroids, as illustrated in Figure 4.2. In 3D the definition is analogous and may be found in [11]. The faces divide the element into smaller regions called subelements, each one associated to a vertex. Figure 4.2 illustrates the main grid entities of a triangular element.

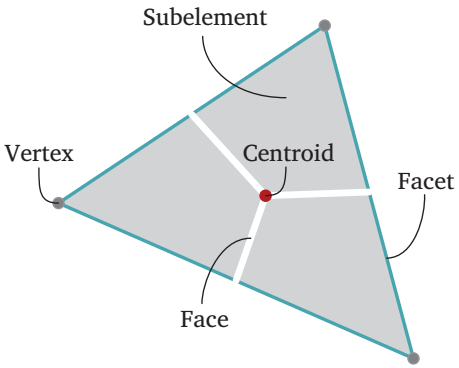


Figure 4.2 – Main entities of a triangular element

In the EbFVM, the unknowns of the problem are associated to elements' vertices, also named nodes. The control volumes are assembled from subelements surrounding the nodes, as illustrated in Figure 4.3. As a consequence, the control surfaces are formed by the faces themselves. Regarding that geometrical arrangement, it is quite convenient to suppose that physical properties such as permeability are homogeneous inside an element. Therefore, no interpolation scheme is required to evaluate them at the surface of the control volumes [6].

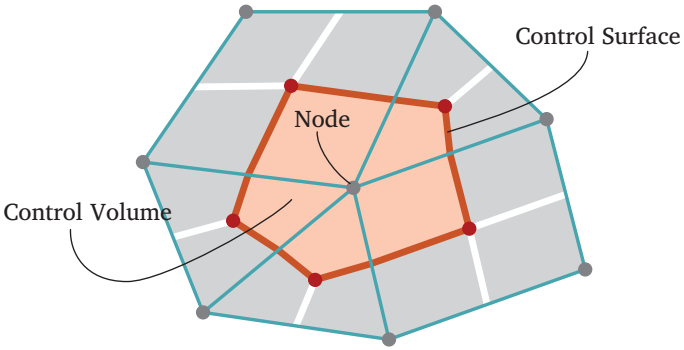


Figure 4.3 – Control volume creation

4.3 Numerical scheme

An EbFVM common approach to handling geometrical distortion of the elements in a grid is the transformation known as mapping. It maps each element from the global coordinate system to a local coordinate system in which the element has a regular representation. Figure 4.4 illustrates a triangle element being mapped to a transformed space represented by the coordinates (ξ, η). As it will be demonstrated later, this procedure simplifies the discretization of the modelling equations.

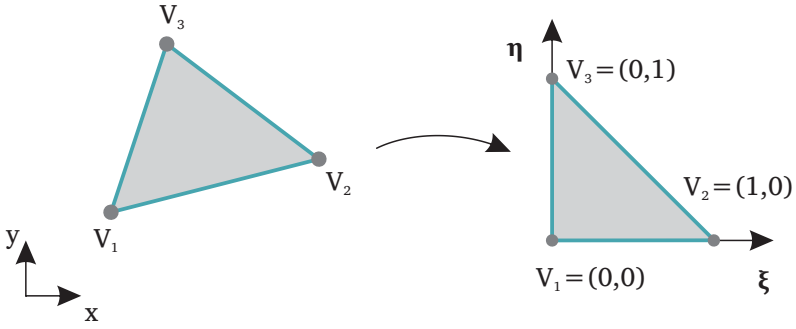


Figure 4.4 – Mapping into a transformed space

The mapping from global to local coordinate systems are performed using first order shape functions from the FEM. For every vertex i of an element there is a shape function $\mathcal{N}_i$ which is 1 at i and 0 at all the other

vertices. In such manner, each point $p=(x,y,z)$ may be represented as

$$p=\sum_{i=1}^{n_v}\mathscr{N}_i(\xi,\eta,\zeta)\,p_i, \tag{4.2}$$

where p_i, $i=1,...,n_v$, represents the element's vertices, n_v the number of vertices, and (ξ,η,ζ) the local coordinates of P. The shape functions must be continuous, differentiable, and partitions of the unity, so that they are positive and obey the relation

$$\sum_{i=1}^{n_v}\mathscr{N}_i(\xi,\eta,\zeta)=1 \tag{4.3}$$

for any point (ξ,η,ζ) in the local coordinate system [24].

Let φ be a function defined over the grid nodes (vertices of the elements). The EbFVM proposes that the value of φ inside an element is given by the relation

$$\varphi(x,y,z)=\sum_{i=1}^{n_v}\mathscr{N}_i(\xi,\eta,\zeta)\,\varphi_i. \tag{4.4}$$

Analyzing the properties of the shape functions shown above, one may realize that the value of φ inside an element is an average of its values at the element vertices and the weight is given by the shape function. This is actually consistent, once φ will not be higher than the maximum value at a vertex nor lower than its minimum value.

Since shape functions are differentiable, the gradient of φ in the global coordinate system may be expressed as

$$\nabla\varphi=\nabla\left(\sum_{i=1}^{n_v}\mathscr{N}_i(\xi,\eta,\zeta)\,\varphi_i\right)=\sum_{i=1}^{n_v}\begin{pmatrix}\partial_x\mathscr{N}_i\\ \partial_y\mathscr{N}_i\\ \partial_z\mathscr{N}_i\end{pmatrix}\varphi_i=\begin{pmatrix}\partial_x\mathscr{N}_1 & \partial_x\mathscr{N}_2 & \cdots & \partial_x\mathscr{N}_{n_v}\\ \partial_y\mathscr{N}_1 & \partial_y\mathscr{N}_2 & \cdots & \partial_y\mathscr{N}_{n_v}\\ \partial_z\mathscr{N}_1 & \partial_z\mathscr{N}_2 & \cdots & \partial_z\mathscr{N}_{n_v}\end{pmatrix}\Phi_e, \tag{4.5}$$

where

$$\Phi_e=\begin{pmatrix}\varphi_1 & \varphi_2 & \cdots & \varphi_{n_v}\end{pmatrix}^T \tag{4.6}$$

is a vector with the values of φ at the element vertices. However, the shape functions are usually given in terms of the local coordinates and thus the derivatives of $\mathscr{N}_i$ with respect to the global coordinates are inconvenient. Applying the chain rule,

$$\begin{pmatrix} \partial_\xi \mathscr{N}_i \\ \partial_\eta \mathscr{N}_i \\ \partial_\zeta \mathscr{N}_i \end{pmatrix} = \begin{pmatrix} \partial_\xi x & \partial_\xi y & \partial_\xi z \\ \partial_\eta x & \partial_\eta y & \partial_\eta z \\ \partial_\zeta x & \partial_\zeta y & \partial_\zeta z \end{pmatrix} \begin{pmatrix} \partial_x \mathscr{N}_i \\ \partial_y \mathscr{N}_i \\ \partial_z \mathscr{N}_i \end{pmatrix}. \tag{4.7}$$

The matrix

$$J = \begin{pmatrix} \partial_\xi x & \partial_\xi y & \partial_\xi z \\ \partial_\eta x & \partial_\eta y & \partial_\eta z \\ \partial_\zeta x & \partial_\zeta y & \partial_\zeta z \end{pmatrix} \tag{4.8}$$

is the well-known Jacobian matrix [11] and may be computed using Equation (4.2). Defining

$$D \equiv \begin{pmatrix} \partial_\xi \mathscr{N}_1 & \partial_\xi \mathscr{N}_2 & \cdots & \partial_\xi \mathscr{N}_{n_v} \\ \partial_\eta \mathscr{N}_1 & \partial_\eta \mathscr{N}_2 & \cdots & \partial_\eta \mathscr{N}_{n_v} \\ \partial_\zeta \mathscr{N}_1 & \partial_\zeta \mathscr{N}_2 & \cdots & \partial_\zeta \mathscr{N}_{n_v} \end{pmatrix}, \tag{4.9}$$

Equation (4.5) may be expressed as

$$\nabla \varphi = J^{-1} D \, \Phi_e. \tag{4.10}$$

Matrix $J^{-1}D$ may be interpreted as a discrete gradient operator.

4.4 Discretization of a conservation equation

According to [16], the conservation equation of a generic property φ associated to a fluid flowing may be written as

$$\frac{\partial}{\partial t}(\rho\varphi) + \nabla\cdot(\rho \mathbf{u}\varphi) = \nabla\cdot(\Gamma \nabla \varphi) + Q, \tag{4.11}$$

where ρ is the specific mass, $\mathbf{u}$ is the fluid velocity, Γ is a diffusivity coefficient, and Q is a source term associated to the property φ. The discretization in a FVM is performed integrating the above equation in each control volume. Consider a control volume Ψ as the one illustrated in Figure 4.3.

Noting that the grid is static, the first integral becomes

$$\int_{\Psi} \frac{\partial}{\partial t}(\rho\varphi)\,\mathrm{d}V = \frac{\partial}{\partial t}\left(\int_{\Psi} \rho\varphi\right) = \frac{\partial}{\partial t}\left(M_{\Psi}\bar{\varphi}_{\Psi}\right) \approx \frac{\partial}{\partial t}\left(M_{\Psi}\varphi_{\Psi,n}\right), \tag{4.12}$$

where M_{Ψ} is the mass contained in Ψ and $\bar{\varphi}_{\Psi}$ is the average value of φ in Ψ. Such average value was approximated by the value of φ at the node itself, which is a reasonable approximation since the node is usually closed to the center of the control volume. The discretization of the source term is similar and gives

$$\int_{\Psi} Q\,\mathrm{d}V \approx Q_{\Psi,n}V_{\Psi}, \tag{4.13}$$

noting that V_{Ψ} is the volume of Ψ.

The second term of Equation (4.11) models the advective transportation of φ. Integrating and applying the divergence theorem

$$\int_{\Psi} \nabla\cdot\left(\rho\mathbf{u}\varphi\right)\mathrm{d}V = \int_{\partial\Psi} (\rho\mathbf{u}\cdot\hat{\mathbf{n}})\varphi\,\mathrm{d}A = \sum_{f}\int_{\partial\Psi_f} (\rho\mathbf{u}\cdot\hat{\mathbf{n}})\varphi\,\mathrm{d}A. \tag{4.14}$$

Approximating the average value of the integral above by the value of the center of each face result

$$\int_{\Psi} \nabla\cdot\left(\rho\mathbf{u}\varphi\right)\mathrm{d}V \approx \sum_{f}\left(\rho\mathbf{u}\cdot\Delta\mathbf{A}_f\right)\varphi_f = \sum_{f}\dot{m}_f\varphi_f, \tag{4.15}$$

where $\dot{m}_f$ is the mass flow rate crossing the face f. It would be intuitive to evaluate φ_f using Equation (4.4). However, this is usually not recommended because the method may become numerically unstable. Upwind-like methods are preferably, despite they may introduce numerical diffusion to the solution obtained [16].

The last term from Equation (4.11) to be integrated is the diffusive term. Integrating and applying the divergence theorem again:

$$\int_{\Psi} \Gamma\nabla\varphi\,\mathrm{d}V = \int_{\Psi} (\Gamma\nabla\varphi)\cdot\hat{\mathbf{n}}\mathrm{d}A \approx \sum_{f}(\Gamma\nabla\varphi)_f\cdot\Delta\mathbf{A}_f. \tag{4.16}$$

Using the discrete gradient operator defined in Equation (4.10),

$$\int_{\Psi} \Gamma \nabla \varphi \, \mathrm{d}V \approx \sum_{f} \left(\Gamma J^{-1} D \, \Phi_e \right)_f \cdot \Delta \mathbf{A}_f . \tag{4.17}$$

The stencil of a numerical method is defined as the set that contains all the nodes that are involved in the discrete equation of a given node. Regarding the four terms of Equation (4.11) discretized previously, only two of them use values at neighbor nodes: the advection – Equation (4.15) – and the diffusive – Equation (4.17) – terms. According to those equations, in the EbFVM context, the stencil is all the nodes that share an element with the given node. Figure 4.5 illustrates the stencil of a general node p.

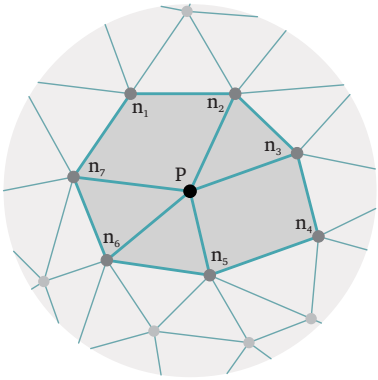


Figure 4.5 – Stencil of p

CHAPTER

5

The Proposed Approach

5.1 UTCHEM

Aiming a performance improvement, the methodology adopted in the parallelization process must lay its focus on UTCHEM's operations that require the most significant part of the computational efforts. Evidently, refined grids employed in large and/or complex domain simulations are a relevant matter. This fact motivated employing a domain decomposition based methodology. The reservoir grid is divided among the available processors. Each one of them works in only a part of the domain, while MPI routines are in charge of the communication. Since all implementations are based on MPI routines, the code is fully functional for the most general parallel architecture: hybrid-memory architecture. As a consequence, UTCHEM can run in parallel not only in CPU clusters, but also in personal computers. The use of the IPARS Framework (described in Section 5.1.3) made viable the introduction of a new input file format for the simulator. This format is more user friendly and error prone, as described in Appendix A.

5.1.1 Domain decomposition

UTCHEM's parallelization applies only to structured grids. A structured grid has such an organization that is completely defined by the number of grid blocks on each coordinate direction and their dimension. Here it is supposed, without losing generality, that the grid is Cartesian and we denote the number of grid blocks on each direction by N_x, N_y, and N_z, related to the directions x, y, and z, respectively.

Seemingly, the main target in parallelization is improving computational performance. In order to achieve that goal, it's crucial to decompose the domain in such a way that each processor take care of almost the same number of grid blocks. That is, overloaded processors must be avoided. Furthermore, dividing a structured grid is much easier than an unstructured one, since its topology is strictly dependent of the coordinate system. To each block is associated a topological coordinate (i, j, k), which represents its localization indexes along the directions x,y and z, respectively. The division applied in this study occurs, in fact, along the directions y and z (does not depend on x). The pivotal reference in the decomposition procedure is the j direction. The k one, on its turn, is regarded as a layer. For each layer, then, is necessary to define the range in the j direction. This can be done by two numbers, namely j_{P^-} and j_{P^+} in the following way: if a grid block S has topological coordinates (i_S, j_S, k_S) and $j_{P^-} \leq j_S \leq j_{P^+}$, then S is in the domain of processor P. One may note that this procedure could be simplified by limiting the division solely to the y direction. This is not recommended since depending on N_y and on the number of processors the load balance might be not good enough. Hence, the values of j_{P^-} and j_{P^+} may change according to the topological coordinate k. For example, if a 3x3x2 case will be simulated with two processors, the first processor may take the first two grid blocks of the y direction in the first horizontal layer, but only one in the last one. The grid thus is equally divided: each processor works on nine grid blocks. Figure 5.1 exemplifies a grid division.

In UTCHEM there is an important variable named NBL. It was used to store the total number of grid blocks. In the Parallel UTCHEM, this would not make sense anymore, given that each processor sees only a part of the reservoir domain. Now NBL stores the number of local active grid blocks (inactive grid blocks will be discussed in the following section). The size of

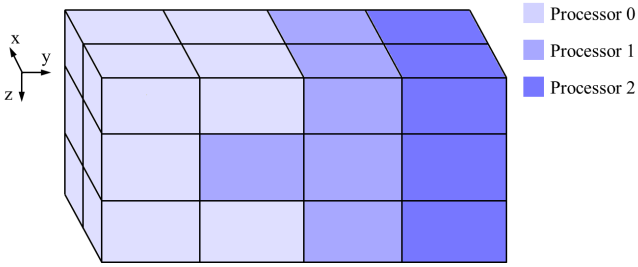


Figure 5.1 – Example of a grid division

grid-dependent arrays is set to be precisely NBL. As a consequence, grid-related operations do not need to be modified at all. There is however an exception: if a grid-related array is used in a stencil computation (computation that requires values at neighbor grid blocks), values at grid blocks that do not belong to the current processor domain may be needed. For such variables a continuous indexing such that the one that starts in 1 and ends in NBL is not feasible. In this case, a coordinate-based indexing was applied: each processor domain is extended so that it contains grid blocks of other processor domains. The additional grid blocks are called ghost cells and no computation is performed for them. They are used only to store values from other processor domains. Values at ghost cells are updated using MPI routines every time stencil computations are performed. In Figure 5.2 ghost cells are illustrated.

Grid-dependent arrays with continuum indexation will be labeled here as type 1, while the others will be labeled as type 2. The indexes of a type 2 array are named I1, J1, and K1, which are related to the directions x, y, and z. The range of I1 is from IL1 to IL2, while the range of K1 is from KL1 to KL2. For J1 the range is from JL1V(K1) to JL2V(K1). One must note that the range of J1 depends on the local coordinate K1, reflecting the nature of the grid division. Because of the changing in the indexation of some grid-dependent arrays, every loop of UTCHEM in which type 2 arrays are used must be changed from continuum to coordinate based, as illustrated in Figure 5.3. If the continuum index is needed but the loop is coordinate-based, then the continuum index may be accessed using the variable IJKPOS. Due to inactive grid blocks, explained in the following section, the opposite can not be performed: access the coordinate indexes I1, J1, and K1 from a continuum index I.

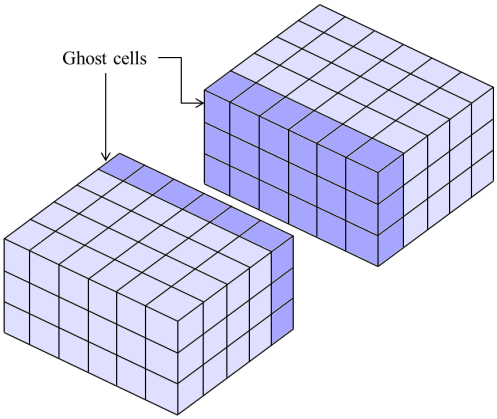


Figure 5.2 – Example of ghost cells from a 6x8x3 grid

```
  DO 1 I = 1,NBL
     …
1 CONTINUE
```

→

```
  DO 1 K1 = KL1,KL2
  DO 1 J1 = JL1V(K1),JL2V(K1)
  DO 1 I1 = IL1,IL2
    IF (KEYOUT(I1,J1,K1).NE.0) THEN
      I = IJKPOS(I1,J1,K1)
       …
    ENDIF
1 CONTINUE
```

Figure 5.3 – Loop modification when type 2 arrays are presented

5.1.2 Inactive grid blocks

Inactive grid blocks are characterized by very low porosity and permeability such that the fluid flow through them is negligible. They facilitate an accurate description of a reservoir geometry using a structured grid, such as a cartesian or a corner-point grid. Figure 5.4 illustrates a case in which some grid blocks were set as inactive (those in grey), adjusting the grid's geometry to the reservoir's shape. In UTCHEM's previous versions, inactive grid blocks were kept in almost all computation. It was assigned to them small values – but not null, due to convergence problems – of porosity and permeability and they were supposed to be fully saturated by water. Sometimes, however, using them in computations may be physically inconsistent (e.g. computation of concentration derivatives).

Along the parallelization process, the same approach used in [8] was

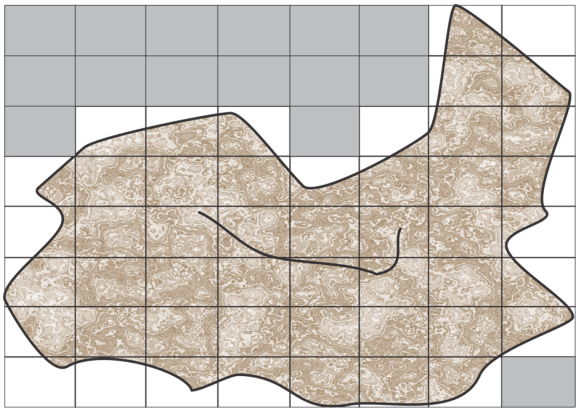


Figure 5.4 – Example of inactive grid blocks been used to better describe a reservoir domain

implemented: excluding inactive grid blocks from all computations. This is automatically done in type 1 arrays because their size is equal to the number of active grid blocks (inactive grid blocks do not have a continuum index). For type 2 arrays, on the other hand, a variable named KEYOUT is used to indicate if a grid block is active or not. KEYOUT is a type 2 array that has the value 1 at active grid blocks, 0 at inactive grid blocks, and -1 at ghost cells. If a grid block has KEYOUT 1, then it is an active grid block. If KEYOUT is -1 the grid block is a ghost cell and if KEYOUT is 0 the grid block is inactive. Hence, before each computation the value of KEYOUT is verified, as described in Figure 5.3. For stencil computations, inactive grid blocks are treated as impermeable boundaries and a proper methodology is applied in every case. For example, if a concentration derivative needs to be computed but there is a neighbor grid block that is inactive, a central difference scheme will substituted by a forward or backward difference scheme, as it is performed near to the reservoir boundaries.

