

Massively Parallel Oil Reservoir Simulation for History Matching

BY

Ayham Horiah Zaza

A Dissertation Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

COMPUTER SCIENCE AND ENGINEERING


December 2015

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Ayham Horiah Zaza** under the direction his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING**.



Dr. Adel Ahmed
Department Chairman




Dr. Salam A. Zummo
Dean of Graduate Studies

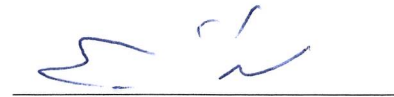
21/3/16
Date



Dr. Mayez Al-Mouhammed
(Advisor)



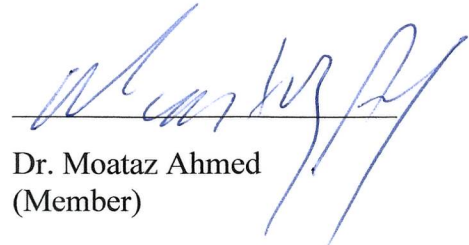
Dr. Faisal Fairag
(Co-Advisor)



Dr. Shokri Selim
(Member)



Dr. Gabor Korvin
(Member)



Dr. Moataz Ahmed
(Member)

**MASSIVELY PARALLEL OIL RESERVOIR SIMULATION FOR
HISTORY MATCHING**

Ayham Horiah Zaza

Computer Science and Engineering Department

December 2015

© Ayham Horiah Zaza

2015

This dissertation is dedicated to my beloved sister, who had been my emotional anchors, in memoriam. For my mother, for all her personal sacrifices and support. Dad, family and friends who shared with me moments of joy and sadness. For my future wife who would be proud of my achievements. For anyone who instilled in me the inspiration to set high goals and the confidence to achieve them.

ACKNOWLEDGMENTS

This work would have never come to existence without the continuous help of many great people around me. Whether it was in the form of direct mentoring, general advice or life experience, they were always there supporting, encouraging and inspiring.

I would like to start by thanking my thesis adviser Dr. Mayez Al-Mouhammed for all the feedback he provided, the time he spent clarifying things and his patience. Thanks is extended to both my co-adviser Dr. Faisal Fairag and Dr. Gabor Korvin for their continuous motivation, kind review, advice and support especially in mathematical related issues. I am also grateful to Dr. Shokri Selim for all the time he spent explaining things, all his guidance, encouragement and for his patience. Thanks also to Dr. Moataz Ahmed for his kind suggestions, inspiration and advice. I would like to thank all of them for all their exerted efforts and the time they spent following up with my many emails and attending several presentations. Apology is presented for any unintentional inconvenience in the course of this work.

I would like to dedicate a special thanks to Dr. Abee Awotunde for his extensive, kind help, patience and motivation when developing the physical model and for introducing me to the field of petroleum engineering and reservoir simulation. Thanks also to my colleague Anas Al-Mousa for his kind help and support especially in technical configuration related issues. I am also grateful to many kind friends and many previous great instructors who were supporting and encouraging.

Thanks also the Dean of Graduate Studies Dr. Salam Zummo, the Dean of Computer Science and Engineering, Dr. Adel Ahmed, and both the current and former Chairman of Computer Engineering Department, Dr. Ahmad Almulhem and Dr. Basem Almadani, for all their exerted efforts to develop the program. Finally, I would like to acknowledge all the great facilities available at KFUPM, you have been just a wonderful university!

TABLE OF CONTENTS

ACKNOWLEDGMENTS.....	V
TABLE OF CONTENTS.....	VI
LIST OF TABLES.....	IX
LIST OF FIGURES.....	X
LIST OF ABBREVIATIONS.....	XIII
ABSTRACT.....	XV
ملخص الرسالة.....	XVII
CHAPTER ONE: INTRODUCTION.....	19
1.1 Statement of the Problem.....	20
1.2 Parallel Computer Architecture: Past and Present	22
1.2.1 Multicore System	25
1.2.2 From Multicore to Many-core	28
1.2.3 Intel Xeon Phi	29
1.2.4 NVidia’s GPUs.....	30
1.3 The Programming Model of the Compute Unified Device Architecture (CUDA)	31
1.4 The Forward Reservoir Model.....	33
1.4.1 The Discretization Process	35

1.4.2	Assembling the System.....	36
1.4.3	The Linearization Step	38
1.5	Computation of Sensitivity Coefficients	39
1.5.1	Forward Sensitivity Method	41
1.5.2	Adjoint Sensitivity Approach	44
CHAPTER TWO: LITERATURE REVIEW.....		49
2.1	Review of Discretization Approaches	49
2.2	Linear Solvers Review	51
2.3	Review of Sparse Storage Techniques	56
2.4	Review of Linear Solver Libraries	71
CHAPTER THREE: COMPUTATIONAL MODELS, EXPERIMENTATIONS AND RESULTS		74
3.1	Computational Model for Reservoir Simulation	75
3.1.1	The Computational Model of the Forward Simulation Scheme.....	75
3.1.2	The Computational Model of the Inverse Simulation Scheme	78
3.2	Analytical Parallel Linear Solver Selection.....	84
3.2.1	The Generalized Minimum Residual Method (GMRES).....	87
3.2.2	The Bi-Conjugate Gradient Method (BiCG)	90
3.2.3	The Bi-Conjugate Gradient Stabilized Method (BiCGSTAB)	92
3.2.4	The Quasi-Minimal Residual Method (QMR)	94
3.2.5	Linear Solver Selection Based Tradeoffs	96
3.2.6	The Study of Concurrency Profile for BiCGSTAB and QMR	99
3.3	Experimental Parallel Linear Solver Selection	115
3.4	Special Case: Sparse Matrix Vector Multiplication for Hepta-Diagonal Matrices.....	132

3.5	Parallel Implementation of the selected Linear Solver for Matrices with Single (RHS)	136
3.5.1	Introduction	136
3.5.2	Merging Operations	138
3.5.3	Experiments and Comparisons	149
3.6	Parallel Implementation of the selected Linear Solver for Matrices with Single (RHS)	159
3.6.1	Introduction and Motivation	159
3.6.2	Implementation Strategy.....	161
3.6.3	Performance Evaluation	164
3.6.4	Concluding Remarks for this Section.....	170
 CHAPTER FOUR: PARALLEL MODELING AND IMPLEMENTATION OF FORWARD RESERVOIR SIMULATION		171
4.1	The Parallel Model	171
4.2	Experiments and Comparisons.....	180
4.3	The Parallel FRS Graphical User Interface (GUI)	184
4.4	Concluding Remarks and Future Work	189
 APPENDIX A WORK COMPLETED UNDER DIRECTED RESEARCH		191
A.1	Computational Model for Reservoir Simulation	191
A.2	Validating Reservoir Results.....	196
 APPENDIX B CUDA KERNELS UTILIZED IN THIS WORK.....		203
B.1	BiCGSTAB Merged Implementation.....	203
B.2	BiCGSTAB for MRHS System	212
 REFERENCES.....		216
 VITAE.....		223

LIST OF TABLES

Table 1	: Flynn Taxonomy for classifying computer systems.....	26
Table 2	: MIMD machines according to their attached memory and communication schemes.....	27
Table 3	: The main tasks the constitute Krylov Linear Solvers with an anticipated associated parallel complexities	87
Table 4	: The Storage Requirement for the four solvers.....	96
Table 5	: Number of reductions in the four nominated algorithms.....	97
Table 6	: Summary of the number of main transactions within an iteration	97
Table 7	: The number of available concurrent operations in QMR and BiCGSTAB. N is matrix leading dimension.....	100
Table 8	: Estimated parallel cost based on the perspective required steps to complete its operations in parallel when assuming Infinite Resources.....	106
Table 9	: Two consecutive patterns possibilities for tasks representing Krylov Solvers	113
Table 10	: Three consecutive patterns possibilities for tasks representing Krylov Solvers	114
Table 11	: Condition number for various samples of the reservoir simulator	116
Table 12	: TESLA K20X GPU ACCELERATOR.....	117
Table 13	: Comparison of different compute capabilities for GPU Architecture	117
Table 14	: Performance FLOPS for the kernels constituting the BiCGSTAB merged implementation	156
Table 15	: Performance FLOPS for the kernels constituting the BiCGSTAB merged implementation	168
Table 16	: List of utilized optimizations in the developed parallel FRS code.....	179
Table 17	: The Execution time (ET) for serial and parallel FRS	181
Table 18	: The parallel execution time of FRS various grid dimensions	182
Table 19	: Well distribution for both the producer and the injector over grid space of (20 x 30 x 2).....	196

LIST OF FIGURES

Figure 1	: Memory Hierarchy in NVidia GPU.....	32
Figure 2	: Sample Sparse Matrix with arbitrary values.....	57
Figure 3	: From top to bottom: COO, CRS and CCS representation for matrix shown in Figure 2.....	59
Figure 4	: JDS representation for matrix shown in Figure 2.....	62
Figure 5	: TJDS representation for matrix shown in Figure 2	64
Figure 6	: Block coordinate storage representation.....	66
Figure 7	: Overhead of using BCOO for various block sizes	68
Figure 8	: The Computational Model for Oil Reservoir History Matching	76
Figure 9	: Sample snapshot of the assembled linear system for FSR, (J, I and H): is the maximum number of steps in the z, x and y directions, respectively	77
Figure 10	: Inverse Model: Forward Sensitivity Approach	81
Figure 11	: The computational model for the Adjoint Sensitivity Approach	83
Figure 12	: QMR Data Dependency Graph.....	101
Figure 13	: QMR Data Dependency Graph.....	102
Figure 14	: Concurrency Profile of QMR with N=64	104
Figure 15	: Concurrency Profile of BiCGSTAB with N=64.....	105
Figure 16	: QMR Span	107
Figure 17	: BiCGSTAB Span.....	108
Figure 18	: A comparison between the estimated parallel cost based on the perspective required steps for QMR and BiCGSTAB Algorithms, with matrix leading dimension N = 1024. The smaller the parallel cost, the better.	109
Figure 19	: The Abstract Parallel Complexity Graph (APCG) for BiCGSTAB and QMR	112
Figure 20	: Average Parallel Execution Times for Sample_0.....	120
Figure 21	: Average Parallel Execution Times for Sample_31.....	121
Figure 22	: Average Parallel Execution Times for Sample_62.....	122
Figure 23	: Average Parallel Execution Times for Sample_93.....	123
Figure 24	: Average Parallel Execution Times for Sample_124.....	124
Figure 25	: Solvers Parallel Execution time for various storage formats with relative residual semi-log plot. Sample_0. Convergence is independent of the utilized storage scheme.....	125