5.1.3 IPARS framework

UTCHEM was parallelized with support of the IPARS framework. IPARS provides several routines that facilitate good strategy execution of parallel

related operations. For example, it has routines to divides the grid according to the number of processors, update type 2 array values at ghost cells, and measure the execution time expended by each processor. Figure 5.5 delineates the organization of the parallel version of UTCHEM, which for now on will be named UTCHEMP.

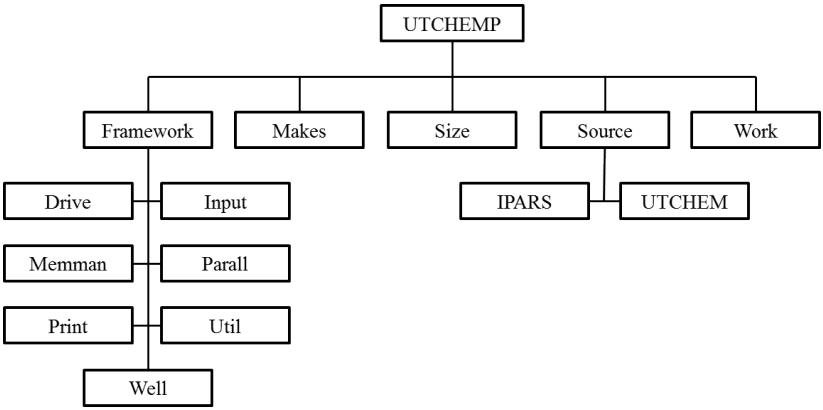


Figure 5.5 – Organization of UTCHEMP

The main routines of IPARS framework are placed in the folder Framework. The folder Drive, inside Framework, contains the main subroutine, called IPARS, which directly or indirectly calls all the other subroutines. In folder Input there are subroutines to manage the reading, and the initialization of the simulation. Memman, on the other hand, contains several functions implemented in C++ that are used by the IPARS framework. The types of the subroutines presented in the folders Parall, Print, Well, and Util are straightforward: subroutines for communication between processors, to export results, to help well related operations, and some of general purpose. In folder IPARS, inside folder Source, there are some additional IPARS subroutines that are used mainly to initialize the simulation. On the other side, the folder UTCHEM contains almost all subroutines that were previously presented in UTCHEM, as well as some new ones. Since now the simulation is driven by the IPARS framework, some operations were moved from subroutines originally in UTCHEM to subroutines of the IPARS framework. For example, the part of the subroutines INOUT that were used to read geometry data were moved to a

subroutine from the folder Input because such data is needed before the calling of the subroutine that divides the grid. Thus, INOUT now executes less operations that it did before. Finally, the folders Makes, Size, and Work have files for the code compilation.

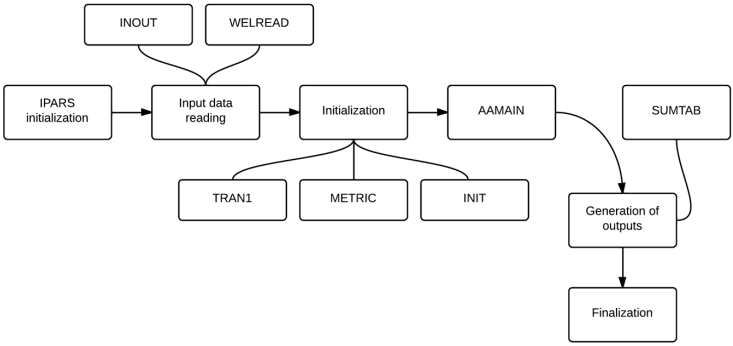


Figure 5.6 – Simulator workflow

The algorithm workflow of UTCHEMP is sketched in Figure 5.6. At first the IPARS framework is initialized and the input files are read. Almost all data is read by the subroutine INOUT or its inner subroutines. An exception to this are the geometry data reading, which occurs before INOUT is called, and the well data reading, which is performed by the subroutines WELLREAD, now called outside INOUT. After reading the input files, some additional initializations are performed and then it is called the subroutine AAMAIN. AAMAIN was originally the the main UTCHEM's subroutine, directly or indirectly calling all the other ones. In UTCHEMP, however, AAMAIN contains only the part of the code used to execute a time step. After AAMAIN some additional outputs are generated and the simulation is finalized.

5.2 EFVLib

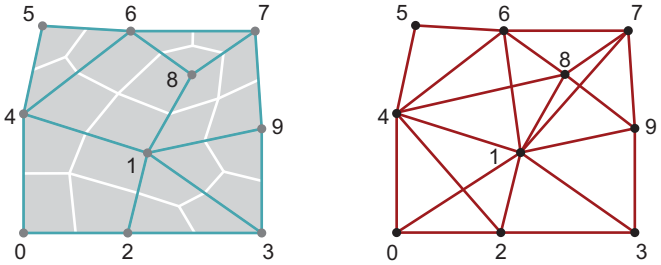
The parallel computing methodology applied in the EFVLib is also based on domain decomposition. The reasons are the same: the main computational cost from the simulation usually comes from the grid-related operations. When the domain is decomposed, each processing unit works

only in a part of the domain, which is much smaller than the whole domain. Since the processors operate simultaneously, the time required to complete a large scale simulation using several processors is equivalent to the time that a single processing unit takes to run a small simulation case.

The straightforward methodology for dividing a domain is to distribute the grid elements from Equation (4.1) among the available processors. This was indeed the methodology applied in UTCHEM, which uses a cell-cen-tered numerical scheme. However in the EbFVM the unknowns are at the vertices of the elements. A division based on elements would then cause redundant computations since a vertex might be replicated in multiple computational domains. Furthermore, usually there are many more elements than vertices. For example, in a typical tetrahedron grid, the number of elements is six times the number of vertices [19]. Thus the effort to divide a grid tends to be smaller if the division is based on vertices rather than on elements. For these reasons, the grid division methodology adopted in this work is based on vertices: each vertex is addressed to a unique computational domain.

In Section 4.4 a general conservation equation was discretized using the Element-based Finite Volume Method. From Equation (4.17) one may note that the flux of a property through a face depends on the value of that property at the element vertices. Thus, a vertex is in a sense connected to all the other vertices from the elements to which it belongs. The notion of connection leads to the representation of a grid as a graph. The vertices are the graph's nodes and if two vertices belongs to the same element, then there is an edge connecting them. Figure 5.7 illustrates the graph representation of a grid.

The graph representation of a grid is convenient for the division task. Divide a grid means separate the nodes of its graph into different groups. If two connected nodes go to different groups, then the edge connecting them was crossed. However, those two nodes depend on each other for the flux computation. If they are in separate groups, data must be interchanged between those groups. So, minimizing the number of edges crossed improves the quality of the grid division.



(a) Grid **(b)** Graph

Figure 5.7 – Graph in **(b)** is the graph of grid **(a)**

5.2.1 Graph partitioning

In this study the partition of graphs is supported by Metis library [1]. This library gives back a vector P whose size is the number of nodes. Each component of P is the partition to which the node belongs. There are two graph partitioning methods available: `METIS_PartGraphRecursive` and `METIS_PartGraphKway`. The first one implements a multilevel recursive bisection algorithm [14] and will be referred to as bisection. The second method implements a k-way partitioning scheme [15] and will be referred to as k-way. The method chosen to parallelize the studied codes was the `METIS_PartGraphKway` because it produces partitions of comparable or even better quality and it also requires less time [15].

Both graph partitioning methods available in Metis – bisection and k-way – are based on the multilevel paradigm. The partitioning itself goes through three phases: coarsening, initial partitioning, and uncoarsening.

Coarsening phase

In the coarsening phase, adjacent nodes are grouped yielding a coarser graph in order to diminish its partitioning complexity. This grouping proceeds until a sufficiently small graph is achieved.

A weight is attributed to each node and to each connection. Let N^n be a group of nodes that were collapsed into a single node n. The weight of n is simply the sum of the weights of the nodes in N^n. Edges connecting two nodes that are both in N^n are eliminated. On the other hand, all edges connecting a node from N^n to an external node p are collapsed into a single edge connecting n to p. The weight of this edge is of course the sum

of the weights of the collapsed edges. This weight-based strategy helps the construction of good partitions in the coarser graphs with respect to the finer graph. The edge-cut of the graphs are the same and a balanced partitioning in the coarser graphs tends to be also balanced in the finer graph [15].

The method chosen to group the nodes is the Heavy Edge Matching (HEM) [15]. The objective of this method is the production of coarser graphs which lead to partitions that minimize the edge-cut. This is achieved by looking for coarsened graphs that reduce the sum of the edge weights. Denoting by e_{pq} the edge that connects the nodes p and q and by $w(e_{p,q})$ the weight of such edge, the following relation holds

$$\sum_{e_{p,q} \in G^{i+1}} w(e_{p,q}) = \sum_{e_{p,q} \in G^{i}} w(e_{p,q}) - \sum_{p,q \in N^n; n \in G^i} w(e_{p,q}), \tag{5.1}$$

where G^{i+1} is the graph obtained by coarsening G^i. Thus in order to minimize the edge weight sum it is interesting to collapse the edges whose weights are the biggest. In the HEM method all nodes are visited in a random order. If a node n was not matched yet (grouped to other nodes), it is matched to the unmatched adjacent node whose corresponding edge weight is the biggest. If such adjacent node does not exist, the node n remains unmatched. According to [15], this methods produces good results in practice, despite the fact that it does not guarantee that coarsened graph obtained is the one that minimize the edge weight sum.

Initial partitioning phase

The second phase is the initial partitioning phase. It takes place when the coarsening method is not effective anymore, that is, the graph size reduction factor of successive graphs is greater than 0.8. This phase is the major difference between both Metis partitioning methods: the bisection method divides the coarsest graph in only two parts and the k-way, on the other hand, splits it directly into k parts.

As already mentioned, the bisection method divides the coarsest graph in two parts, which are then uncoarsened. The resulting graphs are then divided again until the required number of partitions is achieved. The method must be applied $\log_2 k$ times in order to obtain k partitions.

The complexity of the method is $\mathcal{O}(|E|\log_2 k)$, where $|E|$ is the number of connections (edges) [13].

In the k-way partition, the coarsest graph is divided directly into k parts and thus applying the method once is enough to obtain the number of partitions wanted. Curiously, the k-way method divides the coarsest graph using the multilevel bisection algorithm [14]. Despite the fact that a k-way division is much more laborious than a bisection, the coarsest graph is so coarse that its division is computationally cheap. Since the method is applied only once, its complexity is reduced to $\mathcal{O}(|E|)$ [15].

Uncoarsening phase

The last phase is referred to as uncoarsening. A straightforward methodology would be simply ungrouping the nodes until the finer graph is reached. However, graphs with a larger number of nodes have also a larger number of degrees of freedom so are much more complex. Furthermore, the best partition of a coarse graph may not be the best one of a finer graph. For this reason, Metis improves the quality of the graph partition by swapping nodes among the partitions as the graph is uncoarsened [15]. The method chosen in this study is the Greedy Refinement (GR). In this approach, the nodes are visited in a random order and at each node n it is computed a gain function with respect to each partition b using the formula

$$g_b(n) = ED_b(n) - ID(n). \tag{5.2}$$

ED_b (external degree) is the sum of edge weights of the adjacent nodes that are in b while ID (internal degree) is the sum of the edge weights of the nodes that are also in the partition a to which n belongs. The node n will be moved to partition b if g_b is positive and greater than the gain of any other partition. Besides, in order to preserve load balance the following conditions must also be achieved:

$$\sum_{p\in b} w(p) + w(n) \leq W^{\max} \tag{5.3}$$

$$\sum_{p\in a} w(p) - w(n) \geq W^{\min}. \tag{5.4}$$

The values used in Metis for $W^{\max}$ and $W^{\min}$ are $0.9|V_0|/k$ and $1.03|V_0|/k$ respectively, where $|V_0|$ is the sum of all the node weights and k is the number of partitions. The GR algorithm is iterated until convergence. According to [15], the number of iteration is small.

5.2.2 Ghost nodes

In Section 4.4 a general conservation equation was discretized using the Element-based Finite Volume Method. It was noted that the stencil associated to a node n contains all the nodes that share an element with this node, as illustraded in Figure 4.5. Those nodes are necessary not only to compute the shape functions, but also to store values of fields defined on the grid. As a consequence, the subdomains obtained by partitioning the related graph of a grid must be extended in order to embrace every neighboring node, resulting in subdomain overlapping.

Consider for example the grid illustrated in Figure 5.8. This grid was divided into two subdomains, each subdomain being assigned to a processor unity. The nodes and their corresponding control volumes addressed to processor 0 are painted in yellow and the ones addressed to processor 1 are painted in red. Due to the stencil of the EbFVM, the discretized equations of nodes 1, 5, 8, 11, and 12 in subdomain 0 require values at nodes 0, 9, 10, and 13, which are in another computational domain. The same applies for subdomain 1: discretized equations of nodes 0, 9, 10, and 13 demand values at nodes 1, 5, 8, 11, and 12. The subdomains thus must be extended, resulting in the grids illustrated in Figure 5.9.

The nodes that come from another computational domain are called ghost nodes and are similar to the ghost cells used in UTCHEM. No computation is performed for them. They are used only to store values. If a value of a field changes in a node, then its value should be updated at all computational domains where such node is a ghost node. This operation is actually quite complicated, but the code developed for it hides most of the technical details, leaving for the final user a simple interface. Subsection 5.2.6 shows how simple such interface is.

The nodes as well as the elements have local indexes at their local subdomains. The ordering of the nodes is performed in the following way: the first nodes are local – which means that they are addressed to

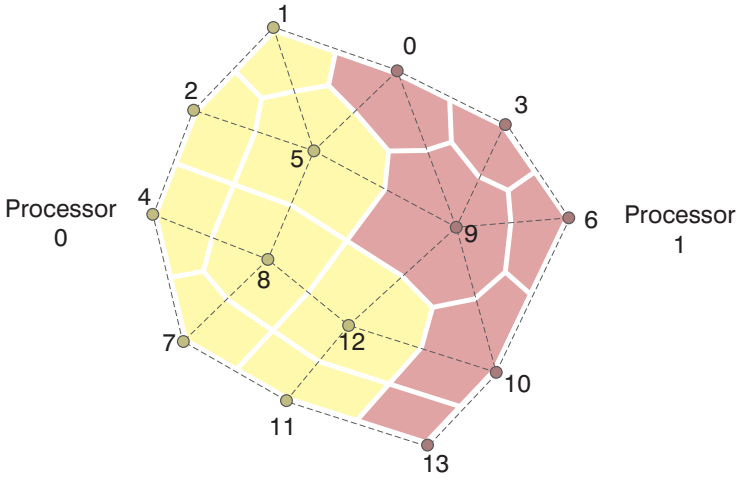


Figure 5.8 – Unstructured grid divided into two subdomains

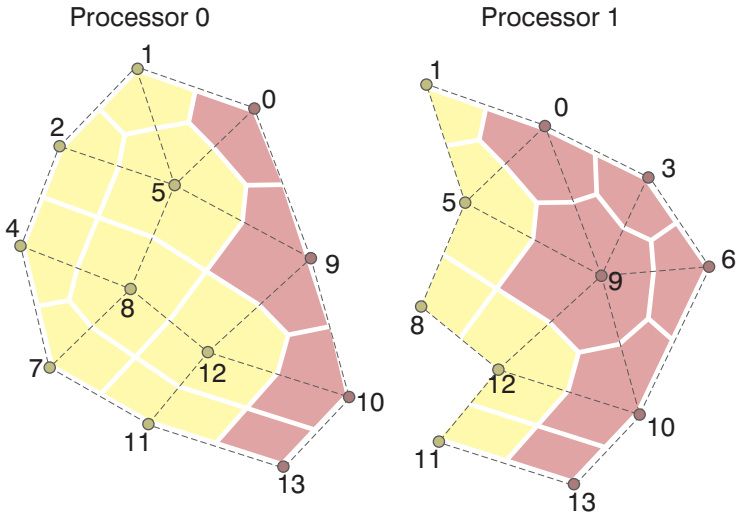


Figure 5.9 – Subdomains extended

the current subdomain – leaving the last indexes for the ghost nodes. This way of ordering is convenient for three reasons:

- If the user intends to access only the local nodes, it is sufficient swapping the vector of vertices until the last local vertex is reached;
- The user may know if a node is local simply by checking if its index is lower than the number of local vertices;
- It is not necessary to store the vertices in separate vector, which is computationally very convenient.

Figure 5.10 illustrates the local indexes of the nodes of the grid from Figure 5.9.

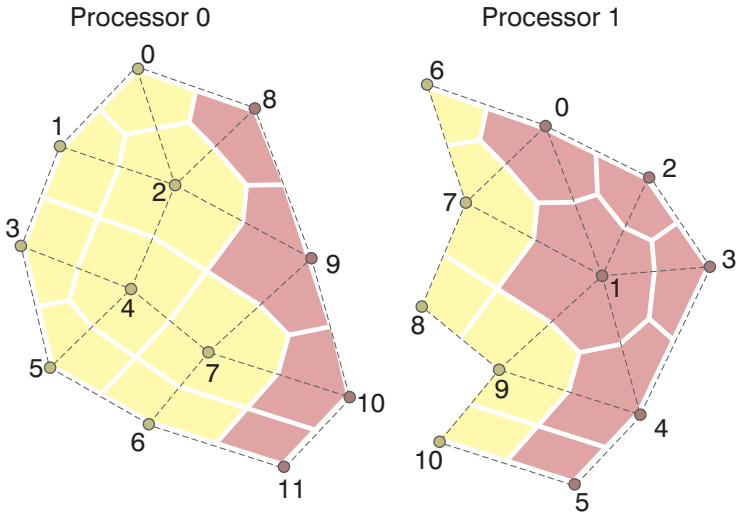


Figure 5.10 – Local indexes of two subdomains

5.2.3 Assembling in parallel a system of linear equations

In almost all simulations it is necessary to solve a system of linear equations (SLE). This is one of the most time consuming operations. It is estimated that about 60 to 70% of the computational time is spend solving systems of linear equation [16]. Since the main focus of this work is on computational performance, the SLE must be solved very efficiently. However, the development of a parallel solver for SLE is beyond the objectives of this work. It was used instead the Portable, Extensible Toolkit for

Scientific Computation (PETSc) [4] as an external library to solve the SLE's. PETSc provides a lot of tools specially designed for scientific applications and is recognized as one of the most powerful libraries available.

Since PETSc is intended to support very general applications, there are many details that must be configured in order to assemble and solve a system of linear equations. It is desired however the product of this work to be simple in order to encourage people to employ parallel computing in their future developments. For this reason, a relatively simple computational interface was created to hide most of the PETSc details. The matrix are supposed to be sparse, while both matrix and the vectors of the linear system are distributed along the available processing units. In other words, matrix and vectors are parallel. The default preconditioner and solver are the incomplete LU factorization and GMRES, respectively, but the user is allowed to choose any other methods available in PETSc.

Considering a generic SLE $Ax = b$, PETSc distribute its components among the available processors. It is possible either to define the size of the global SLE and let PETSc divide the SLE among the processors or define the size of each local SLE and thus the global size will be the sum of the local sizes. In any case, the first processor will have the first rows, the second processor will have the rows following the first processor rows, and so on. Each processor p has a submatrix A_p with the rows of A as well as subvectors x_p and b_p with the corresponding components of x and b. Despite each processor having only a portion of the SLE, PETSc handles the SLE as a global one. This means that the indices used to set a SLE are global instead of local. A processor may set a value even if the corresponding row is outside its range. If that is the case, the value is communicated to the processor that actually have the corresponding row. Such communication however should be avoided, since it degrades performance. It is recommended each processor to set only local values.

In order to better handle communication and thus optimize performance, PETSc divide the matrix A_p into two other ones: the "diagonal" matrix $A_{p,d}$ and the "off-diagonal" matrix $A_{p,o}$. $A_{p,d}$ is a square matrix composed by the entries A_{ij} of A such that the rows i and j are local to the processor p. $A_{p,o}$ on the other hand contains all the entries of A_p that are not in $A_{p,d}$. Such way of interpret A_p is convenient for example in a matrix-vector multiplication, which is an operation that may be executed several times while solving a SLE. Let w be the vector to which A will be

multiplied, w_p its local segment in the processor p, and w_o the segment of w outside p. The operation performed locally in p is

$$A_p w = \begin{pmatrix} A_{p,d} & A_{p,o} \end{pmatrix} \begin{pmatrix} w_p \\ w_o \end{pmatrix} = A_{p,d} w_p + A_{p,o} w_o. \tag{5.5}$$

The multiplication $A_{p,d} w_p$ can be promptly executed, since it involves only local values. The values w_o on the other hand must be received from another processors. The division of A_p in diagonal and off-diagonal portions is also convenient for Block Jacobi preconditioners, described later. In this case, operations with the off-diagonal matrix $A_{p,o}$ are simply ignored.