Figure 26 :	Solvers Parallel Execution time for various storage formats with relative residual semi-log plot. Sample_31. Convergence is independent of the utilized storage scheme.....	126
Figure 27 :	Solvers Parallel Execution time for various storage formats with relative residual semi-log plot. Sample_62. Convergence is independent of the utilized storage scheme.....	127
Figure 28 :	Solvers Parallel Execution time for various storage formats with relative residual semi-log plot. Sample_93. Convergence is independent of the utilized storage scheme.....	128
Figure 29 :	Solvers Parallel Execution time for various storage formats with relative residual semi-log plot. Sample_124. Convergence is independent of the utilized storage scheme.....	129
Figure 30 :	BiCGSTAB Parallel Execution time for various storage formats with relative residual semi-log plot. All Extracted Sample. Convergence is independent of the utilized storage scheme.....	131
Figure 31 :	The average execution time of SpMV for various storage schemes and different related matrix dimensions. Here the input size has been studied within each scheme separately.	134
Figure 32 :	The average execution time of SpMV for various storage schemes and different related matrix dimensions. The focus here is see how each storage scheme behaves for a given matrix dimension.....	135
Figure 33 :	BiCGSTAB Data Dependency Graph (DDG), main operations are highlighted.....	137
Figure 34 :	The normal flow for various threads cooperating to compute sequence of operations in BiCGSTAB Algorithm	139
Figure 35 :	One possibility for merging arithmetic operations of the snippet of BiCGSTAB code, shown in Figure 34	142
Figure 36 :	Average Parallel Execution time for the two versions of the implemented solvers.....	154
Figure 37 :	Average Parallel Execution time for the two versions of the implemented solvers.....	155
Figure 38 :	GFLOPS/s for the kernels used to program the BiCGSTAB merged for various matrix dimensions.....	157
Figure 39 :	The Kernel Function for compute_alpha.....	163
Figure 40:	Data and some statistics for a version of BiCGSTAB that solves a system with MRHS. Whenever GPU memory cannot be allocated on the device, device allocation fail flag is raised	166
Figure 41 :	A double log plot for the average execution time of MRHS BiCGStab solver for various matrix dimensions and different MRHS widths.....	167

Figure 42 :	GFLOPS/s for the kernels used to program the BiCGSTAB merged for various matrix dimensions.....	169
Figure 43 :	The Activity Diagram for the reservoir simulator, with its computational scheme shown in Figure 52	174
Figure 44 :	The Activity Diagram for a sample North-South flow calculation inside the Newton Iteration	175
Figure 45 :	A double-log plot for the parallel execution time of our developed FRS for various geometries	183
Figure 46 :	GUI Snapshot showing the resulting pressure at Injectors.....	185
Figure 47 :	GUI Snapshot showing the resulting pressure at Producers.....	186
Figure 48 :	GUI Snapshot showing water cut values.....	187
Figure 49 :	GUI Snapshot showing how data is loaded into the system.....	188
Figure 50 :	General Scheme for Forward Reservoir Simulation.....	191
Figure 51 :	General Description for the Forward Reservoir Simulation Model	194
Figure 52 :	General Computational Scheme for the Forward Oil-Black model: When assembling the linear system. All grid points are visited. Newton Iteration repeatedly solves the system of linear equations formed in the grid iteration.....	195
Figure 53 :	Permeability map for the utilized wells shown in Table 19	197
Figure 54 :	Pwf at Injectors, Pc is included, No Flow BC for 20*30*2, specified flow rate at injector	199
Figure 55 :	Pwf at Producers, Pc is included, No Flow BC for 20*30*2, specified total rate at producer	200
Figure 56 :	Pwf at Injectors. Constant Flow BC (5000psi) at m-J and m-HJ, No Flow BC for the rest. Water-oil reservoir of dimensions (20*30*2) and specified flow rate at 6 injectors.....	201
Figure 57 :	Pwf at Producers. Constant Flow BC (5000psi) at m-J and m-HJ, No Flow BC for the rest. Water-oil reservoir of dimensions (20*30*2) and specified total rate at 7 producers	202

LIST OF ABBREVIATIONS

- FRS** : Forward Reservoir Simulator
- RHS** : Right Hand Side
- MRHS** : Multiple Right Hand Side
- GUI** : Graphical User Interface
- PDE** : Partial Differential Equation
- CPU** : Central Processing Unit
- GPU** : Graphical Processing Unit
- ILP** : Instruction Level Parallelism
- SMX** : Streaming Multi-processors
- CUDA** : Compute Unified Device Architecture
- FVM** : Finite Volume Method
- C.V** : Control Volume
- ILU** : Incomplete Lower Upper
- GMRES** : Generalized Minimum Residual Method

BiCG : Bi-Conjugate Gradient Method

BiCGSTAB : Bi-Conjugate Gradient Stabilized Method

QMR : Quasi-Minimal Residual Method

COO : Coordinate Storage Scheme

CSR : Compressed Row Storage Scheme

CCS : Compressed Column Storage Scheme

ELLPACK : A form of Jagged Diagonal Storage Scheme

HYB : Hybrid Storage Scheme

MV : Matrix Vector Multiplication

SpMV : Sparse Matrix Vector Multiplication

LAS : Linear Algebra Solver

ABSTRACT

Full Name : Ayham Horiah Zaza
Thesis Title : Massively Parallel Oil Reservoir Simulation for History Matching
Major Field : Computer Science and Engineering
Date of Degree : Dec. 2015

Petroleum Reservoir modeling is a challenging process that attempts inferring reservoir structure and configurations through the estimation of essential spatial properties like porosity and permeability. The general model consists of two consecutive and computationally expensive simulated paradigms; the forward and the inverse models. The goal of Forward Reservoir Simulation (FRS) is to model fluid flow and mass transfer in porous media to eventually draw conclusions about the behavior of certain flow variables and well responses. Any developed (FRS) is prone to significant errors as the initial data that defines the reservoir and the actual values of reservoir parameters are not necessarily the same. As a result, history matching or the inverse model repeatedly improves the simulated reservoir past performance after observing weaknesses in current data to suggest modifications in subsequent iterations. Both models eventually attempt solving a huge and computationally very expensive sparse linear system having either one or multiple right hand side (RHS) in the forward or the inverse model, respectively. By considering the state of art advances in massively parallel computing and the accompanying parallel architecture, this work aims primarily at developing a parallel simulator for oil reservoir on many-core processors by implementing a suitable parallel preconditioned linear solver

for both (single & multiple RHS) and exploiting several optimizations in both storage and implementation, to speed up the computation and minimize the overall simulator execution time. To offer more flexibility a graphical user interface (GUI) with simple visualization and controls will also be offered.

ملخص الرسالة

الاسم الكامل: أيهم نواف حورية ظاظا

عنوان الرسالة:

التخصص: علوم وهندسة الحاسب الآلي

تاريخ الدرجة العلمية: كانون الأول ٢٠١٥م

على الرغم من التحديات العلمية المصاحبة لها، تهدف عملية المحاكاة الحاسوبية لطريقة عمل الحقول النفطية أساساً إلى التنبؤ بماهية هذه الحقول من خلال استقراء لجملة من الخصائص البنوية الرئيسية كصفات الصخور المكونة ونفاذية الموائع من خلالها. يقوم الهيكل العام لهذه المحاكاة على نموذجين متتابعين يعتمدان بشكل كبير على جم هائل من العمليات الحسابية المعقدة. غرض أول هذين النموذجين معرفة كيفية سريان وتدفق الموائع من خلال الطبقات المكونة للحقل النفطي واستنتاج قيم أولية من خلال افتراضات ليست بالضرورة دقيقة، لبعض المتغيرات المصاحبة. يأتي دور النموذج الثاني لتحسين القيم الخارجة من النموذج الأول من خلال مقارنته النتائج المتنبئة مع قراءات سابقة، وتحليلها واستنباط شروط أفضل لتوليد نتائج أدق وافتراضات أحسن. يتطلب الأمر في كلتا الحالتين وبشكل متزامن، حل سلسلة نظم من المعادلات الخطية، مجموعة في مصفوفة مربعة هيكلية وهائلة مليئة بعناصر صفرية لطرف أيمن أوحد كما في النموذج الأول، أو متعدد كما في الثاني. يهدف هذا البحث أساساً، وبالاستفادة من أحدث التطورات السريعة المتعاقبة في مجال الحوسبة المتزامنة المتوازية، إلى تطوير برنامج متكامل يعمل على حواسيب كثيرة الأنوية، لمحاكاة عملية استخراج النفط من الخزانات الأرضية بغرض تسريع عملية الحصول على النتائج

المرجوة. لإعطاء صورة متكاملة، سيتم أيضاً تطوير واجهة للمستخدمين تمكنهم من التحكم ببعض متغيرات البرنامج واستعراض النتائج.

CHAPTER 1

INTRODUCTION

From food production, power generation to transportation systems and almost every other aspect of daily life, our modern society continues to ask for more and more energy with oil being the number one resource that addresses that heavily increasing demands. Despite the huge technological advances in oil industry, recovering the remaining available oil is limited by our knowledge and understanding of oil reservoirs [1]. The process of Reservoir Simulation requires large amount of memory storage as well as extensive computations to eventually provide vital information about the production rate, cost management, optimal well placement and many other reservoir parameters. As the computation for practical reservoir dimensions may last for days, speeding up the process by taking advantage of parallel computing is indispensable.

Like many other complex systems in nature, the behavior of oil reservoir can be modeled using a set of non-linear partial differential equations (PDEs) that describe how the entire system evolve in time, space or both. For many practical scenarios, obtaining a closed form analytical solution for the governing (PDEs) that completely describe the problem is extremely difficult or even impossible. For that reason various discretization schemes have been developed and utilized to approximate the solution of the governing (PDEs), yet

maintaining stability and leading sound results with accepted convergence level. Such approximations result in a large sparse system of algebraic equations that needs to be further solved.

The details of the problem are described in the next section. After that, the entire system model is presented followed by shedding some light on the computational model. Literature survey for discretization schemes and linear solvers is then introduced before finally stating the deliverables, methodology and the objectives out of this research.

1.1 Statement of the Problem

Petroleum Reservoir modeling is a challenging process for inferring reservoir structure and configurations through the estimation of essential spatial properties like porosity and permeability. As reservoirs extend over wide geographical areas, collecting enough samples efficiently and accurately to approximate flow conditions over a reasonable grid size is impracticable both economically and technically. This is mainly attributed to the fact that, wells are the only window through which various samples could be drawn. As a result and in order to approximate the estimation of reservoir parameters, indirect measurements or inverse modeling is a widely utilized alternative. When applied in petroleum engineering context, the inverse problem consists of two iterative and consecutive parts: the forward model and history matching – the inverse model.

At the beginning of the first process, the forward model assumes initial values for porosity and permeability and tries to predict resulting estimates of pressure and saturation by

discretizing the governing partial differential equations (PDE's) using a previously defined numerical scheme. Suitable desired boundary conditions and well constraints are imposed before finally and simultaneously solving the resulting set of nonlinear algebraic equations. After finishing all time iterates, the final resulting solution is fed to the inverse model which in turns searches the reservoir characteristics space to find the best variable estimate that matches the calculated pressure and saturation values.

The inverse process is very challenging as its obtained solution is very sensitive to the input data that is naturally subjected to measurement and modeling errors. At the heart of inverse model lies the formulation and computation of sensitivity matrix that measures how an induced change in reservoir behavior at one place could be carried out throughout the entire system. It is computationally very expensive, and various methods were suggested to compute it. Two such famous approaches are the forward sensitivity and the adjoint sensitivity methods. Moreover, when the size of this sensitivity matrix is even large, approximation techniques may be utilized to further reduce its dimension.

By considering the state of art advances in massively parallel computing and the accompanying parallel architecture, this work aims primarily at developing a parallel simulator for oil reservoir on many-core processors by implementing a suitable parallel preconditioned linear solver for both (single & multiple RHS) and exploiting several optimizations in both storage and implementation, to speed up the computation and to minimize the overall simulator execution time. To offer more flexibility a graphical user interface (GUI) with simple visualization and controls will also be offered.

1.2 Parallel Computer Architecture: Past and Present

Aiming for more and more performance has always been a driving force for any technological advances in computer systems ever since it was invented. Despite all the ambiguities associated with quantifying what the word performance solely indicates, the development trend was geared and motivated by a necessity of solving complex, practical and large scale real life problems,. As a result, machines with several architectural taxonomies have been built to serve different needs.

With a Central Processing Unit (CPU) interconnected with parallel wires to a memory chip that stores low level instructions and user data, the classical von Neumann model [2] laid the most successful foundational architecture that both dominated and advanced computer industry for quite some time. The (CPU) that features special fast storage elements called registers, comprise a control unit responsible not only for tracking program flow but also determining the next fetched instruction to be later executed by the arithmetic and logic unit (ALU).

As processor's throughput, the amount of work that can be completed per unit time, is much higher than the rate at which data arrives from main memory, various considerations over the years of computer system development were suggested to overcome that bottleneck. The improvements took many directions ranging from enhancing the performance of existing components and inventing novel technologies up to introducing new architectural taxonomies.

The presence of different memory hierarchies that originally revolved around exploiting the concept of data temporal and spatial localities, helped to some extent in bridging the previous latency gap. The idea was based on trading off space and power consumption with speed. This led to introducing and manufacturing special cache memories which are small in size but supports fast data access, organized at different levels between the CPU and main memory. According to a predefined scheme, cache memory maps a portion of data from main memory to its lines and serve them directly, upon a hit, to the processor when requested. If a processor requested data that is not available in cache, then data is fetched from main memory and some unused old data blocks are then replaced according to certain mechanism. Regardless of the mapping scheme or any resulting coherency overhead, the effectiveness of caches is prominent when the probability of not finding requested data in cache (miss rate) is small. At the first glance, it is obvious that, the miss rate is lowered when the cache size is made bigger. Nevertheless, and based on the intensive study of [3] that relates the cache sizes and the program working set, [4] has indicated that the benefit of further increasing cache size would be minimal and will not contribute to the overall performance as used to be in the past. [4] indicated that currently available cache sizes are big enough to hold the data needed to be accessed through out the lifecycle a given program in order to complete its needed calculations.

Dating back to 1965, Gordon Moore, co-founder of Intel Corporation, formulated an observation that was later known as Moor's Law and predicted the number of transistors per inch on integrated circuits to be doubled every 18 months [5]. The observation held true for quite good time until it finally hit classical physics walls. The more transistors shrink in size, the faster the electronic response becomes and hence the faster the integrated

circuit is [4]. However, as the frequency of operation increases, the associated power consumption increases in a quadratic relation¹. Current technology still cannot cope with that excessive amount of dissipated resulting heat that if pushed further, may either melt the chip or result in an unreliable behavior [6].

Moor's prediction of the huge increase in transistors' count, had paved the road for a new speed optimization era where more space is invested to deliver better performance. Instruction Level Parallelism (ILP) techniques [7] such as Superscalar Instruction Issue and Instruction Pipelining, are two currently widely utilized strategies that utilized the previous tradeoff and often been exploited to their possible extreme. Pipelining is centered on breaking down instructions into smaller pieces to be later processed at multiple staggered independent stages. The simultaneous work flow among different stages will eventually achieve a throughput of executing one instruction per clock cycle. Moreover and in addition to utilize complex circuitry as in pipelining, superscalar machines make use of duplicated additional hardware functional units to dynamically fetch, issue and process multiple instructions at the same time. While simultaneous issue of six instructions in superscalar machines, is about the useful limit for most programs on real processors,

¹ The capacitance is the ability of the circuit to store energy $C = \frac{q}{V}$ Or $q = C.V$.

Work is moving the charge against the voltage: $W = V * q ==> W = C.V^2$.

Power is work per unit time: $P = \frac{W}{t} = W.f ==> P = C.V^2.f$

increasing the size of the pipeline beyond a certain depth has not been proven contribute to better performance of the processor because of the inherent practical limits² [4, 6].

Just as the previous two techniques, Speculative Execution and Branch Prediction are also other forms of ILP. They again take advantage of the exponential increase in the number of transistors and advanced manufacturing technologies to introduce other components for boosting up performance [7]. In order to enhance speculation, a buffer is utilized to keep a history record of already taken branches inside a program, so that they are utilized later by processors for any upcoming branches. Although keeping such records consumes space and power [6], and despite the fact that programs' behavior is not completely predictable, such statistical inference had lead a boost in performance but only up to a certain point [4].

1.2.1 Multicore System

In parallel and not far from the previous chronological development, many attempts were dedicated to making use of multiple cooperating processors to either reduce the overall execution time of very intensive computational simulations or to solve a given problem at larger scales. By taking the combination of instructions' flow and data streams, Flynn [2, 5, 7] proposed a coarse famous taxonomy to categorize computer systems. Table 1

Although SIMD machines may yield a very high throughput especially when processing vector instructions, such machines suffer from a main drawback stemmed from their

² Pipelining is accomplished by reducing the amount of logic per stage to reduce the time between clocked circuits, and there is a practical limit to the number of stages into which instruction processing can be decomposed

original design; all computations must proceed in lock step and therefore free processing elements that had completed their job cannot start other tasks [8].

Table 1: Flynn Taxonomy for classifying computer systems

		Data Streams	
		Single	Multiple
Instructions	Single	<p>The uniprocessor</p> <p><i>Ex. von Neumann Architecture</i></p>	<p>SIMD</p> <p>The same instruction is executed by multiple processors while operating on different data streams.</p> <p><i>Ex. Vector Architecture</i></p>
	Multiple	<p>MISD</p> <p>A single data stream that utilizes successive functional units</p> <p><i>Ex. No Commercial model available yet</i></p>	<p>MIMD</p> <p>Each processor fetches its own instructions and uses its own data.</p> <p><i>Ex. General-Purpose Multiprocessors</i></p>

MIMD can be further classified into two categories based on their attached memory organizations: shared memory systems and distributed memory systems. Distributed memory system is also categorized according to the access pattern to be either distributed shared memory or clusters, Table 2.

Table 2: MIMD machines according to their attached memory and communication schemes

	Memory System	
	Shared	Distributed
Organization	Processors share a single centralized memory	Memory is physically distributed and private to each processor.
communication	Buses or switches	Switches, Multidimensional meshes, communication networks, internet
Characteristics	<ul style="list-style-type: none"> • The main memory has a uniform (symmetric) access time from any processor. • Implicit communication via load and store from a shared variable. • Explicit Synchronization <p>Also known as:</p> <ul style="list-style-type: none"> • Symmetric multiprocessors (SMPs) • Uniform Memory Access (UMA) 	<p>Two communication schemes:</p> <ul style="list-style-type: none"> • <i>Distributed Shared Memory (DSM)</i>: <ul style="list-style-type: none"> ○ Communication via a logical shared address space. ○ Also called non-uniform memory access (NUMAs), as the access time for varies according to the location of a data in memory ○ Implicit communication ○ To mitigate the discrepancy in memory access time, processors are shipped with caches and a coherency protocol. • <i>Multicomputers</i> <ul style="list-style-type: none"> ○ Separate computers connected on a local area network ○ Popularly called clusters ○ Explicit Communication via message passing ○ Implicit Synchronization
Famous Programming Environment³ [9]	OpenMP [10]: Implemented as set of extensions to (C/Fortran)	MPI [11] Implemented as a library called from programs written in a sequential programming language

³ Java is also famous for both memory systems and enjoys lot of software engineering benefits. However, it is slow compared to the other two environments and suffer from several deficiencies in the domain.

The programming effort needed to write parallel applications targeted to run on shared address space is minimal compared to other schemes as no data structure is needed to be distributed among processors. It is worth mentioning that such systems do not scale. This is due to the fact that increasing the number of processors, will increase the contention for memory bandwidth which is already a limiting factor [9].

1.2.2 From Multicore to Many-core

Many-core machines have emerged naturally as an answer to the continuous demand and need for more performance. They have been developed by considering the tricks and limitations that has been learnt over the years of continuous improvement on the design of both single and multicore systems. In addition to exploiting all possible optimizations to their limits, many-core machines came to existence after realizing that ILP could only deliver constant factors of speedup [6]. Moreover, it has been firmly realized that clock speed could not be increased anymore without melting the chip. As a result, the design consideration for many-core systems was centered on optimizing the architecture for power rather than performance [9]. On NVidia's GPUs for instance and being generated from simple cores operating at MHz clock, teraflop performance, or even exaflop in the near future, is achieved via hundreds of thousands cooperating threads⁴ performing the same task simultaneously.

⁴ Multiple threads exploit parallelism through latency hiding

Unlike the previous trend in manufacturing high performance computing machines, designing dedicated throughput oriented devices rather than utilizing general purpose latency oriented ones had enabled smarter utilization of Moor's observation. Doubling the number of transistors every eighteen month on a chip is now used to create either many-core processors, or single chips having multiple processor cores [4, 6, 12].

The details for the most widely used many core systems is presented next.