Based on the observations above and on the domain decomposition methodology described in the previous sections, have in mind that a processor have local vertices and local SLE rows and that they are not necessarily related. The first processor for example will have for sure the first SPE rows but probably not all the first vertices. In fact, the processor 0 in Figure 5.8 has the set of vertices $\{1,2,4,5,7,8,11,12\}$ and not the set $\{0,1,2,3,4,5,6,7\}$. It is important however to ensure that the local rows corresponds to the local vertices. A processor has only the data to compute the discretized conservation equation coefficients of the local vertices. If the SLE row of a local vertex is not local, the coefficient must be transmitted to the processor in which the row is local, resulting in communication overhead. Furthermore, the solution x retrieved from PETSc is the solution at the local rows. If the local rows are not related to local vertices, such solution must be manually transmitted to the corresponding processors. Another way of retrieving the solution is ask PETSc for the global solution. This of course is not smart because it requires PETSc to make lots of communication to have a copy of the global solution on each processor and also demands too much memory.

To make sure the local rows are related to the local vertices, two measures are taken. First, the local size of the SLE must be equal to the number of local vertices (supposing only a single variable is being solved). In order to ensure this, the local size of the SLE is directly set in PETSc instead of setting the global size and let PETSc to divide the SLE. Secondly, a new global indexation is set to the vertices. The new indices are called PETSc indices. The PETSc index of a vertex in a processor p is its local index in

p plus the sum of the number of local nodes from processors q such that $q < p$. In Figure 5.11 it is illustrated the PETSc indexes of the nodes of the grid from Figure 5.9. Note that now the processor 0 has in fact the set of vertices $\{0,1,2,3,4,5,6,7\}$ as it was desired. It is worth noting that this new global indices are used only to handle the local SLE. The results exported from a simulation still respecting the original global indexation.

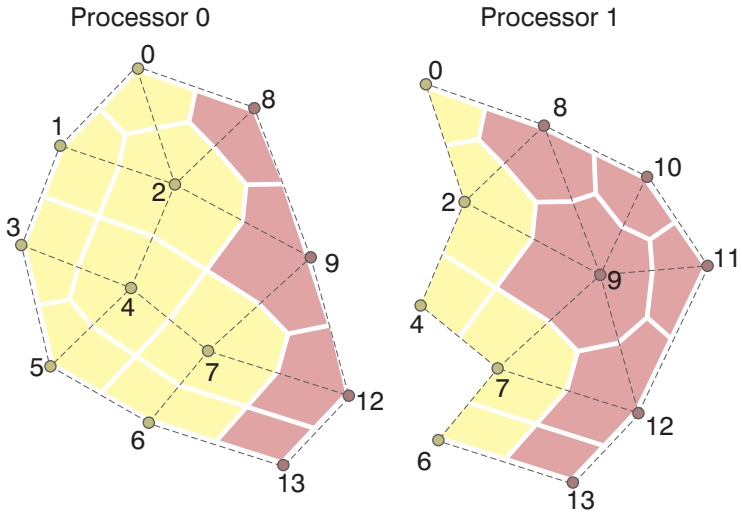


Figure 5.11 – PETSc indexes of two subdomains

Figure 5.12 illustrates the structure of a matrix assembled in parallel using the EbFVM on the grid of Figure 5.11. This matrix has four submatrices: $A_{0,d}$, $A_{0,o}$, $A_{1,d}$, and $A_{1,o}$. $A_{0,d}$ and $A_{1,d}$ are the diagonal submatrices of processors 0 and 1, respectively. Their coefficients are associated to local nodes only. Matrices $A_{0,o}$ and $A_{1,o}$ on the other hand are the off-diagonal submatrices. Their coefficients are associated to a local and a ghost node. It is important to emphasize that each processor assembles its diagonal and off-diagonal submatrices, not interfering in submatrices of other processors. As an example, consider without losing generality that the problem to be solved is purely diffusive. The matrix is symmetric, which means that for every pair of nodes (m,n) the coefficients $a_{m,n}$ and $a_{n,m}$ will be the same. If m and n are in different subdomains, then such coefficient will be calculated twice: one time by the processor of the

subdomain of m and another time by the processor of the subdomain of n. Regarding that values at ghost nodes are updated, the results at both processors will be the same. This shows that reducing interface between subdomains is important not only to reduce the amount of data communicated between processors, but also to reduce repeated calculations.

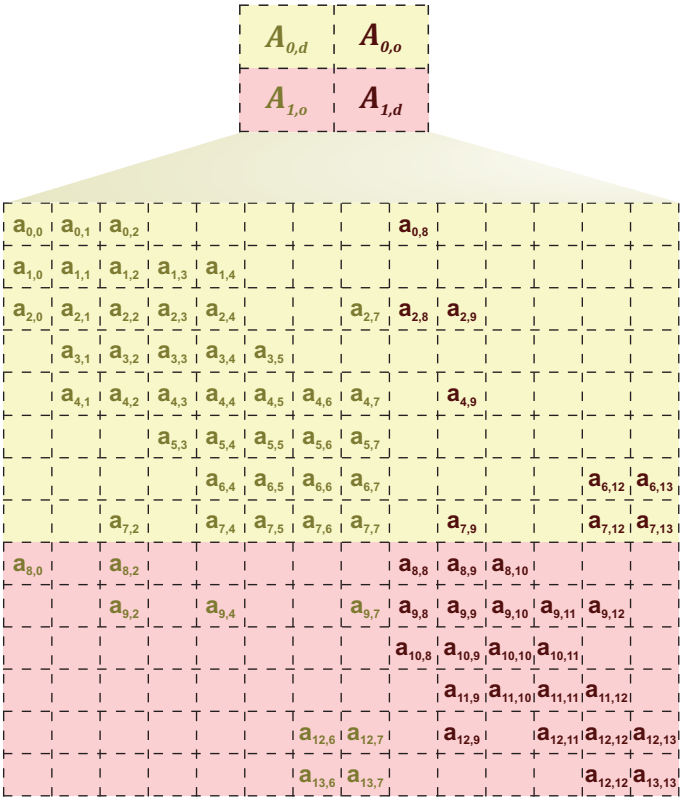


Figure 5.12 – Matrix assembled in parallel

5.2.4 Solving in parallel a system of linear equations

The matrix coming from the discretization of the conservation equations is sparse and thus it is not feasible to apply a direct method to solve the linear system. Instead, iterative methods are used for that task. In iterative

methods, instead of solving $Ax = b$ it is proposed to solve a much more cheap linear system $Kx = b$. Since K is different from A, the solution $x_0 = K^{-1}b$ probably is not the solution of the desired linear system. There is an error $e_0 = x_0 - x$ and a residual $r_0 = Ax_0 - b$ such that $Ae_0 = r_0$. However, it is not practical to solve $Ae_0 = r_0$ and thus the linear system $K\tilde{e}_0 = r_0$ is solved instead. $\tilde{e}_0$ is an approximation of the actual error and it is used to correct the approximate solution: $x_1 \equiv x_0 - \tilde{e}_0$. This process is executed until convergence is reached. The general formula of such scheme is

$$x_{i+1} = x_i - K^{-1}r_i. \tag{5.6}$$

All methods that obey the above formula are called *stationary*. The term *stationary* comes from the fact that the operations applied in each iteration are the same. There is a generalization of stationary methods that contemplates almost all known iterative methods. Instead of using only the last residue to compute the new approximate solution, all the previous residues are used:

$$x_{i+1} = x_i + \sum_{j \le i} K^{-1}r_j\alpha_{ij}, \tag{5.7}$$

where α_{ij} is the weight of the term $K^{-1}r_j$. For stationary methods,

$$\alpha_{ij} = \begin{cases} -1, \text{if} i = j \\ 0, \text{if} i \neq j \end{cases} \tag{5.8}$$

K is a linear system preconditioner and the convergence of an iterative method depends on how close to the original matrix A is the matrix K. Letting D_A, L_A, and U_A be the diagonal, lower triangle, and upper triangle parts of A, some classical iterative methods are:

- Richardson: $K = \alpha I$;
- Jacobi: $K = D_A$;
- Gauss-Seidel: $K = D_A + L_A$;
- SOR: $K = \alpha^{-1}D_A + L_A$;
- Symmetric SOR (SSOR): $K = (D_A + L_A)D_A^{-1}(D_A + U_A)$.

Based on Equation (5.6) (and on its general form given by Equation (5.7)), one may see that the following operations are common to every iterative linear system solver:

- Vector operations (like additions and inner products);
- Matrix-vector product;
- Construction of a preconditioner matrix $K \approx A$ and the solution of a linear system $Kx = y$.

Some methodologies to make these operations parallel are presented in [9].

Vector operations

The main vector operations are vector additions and inner products. Vector additions are operations of the form $x \leftarrow \alpha x + \beta y$. If all vectors are distributed in the same way, this operation is intrinsically parallel: all processors can execute it without any communication. Inner products, on the other hand, are of the form $\alpha = x^T y$. This is a reduction operation, since the inputs are vectors and the result is a scalar. This operation cannot be executed without communication because each processor contains only a part of the vectors x and y. The common strategy is to compute the quantities $\alpha_P = x_P^T y_P$ at each processor, transmit the values obtained to all processors, and then perform the sum

$$\alpha = \sum_{i=1}^{N} \alpha_P, \tag{5.9}$$

where x_P and y_P are the local segments of x and y and N is the number of processors.

Matrix-vector product

The stencil of the numerical methods used to solve partial differential equations makes the matrix of the resulting linear system to be sparse. Thus, for most j's, $1 \leq j \leq n$, where n is the size of the linear system, the operation $y_i \leftarrow y_i + a_{ij} x_j$ is redundant. Let I_P be the index set of the matrix rows owned by a processor P (indexes of the diagonal portion of the

sumbmatrix A_P, described in the beginning of this section). The values x_j that the processor P actually needs in order to perform the matrix-vector multiplication are those such that j is in the set

$$S_{P,i} = \{j : j \notin I_P, a_{ij} \neq 0\} \tag{5.10}$$

It is not clever however to send to P a package for each set $S_{P,i}$. Instead, the sets $S_{P,i}$ are combined into a single one defined by $S_P \equiv \cup_{i \in I_p} S_{P,i}$. The processor P receives only one package with the values of x in S_P and then is able to perform the multiplication.

Preconditioner

The parallelization strategy applied for the preconditioner strongly depends on what is the chosen of K. The Jacobi method is intrinsically parallel, since K is the diagonal of A and the operation $x = K^{-1}y$ can be performed to each component of x independently. For other methods, on the other hand, there are difficulties. One of the most common preconditioners is the incomplete LU factorization (ILU). This type of factorization is performed using Gaussian elimination just like the complete LU factorization, but the elements that would become non-zero are simply ignored. This avoids the fill-in phenomenon, letting the matrix $L+U$ with the same sparsity of the original matrix A.

The problem with ILU happens when it is necessary to solve a system $Lx = y$ in parallel. The second processor must wait the first one solve its unknowns to start working. The third processor, by its turn, must wait the second one, and so on. Solving $Lx = y$ is recursive and thus sequential. One of the strategies that has been proposed to scale the ILU factorization is the Block Jacobi method. The idea of the Block Jacobi method is to ignore all matrix components outside the processor subdomain. In other words, the connection between processors is simply ignored. Doing this is actually not wrong, since one must remember that K is only an approximation of A. More iterations will be needed, but the method becomes scalable.

Another strategy that may be applied to scale the ILU factorization is the graph coloring associated with permutation. To each node p it is associated a color in such a way that all its neighbours have a color different from p. Hence, there are no neighbours with same color. The

matrix is then permuted to group the nodes with same color. The resulting matrix have diagonal blocks that are diagonal matrices. Each color is then solved at a time. Since nodes of same color does not depend on each other, the process can be executed in parallel. After a color is processed, data is exchanged between the processors so that all of them have the necessary values to start processing another color.

5.2.5 Wells and boundaries

Wells and boundaries are also geometrical entities and thus they must be considered in the domain decomposition. The treatment for both of them is straightforward.

A boundary is a collection of element facets that are at the contour of the solution domain [11]. These element facets are defined as boundary elements. In two-dimensional grids the boundary elements are lines whereas in three-dimensional grids they may be triangles or quadrangles. It is possible to define several boundaries, each one having a group of boundary elements and a particular boundary condition.

The methodology used to divide the boundaries is similar to the one applied for the elements. Consider again the grid of Figure 5.8 whose boundaries are defined in Figure 5.13. For a control volume whose vertex p is at the boundary of the domain, it is also necessary to compute fluxes through boundary elements. All boundary elements that have p as one of their vertices are involved in the computation. Thus, the condition to divide the boundaries among the processors is: if a boundary element e of a boundary E is in a subdomain Ω_i if there is a vertex v in Ω_i such that v is a vertex of e. If e is in Ω_i, not necessarily all boundary elements of E are in Ω_i: only the one with vertices in Ω_i. Figure 5.14 shows how the boundaries from the grid of Figure 5.13 may be split into two subdomains. Subdomain 0 have the boundaries 0 and 1 and subdomain 1 have the boundaries 0, 1, and 2. Note that neither subdomain 0 nor subdomain 1 have all the elements of boundary 1.

According to Figure 5.14, the nodes at the contour of a subdomain either are at the contour of the global domain or are ghost nodes. To the ones at the global subdomain contour, it is simply applied the boundary condition associated to the boundary to which they belong. On the other

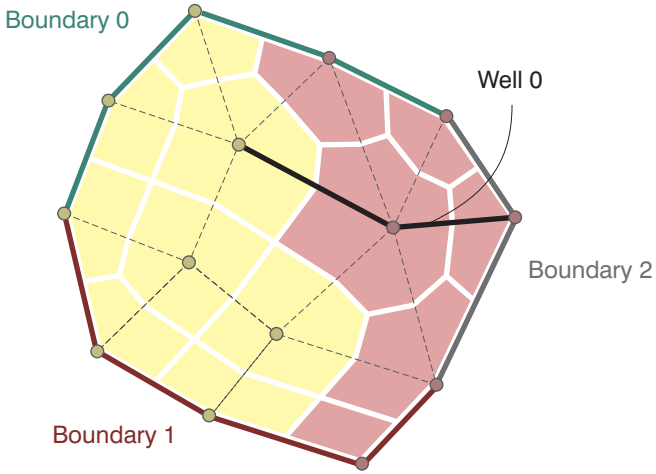


Figure 5.13 – Boundaries and well

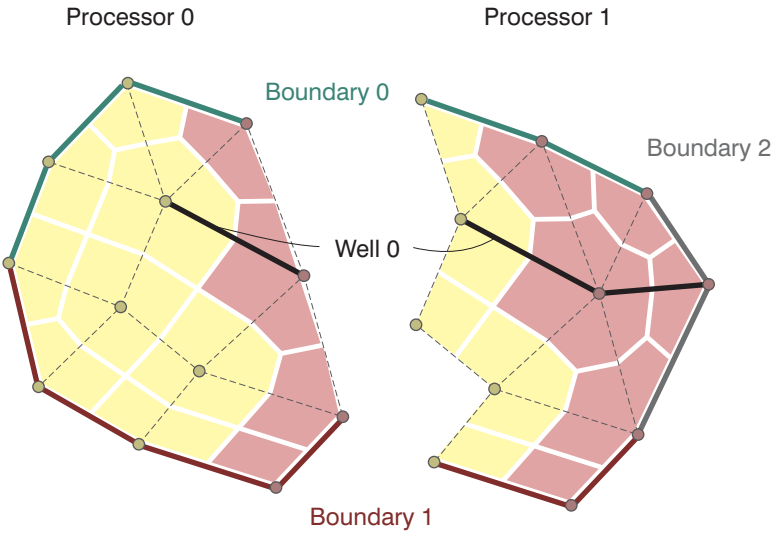


Figure 5.14 – Boundaries and well

hand, for the ghost nodes nothing needs to be done, since their purpose is only store data and thus no computation is performed at them.

Wells are represented in EFVLib as a sequence of line segments that

are coincident with element edges. One of those line segment is called well element and is limited by two nodes. Each one of this nodes is associated to half of the well element. The flow rate of a phase α going from a well to the control volume of a node p is the sum of the flow rates of each of the half well elements associated to p. he phase flow rate is expressed by

$$q_p = \lambda_{\alpha,p} \mathrm{WI}_p \left(P_p - \Pi_p\right), \tag{5.11}$$

where $\lambda_{\alpha,p}$ is the phase mobility, WI_p is the well index, P_p is the reservoir pressure, and Π_p is the well pressure [11]. All of those variables excepting are evaluated at the reservoir, except for Π_p, which is evaluated inside the well. Since the above equation depends on the reservoir condition, it is straightforward that wells should be segmented following the grid division. Two well elements that share a node must be present in the subdomain in which this node is local. Figure 5.14 shows the division of a well into two subdomains.

5.2.6 The code

The implementations in EFVLib for parallel running does not properly solve a case in parallel. They are actually a set of utilities that helps an EFVLib user to adapt his/her code to be solved in parallel. The user must have in mind that, instead of dealing with the whole domain, the code are now being used for only a portion of the domain. Sometimes it is necessary to update values at ghost nodes, divide a global field into local fields, or reunite local fields into a global one. The parallel utilities developed for EFVLib hide most of the technical details and provide a simple user interface.

Three external libraries were used: Metis 5.1.0 [1], PETSc 3.4.4 [4], and Boost 1.55 [5]. The first one is used to partition the grid nodes according to what was explained in Subsection 5.2.1. The second one is devoted to solve in parallel systems of linear equations coming from the discretization of the conservation equations. Both Metis and PETSc are hidden inside the parallel utilities developed here and hence the user has almost no contact with them. Boost on the other hand provides a simplified interface for the MPI functions that was made available to the user and also used in the implementation of the parallel utilities.

There are four classes for supporting parallel computing in EFVLib: `VertexPooler`, `GridDivider`, `FieldDivider`, and `FieldOnVerticesSynchronizer`. The interface of class `VertexPooler` is presented in Listing 5.1. This class is used to pool the nodes (vertices) into subdomains using Metis functions. Both k-way and bisection Metis graph partitioning methods are available and can be chosen using the variable `partitionMethod`. K-way is the default method. The class `VertexPooler` is actually an abstract class and thus cannot be instantiated. It is necessary to create a class derived from `VertexPooler` that implements the method `computeWeightArray`, which is responsible by setting the weights of the graph edges. Two of such classes were implemented – `UnweightedVertexPooler` and `InverseDistanceWeightedVertexPooler` – but the user is free to create a custom `VertexPooler`.

The function `divide` is the main function of `VertexPooler`. This function properly does the partition of the nodes of a grid. There are two input parameters: `GridData` and `nParts`. `GridData` is a temporary computational structure that stores the essential data of a grid that will be divided[18]. `nParts` is the number of partitions, which will be usually equal to the number of processors. The output of `divide` is the variable `subdomains`, which is an array that has the subdomain index of each node.

Listing 5.1 – class VertexPooler

```
1  class VertexPooler {
2  public:
3      VertexPooler( PartitioningMethod partitioningMethod = KWAY );
4
5      void setPartitioningMethod( PartitioningMethod partitioningMethod );
6      void divide( GridDataPtr gridData, int nParts, idx_t*& subdomains );
7
8      virtual ~VertexPooler(){}
9
10 protected:
11     // ... protected attributes
12
13 private:
14     virtual void computeWeightArray( GridDataPtr gridData, idx_t*& weights ) = 0;
15     // ... other private methods
16 };//class VertexPooler
17
18 typedef SharedPointer< VertexPooler > VertexPoolerPtr;
```

`GridDivider` is a class that takes a global `GridData` and creates the local `GridData` for each subdomain. This is a very important class. It partitions the nodes using a `VertexPooler`, creates the subdomains's elements, divides wells and boundaries, and extends the subdomains to include ghost nodes. The interface of this class is in Listing 5.2. `divide` is the function that should be called to perform the actual division a grid. The master processor (processor whose rank is 0) should call the function `divide` passing as parameters the `GridData` of the global grid (only the master processor has access to the whole grid) and, if the desired `VertexPooler` is different from `UnweightedVertexPooler`, a `VertexPooler`. The other processors call the function `divide` passing no parameter. They receive from the master processor the `GridData` of their local subdomain. The output of `divide` is a structure that contains the local `GridData` as well as vectors that may be used by `FieldDivider` and `FieldOnVerticesSynchronizer`, classes that will be described next.