1.2.3 Intel Xeon Phi

Taking advantage of the new implemented 218 instructions not to mention the dedicated vector processing unit (VPU) and if a given code is highly parallel, efficiently vectorizable, scalable and able to hide the I/O communication [13], then it can effectively enjoy the teraflop performance offered by the power efficient Xeon Phi coprocessor [14]. The accelerator that coexists with the main processor and operates at about its third speed supports various execution models including heterogeneous programming mode⁵, coprocessor native execution mode⁶ and Symmetric execution⁷ mode [13]. Through either data marshaling or virtual shared memory model, the host processor and Intel Xeon Phi communicate for exchanging data [13].

⁵ Also known the offload mode, supported by OpenMP 4.0

⁶ As the Intel Xeon phi has its own micro OS, it can be viewed as another node connected to the main system. Cross compilation is required.

⁷ The application runs on both the main processor and the accelerator. Communication is done through message passing interface.

1.2.4 NVidia's GPUs

The product line at NVidia is continuously introducing new generations of high performance power efficient hardware. Besides the offered extreme computing capabilities, the new Kepler architecture [15] has introduced more features that enables increased GPU utilization and simplify parallel program design. For example, by allowing kernels to have full control on spawning other kernels, dynamic parallelism gives more flexibility for parallelizing nested loop iterations and performing recursion. Moreover, and to better utilize the system's multicores, Hyper-Q allows multiple simultaneous connection lines from those cores to launch work on the GPU, thus supporting computation and communication overlapping optimization.

With a support to 2688 CUDA Cores, 6 GB memory with 250 GB/s bandwidth, the Tesla K20 GPU is capable of delivering 1.32 TF and 3.95 TF double and single precision peak performance, respectively. The accelerator that is made of more than 7.1 Billion transistors is shipped with 15 streaming multiprocessors (SMX) and 1.5 MB L2 cache. Each SMX supports a maximum of 2048 threads, 16 thread blocks, 64K 32-bit registers, up to 48K shared memory. Each thread block can have a maximum of 1024 threads, while every thread can have a maximum of 255 registers. The computing Grid can support a maximum of $2^{32} - 1$ threads. Four warps each containing 32 threads can be issued and executed concurrently⁸. Threads within a warp can share data through the new implemented Shuffle instruction and therefore reduce the amount of shared memory needed per thread block⁹.

⁸ This is because of the available quad warp scheduler and the eight instruction dispatch units.

⁹ This has a direct relation with the amount of threads and thread blocks that can be allocated.

As this work is implemented on this architecture by utilizing its accompanying parallel computing platform -CUDA, the next section is dedicated to describing this programming model and its associated optimizations in more details.

1.3 The Programming Model of the Compute Unified Device Architecture (CUDA)

The NVidia GPU memory, Figure 1, is organized at different levels each of which varies in speed, usage, size, and scope¹⁰ [15]. Tesla K20x GPU features a 6 GB global memory with 250 GB/s bandwidth. Data stored in global memory are allocated and destroyed from the host and are visible by all threads in the application. With a similar scope and certain considerations¹¹, the read only 512 KB Constant Memory provides a relatively faster access speed than the global memory by reducing bandwidth usage through caching constant values and broadcasting them to all threads in a warp. At the block level and being visible to all threads in the block, the configurable 64 KB shared memory and in the absence of bank conflicts, provide even much faster access speed and allow data sharing and reuse among threads within the block. Finally, and with a lifetime of the thread that created it, registers are considered the fastest memory elements requiring zero clock cycle per instruction in the absence data dependency. Kepler based devices support a maximum of 255 32-bit register per thread.

¹⁰ See also: <http://docs.nvidia.com/cuda/kepler-tuning-guide/#axzz3V6tnqhWI>

¹¹ Warps of threads read the same location

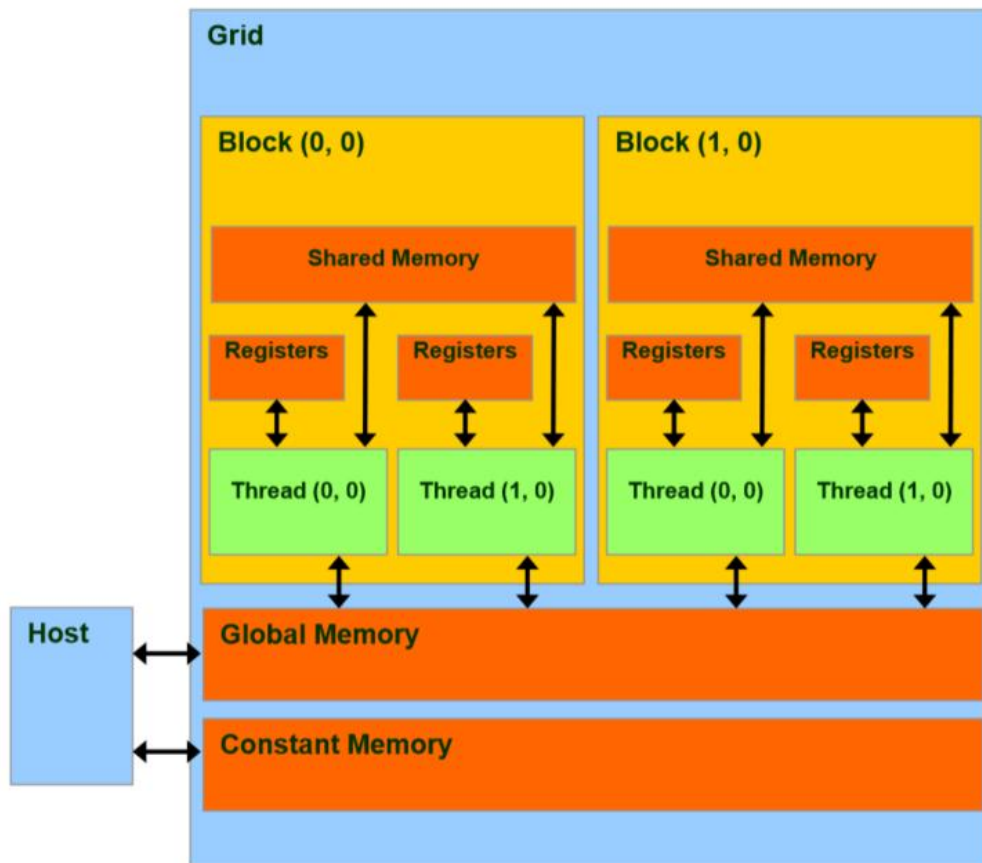


Figure 1: Memory Hierarchy in NVidia GPU

1.4 The Forward Reservoir Model

Forward Reservoir Simulation (FRS) is a predictive mathematical process that models fluid flow and mass transfer in porous media. Regardless of the discretization approach or grid mesh type, FRS will eventually draw conclusions at the behavior of certain flow variables and well responses to either utilize it in the development of new fields to estimate the production rate for instance, or to instantiate another process, namely, the inverse model and history matching.

Our implemented model fully describes the 3D flow process of the two immiscible phases (water, oil) and accounts for various physical properties in the flowing medium like permeability, porosity, oil pressure, water saturation as well as the interacting forces such as gravity and capillary. Permeability is the capacity of the rock to transmit fluid through its connected pores when the same fluid fills all the interconnected pores [16]. A porous medium is a solid containing void spaces (pores), connected or unconnected, dispersed within it in either a regular or random manner. And porosity is the ratio of the volume of the pores to the total bulk volume of the media [17]. Our simulated reservoir will be described as having isotropic permeability distribution and a heterogeneous geometry¹². At the analysis stage, the mass balance equation for every phase is constructed and the associated velocities are expressed by means of Darcy's law that linearly relates

¹² Those are properties of the porous media:

Isotropic: permeability is constant in all directions, i.e. it does not exhibit directional bias.

Heterogeneous: porosity is changing with location.

the flow rate to pressure drop through geometry, viscosity and permeability.

Mathematically, the mass balance equation can be derived as:

$$-\nabla(\rho_f \vec{u}_f) + \rho_f \frac{q_f^{well}}{V} = \frac{\partial}{\partial t}(\phi_f \rho_f S_f), \quad (1.1)$$

where, the subscript $f \in [\text{oil (o), water (w)}]$, \vec{u}_f is velocity vector, ρ_f the density, q_f^{well} the flow rate, V_b is the bulk volume, ϕ the porosity of the medium, and S_f is phase saturation.

Darcy Law is given by:

$$\vec{u}_f = -\frac{1}{\mu_f} k_{rf} \underline{k} (\nabla p_f + \gamma_f Z), \quad (1.2)$$

with \underline{k} representing absolute permeability tensor of the medium, k_{rf} is the relative permeability of phase f , μ_f is the viscosity of phase f , \vec{u}_f is the velocity of phase f , p the applied pressure drop, Z is the depth of the reservoir and γ is the specific gravity of the fluid.

Expanding equation (1.1) using suitable flow units, and after substituting the velocity from equation (1.2) we obtain the following equations for each phase,

$$\begin{aligned} \frac{\partial}{\partial x} \left(\frac{\beta_c k_{ro} K_x A_x}{\mu_o B_o} \frac{\partial p_o}{\partial x} \right) \Delta x + \frac{\partial}{\partial y} \left(\frac{\beta_c k_{ro} K_y A_y}{\mu_o B_o} \frac{\partial p_o}{\partial y} \right) \Delta y \\ + \frac{\partial}{\partial z} \left(\frac{\beta_c k_{ro} K_z A_z}{\mu_o B_o} \frac{\partial \phi_o}{\partial z} \right) \Delta z + q_{osc} = \frac{V_b}{\alpha_c} \frac{\partial}{\partial t} \left(\frac{\phi S_o}{B_o} \right). \end{aligned} \quad (1.3)$$

$$\begin{aligned}
& \frac{\partial}{\partial x} \left(\frac{\beta_c k_{rw} K_x A_x}{\mu_w B_w} \frac{\partial p_w}{\partial x} \right) \Delta x + \frac{\partial}{\partial y} \left(\frac{\beta_c k_{rw} K_y A_y}{\mu_w B_w} \frac{\partial p_w}{\partial y} \right) \Delta y \\
& + \frac{\partial}{\partial z} \left(\frac{\beta_c k_{rw} K_z A_z}{\mu_w B_w} \frac{\partial \phi_w}{\partial z} \right) \Delta z + q_{wsc} = \frac{V_b}{\alpha_c} \frac{\partial}{\partial t} \left(\frac{\phi S_w}{B_w} \right).
\end{aligned} \tag{1.4}$$

Two more equations are then needed to close the system. In the two-phase system considered in this work, we require that:

$$p_c = p_o - p_w, \tag{1.5}$$

$$S_o + S_w = 1, \tag{1.6}$$

where, B is the formation volume factor, α_c and β_c are constants, k_{ro} and k_{rw} are the relative permeability for oil and water respectively. Finally p_c is the capillary pressure.

The simulator will handle different boundary conditions and well constrains. Natural grid indexing is utilized and the above equation is then discretized using the finite volume method [18] on a structured grid.

After discretizing equations (1.3) and (1.4), and after providing initial state variables (p_o & S_w) as well as reservoir properties, FRS solve for the corresponding state variables values at each iteration. The details of the process are described next.

1.4.1 The Discretization Process

The goal of this step is to approximate the solution of the governing non-linear partial differential equations provided by (1.3) and (1.4) after imposing certain boundary conditions of interest, by a system of non-linear algebraic equations that are iteratively solved. At the analysis stage, the domain of interest, the reservoir, is subdivided into a finite

number of grid cubes, control volumes, that spans the entire 3D space. By following the finite volume approach, the flow equations at the center of each grid cube are then integrated over that volume, shape functions between the center and the edges are then assumed and interpolation¹³ is performed in an attempt to summarize the total flow across and within the cube by a single point in the center. This will lead to a non-linear algebraic equation that approximates the original (PDE) and resembles the flow at the center of the control volume taking into account the contribution of other flows coming from all the six neighboring directions (North, South, East, West, Top and Bottom) as well as extra flow sources coming from the wells for instance. The previous process is repeated until all the originally subdivided volumes are visited.

1.4.2 Assembling the System

The mathematical derivation for the developed models follows exactly the formulation presented by Abeer [19, 20].

The residual equation of the discretized system is given by the following¹⁴

$$\vec{R}^{n+1}(\vec{u}^{n+1}, \vec{u}^n, \vec{v}, \Delta t; \vec{\alpha}) = \vec{0}, \quad (1.7)$$

where \vec{v} is the vector of known reservoir properties and \vec{u} is the vector of the state variables given by:

$$\vec{u} = [p_{0,1}, S_{w,1}, \dots, p_{0,M}, S_{w,M}, p_{wf,1}, \dots, p_{wf,Nwell}]^T \quad (1.8)$$

¹³ Depending on the required accuracy, the shape functions and the interpolation could be linear, quadratic or any other higher order.

¹⁴ Assumptions include: fully implicit approach, three-dimensional reservoir system with M grid blocks and $Nwells$ wells.

\vec{R}^{n+1} consists of the residual due to flow in and out of reservoir grid blocks, \vec{R}_{blk}^{n+1} and the residual due to flow into or out of the wells in the reservoir, \vec{R}_{well}^{n+1} . Thus \vec{R}^{n+1} may be represented by:

$$\vec{R}^{n+1} = \begin{bmatrix} \vec{R}_{blk}^{n+1} \\ \vec{R}_{well}^{n+1} \end{bmatrix}, \quad (1.9)$$

where:

$$\vec{R}_{blk}^{n+1} = [R_{w,1}^{n+1}, R_{o,1}^{n+1}, R_{w,2}^{n+1}, R_{o,2}^{n+1}, \dots, R_{w,M}^{n+1}, R_{o,M}^{n+1}]^T \quad (1.10)$$

and

$$\vec{R}_{well}^{n+1} = [R_{well,1}^{n+1}, R_{well,2}^{n+1}, \dots, R_{well,Nwell}^{n+1}]^T, \quad (1.11)$$

\vec{R}_{blk}^{n+1} consists of the residuals representing the two phases present in the reservoir:

$$\vec{R}_w^{n+1}(\vec{p}_o^{n+1}, S_w^{n+1}, \vec{p}_{wf}^{n+1}, \vec{p}_o^n, S_w^n, \vec{\varphi}_{ini}, \Delta t; \vec{k}) = \vec{0}, \quad (1.12)$$

and:

$$\vec{R}_o^{n+1}(\vec{p}_o^{n+1}, S_w^{n+1}, \vec{p}_{wf}^{n+1}, \vec{p}_o^n, S_w^n, \vec{\varphi}_{ini}, \Delta t; \vec{k}) = \vec{0}, \quad (1.13)$$

while \vec{R}_{well}^{n+1} is the well residual given by:

$$\vec{R}_{well}^{n+1}(\vec{p}_o^{n+1}, S_w^{n+1}, \vec{p}_{wf}^{n+1}, \vec{p}_o^n, S_w^n, \vec{\varphi}_{ini}, \Delta t; \vec{k}) = \vec{0} \quad (1.14)$$

In Equations (1.12) through (1.14), $\vec{\varphi}_{ini}$ is the initial porosity distribution and \vec{k} is the permeability distribution in the reservoir. For a fixed total production rate constraint, we have

$$\vec{R}_{well,i}^{n+1} = \sum_{ph=o,w} \sum_j^{N\ comp} q_{ph,j}^{well} - q_{t,i} = 0 \quad (1.15)$$

where $q_{ph,j}^{well}$ is the flow rate of phase ph (ph is either oil or water) at the j^{th} completion given by:

$$q_{ph,j}^{well} = \lambda_{ph,j}^{n+1} WI_j (p_{ph,j}^{n+1} - p_{wf}^{n+1} - \gamma_{ph,j}^{n+1} \Delta z_j) \quad (1.16)$$

while $\lambda_{ph,j}^{n+1}$ and $\gamma_{ph,j}^{n+1}$ are the mobility ratio and specific gravity respectively of phases ph at the j^{th} completion in well i . WI_j is the well index at completion j .

1.4.3 The Linearization Step

Before the system of non-linear equations that was presented in equation (1.7) is simultaneously solved, a linearization step is necessary. The Newton Iteration achieves that goal by repeatedly refining a nearby approximation obtained after solving a linear system with the Jacobian as the coefficient matrix. For every iteration we solve the linear system

$$J^{n+1,l+1} \delta \vec{u}^{n+1,l} = -\vec{R}^{n+1,l} \quad (1.17)$$

and updated the solution:

$$\vec{u}^{n+1,l+1} = \vec{u}^{n+1,l} + \delta \vec{u}^{n+1,l} \quad (1.18)$$

In Equation (1.17), the Jacobian matrix $J^{n+1,l}$ is given by:

$$J^{n+1,l} = \frac{\partial \vec{R}^{n+1,l}}{\partial \vec{u}^{n+1,l}} \quad (1.19)$$

At the l^{th} iteration.

As the practical dimensions of the modeled space are very high (M ~ billions), special care should be taken for choosing a suitable solver¹⁵.

1.5 Computation of Sensitivity Coefficients

Any developed forward reservoir simulation is prone to significant errors as the initial data that define the reservoir model and the actual values of reservoir parameters are not necessarily the same. This lack of information is due to the fact that wells are the only window to the reservoir where some properties can be drawn. Not only well dimensions are very narrow, but also they are distributed over wide areas. As a result drawing

¹⁵ Details will be provided later for various solvers comparisons.

conclusions about reservoir behavior in-between wells or interpolating reservoir parameters among wells is subjected to significant mismatch with the actual values. To counter this mismatch, repeated improvement of the simulated reservoir past performance are performed after observing weaknesses in data and suggesting modifications needed to improve the model [21].

History matching is the application of inverse theory to petroleum reservoir engineering, where direct or indirect observations at either well locations or well-head respectively are used to estimate variables that describe the physical properties of the system. Such information could be described by sensitivity coefficients which relate small changes in model variables such as permeability, to changes in the state variables such as pressure or saturation. The high computational cost required when processing sensitivity coefficients not only influences the optimization methodology, but also forces certain compromises and tradeoffs. [19, 22] Two famous approaches for computing sensitivity coefficients are the forward sensitivity [19, 22-24] and the adjoint-state [19, 22, 25, 26].

Both the forward sensitivity approach and the adjoint method require the simultaneous solution of a linear system with multiple right hand side independent vectors, assembled in a matrix that has a column dimension Σ . Although, the two methods produce the same results, Σ in both approaches differs widely and the choice for which one to apply is highly driven by the size of data and model spaces. When the number of data to match is significantly smaller than the number of parameters to estimate, the adjoint method is favored over the forward sensitivity approach. Σ in this case contains information about data for which sensitivities are to be calculated and independent of the number of

parameters. On the other hand, Σ in the forward sensitivity case stores redundant information about model variables but is preferred when the number of parameters is small. When both data space and model space are of high dimensions, the computation of sensitivity coefficients is very expensive and the use of parallel machines is a must or other approximations are utilized. One of which is presented in [19].

To start with, the following sections present the mathematical derivations for both approaches. Again, we follow the same formulation as presented by Abeebe in [19]

1.5.1 Forward Sensitivity Method

Recall the general representation of the residual equations in (1.1)

$$\vec{R}^{n+1}(\vec{u}^{n+1}, \vec{u}^n, \vec{v}, \Delta t; \vec{\alpha}) = \vec{0} \quad (1.20)$$

A perturbation, $\delta\vec{\alpha}$, of the model parameter, $\vec{\alpha}$, induces a perturbation, $\delta\vec{u}$, of the state variable, \vec{u} , and a perturbation of the residual \vec{R} as given by

$$\vec{R}(\vec{u}^{n+1} + \delta\vec{u}^{n+1}, \vec{u}^n + \delta\vec{u}^n, \vec{v}, \Delta t; \vec{\alpha} + \delta\vec{\alpha}) = \vec{0} \quad (1.21)$$

An expansion of Equation (1.20) leads to

$$\vec{R}^{n+1} + \frac{\partial \vec{R}^{n+1}}{\partial \vec{u}^{n+1}} \delta\vec{u}^{n+1} + \frac{\partial \vec{R}^{n+1}}{\partial \vec{u}^n} \delta\vec{u}^n + \frac{\partial \vec{R}^{n+1}}{\partial \vec{\alpha}} \delta\vec{\alpha} + O(\delta^2) = \vec{0}. \quad (1.22)$$

Dropping higher order terms and recognizing that $\vec{R}^{n+1} = 0$ leads to the first order approximation

$$J^{n+1} \delta \vec{u}^{n+1} = -D^{n+1} \delta \vec{u}^n - Y^{n+1} \delta \vec{\alpha}, \quad (1.23)$$

in which

$$J^{n+1} = \frac{\partial \vec{R}^{n+1}}{\partial \vec{u}^{n+1}} \quad (1.24)$$

is the Jacobian matrix obtained from the simulator at the last step of the Newton-Raphson iteration,

$$D^{n+1} = \frac{\partial \vec{R}^{n+1}}{\partial \vec{u}^n} \quad (1.25)$$

is a block-diagonal matrix containing the derivative of the accumulation terms with respect to the state variables, at the previous time step n and

$$Y^{n+1} = \frac{\partial \vec{R}^{n+1}}{\partial \vec{\alpha}} \quad (1.26)$$

is a very sparse matrix programmed into the simulator and obtained at the last step of the Newton-Raphson iteration. Differentiating Equation (1.23) with respect to $\vec{\alpha}$ gives

$$J^{n+1} S^{n+1} = -D^{n+1} S^n - Y^{n+1}, \quad (1.27)$$

where

$$S^n = \frac{\delta \vec{u}^n}{\delta \vec{\alpha}} \quad (1.28)$$

is the sensitivity matrix required to solve the inverse problem.