Listing 5.2 – class GridDivider

```
1  class GridDivider {
2  public:
3      GridDivider(){}
4
5      GridDividerOutputPtr divide( GridDataPtr gridData, VertexPoolerPtr vertexPooler
6          = VertexPoolerPtr( new UnweightedVertexPooler ) );
7      GridDividerOutputPtr divide();
8
9      virtual ~GridDivider(){}
10
11 private:
12     // ... private methods
13 };
14
15 typedef SharedPointer< GridDivider > GridDividerPtr;
```

`FieldOnVerticesSynchronizer`, listed in Listing 5.3, is a class whose purpose is updating values at ghost nodes. It can update any type of field defined in EFVLib: fields of scalars, vectors, vectors of vectors (they are not necessarily a matrix because the size of the vectors may not be the same), or symmetric tensors. Other types of fields can also be updated as long as the field type is properly serialized (class serialization

is described in [5]). The class `FieldOnVerticesSynchronizer` uses a structure called `SynchronizerVerticesVectors`, which is an output of the grid division. The function that updates a field is the function `synchronize`. All processors must call this function passing as parameter the local field to be updated. It was used in the function an optimization provided by Boost that consist on the separation of the structure of a vector from its content [5]. The structure of the vectors that will be communicated using MPI methods is broadcast to the processors when the class `FieldOnVerticesSynchronizer` is instantiated. When the function `synchronize` is called later, each processor already knows the structure of the MPI package that will be received, not being necessary the allocation of more memory than what is needed.

Listing 5.3 – class FieldOnVerticesSynchronizer

```
1  template< class _FieldType >
2  class FieldOnVerticesSynchronizer{
3  public:
4      // ... some typedefs
5
6      FieldOnVerticesSynchronizer( SynchronizerVerticesVectorsPtr
7          synchronizerVerticesVectors );
8
9      void synchronize( FieldOnVerticesPtr field );
10
11     virtual ~FieldOnVerticesSynchronizer(){}
12
13 protected:
14     // ... protected attributes
15
16 private:
17     // ... private methods
18 };
```

`FieldDivider` is a class intended for scattering a global field into local ones and to gather local fields into a global one. The scatter operation is usually performed at the beginning of a simulation when the master processor have read the initial conditions and must spread the data to the local subdomains. The gather operation on the other hand is necessary to collect and export results using the master processor. There are two functions for scattering (lines 8 and 9 of Listing 5.4) and two functions for

gathering (lines 11 and 12 of Listing 5.4). The ones with two arguments are intended to be called only by the master processor, which is the only processor that has access to the global field. The ones with a single argument are called by the other processors.

Listing 5.4 – class FieldDivider

```
1  template< class _EntityType, class _FieldType >
2  class  FieldDivider{
3  public:
4      // ...  some typedefs
5
6      FieldDivider( IntVector2DPtr entitiesOfSubdomains);
7
8      void scatter(  FieldTypeVectorPtr global, FieldTypeVectorPtr local );
9      void scatter(  FieldTypeVectorPtr local );
10
11     void gather( FieldTypeVectorPtr local, FieldTypeVectorPtr global );
12     void gather( FieldTypeVectorPtr local );
13
14     virtual  ~FieldDivider(){}
15
16 protected:
17     // ...  protected attributes
18
19 private:
20     // ...  private methods
21 };
```

An example of a simple program that solves a two-dimensional heat transfer problem using EFVLib is presented in Appendix B.

CHAPTER

6

Experimental Environment and Results

This chapter presents results obtained from the new parallel versions of UTCHEM and EFVLib. The parallel version of UTCHEM will be called UTCHEMP to distinguish it to its previous serial version. UTCHEMP is a reservoir simulator that have many models which enables the simulation of very complex fluid flow phenomena in reservoirs. It is necessary thus to validate UTCHEMP against several different cases to guarantee that all of its features are working as expected. Four cases are presented here. Each one of them explores different models implemented in the simulator.

The cases used to evaluate UTCHEMP were run in the TACC-Lonestar cluster [3], a cluster from The University of Texas at Austin. It has a total of 1,888 computer nodes, each one with 12 cores. The clock frequency of the cores is 3.33 GHz and each node has 24 Gb of RAM memory available. The compiler used is an Intel Fortran, with the optimization flag O3. A single cluster node was used in the cases up to 8 processors and several nodes with 8 processors per node in the cases with more than 8 processors. The evaluation of UTCHEMP started with its validation against some

of its benchmark cases. The number of processors that was possible to be employed was small because of the small number of grid blocks. It was used in this study only 1, 2, 4, and 8 processors. The second part is a performance evaluation. The number of grid blocks were sharply increased keeping the size of the grid blocks the same to avoid numerical instabilities. The wells were rearranged in a pattern similar to the original case. It is very important to emphasize that the performance evaluation case is different from the validation case. However, the workflow of the simulation is the same so that the results are expected to be right. Furthermore, some cases had their original final simulation time reduced in the performance evaluation due restrictions in the TACC utilization policy, which does not permit that a simulation take more than one day to finish.

The methodology used with EFVLib was slightly different. EFVLib is a numerical library intended to help the user to develop its own applications. So, it is not necessary to validate the library against cases that are physically different. What is important is that the cases cover most of EFVLib's functions. Only two cases are presented here, both of them simulating a two-phase, incompressible, immiscible flow. The first one has a small but geometrically complex grid intended mainly to validate the library, despite some performance tests were executed. The geometry of the second case is much more simple, although the grid has many more elements. Both cases were run in the CPU cluster of SINMEC, a CFD laboratory from the Federal University of Santa Catarina. The cluster has 64 computer nodes, each one with 8 Gb of RAM memory and 8 cores with 2.00 GHz of clock frequency. The compiler used is a GCC 4.1.2.

6.1 Case 1

This is a water flooding case. There are four wells injecting water and 13 producer wells. All wells operate according to a flow constrained condition. Figure 6.1 illustrates the water saturation field after 800 days. The reservoir permeability is anisotropic and heterogeneous. The simulation runs for 2526 simulation days. The coarse grid used has 31 grid blocks in direction x, 45 grid blocks in direction y, and three in direction z. The grid block sizes are constant and equal to 100 ft in directions x and y, and

Table 6.1 – Case 1 properties

Property	Value
Reservoir lenght	3100 ft
Reservoir width	4500 ft
Reservoir thickness	24 ft
Coarse grid	31x45x3 (4185)
Fine grid	200x200x5 (200000)
Number of components	11
Max. number of phases	4
Porosity	0.1371
Permeability in x dir.	from 10 to 1250 mD
Permeability in y dir.	from 20 to 2500 mD
Permeability in z dir.	5 mD
Initial water saturation	0.1
Initial reservoir pressure	1500 psi
Depth	3400 ft
Number of injector wells	4
Number of producer wells	12
Simulation days	2526 days

size 11, 9 and 4 ft (from top to bottom) in direction z. The main properties of the case are shown in Table 6.1.

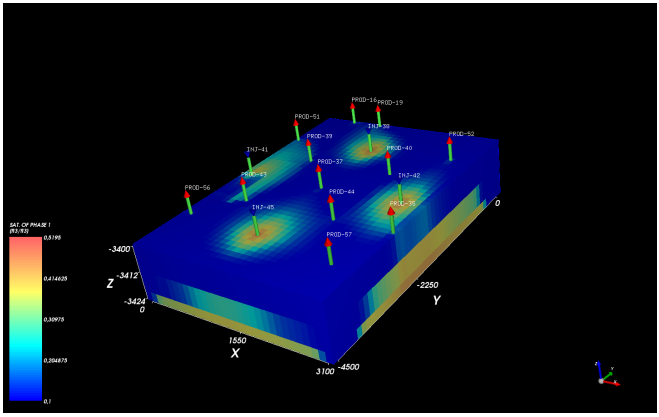


Figure 6.1 – Water saturation after 800 simulation days of case 1

The validation of UTCHEMP against UTCHEM is based on results of

average aqueous phase pressure, average water saturation, and average oil production rate. UTCHEMP was run using 1, 2, 4, and 8 processors. The results obtained are presented in the Figures 6.2, 6.3, and 6.4. As one may see, the results are in good match.

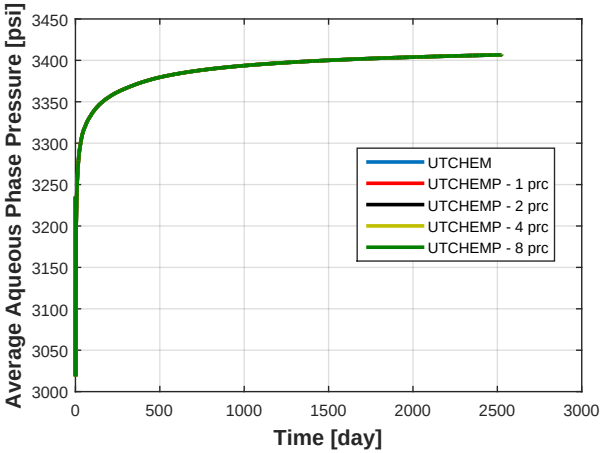


Figure 6.2 – Average aqueous phase pressure of case 1

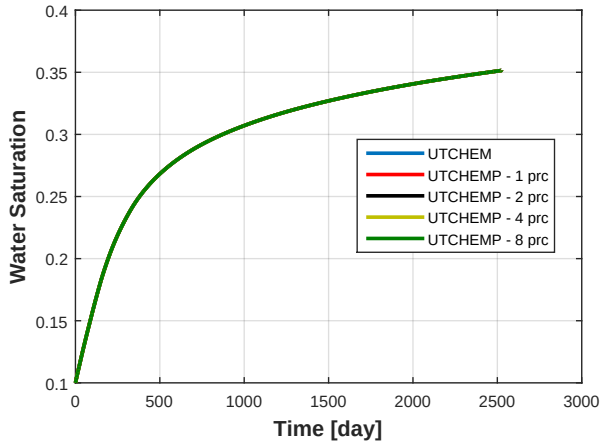


Figure 6.3 – Average water saturation of case 1

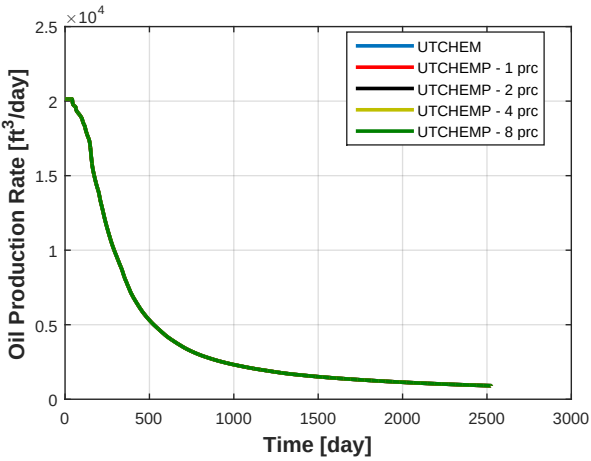


Figure 6.4 – Overall oil production rate of case 1

With only near 4000 grid blocks, the grid chosen is too coarse for an appropriate performance evaluation. Following the methodology described in the beginning of this chapter, the grid size was increased to 200 x 200 x 5 (200, 200, and 5 grid blocks in directions x, y, and z, respectively), which gives a total of 200,000 grid blocks. The wells were rearranged in a similar pattern. It was used 1, 2, 4, 8, 16, 32, 64, and 128 processors. Figure 6.5 presents the total computational time required by the simulations. It was reduced from almost 5 hours using a single processor to about 7 minutes with 64 and 128 processors. Figure 6.6 shows the speedup, defined in Equation (2.1). The red dashed line is the ideal speedup, defined in Equation (2.2). As one may see, the computational time achieved with 64 processors is almost the same achieved with 128. This is an indication that the increase in performance that it is possible to have with parallel computing is saturating. The relative computational weight of non-parallel operations and the cost of communication between processors becomes high compared to the cost of parallel operations. In fact, when 128 processors are used, the division based on direction y makes each processor have one or two grid blocks in this direction. Thus, the number of ghost nodes at each subdomain is approximately the same of the number of ghost nodes.

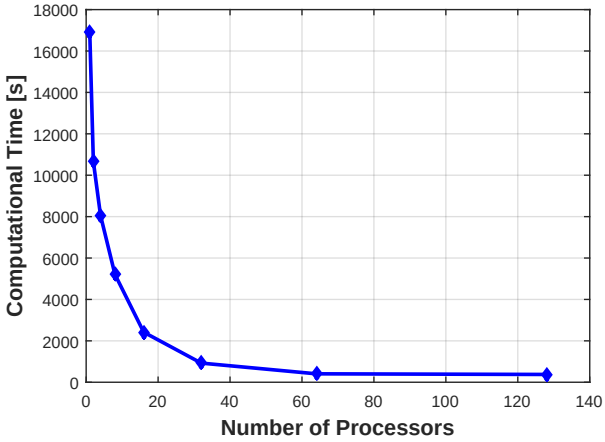


Figure 6.5 – Total computational time according to the number of processors of case 1

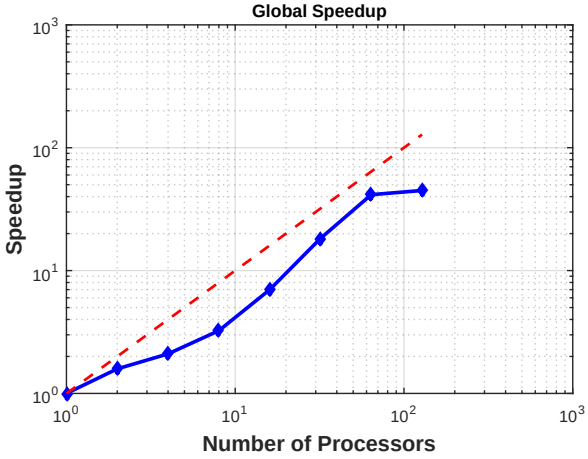


Figure 6.6 – Global speedup of case 1

Table 6.2 presents the computational time of some UTCHEMP operations. The results indicate that the initialization time grows as the number of processors increase, which is reasonable since the input data must be broadcast to a larger number of processors. The computation of

transmissibilities and salinity seems to be scalable operations due to their time reduction. The computation of concentrations had its time reduced with the increase of the number of processors, but then starts to grow up again with a large number of processors. This happens because the main routine of UTCHEMP that computes concentrations is actually huge and contains not only explicit grid-related operations, but also communication in order to update variables at ghost nodes.

Table 6.2 – Computational times in seconds of case 1

Prcs	Total Time	Init.	Trans.	Conc.	Salinity	LS Solving
1	16894.7	3.044	143.9	1524.3	43.81	15122.2
2	10656.8	2.677	84.53	783.24	31.39	9718.48
4	8050.04	2.824	58.75	446.21	26.15	7487.80
8	5210.12	2.738	33.80	265.18	14.77	4874.41
16	2418.94	5.800	15.64	137.32	6.196	2241.78
32	925.175	10.80	7.929	97.959	2.737	798.128
64	407.275	18.06	4.584	83.246	1.504	290.474
128	376.592	35.556	3.199	109.116	1.318	202.79

Figure 6.7 shows the contribution of some operations over the global computational time. As one may see, the biggest contribution is the solving of linear systems coming from the aqueous phase pressure equation discretization. Following that, there are the explicit solving of concentration equations and the computation of transmissibilities. Since PETSc is used to solve systems of linear equations, this library is the main responsible for the total computational time of this case.

6.2 Case 2

The second case is a gel treatment problem. The main features of this case are summarized in Table 6.3. The reservoir has length 1100 ft, width 1000 ft, and thickness 27 ft. It is inclined in directions x and y with dip angles $\theta_x = 0.02923$ rad and $\theta_y = 0.00277$ rad respectively. The reservoir has heterogeneous permeability and porosity field: the porosity field changes from 0.083 to 0.499 whilst the permeability is 72 mD at the first and 900 mD at the second reservoir layers. The coarse grid has 14 grid blocks in

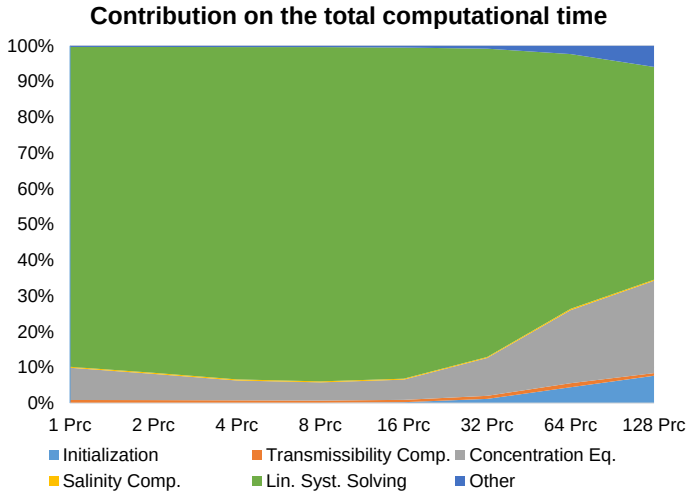


Figure 6.7 – Contribution on the total computational time of case 1

direction x, 13 grid blocks in direction y, and 2 grid blocks in direction z, which gives a total of 364 grid blocks. There is a single injector well and three producer wells. The fluid injected is water and the producers operate at constant bottom hole pressure. The flow is not isothermal and thus the energy equation must be solved. Initially the reservoir is at a uniform temperature of 103°F, but the fluid injected is at 68°F, which makes the average temperature decrease. The case runs until 816 simulation days. Figure 6.8 shows the temperature field after 510 simulation days.

Figures 6.9, 6.10, and 6.11 presents the results of the validation of this case. As one may see, the results are in good match, indicating that probably there is no bug in UTCHEMP in this case. We should always emphasize that results correctness is much more important than a better software performance once of course wrong results are worthless.

After the validation the grid size was increased to 200 x 200 x 10, giving a total of 400,000 grid blocks, to evaluate the software parallel performance. The case were run with 1, 2, 4, 8, 16, 32, 64, and 128 processors. The final simulation time was decreased to 400 days, otherwise using a single processor would require more than one day to finish the simulation, exceeding the maximum time allowed in TACC. Figure 6.12 shows the total computational time spent by the simulations. With a single pro-

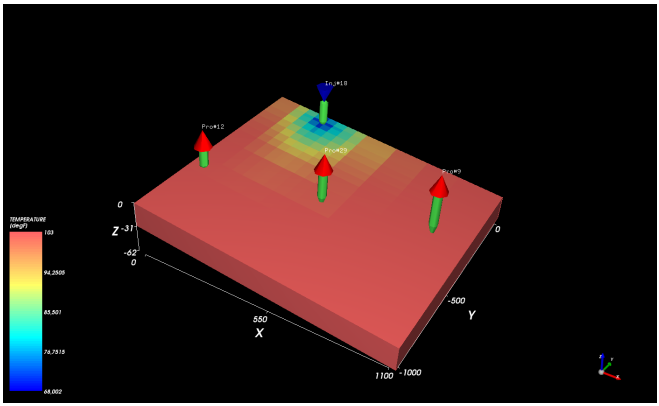


Figure 6.8 – Temperature after 510 simulation days of case 2

Table 6.3 – Case 2 properties

Property	Value
Reservoir lenght	1100 ft
Reservoir width	1000 ft
Reservoir thickness	27 ft
Coarse grid	14x13x2 (364)
Fine grid	200x200x10 (400000)
Number of components	13
Porosity	from 0.083 to 0.499
Permeability in x dir.	first layer 72 mD, second layer 900 mD
Permeability in y dir.	first layer 72 mD, second layer 900 mD
Permeability in z dir.	first layer 72 mD, second layer 900 mD
Initial water saturation	0.3001
Initial reservoir pressure	2000 psi at 1300 ft
Depth	1300 ft
Number of injector wells	1
Number of producer wells	3
Simulation days	816 days

cessor the simulation took about half a day to finish, while with 128 it was only about 20 minutes. Figure 6.13 presents the speedup of this case. It is interesting to note that up to 32 processors the results are similar to what is predicted by the Gustafson-Barsis's Law, discussed in section 2.5, but then the speedup starts to saturate, as predicted by Amdahl's Law.