In Equation (1.25) the entries of the block-diagonal matrix, D^{n+1} , are

$$D_{2m-1,2m-1}^{n+1} = \frac{\Delta V_m}{\Delta t} \left[\frac{\partial(\phi S_o b_o)}{\partial p_o} \right]_m^n, \quad (1.29)$$

$$D_{2m-1,2m}^{n+1} = \frac{\Delta V_m}{\Delta t} \left[\frac{\partial(\phi S_o b_o)}{\partial S_w} \right]_m^n, \quad (1.30)$$

$$D_{2m,2m-1}^{n+1} = \frac{\Delta V_m}{\Delta t} \left[\frac{\partial(\phi S_w b_w)}{\partial p_o} \right]_m^n, \quad (1.31)$$

and

$$D_{2m,2m}^{n+1} = \frac{\Delta V_m}{\Delta t} \left[\frac{\partial(\phi S_w b_w)}{\partial S_w} \right]_m^n, \quad (1.32)$$

for $m = 1, 2, \dots, M$. The Jacobian matrix, the matrix containing partial derivatives of the accumulation terms, does not change for all parameters. Thus we only need to compute them once at every time. In Equation (1.26) Y^{n+1} is the derivative of the residual with respect to model parameters and is given by:

$$Y^{n+1} = \frac{\delta \vec{R}^{n+1}}{\partial \vec{\alpha}} = \begin{bmatrix} \frac{\vec{R}_{blk}^{n+1}}{\partial \vec{\alpha}} \\ \frac{\vec{R}_{well}^{n+1}}{\partial \vec{\alpha}} \end{bmatrix}, \quad (1.33)$$

Except where otherwise noted,

$$\vec{\alpha} = \ln \vec{k}. \quad (1.34)$$

The derivative of the state variables with respect to $\ln \vec{k}$ is given by

$$\frac{\partial \vec{u}}{\partial \ln \vec{k}} = k \frac{\partial \vec{u}}{\partial k} \quad (1.35)$$

If we use the wavelets¹⁶ of $\vec{\alpha}$ as model parameters, Equation (1.27) becomes:

$$J^{n+1} S_c^{n+1} = -D^{n+1} S_c^n - Y^{n+1} W^T, \quad (1.36)$$

where

$$S_c^{n+1} = \frac{\partial \vec{u}^{n+1}}{\partial \vec{c}} = \frac{\partial \vec{u}^{n+1}}{\partial \vec{\alpha}} W^T, \quad (1.37)$$

and

$$\vec{c} = W \vec{\alpha}, \quad (1.38)$$

1.5.2 Adjoint Sensitivity Approach

¹⁶ The wavelet transform is a tool that cuts up data into different frequency components, and then studies each component with a resolution matched to its scale. [27] I. Daubechies, *Ten lectures on wavelets* vol. 61: SIAM, 1992.

Consider any scalar-valued function $\Psi(\vec{\alpha})$ which depends on $\vec{u}^n(\vec{\alpha})$ and is thus represented by

$$\Psi(\vec{\alpha}) = \eta \left(\Omega \vec{u}^n(\vec{\alpha}), \vec{\alpha} \right). \quad (1.39)$$

in which

$$\Omega \vec{u}^n = \{ \vec{u}^n : n = 1, 2, \dots, N \}. \quad (1.40)$$

Where:

$$\eta \left(\Omega \vec{u}^n(\vec{\alpha}), \vec{\alpha} \right)$$

represents the computed data at time index (n) where the measurement are made. Define Ψ_a by adjoining the constraints \vec{f} in (1.20) to η using adjoint variables¹⁷ $\vec{\lambda}$

$$\Psi_a(\vec{u}^{n+1}, \vec{\lambda}, \vec{\alpha}) = \eta + \sum_{n=0}^N [(\vec{\lambda}^{n+1})^T \vec{f}^{n+1}] \quad (1.41)$$

In Equation (1.41) $\vec{\lambda}^{n+1}$ is the vector of adjoint variables at time-step $n + 1$ and it is of the same dimension as $\delta \vec{u}^{n+1}$, the solution of Equation(1.18). At any feasible solution, $\delta \vec{u}_{sol}^{n+1}$,

$$\vec{f}^{n+1}(\vec{u}_{sol}^{n+1}, \vec{u}_{sol}^n, \vec{\alpha}) = 0 \quad (1.42)$$

¹⁷ For comprehensive description please see [19] A. A. Awotunde, "Relating time series in data to spatial variation in the reservoir using wavelets," Ph.D. Thesis, Department of Energy Resource Engineering, Stanford University, 2010.

and as such

$$\Psi_a(\bar{u}_{sol}^{n+1}, \vec{\lambda}, \bar{\alpha}) = \eta(\bar{u}_{sol}^{n+1}, \bar{\alpha}) = \Psi(\bar{\alpha}). \quad (1.43)$$

Taking the total differential of Equation (1.41) we have

$$\begin{aligned} \partial\Psi_a = \partial\eta + \sum_{n=0}^N \left[(\vec{\lambda}^{n+1})^T \frac{\partial \vec{f}^{n+1}}{\partial \bar{u}^{n+1}} \delta \bar{u}^{n+1} + (\vec{\lambda}^{n+1})^T \frac{\partial \vec{f}^{n+1}}{\partial \bar{u}^n} \delta \bar{u}^n \right. \\ \left. + (\vec{\lambda}^{n+1})^T \frac{\partial \vec{f}^{n+1}}{\partial \bar{\alpha}} \delta \bar{\alpha} \right] \end{aligned} \quad (1.44)$$

By considering the fact that the initial conditions are fixed

$$\delta \bar{u}^0 = \vec{0} \quad (1.45)$$

And after certain manipulations, it can be shown that Eq. (1.44) will lead to [19]:

$$\begin{aligned} \partial\Psi_a = \sum_{n=1}^N \left\{ \left[(\vec{\lambda}^n)^T \frac{\partial \vec{f}^n}{\partial \bar{u}^n} + (\vec{\lambda}^{n+1})^T \frac{\partial \vec{f}^{n+1}}{\partial \bar{u}^n} \right. \right. \\ \left. \left. + \frac{\partial \eta}{\partial \bar{u}^n} \right] \delta \bar{u}^n \right\} + \left\{ \frac{\partial \eta}{\partial \bar{\alpha}} \right. \\ \left. + \sum_{n=1}^N \left[(\vec{\lambda}^n)^T \frac{\partial \vec{f}^n}{\partial \bar{\alpha}} \right] \right\} \delta \bar{\alpha} \end{aligned} \quad (1.46)$$

We choose $\vec{\lambda}^n$ so that the first term in Equation (1.46) vanishes. That is,

$$(\vec{\lambda}^n)^T \frac{\partial \vec{f}^n}{\partial \vec{u}^n} + (\vec{\lambda}^{n+1})^T \frac{\partial \vec{f}^{n+1}}{\partial \vec{u}^n} + \frac{\partial \eta}{\partial \vec{u}^n} = \vec{0} \quad (1.47)$$

Equation (1.45) may be written as

$$(\mathbf{J}^n)^T \vec{\lambda}^n = - \left[(\mathbf{D}^{n+1})^T \vec{\lambda}^{n+1} + \left(\frac{\partial \eta}{\partial \vec{u}^n} \right)^T \right] \quad (1.48)$$

At the last time step $\vec{\lambda}^{N+1}$ is zero. Thus

$$(\mathbf{J}^N)^T \vec{\lambda}^N = - \left(\frac{\partial \eta}{\partial \vec{u}^N} \right)^T \quad (1.49)$$

Equations (1.48) and (1.49) are the adjoint equations through which all the adjoint variables $\vec{\lambda}^n$ are evaluated. Substituting Equation (1.48) into (1.47) and using the definition of \mathbf{Y}^n we obtain

$$\partial \Psi_a = \left\{ \frac{\partial \eta}{\partial \vec{\alpha}} + \sum_{n=1}^N [(\vec{\lambda}^n)^T \mathbf{Y}^n] \right\} \partial \vec{\alpha} \quad (1.50)$$

Differentiating Equation (1.50) with respect to α results in

$$\frac{\partial \Psi_a}{\partial \vec{\alpha}} = \frac{\partial \eta}{\partial \vec{\alpha}} + \sum_{n=1}^N [(\vec{\lambda}^n)^T \mathbf{Y}^n] \quad (1.51)$$

Equation (1.51) gives the sensitivity of the scalar-valued function η to model parameters $\vec{\alpha}$.

Equations (1.49) and (1.50) are solved backward in time for $n = N, N - 1, \dots, 1$. Consider

that we have measurements of well pressure, p_{wf} for all the N time steps. We may choose to compute the sensitivity of $p_{wf}(t_n)$ for any $(n \in 1, 2, \dots)$ or a linear or nonlinear combination of all the $p_{wf}(t_n)$. In fact, to compute gradient of the objective function, Φ we only need to replace η with Φ in Equations (1.48), (1.49) and (1.51). [19]

CHAPTER 2

LITERATURE REVIEW

In this section, we review existing literature in areas relevant to this study. It covers a review for the discretization methodologies, and linear solvers.

2.1 Review of Discretization Approaches

Due to its simplistic formulation, ease of programming and previously accepted consistency, stability, and convergence, Finite Difference Method (FDM) was very famous in old literature. After the domain of interest is partitioned into structured grids, FDM approximate the derivatives in the domain's governing Partial Differential Equations (PDE's) by manipulating the equation's Taylor Series Expansion. Depending on the aimed accuracy, several schemes are derived and utilized. For example, in one dimensional discretization, the truncation error decreases by $O(\Delta x^2)$ in the case of Central Difference and by $O(\Delta x)$ when Forward or Backward Differences are used. Whether block centered or point distributed discretization is considered, and after imposing suitable boundary conditions, such difference approximations yield a system of algebraic equations that eventually reduces to a banded sparse linear system. [28, 29]

Unlike FDM, Finite Element Method (FEM) has the ability to handle complex geometries and deal with variable material properties not to mention its rigorous mathematical foundations primarily reflected in error estimation. Moreover, when applied to reservoir simulation, it plays a role in reducing grid orientation effects [30]. Over the years and after its deployment as a numerical procedure for solving (PDE's), various flavors and enhancements were suggested. In their book "Computational Methods for Multiphase Flows in Porous Media" Zhangxin et al. [30], detailed the previous issues and presented in depth elaboration on various (FEM) as well as case studies. Such variations include: Control Volume Finite Element, Discontinuous Finite Elements, Mixed Finite Element, Characteristic Finite Element and Adaptive Finite Element Methods. The general (FEM) approach could be described as follows: The domain of interests is first subdivided into unstructured non-overlapping elements that are usually triangles or tetrahedral in 2D or 3D cases respectively. After that, the variation of the solution inside an element is expressed by a shape-interpolation- function that form a linear distribution having its values vanish outside the corresponding element. The differential form of the governing PDE's is transformed to their equivalent integral form by either utilizing the variation principle or through the method of weighted residuals of the weak formulation if preserving physical laws is desired. Finally, element equations and load vectors for each element are determined to form matrix equations, boundary conditions are imposed, and the final assembled system of simultaneous equations is solved. [31, 32]

The Finite Volume Method (FVM) has become widely accepted in simulating fluid behavior not only because it naturally produces conserved discretization for the associated physical laws, but most importantly because of its flexibility. The method utilizes mesh

dependent control volumes instead of grid intersection points to model unstructured grids without the need to perform coordinate transformation. As a direct result, the programming effort is much less compared to (FEM). The process begins by subdividing the domain of interests into a finite number of contiguous non-overlapping elements called control volumes (C.V). At the center of each (C.V) the associated governing (PDE's) are integrated with respect to the variables of interest. Interpolation is used to express variable values at the (C.V.) surfaces before the final assembly of the algebraic equations is formed and solved. [18, 33, 34]

2.2 Linear Solvers Review

As the discretization process of PDE's for practical problems will eventually lead to a set of algebraic equations with huge sparse coefficient matrix, Equation (2.1) , and given the associated storage issues and other limitations in direct linear solvers, researchers in the field of computational science and engineering continued to favor iterative methods in their applications.

$$Ax = b, \quad (2.1)$$

where: A is the coefficient matrix of the system

Although a clear boundary between the two classifications is very blur as indicated by [35], and since they are context specific, one can still classify linear solvers into direct and

iterative, to provide better rationalization when picking up the right solver for any application of interest.

If the coefficient matrix (A) is non-degenerate, non-singular, direct solvers in the absence of rounding errors, offer the exact solution in finite steps with robust and predictable behavior without putting any constraints on the type of A . On the other hand, as the problem size gets bigger, direct solvers start exhibiting memory problems given their demand for long recurrence. Moreover, and because of the fill in problem, data structure used to store the original sparse coefficients is continuously altered and never preserved as lot of previously zero entries become non zero as the factorization proceeds [36, 37].

Over the past 30 years, sparse direct solvers continued to develop and various strategies were introduced to guarantee more stable LU decomposition with minimal fill-in [38] or that preserves sparsity [39]. Despite all of the attempts, and because of the large storage demand and the processing requirements that is inherently sequential, some authors believe that the use of direct methods in practice is still limited to 2D mathematical modeling as reported by [40]. On the other hand, because of direct methods' superior robustness and because computers are getting faster, many other authors [35] believe many problems will be solved by methods from both approaches.

The most famous direct approach is Gauss elimination. In its general form, the method decomposes matrix (A) into both lower and upper triangular forms (LU). To solve the system in (2.1) forward elimination is performed first before back substitution takes place. With special consideration for the sparse case, Scott in [40] considered many numerical examples and reviewed frontal and multifrontal methods that are derived by combining

Gauss elimination and finite element approaches. Such methods are characterized by reducing storage and processing demands by interleaving matrix assembly with the elimination steps.

Motivated by Strassen's algorithm [41] that utilizes recursion to speed up matrix multiplication, not to mention recursion highlighted success in computational problems when applied to dense matrices, Dongarra and others in [42] attempted a recursive approach for the LU factorization of sparse matrices. Although, they reported an efficient storage and speedup compared to multifrontal methods for most sparse matrix profiles, recursion suffers from substantial drawbacks from software engineering perspective [43], which in turns limit its scalability and performance in parallel computing. First although recursion leads a very concise and readable code, it is sequential in nature as it is executed in memory stack that forces Last In First Out (LIFO) sequence of function calls. Second, recursion relies on long recurrence making it not suitable for practical problems with big A as it demands excessive memory storage.

On the other hand, and although they, might suffer from convergence issues and compromised accuracy, Iterative Methods are highly favored in the solution of large sparse systems. First, they preserve system sparsity as they do not modify the coefficient matrix. Second and most important, beside vector updates, the essential operation in almost all iterative solvers is matrix vector multiplication [36] that is characterized by its inherent parallelism. Moreover, and although iterative approaches are problem specific, it has been shown that the convergence could be enhanced by the use of suitable preconditioner.

Starting with an initial guess for the vector x in equation (2.1), iterative methods continue to refine that solution according to a certain criteria until convergence, if exists, to the desired accuracy. The overall idea lies behind replacing the system of equations by some nearby system which is easily solved [37, 44]. Such methods could be further classified into two main groups: stationary methods like Jacobi, Gauss Seidel, Successive over Relaxation, and non-stationary like Krylov subspace based methods [36, 45, 46].

The discretization of flow equations that governs two-phase oil water reservoir behavior that results from the forward modeling, using finite volume approach will yield a sparse system having ill-conditioned unsymmetrical coefficient matrix with Hepta-diagonal profile and 2×2 block representing each entry. Moreover, the inverse problem requires solving either the same matrix, forward sensitivity approach, or its transpose in the case of adjoint sensitivity approach, with multiple right hand side.

As a result and with the aim of writing the parallel code for the complete simulator, we review four applicable preconditioned Krylov methods of interest. In order to select one solver for our final implementation, we will be analyzing the part of a computation that can be parallelized as well as the usually addressed issues of storage and convergence. At this stage, we will only focus on general observations and leaving the detailed parallel analysis to a later stage. Given that perspective, the remaining lines in this section will review the following suggested solvers: The Generalized Minimum Residual Method (GMRES) by Saad and Schultz [47], The Bi-Conjugate Gradient Method (BiCG) by Fletcher [48], the quasi-minimal residual method (QMR) by Freund and Nachtigal [49], and finally, the Bi-Conjugate Gradient Stabilized (Bi-CGSTAB) by Van der Vorst [50].

By definition, the Krylov subspace generated by the coefficient matrix A and the accompanying residual $r_0 = b - Ax_0$ is denoted by: $K^k(A; r_0)$, with k indicating the iteration and given by: $K^k(A; r_0) \in \text{span}(r_0, Ar_0, A^2 r_0, \dots, A^{k-1} r_0)$

Krylov methods are classified according to the way x is chosen from the constructed subspace that contains the successive approximate solutions into: [37]

1. **The Ritz–Galerkin Approach:** constructs x_k for which the residual is orthogonal to the current subspace: $b - Ax_k \perp K^k(A; r_0)$. This leads to Conjugate Gradients, The Lanczos method, FOM, GENCG methods.
2. **The Minimum Norm Residual Approach:** identifies x_k for which the Euclidean norm $\|b - Ax_k\|_2$ is minimal over $K^k(A; r_0)$, then we have: GMRES, MINRES, ORTHODIR
3. **The Petrov–Galerkin Approach:** x_k is found so that the residual $b - Ax_k$ is orthogonal to some other suitable k -dimensional subspace. This leads to BiCG and QMR.
4. **The Minimum Norm Error Approach:** Determine x_k in $A^T K^k(A^T; r_0)$ for which the Euclidean norm $\|x_k - x\|_2$ is minimal. This leads to SYMMLQ and GMERR
5. **Hybrid Approaches**
 - a. CGS, Bi-CGSTAB
 - b. Bi-CGSTAB(L), TFQMR, FGMRES, and GMRESR

For extensive review of direct solvers and various implementation variations, one might consider [45, 51-53]. A comprehensive survey for preconditioning techniques is presented in [54]. For a complete survey on iterative solution methods, please check [35]. For a very

quick algorithmic treatment and comparison [45]. The books by [37, 55] presents a comprehensive treatment of the subject with a focus on the theory and finally, [56] describes various aspects of the parallel implementation of iterative solvers.

2.3 Review of Sparse Storage Techniques

Motivated by reducing storage requirements and avoiding unnecessary computations, sparse matrix representations have evolved to efficiently identify, operate on, and manipulate all non-zero matrix elements. As opposed to dense matrices, a sparse matrix is a matrix in which most of the elements are zero. Sparsity is the associated term that measures the fraction of non-zero elements to the total sparse matrix dimension. For example the sparsity of the general sparse matrix with arbitrary values shown in Figure 2 is calculated as:

$$Sparsity = \frac{\text{Number of Nonzeros}}{\text{Total Number of Elements}} = \frac{20}{81}$$

$$\Rightarrow Sparsity = 0.247$$

Perhaps, the most easy and obvious approach to store sparse matrices, is to store the spatial coordinates of their elements according to some traversing rules together with their corresponding values. For a 2-D matrix, such index representation could be abstracted by a state graph with nodes representing the first spatial coordinate, and directions representing the second [55]. The famous coordinate storage scheme (COO) [57] stores matrix information in three separate arrays (value, column-coordinate, row-coordinate) each with a length equals the total count of the non-zero elements Figure 3.

	1	2	3	4	5	6	7	8	9
1	1	0	0	0	0	0	0	0	0
2	0	25	0	0	0	0	0	0	0
3	0	16	8	0	0	0	0	0	0
4	0	0	8	1	0	0	0	0	3
5	0	11	0	4	13	0	0	0	15
6	0	6	0	0	0	26	0	0	5
7	0	21	0	0	0	0	1	0	0
8	0	16	0	0	0	0	0	26	0
9	0	13	0	0	0	0	0	0	1

Figure 2: Sample Sparse Matrix with arbitrary values

(COO) could be further optimized by trading off some computation with storage leading to two other representations: compressed row storage (CRS), and compressed column storage (CCS), Figure 3. In those schemes either the row-coordinate vector in (CRS) or the column-coordinate vector in (CCS) are replaced by another smaller vector that only stores values pointing to the first corresponding non-zero element in the value vector. The corresponding number of non-zero elements is then easily calculated by subtracting two consecutive indices in the replaced vector [45].

It can be observed from previous figures, that the maximum length of the vector holding the pointers in both (CRS) and (CCS) equals respectively the number of rows and columns in the original matrix. As a result, establishing a case where either of those representations outperforms the other in terms of minimizing storage space is an easy task indeed. For example, let $R \times C$ be the row and column dimensions of matrix M . If $(R < C)$ and the matrix is full rank, then CRS is more favorable. The reverse also holds true. This indeed motivates the necessity for either developing an intelligent algorithm that statically detects and selects the best storage scheme for a given input matrix, or a reconfigurable one that dynamically changes its internal data structure to fulfill the previous need.