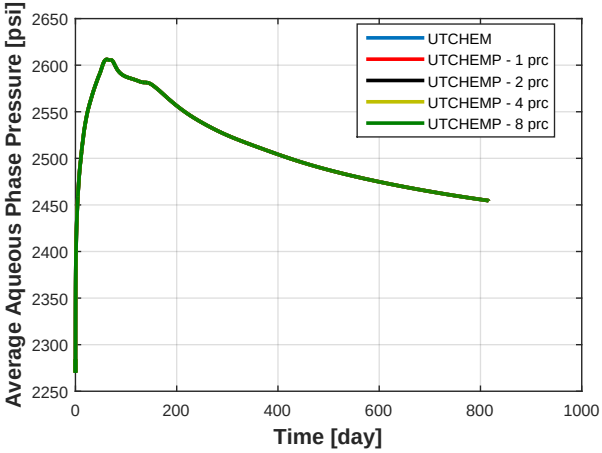


Figure 6.9 – Average aqueous phase pressure of case 2

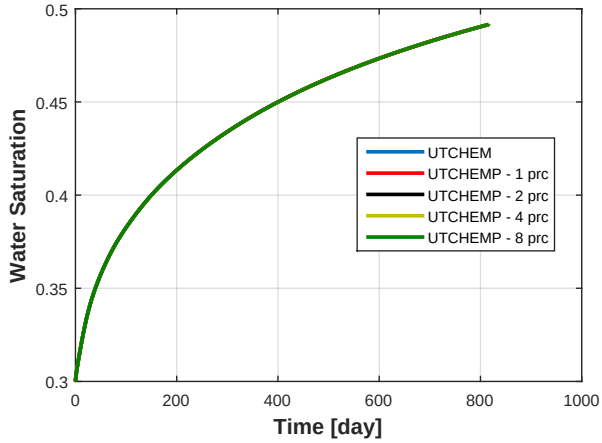


Figure 6.10 – Average water saturation of case 2

Table 6.4 has the computational time values obtained at the main operations executed by the simulator. The results are similar to the last case. As the number of processors increase, the initialization time increase, but the other times decrease. It was not noted increasing in the time to compute concentrations, but its decreasing is not so sharp as the other

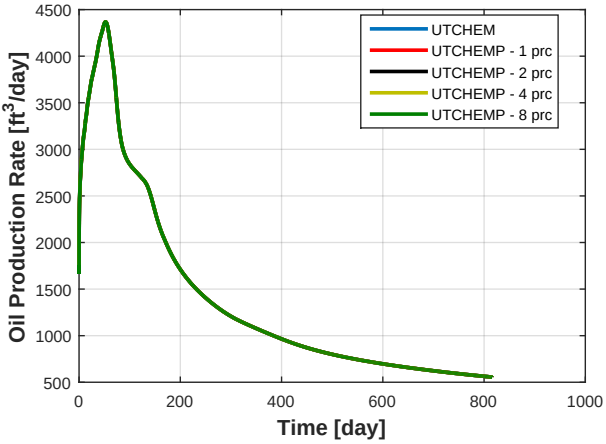


Figure 6.11 – Overall oil production rate of case 2

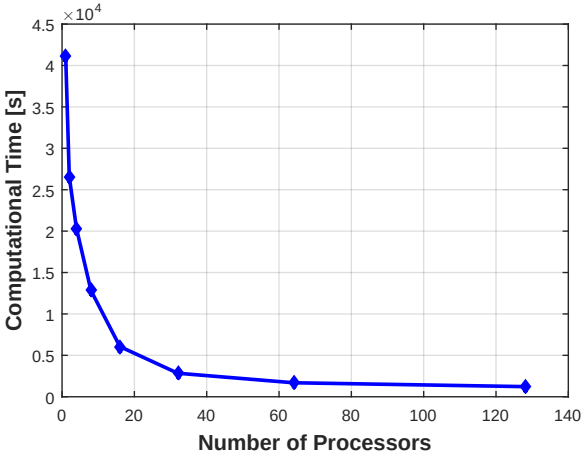


Figure 6.12 – Computational time according to the number of processors of case 2

variables. Furthermore, the grid is more refined and as a consequence the cost of the grid-related operations is higher. The grid operations in the main routine that computes concentrations are explicit and thus its scalability is higher.

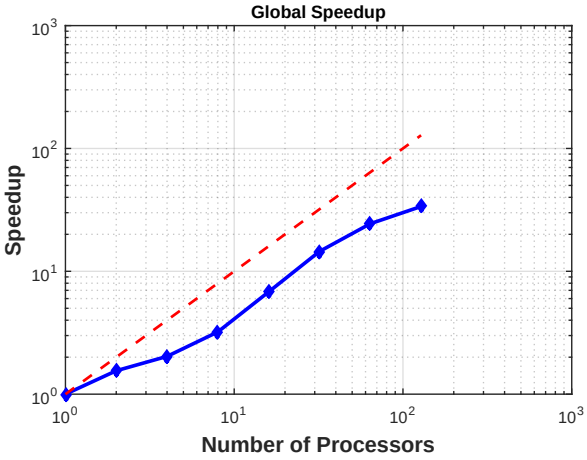


Figure 6.13 – Global speedup of case 2

Table 6.4 – Computational times in seconds of case 2

Prcs	Total Time	Init.	Trans.	Conc.	Salinity	LS Solving
1	41196.65	4.833	466.2	4703.7	82.00	35432.4
2	26555.64	4.525	296.0	2574.4	57.94	23339.6
4	20290.73	4.264	225.6	1590.7	47.97	18242.7
8	12824.40	4.252	114.1	1043.8	28.79	11521.7
16	6056.945	7.186	49.79	514.05	13.73	5420.73
32	2839.301	12.05	24.75	288.88	6.186	2480.31
64	1688.573	17.69	14.74	272.75	3.218	1353.55
128	1222.323	38.79	9.253	186.51	2.699	950.441

Figure 6.14 shows the contribution of the main routines on the total computational time. The results are close to the ones of the last case. Again, the solving of the system of linear equations is by far the most time-consuming operation. This operation however is executed by an external solver (PETSc [4]) and as a consequence we do not have much control on it. The speedup curve of the linear system solving is not presented here because it is very similar to the global speedup.

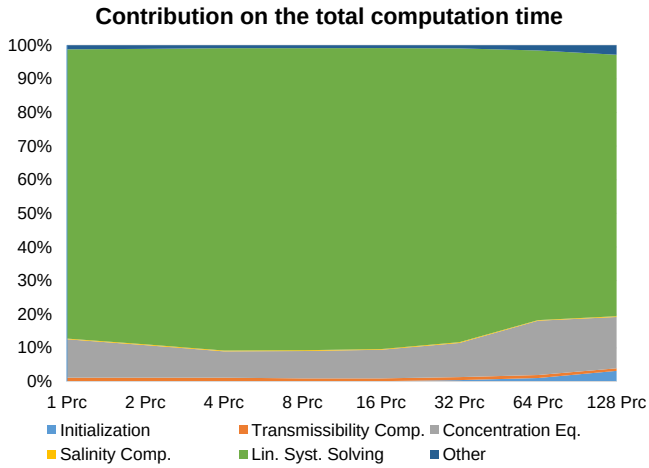


Figure 6.14 – Contribution on the total computational time of case 2

6.3 Case 3

The third case is a polymer flooding. There is a single injector well surrounded by four producer wells, as illustrated in Figure 6.15. It is injected a mixture of water, polymer, chloride, and calcium at a constant flow rate. The producer wells operate with constant bottom hole pressure. The reservoir has length 1640.5 ft, width 1640.5 ft, and thickness 10.8 ft. The coarse grid is 15x15x3, given a total of 675 grid blocks. Permeability and porosity obey a 3D stochastic distribution. Initially the aqueous phase pressure and saturation are uniform and have values of 100 psi and 0.38, respectively. The simulation runs until 1500 simulation days. Table 6.5 shows the main characteristics of this case.

Similarly to the previous cases, this case was validated against an older serial version of the simulator. The results are presented in Figures 6.16, 6.17, and 6.18. Again, it was used 1, 2, 4, and 8 processors in UTCHEMP and the variables compared are the aqueous phase pressure, aqueous phase saturation, and the oil production rate. The results are in good match.

For performance evaluation, the grid size was increased to 500x500x4, which gives a total of 1,000,000 grid blocks. The wells were placed in a similar pattern. It was used 1, 2, 4, 8, 16, 32, 64, 128, and 256 processors.

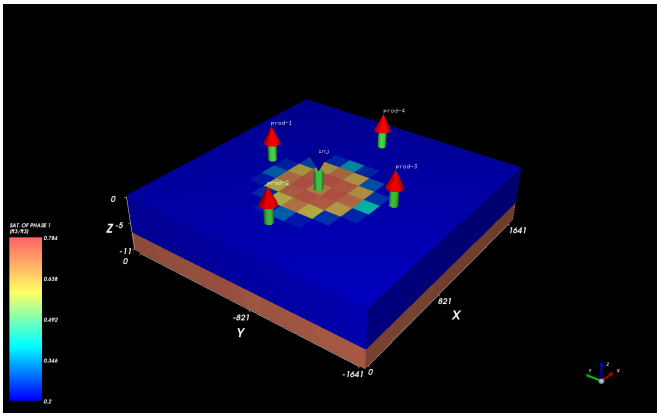


Figure 6.15 – Case 3 aqueous phase saturation after 600 simulation days

Table 6.5 – Case 3 properties

Property	Value
Reservoir lenght	1,640.5 ft
Reservoir width	1,640.5 ft
Reservoir thickness	10.8 ft
Coarse grid	15x15x3 (675)
Fine grid	500x500x4 (1000000)
Number of components	9
Porosity	from 0.184 to 0.416
Permeability in x dir.	477.83 to 3,423.3 mD
Permeability in y dir.	equal to perm. in x dir.
Permeability in z dir.	762.79 to 3,025.8 mD
Initial water saturation	0.38
Initial reservoir pressure	3000 psi
Depth	2000 ft
Number of injector wells	1
Number of producer wells	4
Simulation days	1500 days

The final simulation time was reduced from 1500 days to only 50 days, otherwise the simulation running with small number of processor would take much more than one day to finish, which is the maximum simulation time allowed in TACC. Figure 6.19 shows the results of the total computational time, while Figure 6.20 presents the corresponding speedup. The

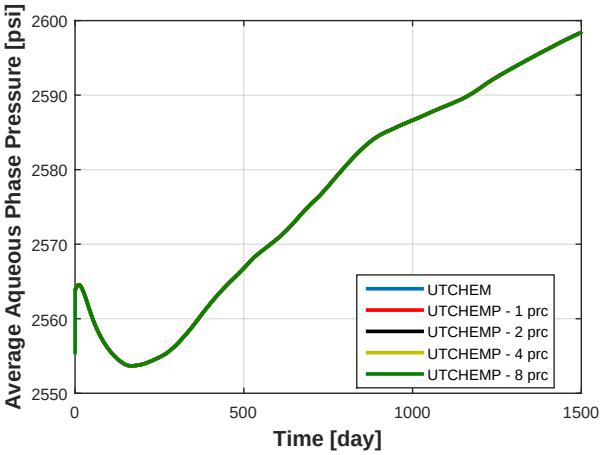


Figure 6.16 – Average aqueous phase pressure of case 3

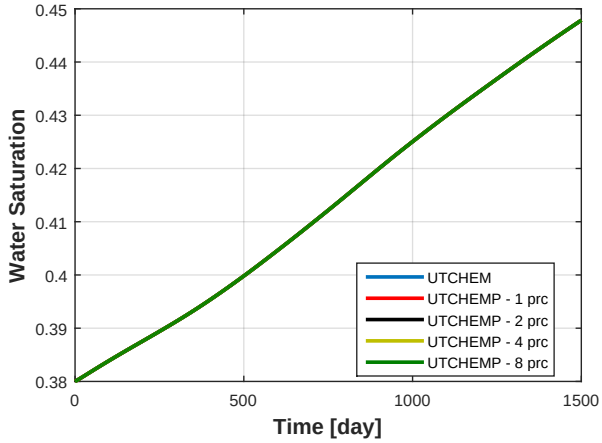


Figure 6.17 – Average water saturation of case 3

computational time could be reduced up to 130 times, from almost half a day running with a single processor to about five minutes running with 256 processors. As the previous case, the speedup is similar to what was predicted by the Gustafson-Barsis's Law. In fact, with up to 8 processors only a single cluster node is used and thus a single memory module is

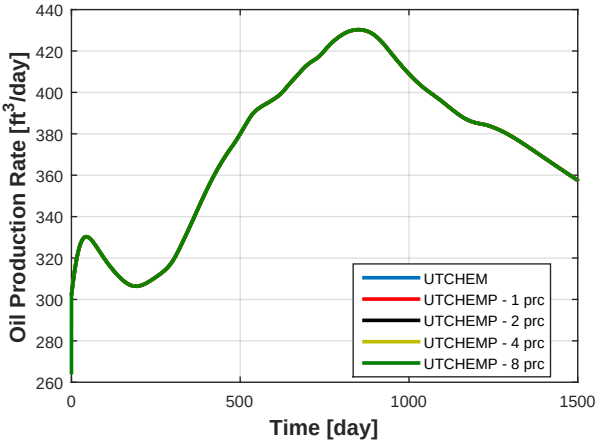


Figure 6.18 – Overall oil production rate of case 3

used. With more processors, more nodes are used and consequently more memory is also available, contributing to make the simulation faster.

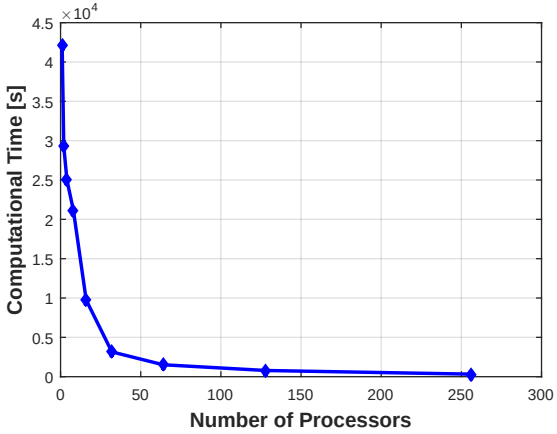


Figure 6.19 – Computational time according to the number of processors of case 3

In Table 6.6 the computational time of the main operations are presented. On the other hand, the contributions on the total computational

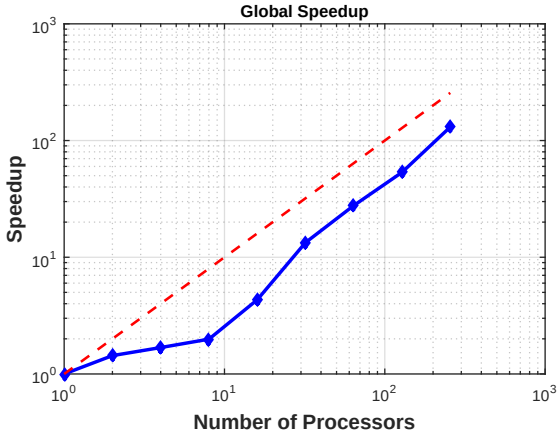


Figure 6.20 – Global speedup of case 3

time of those operations are shown in Figure 6.16. As before, the bottleneck is still solving the system of linear equations. It is interesting to note that the drop on the computational time of this operation is relatively small with up to 8 processors and is sharp with more than 8 processors. Furthermore, as in the previous cases the initialization cost increase as the number of processor increases. With 256 processors, the time required to make the initialization is about 20% of the total time.

Table 6.6 – Computational times in seconds of case 3

Prcs	Total Time	Init.	Trans.	Conc.	Salinity	LS Solving
1	42044.4	11.79	949.7	4065.5	93.77	36533.2
2	29265.9	10.94	643.3	2133.1	66.39	26190.2
4	24969.1	11.70	564.9	1403.8	57.77	22780.8
8	21157.3	10.35	273.2	844.32	33.58	19899.1
16	9733.43	10.74	97.59	408.47	15.42	9157.53
32	3148.89	16.60	39.36	194.72	7.101	2869.94
64	1515.85	27.48	18.97	96.641	3.178	1329.86
128	787.085	49.10	9.603	63.000	1.653	653.203
256	321.943	61.46	6.022	55.234	1.236	185.550

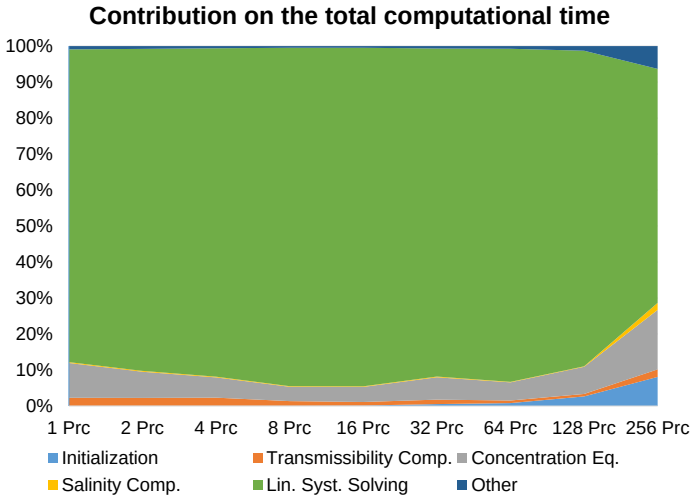


Figure 6.21 – Contribution on the total computational time of case 3

6.4 Case 4

The fourth case is an aquifer recovering simulation. The reservoir initial water and oil saturations are uniform and their values are 0.65 and 0.35, respectively. There are an injector and a producer wells that are placed according to the classic five-spot problem: one well at bottom left corner and the other one at the top right corner, as described in Figure 6.22. A mixture of water, surfactant, and polymer, which forms a microemulsion phase, is injected, helping to move the oil out of the reservoir. The reservoir has dimensions 250 x 250 x 10 ft and the coarse grid used is 11 x 11 x 2 (242 grid blocks). The reservoir is also homogeneous with porosity 0.20, x and y permeabilities 500 mD, and z permeability 50 mD. Table 6.7 presents the main features of the case.

Figures 6.23, 6.24, and 6.25 shows the validation results for this case. Differently from the other cases, the validation results differ. However, Figure 6.25 reveals that the oil production rate is quite unstable and thus a small difference on computations may lead to a relatively large difference on the final results. In fact, the code was extensively debugged and no errors were found. It was verified that even if the linear system is exactly the same, the results may differ a little bit depending on the number of

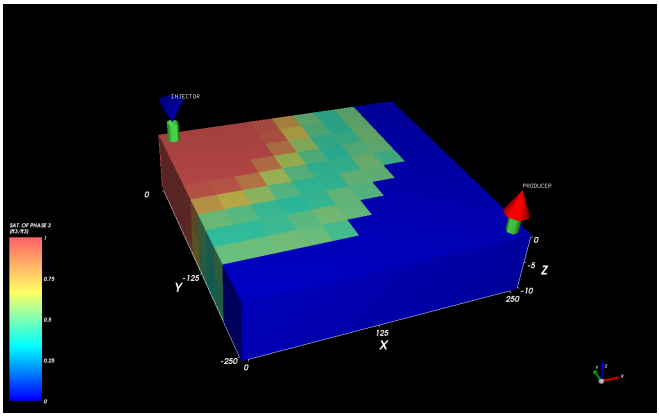


Figure 6.22 – Microemulsion phase saturation after 350 simulation days of case 4

Table 6.7 – Case 4 properties

Property	Value
Reservoir lenght	250 ft
Reservoir width	250 ft
Reservoir thickness	10 ft
Coarse grid	11x11x2 (242)
Fine grid	400x500x10 (2000000)
Number of components	11
Max. number of phases	3
Porosity	0.20
Permeability in x dir.	500 mD
Permeability in y dir.	500 mD.
Permeability in z dir.	50 mD
Initial water saturation	0.65
Initial reservoir pressure	2500 psi
Depth	1000 ft
Number of injector wells	1
Number of producer wells	1
Simulation days	1500 days

processors used. This fact combined with some instabilities probably explain why the results are a little different. Despite the results being a little different, it is worth noting that all validation results follow a similar

pattern, indicating that the code is probably right.

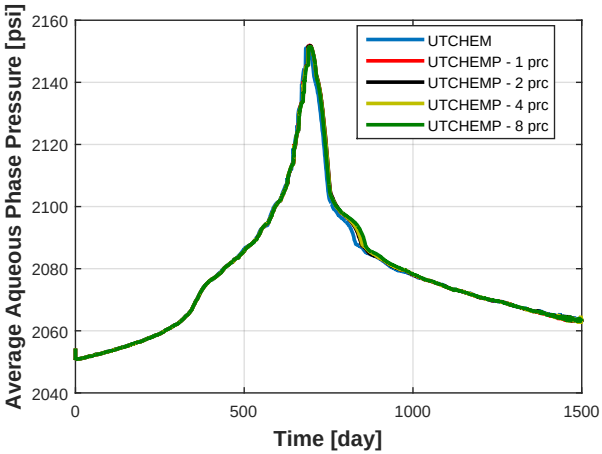


Figure 6.23 – Average aqueous phase pressure of case 4

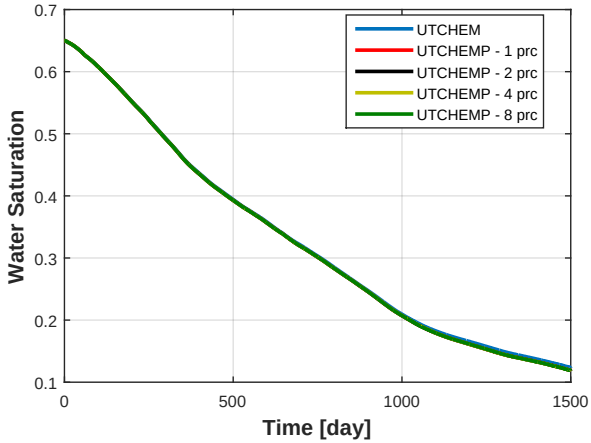


Figure 6.24 – Average water saturation of case 4

Still referencing Figure 6.25, the oil production is close to zero at the beginning of the simulation due to the small relative permeability of phase oil. Its value is about 10^{-4}, while the relative permeability of the aqueous

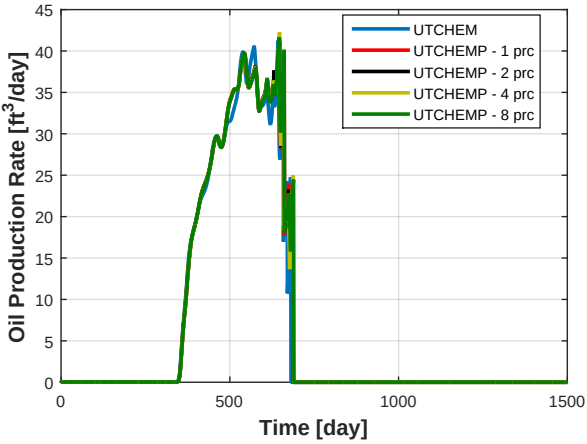


Figure 6.25 – Overall oil production rate of case 4

phase is about 10^{-1}. As a consequence, the oil mobility is small and thus the oil being produced, which is calculated using Equation (5.11), is also small. Another aspect of Figure 6.25 that may seem strange is the oil flow rate dropping to zero at about 700 days. What happened is that phase oil starts to be considered a microemulsion phase because the surfactant concentration surpass a predefined value.

The grid used to evaluate UTCHEMP's performance in this case has 2,000,000 (400x500x10) grid blocks. The number of processors used is 1, 2, 4, 8, 16, 32, 64, 128, and 256. As in the previous case, the final simulation time was reduced to make possible the running of the cases with small number of processors. The total computational time decreased from approximately 2,5 hours with one processors to only about 4 minutes with 256 processors, as one may see in Figure 6.26. The speedup is plotted in Figure 6.27. Again, the speedup is similar to what Gustafson-Barsis's Law predicts, but in this case it saturates with large number of processors, as predicted by the Amdahl's Law.

Table 6.8 presents the computational time of some UTCHEMP"s main operations, whilst Figure 6.28 shows their contribution on the total computational time. The solving of the system of linear equations is the main contributor to the computational time only when not so many processors are used. With 256 processors, the initialization time, which involves op-

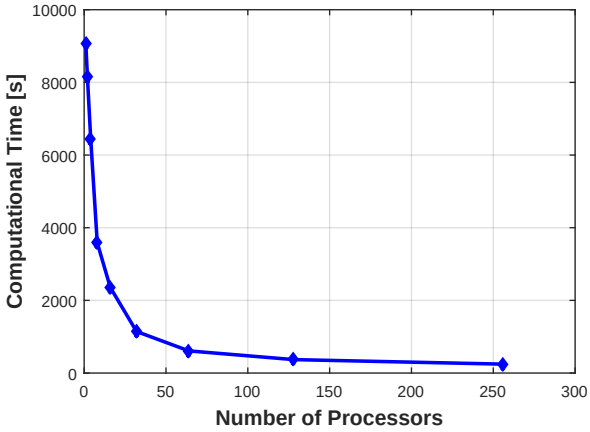


Figure 6.26 – Computational time according to the number of processors of case 4

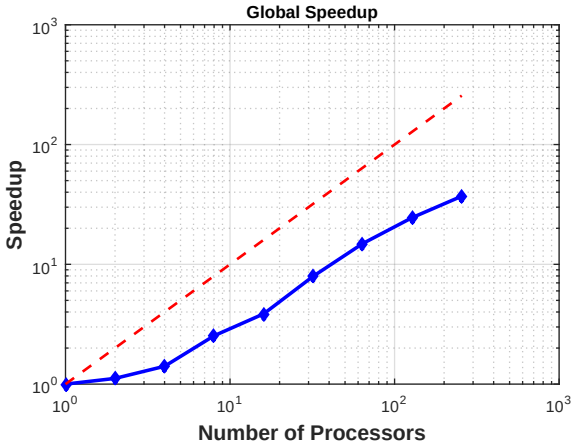


Figure 6.27 – Global speedup of case 4

erations such as reading, transferring the input data, and setting up the reservoir initial state, takes most of the simulation time. This is actually natural, since there is much communication in this process and the number of time steps is relatively small.

This case was also used to evaluate inactive grid blocks. As described

Table 6.8 – Computational times in seconds of case 4

Prcs	Total Time	Init.	Trans.	Conc.	Salinity	LS Solving
1	9086.82	68.93	151.0	1268.0	73.36	7405.0
2	8140.34	66.23	95.64	684.55	41.23	7178.1
4	6448.90	67.88	81.49	462.36	31.10	5751.9
8	3585.60	67.56	45.07	287.22	20.27	3132.9
16	2354.57	63.72	22.11	143.65	9.256	2099.3
32	1143.43	71.43	8.592	77.279	4.488	973.84
64	609.544	79.13	3.892	45.771	2.318	474.68
128	370.061	90.65	2.237	30.796	1.259	242.94
256	245.197	120.9	1.290	24.866	0.818	95.585

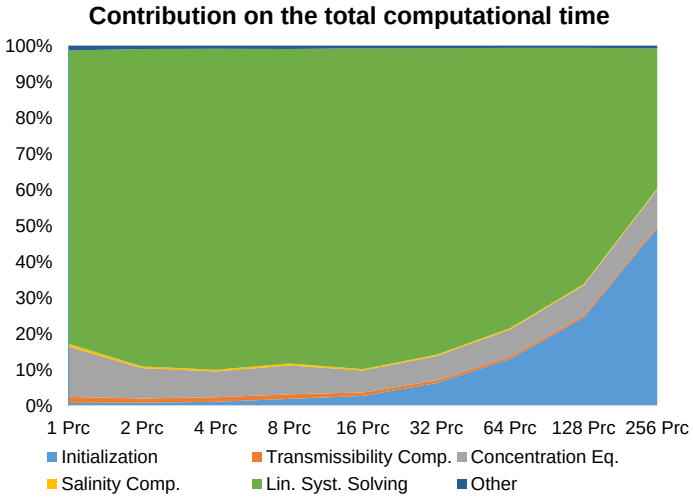


Figure 6.28 – Contribution on the total computational time of case 4

in section 5.1.2, inactive grid blocks may be used to more accurately represent the reservoir geometry. They behave like impermeable boundaries and thus no flux through them should occur. In this case eight grid blocks at the middle of the reservoir was set as inactive and as a consequence the fluids should flow around the inactive cells. Figure 6.29 shows the microemulsion phase saturation after 355 simulation days using the coarse grid. As expected, the microemulsion saturation at inactive grid blocks

remained null, indicating that in fact there is no flow through them.

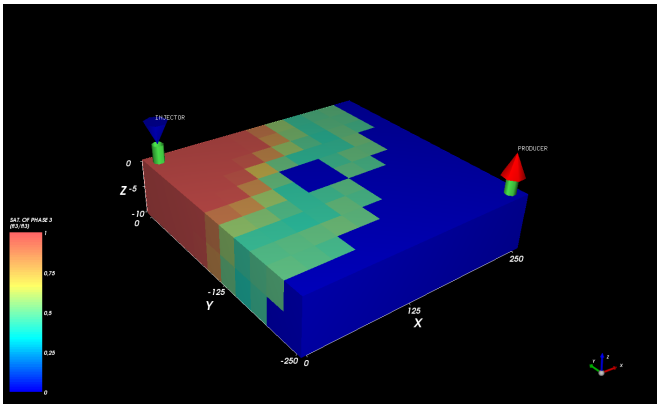


Figure 6.29 – Microemulsion phase saturation after 355 simulation days using inactive cells of case 4

6.5 Case 5

The fifth case is a two-phase problem solved using EFVLib. The grid is hybrid, unstructured, and it is nearly cylindrical around the wells. There are 8073 nodes, 35286 tetrahedra, 544 hexahedra, 544 prims, and 576 pyramids. Its shape is similar to Brazil's territory, as illustrated in Figure 6.30. There are two horizontal producer wells and three vertical injector wells. The injectors injects a total of 50 ft^3/day. The domain is supposed homogeneous and isotropic with permeability 40 mD and porosity 0.1. Table 6.9 summarizes the case properties.

Both water and oil are considered incompressible and immiscible. The problem is modelled by the water mass conservation equation

$$\frac{\partial}{\partial t}\left(\phi S_w\right) = \nabla\cdot\left(\lambda_w \mathbb{K}(\nabla P - \gamma_w \nabla h)\right) + q_w''', \tag{6.1}$$

by the global mass conservation equation

$$\nabla\cdot\left(\lambda_T \mathbb{K}(\nabla P - \gamma_T \nabla h)\right) + q_T''' = 0, \tag{6.2}$$

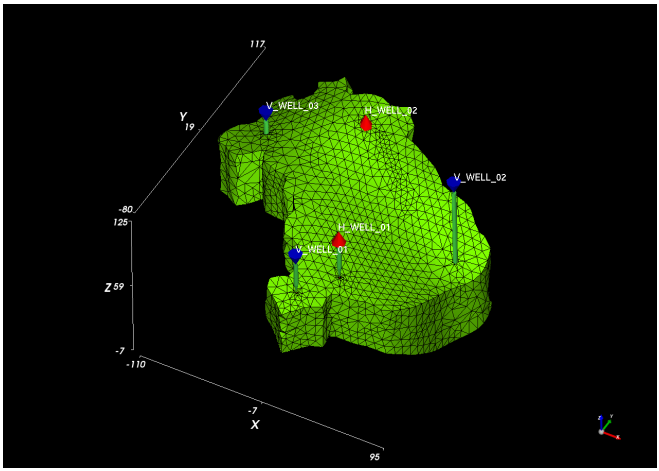


Figure 6.30 – Illustration of the grid and wells of case 5

Table 6.9 – Case 5 properties

Property	Value
Reservoir lenght	197 ft
Reservoir width	205 ft
Reservoir thickness	132 ft
Grid nodes	8073
Tetrahedra	35286
Hexahedra	544
Prims	544
Pyramids	576
Number of phases	2
Number of components	2
Porosity	0.10
Permeability	40 mD
Initial oil saturation	1.0
Number of injector wells	3
Number of producer wells	2
Simulation days	200 days

and by the continuity of the fluid saturations

$$S_w + S_o = 1 \tag{6.3}$$

There are three equations and three unknowns (P, S_w, and S_o). Since the mobility λ depends on the phase saturations, the problem is nonlinear. In order to solve the above equations, it is used an IMPES method: the global mass conservation equation is solved implicitly to obtain the pressure field and then the saturations are computed explicitly with the other two remaining equations. This method however has stability constraints. For this case, it was necessary a time step of 0.001 day, which required 200000 iterations to reach 200 days. Once the time step is really small, the saturation field barely changes from a simulation time to another and thus nonlinearity effects are negligible. In Figure 6.31 it is illustrated the water saturation field after 170 days.

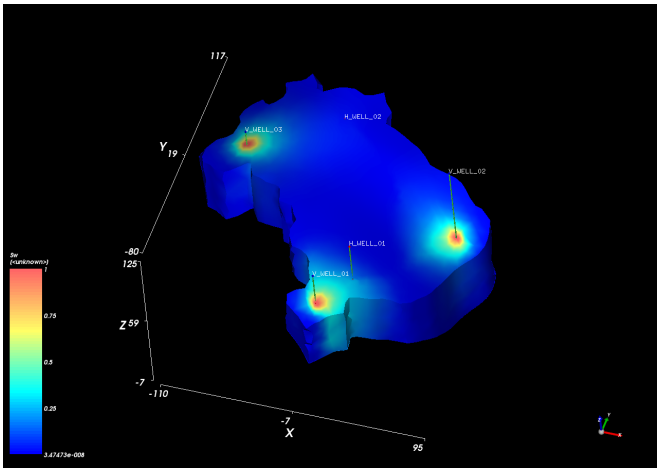
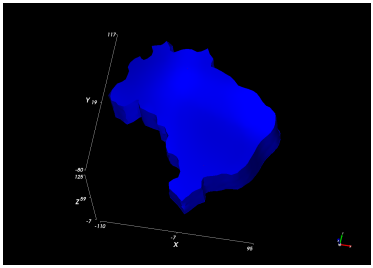


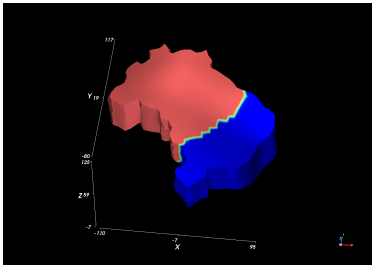
Figure 6.31 – Saturation field after 170 days of case 5

The case was run with 1, 2, 4, 8, 16, and 32 processors. In Figure 6.32 it is illustrated the division of the grid according to the number of processors. All nodes within a same subdomain have same color. It is worth noting that a subdomain may be disconnected, as it happens when four processors are used. This however is not a problem since the methodology using ghost nodes is quite general and does not require connected subdomains.

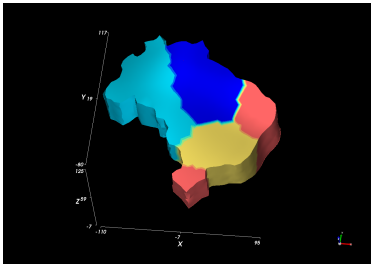
The average values of pressure, water saturation, and oil production rate are plotted in Figures 6.33, 6.34, and 6.35. As one may note, the results match, which is strictly necessary to a parallel simulator. Figure 6.36



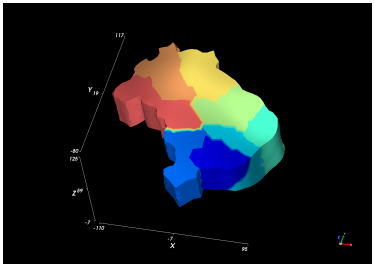
(a) 1 processors



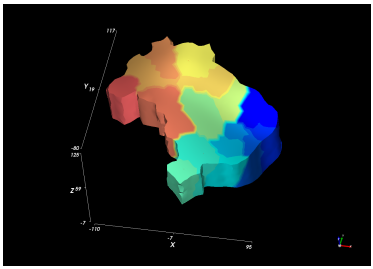
(b) 2 processors



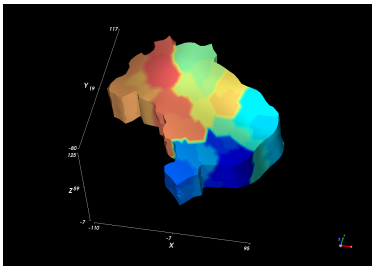
(c) 4 processors



(d) 8 processors



(e) 16 processors



(f) 32 processors

Figure 6.32 – Grid division of case 5 using different number of processors

shows the total computational time spent by the simulation and Figure 6.37, the corresponding speedup. The speedup is almost linear using up to 8 processors. If more processors are employed it will be required to use more than one cluster's node, thus increasing the communication cost. The current case is actually very small. When for example 32 processors are used, each processor handles only about 250 nodes. Despite being small, this case shows that it is possible to achieve a good speedup even when the case is small.

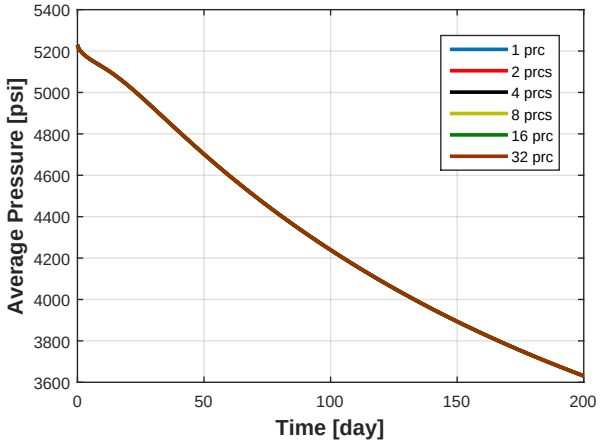


Figure 6.33 – Average pressure of case 5

Figure 6.38 shows the contribution of some operations on the total computational time for different number of processors. Since there is a lot of time steps along the simulation (200000), the contribution of reading, dividing, and assembling the grid on the total computational time is negligible. The main computational cost comes from the computation of water saturation and pressure. In fact, the contribution of pressure computation grows as the number of processors increase, while the contribution of water saturation computation decrease. This is reasonable, because the computation of pressure is implicit – a linear system is solved to obtain the field – and thus data must be exchanged between the processors. Furthermore, the computation of the water saturation is explicit. All of the processors have the data they need to compute the new water

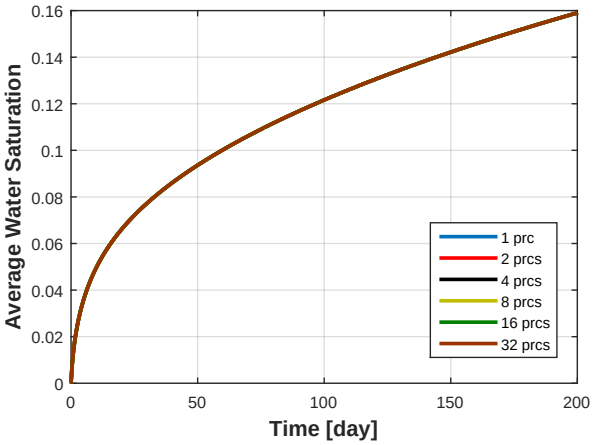


Figure 6.34 – Average water saturation of case 5

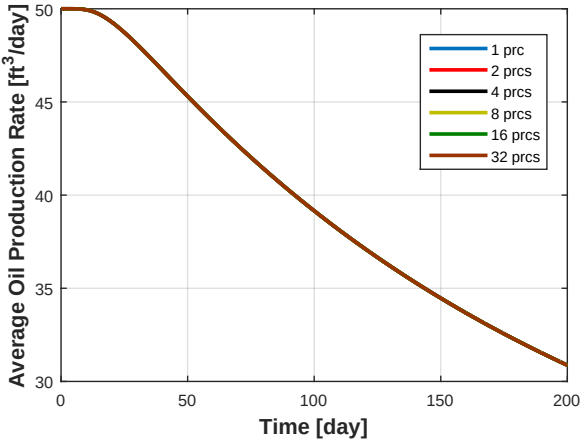


Figure 6.35 – Overall oil production rate of case 5

saturation field based on a new pressure field. The only data exchange regards the updating of such variable at the ghost nodes. Figure 6.39 and Figure 6.40 presents the speedup of the pressure and water saturation computation. It is clear that the former is more scalable than the latter. In fact, the time spent computing the pressure field gets higher when 32

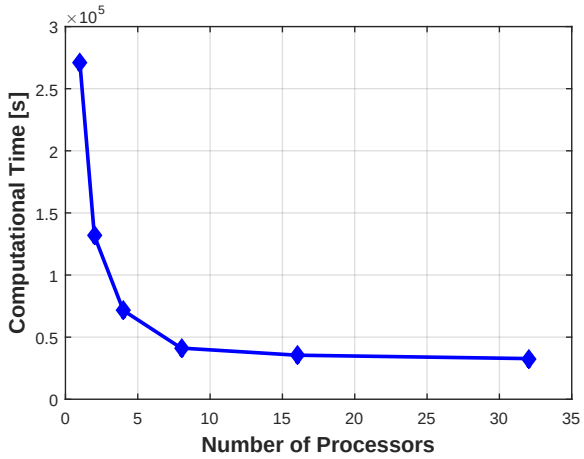


Figure 6.36 – Computational time according to the number of processors of case 5

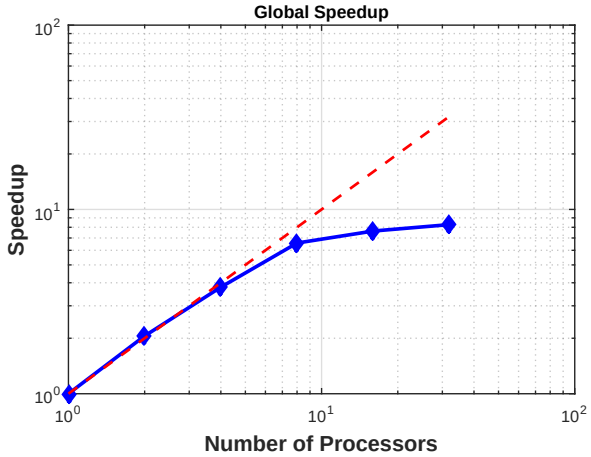


Figure 6.37 – Global speedup of case 5

processors are used instead of 16 processors.

Although in the present case the time spent reading, dividing, and assembling the grid is negligible, it might be relevant if the number of time steps were much smaller. Figure 6.41 shows the computational time

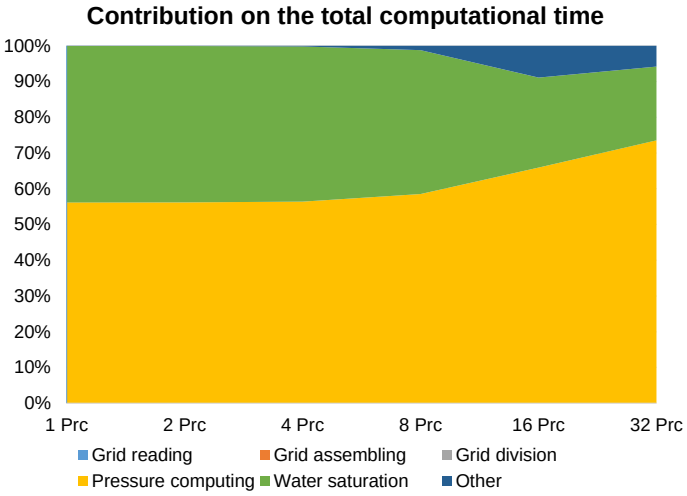


Figure 6.38 – Contribution on the total computational time of case 5

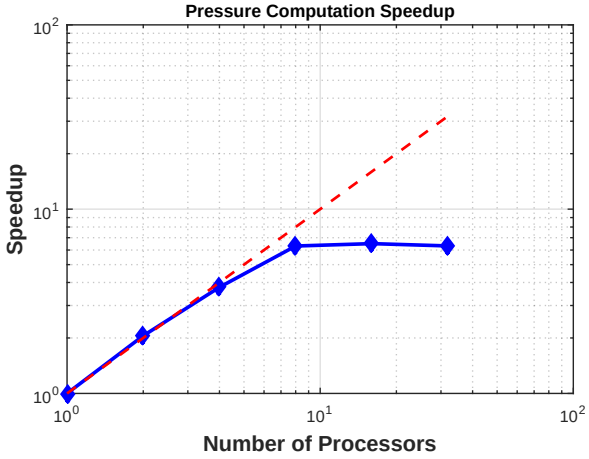


Figure 6.39 – Pressure computation speedup of case 5

according to the number of processors. The cost of reading a grid from a file is always the same because such operation is executed by a single processor, no matter the total number of processors allocated. Furthermore, when a single processor is used there is no cost dividing the grid, but

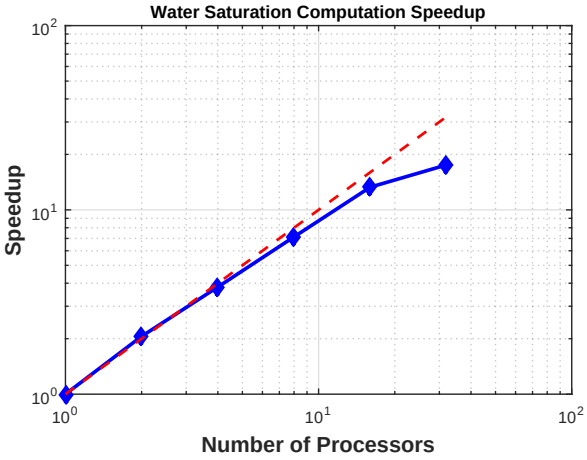


Figure 6.40 – Water saturation computation speedup of case 5

the cost for assembling is higher because the processor must work on the whole grid. As the number of processors is increased, the assembling cost decreases, but the division cost increases. The optimal point happens when 16 processors are used.

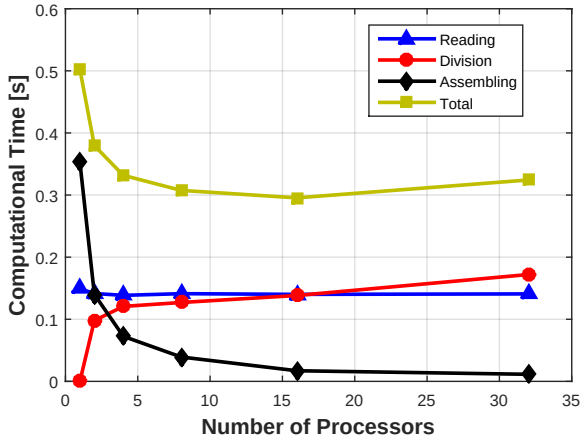


Figure 6.41 – Computational time of some grid operations of case 5

The quality of the grid division was also evaluated for the present

case. It was noted that the load balance is an important parameter to optimize performance. All processors should have almost the same number of nodes, implying that the number of nodes should be the global number of nodes divided by the number of processors. This however is an ideal situation. Assuming the parallel architecture is symmetric, the performance is ruled by the overloaded processor. The ratio between the surplus number of nodes and the ideal number of nodes was used to quantify the load balance. The surplus number of nodes is here defined as the difference between the number of local nodes and the ideal number of nodes. There is a different ratio for each processor, but the ratio that really matters is the maximum. The variation of the maximum ratio are plotted in Figure 6.42. Note that all values are bellow 3%, which is a nice result since it indicates that a performance slow down by load inbalance also should be at most 3%.

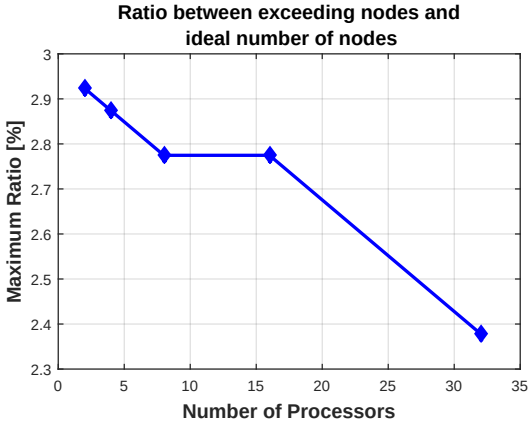


Figure 6.42 – Maximum ratio between the surplus number of nodes and the ideal number of nodes of case 5

Another aspect concerning the grid division quality is the communication. The grid division should minimize the *size* of subdomains's interfaces in order to communicate less data. It is hard however to evaluate if the division is good because we do not know what is the optimal point. Actually, even the concept of interface size is abstract. In this study is straightforward to define the interface's size as the number of ghost nodes. From the communication point of view, the quality of the grid division

is as good as smaller is the ratio between the ghost and the local nodes. Figure 6.43 shows the average ratio obtained using different number of processors. It is clear that as the number of processors increases, the division quality becomes poor. This helps to explain why the speedup results are not good for more than 16 processors. Nevertheless, it does not mean that the division quality can be improved because we do not know what is the optimal ratio.

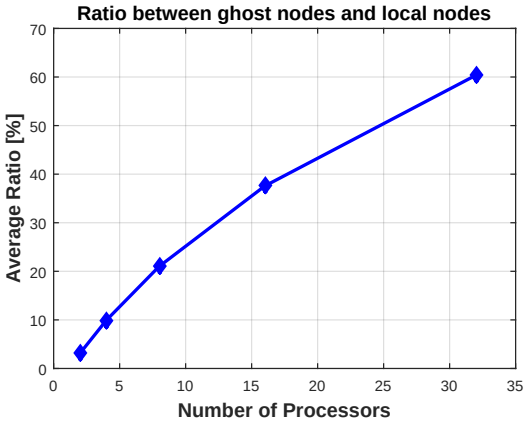


Figure 6.43 – Average ratio between the number of ghost nodes and the number of local nodes of case 5

6.6 Case 6

This case is similar to the last problem considered. A two-phase, incompressible, immiscible model was implemented using EbFVM and IMPES. However, the grid used here is more refined. It has 1971750 tetrahedra and 378259 nodes. There are two vertical well: one producer and one injector, as illustrated in Figure 6.44. We could of course run an even bigger case, but such case would probably require more memory than is available when a single processor is used. The properties of this case are similar to the last cases's and are summarized in Table 6.10. In this case however there is no validation step because the physical model is the same of the last problem.

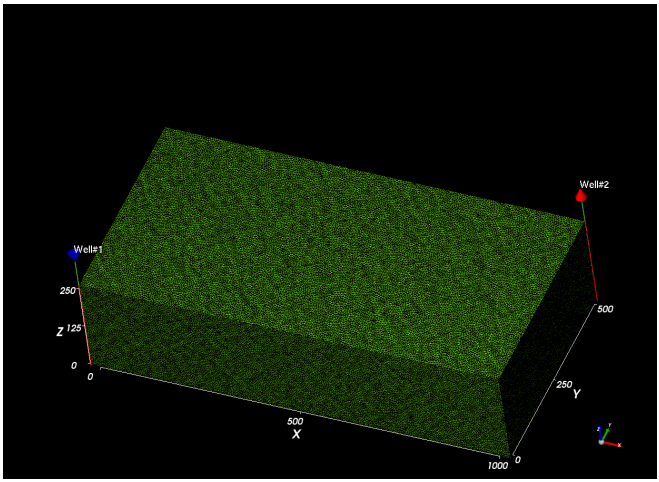


Figure 6.44 – Grid and wells of case 6

Table 6.10 – Case 6 properties

Property	Value
Reservoir lenght	1000 ft
Reservoir width	500 ft
Reservoir thickness	250 ft
Grid nodes	378259
Tetrahedra	1971750
Number of phases	2
Number of components	2
Porosity	0.10
Permeability	10 mD
Initial oil saturation	1.0
Number of injector wells	1
Number of producer wells	1
Number of time steps	100

Figure 6.45 shows the results of the computational time spent by the whole simulation according to the number of processors. As expected, the computational time decreases smoothly as the number of processors used increase. The corresponding speedup is presented in Figure 6.46. Up to 8 processors the speedup is close to linear. With more processors

the communication cost increases and the speedup deviates from linear. With 128 processors for example the speedup is only about 44. In this case each processor handles about 3000 nodes, which is not so many.

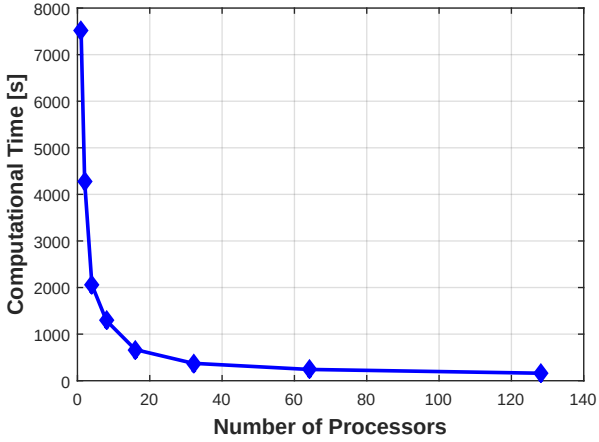


Figure 6.45 – Computational time according to the number of processors of case 6

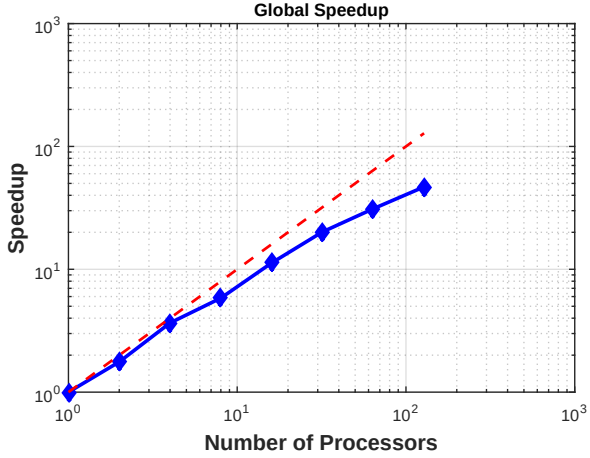


Figure 6.46 – Global speedup of case 6

Figure 6.47 shows the contribution of the main operations to the global computational time. Again, the most costly operations are the pressure and water saturation computation. As the number of processors increases, the cost to compute the water saturation becomes smaller while the cost to compute the pressure is relatively the same. In fact, one may see from the speedups illustrated in Figures 6.48 and 6.49 that the water saturation computation is a highly scalable operation. The computation is explicit. As long as the pressure field are updated at the ghost nodes, no communication is necessary except that for the updating of the saturation values at the ghost nodes. The computation of the pressure field, on the other hand, is implicit and thus demands the assembling and solving of a linear system. As the number of processors increase, the communication becomes more expensive and the speedup saturates. The speedups evaluated in this case are actually similar to the ones of the case 5. However, the performance deteriorates at a higher number of processors since the grid have many more elements and thus the grid related operation contributes more to the global computational time.

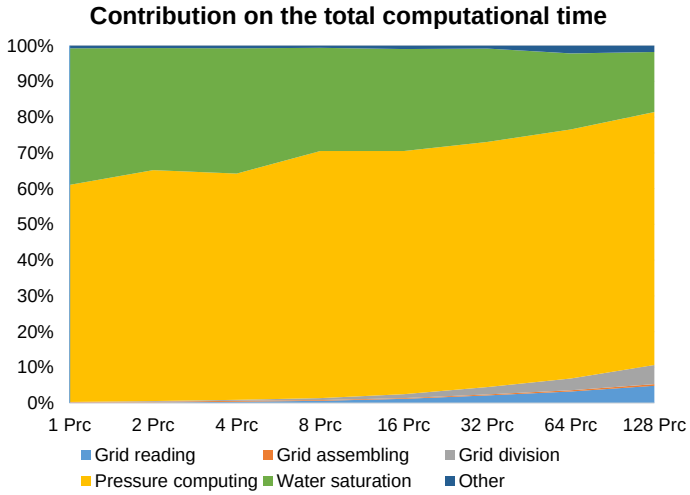


Figure 6.47 – Contribution on the total computational time of case 6

Figure 6.50 shows the computational time associated to some of the operations required to build the grid computational structure. The results are similar to the ones obtained in the last case. The assembly cost is high

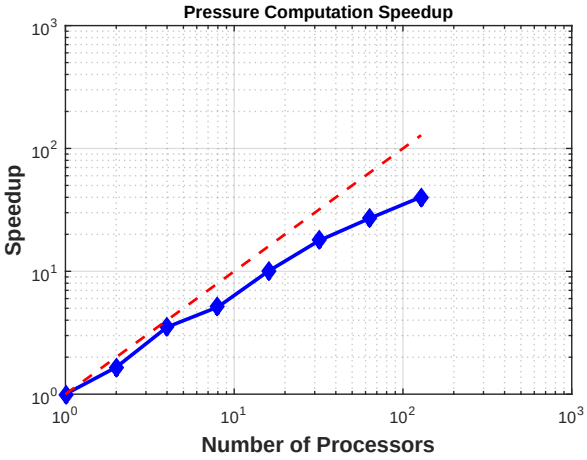


Figure 6.48 – Pressure computation speedup of case 6

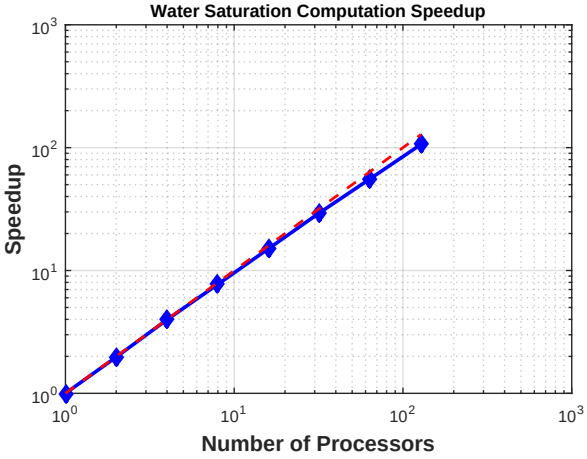


Figure 6.49 – Water saturation computation speedup of case 6

with small number of processors, but it decreases sharply as this number increases. With a large number of processors the operations that have more influence on the computational time are the grid reading and grid division. However, there is no clear optimal point differently from the last case. With 16 processors or more the computational time is almost stable.

As the number of processors increases the cost of division gets higher. The trend is to have a higher total computational cost if more processors are used.

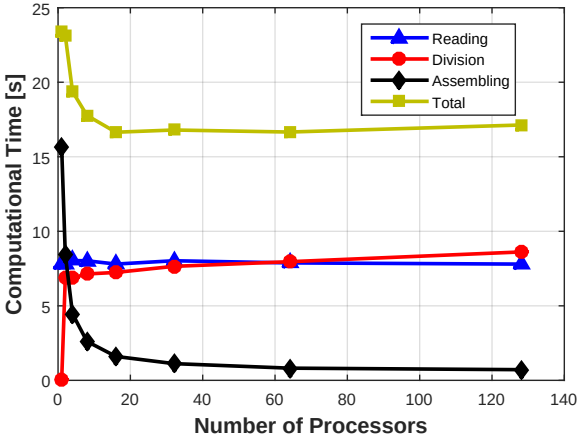


Figure 6.50 – Computational time of some grid operations of case 6

Figures 6.51 and 6.52 present the result of the variables used to evaluate the quality of the grid division. The load balance parameter is surprisingly good for small number of procesors. With two processors, it is almost zero, indicating that the load balance is almost perfect. Even with a highest number of processors the ratio is less than 3%. The communication parameter on the other hand follows the same pattern noted in the last case, although the values are much smaller than that case. With 32 processors, the ratio is about 11% in this case whereas it is about 60% in the other one. This of course happens because the grid in this case has many more nodes than the last one's grid.

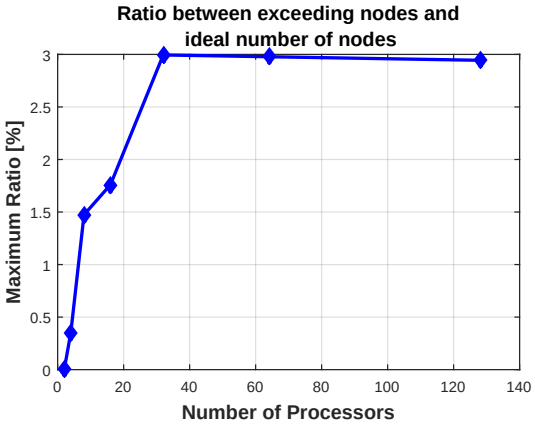


Figure 6.51 – Maximum ratio between the surplus number of nodes and the ideal number of nodes of case 6

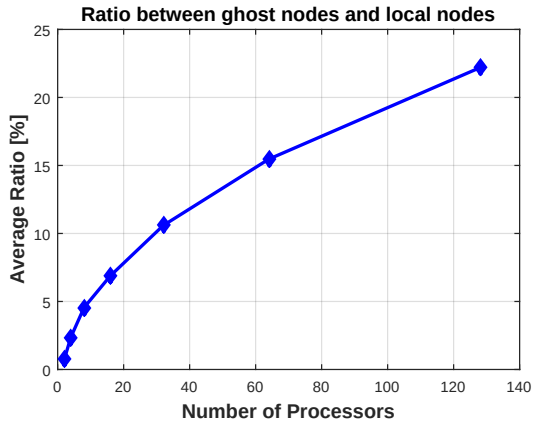


Figure 6.52 – Average ratio between the number of ghost nodes and the number local nodes of case 6

CHAPTER

7

Conclusions

7.1 Summary

This study was developed with the objective of improving computational performance of numerical reservoir simulators. The methodology to improve performance is by the application of parallel computing. Parallel computing was employed by using several processors concurrently. Each processor works on a small part of the problem. By reducing the size of the local problem it is possible to sharply increase the overall performance. The total computational time is the computational time spent by the overloaded processor. Ideally the problem is divided evenly among the processors and as a consequence the speedup is equal to the number of processors.

There are three main types of parallel architecture concerning the memory arrangement: shared, distributed, and hybrid-memory computers. Shared-memory computers are commonly limited to a few number of processors. Distributed-memory computers, on the other hand, may have a large number of processors, each one using a distinct memory module, but require communication between processors, which may slow down the performance. The code developed in this study uses the

MPI paradigm and hence can be used in either distributed or hybrid-memory computers.

Parallel computing was applied to two reservoir simulation softwares: UTCHEM and EFVLib. UTCHEM is a three-dimensional, multicomponent, multiphase, compositional, variable temperature, finite-difference reservoir simulator developed at The University of Texas at Austin. It can be used to simulate enhanced recovery of oil and enhanced remediation of aquifers. EFVLib, on the other hand, is a library that helps its user to develop its own codes using the Element-based Finite Volume Method. This library has several methods that enable the user to read an unstructured grid, assemble control volumes, easily access grid entities, evaluate shape functions, among other features. Despite being developed for application on reservoir simulation, EFVLib is actually general and can be employed for the development of any software that uses the EbFVM.

The idea of employing parallel computing on two softwares instead of just one is that in UTCHEM we deal with complex physical models but the grid is structured and Cartesian. The grid treatment in EFVLib on the other hand is much more complex since it deals with unstructured grids. It was not necessary however treat complex physical models because such models is the final user who implements. So, in this study there was the opportunity of developing methodologies for both structured and unstructured grids, as well as complex physical models.

The parallelization methodology employed in both cases was based on domain decomposition. It is recognized that the main computational cost comes from grid related operations. Each processor works only on a part of the global domain. In UTCHEM the domain is divided taking the direction y as the reference. At the borders of each subdomain created there are ghost cells, which are additional grid blocks used to represent and store values from grid blocks that are in a neighbor subdomain. PETSc was used to solve systems of linear equations due to its efficiency and its parallel computing support. Additionally, it was implemented in UTCHEM inactive cells and a more user-friendly input file format. The parallel version of UTCHEM was called UTCHEMP to distinguish it from its serial version.

The division of the grid in EbFVM is more complex because the grid is unstructured and thus there is no predefined rule of how the control volumes are connected to each other. The notion of control volumes (or

nodes) being connected to each other lead to the representation of the grid as a graph. It was used an external graph partitioning library to partition the graph representation of a grid respecting the load balance and minimum communication criteria. Based on the graph, the local grids are assembled and their borders are extended to contemplate nodes that are in other subdomains. This nodes are called ghost nodes and their purpose is the same of the ghost cells used in UTCHEM: represent and store values from nodes that are in another subdomain. Furthermore, proper methodologies were used to treat wells and boundaries. As in UTCHEM, PETSc was used to solve in parallel system of linear equation rising from the simulations.

7.2 Conclusions

Four cases were executed in UTCHEMP. Each one of them tries to evaluate different physical models from the simulator. There was a good agreement between the validation results of UTCHEM and UTCHEMP running with different number of processors. The only exception was the forth case. It is believed however that the code is correct and the difference is due to some instability and to the fact that the result from the linear system solving is not exactly the same when different number of processors is used.

It was verified in UTCHEMP that usually the more expensive operation is the solution of system of linear equations. This operation is the main responsible by the parallel performance achieved by the software. The evolution of the speedup is not so good when a single cluster node is used probably because there is not much memory available. With more nodes the speedup curve tends to be parallel to the ideal speedup, as predicted by the Gustafson-Barsis's Law. This behaviour is more evident when the grid size is bigger and as a consequence more memory needs to be allocated.

Two EFVLib cases were presented. In both of them it was simulated a two phase flow. Since the physical model is the same, only the first case were used to validate the software. The validations results are in good match. The first case has a small however complex grid. With 8 processors or more the speedup saturates. On the other hand, the second case has

many more control volumes and thus it was achieved a better speedup. Its speedup curve is closer to the Amdahl's Law.

It was also verified in EFVLib that, as the number of processors increases, the time require to divide the grid becomes bigger, but the time to assemble the grid at the same time gets smaller. There should be an optimal number of processors to minimize the cost to initialize a grid. Another analysis was made was referred to the quality of the grid division. There is two main aspects to consider: load balance and communication. It was verified that the load unbalance is of at most 3%, which is relatively small. The communication was evaluated by the ratio of ghost to local nodes. As the number of processors increases, such ratio grow, reaching about 60% in the first case. The increasing on the communication cost helps to explain why in some cases the performance gaining using more processors is not what would be desired.

Although the speedup is not so close to the ideal speedup, the performance improvement is really significant. Depending on the number of processors used, the software can be several times faster. In the forth case for example UTCHEMP was about 130 times faster when running with 256 processors. It is almost impossible to get such performance improvement simply by making the software algorithms more efficient. It is strongly recommended here to take advantage of the computer architectures available nowadays. In fact, the trend is to use parallel computers and our codes should adapted to that.

7.3 Suggestions for future studies

For future studies it is recommended:

- Extend the validation and evaluation of UTCHEMP. This is a huge simulator and there was not enough time to test all of its features;
- Evaluate the PETSc options used in both UTCHEMP and EFVLib. Investigate what is the configuration that optimizes performance;
- Use ParMETIS instead of Metis to improve the division of graphs representing grids;
- Create a code interface in EFVLib to use the non-linear system solving tools available in PETSc;

- Evaluate EFVLib with more complex simulation models.

Bibliography

[1] **METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices**. Department of Computer Science & Engineering / University of Minnesota.

[2] **Technical Documentation for UTCHEM-9.0: a Three-Dimensional Chemical Flood Simulator**. Reservoir Engineering Research Program, Center for Petroleum and Geosystems Engineering, The University of Texas at Austin, Austin, Texas 78712, July, 2000.

[3] **Lonestar User Guide**. *https://portal.tacc.utexas.edu/user-guides/lonestar*, 2015.

[4] BALAY, S., BROWN, J., BUSCHELMAN, K., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., SMITH, B. F., ZHANG, H. **PETSc (Portable, Extensible Toolkit for Scientific Computation)**. *http://www.mcs.anl.gov/petsc*, 2013.

[5] BOOST C++ LIBRARIES. *boost.org*, January, 2015.

[6] CORDAZZO, J. **Simulação de reservatórios de petróleo utilizando o método EbFVM e Multigrid algébrico**. Ph.D. Thesis, Universidade Federal de Santa Catarina, 2006.

[7] DONGARRA, J. J. **Sourcebook of Parallel Computing**. Morgan Kaufmann Publishers, 2003.

[8] DOROH, M. G. **Development and Application of a Parallel Compositional Reservoir Simulator**. Master's Thesis, The University of Texas at Austin, 2012.

[9] EIJKHOUT, V., CHOW, E., VAN DE GEIJN, R. **Introduction to High Performance Scientific Computing**, 2nd edition. Creative Commons Attribution 3.0 Unported, Austin, Texas, 2015.

[10] GEBALI, F. **Algorithms and Parallel Computing**. Wiley, 2011.

[11] HURTADO, F. S. V. **Formulação tridimensional de volumes finitos para simulação de reservatórios de petróleo com malhas não-estruturadas híbridas**. Tese de doutorado, Departamento de Engenharia Mecânica, Universidade Federal de Santa Catarina, Florianópolis, Brasil, 2011.

[12] KARPINSKI, L. **Aplicação do método de volumes finitos baseado em elementos em um simulador de reservatórios químico-composicional**. Master's Thesis, Universidade Federal de Santa Catarina, 2011.

[13] KARYPIS, G., KUMAR, V. **A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs**. *SIAM Journal on Scientific Computing*, v. 20, pp. 359 – 392, 1998.

[14] KARYPIS, G., KUMAR, V. **Multilevel Algorithms for Multi-Constraint Graph Partitioning**. Technical report, University of Minnesota, Department of Computer Science / Army HPC Research Center, 1998.

[15] KARYPIS, G., KUMAR, V. **Multilevel k-way Partitioning Scheme for Irregular Graphs**. *Journal of Parallel and Distributed Computing*, v. 48, pp. 96 – 129, 1998.

[16] MALISKA, C. R. **Transferência de Calor e Mecânica dos Fluidos Computacional**. LTC Editora, Rio de Janeiro, RJ, 2004.

[17] MALISKA, C. R., DA SILVA, A. F. C., HURTADO, F. S. V., DONATTI, C. N., AMBRUS, J. **Especificação e planejamento da biblioteca EFVLib. Relatório técnico SINMEC/SIGER I-02, Parte 2, Departamento de Engenharia Mecânica**. Technical report, Universidade Federal de Santa Catarina, 2008.

[18] MALISKA, C. R., SILVA, A. F. C., HURTADO, F. S. V., DONATTI, C. N., PESCADOR JR., A. V. B., RIBEIRO, G. G. **Manual de classes da biblioteca EFVLib**. Relatório técnico SINMEC/SIGER I-06, Parte 1, Departamento

de Engenharia Mecânica, Universidade Federal de Santa Catarina, 2011.

[19] MALISKA, C. R., DA SILVA, A. F. C., HURTADO, F. S. V., RIBEIRO, G. G., JR, A. A. V. B. P., CERBATO, G., GREIN, E. A. **Comparativo geral dos métodos de discretização e estudos sobre tratamentos de poços e computação paralela, Relatório Técnico SINMEC/SIGER II**. Technical report 04, SINMEC/EMC/UFSC, Florianópolis, SC, 2013.

[20] PACS TRAINING GROUP. **Introduction to MPI**. National Center for Supercomputing Applications (NCSA) located at the University of Illinois at Urbana-Champaign, 2001.

[21] PADUA, D., editor. **Encyclopedia of Parallel Computing**. Springer, 2011.

[22] SHANKLAND, S. **Keeping Moore's Law ticking**. *http://www.cnet.com/*, October, 2012.

[23] TROBEC, R., VAJTERSIC, M., ZINTERHOF, P. **Parallel Computing : Numerics, Applications, and Trends**. Springer, 2009.

[24] ZIENKIEWICZ, O. C., TAYLOR, R. L. **The Finite Element Method**, fifth, vol. 1. Butterworth-Heinemann, 2000.

APPENDIX

A

UTCHEM's Input File

The IPARS framework enables the implementation of a new and much more flexible format for the input files. To each variable that needs data from the input file it is associated a keyword. The keyword may be placed in whichever position in the input file. This is different from the original UTCHEM's input file format, that supported only placing each variable in its own – and single – place. If the datum was misplaced, the simulation could not be executed or, in the worst case, the datum could be assigned to a wrong variable leading to wrong results.

The variables read from the input file are separated in scalars and arrays. Datum from a scalar variable can be placed in the input file as the example bellow

$$VAR = 5$$

This means that the variable whose keyword is "VAR" will be set as 5. The equal sign is not mandatory and thus may be omitted. For array variables, the difference is that it is necessary to specify to each position in the array data should be assigned. In the following example, the keyword

represents the whole array

$$ARR() = 1 \quad 2 \quad 3 \quad 4$$

In this case, the variable whose keyword is "ARR" is initialized with $(1,2,3,4)$. One, however, may initialize each component at a time, as bellow

$$\begin{aligned} ARR(1) &= 1 \\ ARR(2) &= 2 \\ ARR(3) &= 3 \\ ARR(4) &= 4 \end{aligned}$$

Furthermore, it is also possible to initialize a subarray from the original array:

$$\begin{aligned} ARR(1 \text{ TO } 2) &= 1 \quad 2 \\ ARR(3 \text{ TO } 4) &= 3 \quad 4 \end{aligned}$$

Finally, it is worth to comment that if there is less data following a keyword than it is expected, the last components are filled with the last datum. This is very convenient. If, for example, ARR is an array with equal components (let's suppose their values are 6), then in the input file one may write

$$ARR() = 6$$

Matrices are treated as arrays of arrays and thus the same rules apply. However, despite the following initialization

$$\begin{aligned} MAT(,) &= 1 \quad 2 \quad 3 \\ &\quad\;\; 4 \quad 5 \quad 6 \\ &\quad\;\; 7 \quad 8 \quad 9 \end{aligned}$$

is instinctive, it should be avoided. It was discovered that the IPARS framework, from were comes the function that reads the data, puts the first row of data in the first column of the matrix, the second row in the second column, and the third row in the third column. Thus, unless the user knows

IPARS profoundly, it is recommended that the data should be placed as bellow

$$\begin{aligned} MAT(1,) &= 1 \quad 2 \quad 3 \\ MAT(2,) &= 4 \quad 5 \quad 6 \\ MAT(3,) &= 7 \quad 8 \quad 9 \end{aligned}$$

Two more observations should be done. First, to each variable it is associated a data type. In order to avoid reading errors, data following a keyword must respect the type. In the special case in which the data is a string of characters, it must be between quotation marks and must not exceed the maximum number of characters specified for it. The second observation is that comments are allowed. Whatever follow the symbol $ in the same line is ignored. This is very convenient if the user wants to add observations concerning the data from the input file.

APPENDIX

B

Example of code using EFVLib in parallel

Listing B.1 have an example of application of EFVLib with the parallel support implemented in this work. The purpose of this example is only explain the basic modifications required to run a case in parallel. More information about how to use the library may be found in [18]. The example is a two-dimensional heat transfer problem. There are two boundaries – TOP and OTHER –, both of them with prescribed temperature.

Listing B.1 – Example case using EFVLib with parallel processing

```
/* Grid

                TOP_1       TOP_0
             6 ————————7———————— 8
             |          / | \          |
             |   7     /  |  \   4     |
OTHERS_0     |      /     |     \      | OTHERS_5
             |   /    6   |   5   \    |
             | /           |         \  |
             3 ————————4———————— 5
```

```
11                |  \          |          /  |
12                |    \   1    |   2    /    |
13   OTHERS_1     |      \      |      /       |  OTHERS_4
14                |   0    \    |    /    3    |
15                |          \  |  /           |
16                0 ----------1---------- 2
17                  OTHERS_2   OTHERS_3
18  */
19
20  int main( int argc, char* argv[] ){
21      PetscInitialize(NULL, NULL, (char*)0, NULL );{
22      MPICommunicator world;
23
24      int numberOfVertices = 9;
25      GridDataPtr globalGridData;
26      if ( world.rank() == 0 ){
27          globalGridData = ReadGrid::Grid2D( "grid" );
28      }
29
30      // Grid division
31      GridDivider gridDivider;
32      GridDividerOutputPtr gridDividerOutput;
33      if ( world.rank() == 0 ){
34          gridDividerOutput = gridDivider.divide( globalGridData );
35      }
36      else{
37          gridDividerOutput = gridDivider.divide();
38      }
39
40      // GridBuilder
41      GridBuilder builder;
42      GridPtr localGrid = builder.build( gridDividerOutput->localGridData );
43
44      // Diffusive Operator
45      ScalarOrderedFieldOnVerticesPtr temperature( new
46          ScalarOrderedFieldOnVertices( localGrid->getNumberOfVertices(),
47              "T", "°C", "temperature" ));
48      Tensor2DSphericalPtr conductivity( new Tensor2DSpherical( 1.0 ) );
49      TensorConstantPropertyPickerOnElementsPtr condutivityPicker( new
50          TensorConstantPropertyPickerOnElements( conductivity ) );
51      DiffusiveOperatorComputer operatorComputer( localGrid, condutivityPicker );
52      VectorOrderedFieldOnFacesPtr diffusiveOperator = operatorComputer.compute();
53
54      // Linear System
55      PetscMaskPtr mask( new
56          PetscMask( localGrid->getNumberOfLocalVertices(), temperature->getValues() ) );
57      IntVectorPtr localToPetsc = localGrid->getLocalToPetscMapping();
58      mask->setLocalToGlobalMapping( localToPetsc );
59      mask->setTolerance( 1e-9 );
60      mask->setType( "gmres" );
61      mask->setPreconditioner("sor");
62
63      PetscMatrixPtr matrix = StaticPointerCast< PetscMatrix >( mask->getMatrix() );
64      DoubleVectorPtr independent = mask->getIndependentVector();
65
```

```
66      // Matrix
67      foreach( ElementPtr element, localGrid->getInternalElements() ){
68          InternalFacePtrArrayPtr internalFaces = element->getFaces();
69          foreach( InternalFacePtr face, (*internalFaces) ){
70              int vertexLocalHandle = 0;
71              foreach( VertexPtr vertex, element->getVertices() ) {
72                  double coef = -(*diffusiveOperator->getValue( face ))[ vertexLocalHandle ];
73                  VertexPtr backVertex =
74                      StaticPointerCast< efvlib::Vertex >( face->getBackwardNode() );
75                  if ( localGrid->isVertexLocal( backVertex ) ) {
76                      matrix->addValue( backVertex->getPetscHandle(),
77                          vertex->getPetscHandle(), coef );
78                  }
79                  VertexPtr forwardVertex =
80                      StaticPointerCast< efvlib::Vertex >( face->getForwardNode() );
81                  if ( localGrid->isVertexLocal( forwardVertex ) ) {
82                      matrix->addValue( forwardVertex->getPetscHandle(),
83                          vertex->getPetscHandle(), -coef );
84                  }
85                  vertexLocalHandle++;
86              }
87          }
88      }
89
90      // Boundary condition
91      ParserPtr parser(new Parser("../AppParallelTutorial/BoundaryConditions.txt"));
92      BoundaryConditionBuilder bcBuilder;
93      BoundaryConditionSetPtr boundaryConditions =
94          bcBuilder.build( localGrid->getBoundaries(), parser->getRoot());
95
96      matrix->initialize();
97      foreach( DirichletBoundaryConditionPtr bc,
98          boundaryConditions->dirichletBoundaries ){
99      BoundaryPtr boundary = bc->getBoundary();
100         foreach( VertexPtr vertex, boundary->getVertices() ){
101             if ( !localGrid->isVertexLocal( vertex ) ) {
102                 continue;
103             }
104
105             int petscIndex = vertex->getPetscHandle();
106             PetscInt petscArray[1];
107             petscArray[0] = petscIndex;
108             MatZeroRows( matrix->getPetscFormatMatrix(), 1, petscArray,
109                 1.0, NULL, NULL );
110
111             int localIndex = vertex->getHandle();
112             (*independent)[ localIndex ] = bc->getValue( vertex );
113         }
114     }
115
116     mask->solve();
117
118     ScalarFieldOnVerticesSynchronizerPtr sync(new
119         ScalarFieldOnVerticesSynchronizer(
120             gridDividerOutput->synchronizerVerticesVectors));
```

```
121         sync->synchronize( temperature );
122
123         FieldDivider< efvlib::Vertex, double > fieldDivider(
124             gridDividerOutput->verticesOfSubdomains );
125
126         if ( world.rank() == 0 ) {
127             ScalarOrderedFieldOnVerticesPtr globalTemperature( new
128                 ScalarOrderedFieldOnVertices(globalGridData->coordinates.size(),
129                     "T", "°C", "temperature" ) );
130             fieldDivider.gather( temperature, globalTemperature );
131
132             TecPlotSaveFile::vString varNames;
133             varNames.push_back("Temperature");
134             TecPlotSaveFile tecPlotSaveFile;
135             tecPlotSaveFile.initializeForTecplot(globalGridData, "../AppParallelTutorial",
136                 "Output.dat", varNames);
137             tecPlotSaveFile.iniAppendForTecplot( 0.0 );
138             tecPlotSaveFile.appendFieldForTecplot( *globalTemperature );
139             tecPlotSaveFile.endAppendForTecplot();
140             tecPlotSaveFile.close();
141         }else{
142             fieldDivider.gather( temperature );
143         }
144
145         PetscFinalize();
146     }
```

MPI methods can only be used after the MPI environment is initialized. The MPI can be initialized only once during a program execution. Such operation is performed at line 21 by the initialization of PETSc. The actual method in C/C++ to initialize a MPI environment is `MPI_Init`, but since PETSc also use MPI methods, this calling is hidden inside the method `PetscInitialize`.

All processors in a parallel execution run the same code. However, they usually do not perform the same operations. Each one of them have an index, usually called rank, that may be used to distinguish what are the operations that must be executed by the processor. It is usual to define the processor whose rank is 0 as the master processor. The master processor will execute the operation that must be serial. A class from the library BOOST named `communicator` – renamed here to `MPICommunicator` – is used to identify what is the rank of the current processor.

The operation of read a grid from a file is executed by a single processor: the master processor. The reading is performed at line 27 and right berfore that it is checked if the processor is in fact the master processor.

The grid division is executed by calling the method `divide` of an instance of the class `GridDivider`. Only the master processor pass the global grid data as a parameter of the method since only this processor have the data of the global grid. The other processors call the method `divide` without passing any argument. After calling such method all processors receive the data of their local grid and the data required to instantiate some communication classes detailed later.

Lines 57 and 58 are another additional piece of code required for running the program in parallel. A computational interface was created to use either EFVLib's original solver or PETSc. The class `PetscMask` is derived of such interface and should be used if PETSc is intended as the solver, remembering that from the EFVLib's solvers only PETSc has support for parallel running. When the program runs in parallel, `PetscMask` requires a mapping from the local indexes to the global indexes used in PETSc. Such mapping is available in the class `Grid` through the method `getLocalToPetscMapping`. The user only needs to get the mapping and set it in `PetscMask` using the method `setLocalToGlobalMapping`.

Pay attention at lines 75, 81, and 100. The purpose of all of them is to check if the vertex is local. One should remember that all computations are performed for the local vertices, since ghost nodes stand only to store values from other computational domains. This includes the assembling of a linear system: each processor assembles only the lines of its local vertices.

At line 118 the class that performs the updating of values at ghost nodes is instantiated. This class requires some data that comes from the grid division and may be accessed through the grid division output. The temperature field is updated using this class at line 121.

Lines 123 to 143 stand for exporting results. Before line 123 the temperature field still distributed across the processors. However writing a field into a file is a serial operation and thus all processors must sent their data to the master processor. Such operation is execute using the class `FieldDivider`. An instance of this class is created at line 123 and used at lines 130 and 142. Only the master processor pass two parameters to the method, one of them being the a global field in which the data received from the other processors will be stored. Lines after line 130 would be the same whether the code were parallel or not.

Finally PETSc and the MPI environment are finalized at line 145. After this line no MPI operation can be executed.

Despite the description of the example above being a little long, the point is that there is not many modifications to use EFVLib's parallel support. They can be summarized in the following: initialize and finalize PETSc at the beginning and at the end of the code, respectively; let only one processor read a grid; divide the grid; ensure that the processors calculate variables only at their local vertices; update the values of a field at the ghost nodes whenever such field is recalculated; and collect the fields when results of the simulation should be exported.