For very large matrix dimensions and unlike (COO), one drawback of both (CRS) and (CCS) is that restoring and identifying the indices of the original matrix elements after performing some tiling is cumbersome. This in turn presents another scheme's selection compromise namely choosing between saving storage space or flexible tiling with easy indexing and reduced computation. The previous observation goes both ways regardless of tiling precedence, i.e. whether it occurred before compressing the storage or afterwards.

Value	1	25	16	8	8	1	3	11	4	13	15	6	26	5	21	1	16	26	13	1
Row-Coordinate	1	2	3	3	4	4	4	5	5	5	5	6	6	6	7	7	8	8	9	9
Column-Coordinate	1	2	2	3	3	4	9	2	4	5	9	2	6	9	2	7	2	8	2	9

Value	1	25	16	8	8	1	3	11	4	13	15	6	26	5	21	1	16	26	13	1
Column-Coordinate	1	2	2	3	3	4	9	2	4	5	9	2	6	9	2	7	2	8	2	9
Row Pointer	1	2	3	5	8	12	15	17	19											

Value	1	25	16	8	8	1	3	11	4	13	15	6	26	5	21	1	16	26	13	1
Row-Coordinate	1	2	3	3	4	4	4	5	5	5	5	6	6	6	7	7	8	8	9	9
Column-Pointer	1	2	9	11	13	14	15	16	17											

Figure 3: From top to bottom: COO, CRS and CCS representation for matrix shown in Figure 2

To analyze and compare storage requirements, the total number of non-zero elements will be ignored as this is going to be constant among all representations, not to mention that storing the complete array that holds those elements is not optional. Moreover, and without loss of generality and to compare various methods, the following analysis assumes the matrix to be a 2x2 Blocked-Hepta Diagonal. Let $H, I,$ and J be

Let N be the matrix dimension. Then it can be shown¹⁸ that the total number of non-zero elements is less than $(14 N)$. As a result, the total storage required by (COO) is less than

$$14 N \times 3 = 42N$$

On the other hand, the storage required by the compressed scheme is

$$14 N + 14 N + N = 29N$$

The following limit can be established for comparing the compressed storage to the naïve coordinate storage when the matrix is very large,

$$\lim_{N \rightarrow \infty} \frac{42 N}{29 N} = 1.45,$$

which means that (COO) will demands at most around 50% more storage space than either (CRS) or (CCS)! i.e. if (CRS) takes 4GB of memory to store large input matrix then (COO) will at most take 6 GB.

The Jagged Diagonal Storage (JDS) [55, 58] the generalization of ELLPACK-ITPACK [59] first traverses the original sparse matrix row wise, shifts left nonzero elements, and

¹⁸ Assuming 7 diagonals each contains 2 elements per row.

then stores the associated column index, Figure 4. The resulting shifted rows are then rearranged in a descending order according to the maximum nonzero elements count per row; an array of pointers is kept to indicate and later restore those permutations. Nonzero elements are then stored column wise and pointers to indicate the start of each column are recorded [60].

In one way or another, (JDS) is a mix between (CRS) and (CCS) with an extra intermediate permutation stage. Consider the sample matrix we chose for analysis. The initial startup overhead consists of two vectors each of length $14 N$ for storing the values and column indices, and one vector of length N that holds permutations and another varying in length but at most of size N for row pointers. Hence, the storage requirement for (JDS) is

$$S_{jds} = 2(14 N) + 2N = 30 N$$

Therefore comparing the storage space requirement to the compressed formats presented earlier (CRS) and (CCS) yields,

$$\lim_{N \rightarrow \infty} \frac{30N}{29N} = 1.03$$

Which means that as the matrix dimension gets extremely large, then the extra storage demanded by (JDS) could be neglected compared to either (CRS) or (CCS)!

Value	11	8	6	16	21	16	13	25	1	4	1	26	8	1	26	1	13	3	5	15
Column-Coordinate	2	3	2	2	2	2	2	2	1;	4	4	6	3	7	8	9;	5	9	9;	9
Pointer	1	10	19	28																
Permutation	5	4	6	3	7	8	9	2	1											

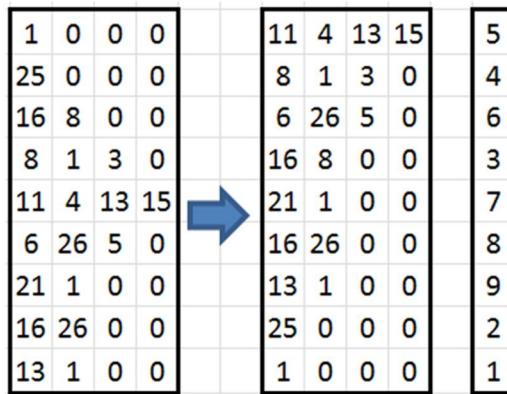


Figure 4: JDS representation for matrix shown in Figure 2

Neglecting start up computational overhead, (JDS) format will force the nonzero elements to be factorized into $(N \times \tau)$ matrix where (τ) is the maximum number of nonzero elements in a given row of the original matrix. Hence, matrices with (JDS) representation can be tiled or partitioned easily to suit different parallel platforms. However, unless dynamic load balancing is established, (JDS) will suffer from severe performance and scalability issues. Moreover, as column indices are stored separately, matrix vector product operation could be performed efficiently.

Just as the relation between (CRS) and (CCS), and as its name implies, the Transposed Jagged Diagonal Storage (TJDS) format, follows exactly the same logic of (JDS) but with main operations being transposed. Instead of being moved left, nonzero elements are shifted upwards. Columns are sorted in a decreasing order and nonzero elements' row indices are saved before storing the value array row by row, Figure 5. In certain applications like matrix vector multiplication and because columns are initially permuted, no extra vector is needed to store these permutations as they are already captured and recovered by reordering the unknown vector accordingly [61]. Thus, it would further save some space. By following the previous analysis on our sample matrix, and evaluating Matrix Vector Multiplication, the following is obtained

$$S_{Tjds} = 2(14 N) + N = 29 N,$$

which is the same space as required by (CSR), but in the same time offering more flexibility and enjoys the characteristics of (JDS).

Value	25	3	8	1	13	26	1	26	1	16	15	8	4	11	5	6	1	21	8	13
Row-Coordinate	2	4	3	4	5	6	7	8	1	3	5	4	5	5	6	6	9	7	2	9
Pointer	1	10	14	16	18	19	20													
Col Permutation	2	9	3	4	5	6	7	8	1											

										2	9	3	4	5	6	7	8	1		
1	25	8	1	13	26	1	26	3		25	3	8	1	13	26	1	26	1		
0	16	8	4	0	0	0	0	15		16	15	9	4	0	0	0	0	0		
0	11	0	0	0	0	0	0	5		11	5	0	0	0	0	0	0	0		
0	6	0	0	0	0	0	0	1		6	1	0	0	0	0	0	0	0		
0	21	0	0	0	0	0	0	0		21	0	0	0	0	0	0	0	0		
0	16	0	0	0	0	0	0	0		16	0	0	0	0	0	0	0	0		
0	13	0	0	0	0	0	0	0		13	0	0	0	0	0	0	0	0		

Figure 5: TJDS representation for matrix shown in Figure 2

Although specialized formats can further optimize storage, their domain of application is very tied and problem specific. Skyline Storage (SS) assumes a triangular matrix, and traverses matrix elements column wise in upper triangular or row wise in lower triangular until it hits the diagonal. It stores data in two arrays: one for the actual values and another is a pointer to the start of each row.

Compressed Diagonal Storage (CDS) traverses the sparse matrix in a diagonal fashion and stores a reference to indicate the diagonal of interest [62].

Despite the fact of their embarrassingly parallel nature [9], matrix vector multiplication (MV) operations are characterized to be bandwidth bound. That is because MV operations suffers from limited temporal locality [63] so they do not enjoy the so called surface to volume effect; i.e. they only perform $O(n^2)$ operations on $O(n^2)$ data [64]. To tackle this issue, and to increase the density of computation per memory transaction especially on modern many-core architectures, various sparse block storage techniques with either padding or by variable block size were utilized [65].

Block coordinate storage (BCOO) approach [66], scans the original sparse matrix row by row and groups nonzero elements into blocks of a predetermined size. Until all blocks are visited, elements at each block are recorded in a separate vector. Two other arrays are used to store the row and column indices to indicate the start of each block while a third vector holds pointers to the start of the first element in the next block. Example for 3x3 blocks is shown in Figure 6.

	1	2	3	4	5	6	7	8	9
1	1	0	0	0	0	0	0	0	0
2	0	25	0	0	0	0	0	0	0
3	0	16	8	0	0	0	0	0	0
4	0	0	8	1	0	0	0	0	3
5	0	11	0	4	13	0	0	0	15
6	0	6	0	0	0	26	0	0	5
7	0	21	0	0	0	0	1	0	0
8	0	16	0	0	0	0	0	26	0
9	0	13	0	0	0	0	0	0	1

Values	1	0	0	0	25	0	0	16	8	0	0	8	0	11	0	0	6	0
	1	0	0	4	13	0	0	0	26	0	0	3	0	0	15	0	0	5
	0	21	0	0	16	0	0	13	0	1	0	0	0	26	0	0	0	1
Row-Coordinate	1	4	4	4	7	7												
Column-Coordinate	1	1	4	7	1	7												
Pointer	1	10	19	29	38	48												

Figure 6: Block coordinate storage representation

Again, let N be the sparse matrix dimension, α the block size and β the number of blocks. It is clear then that the length of the array that holds the desired matrix elements is less than or equal the area of each block times the total number of blocks. i.e.

$$total_elements \leq \alpha^2 \times \beta$$

Specifying the optimal block dimension and shape autonomously is a little bit challenging. All the previously described formats can be thought of as having blocks of dimension (1×1) . Assuming an optimal block dimension has been chosen, it is then obvious how blocked schemes outperform other representations. After all, we are shrinking the size of index arrays to point to group of data rather than a single one, and of course, the larger this group the more saving is achieved. The catch here is that, if blind block decomposition is initiated, the array that was supposed to hold only nonzero elements might be dominated by zeroes. Consider for instance storing an identity matrix using (BCOO), and a block of size α , then the number of nonzero elements per block is (α) and the overhead storage is $(\alpha^2 - \alpha)$! Figure 7

In an attempt to handle the previous issue, and to achieve better performance, Hierarchical Sparse Matrix Storage Format (HSF) were suggested in [67] as well as some adaptive blocking techniques were suggested [68].



Figure 7: Overhead of using BCOO for various block sizes

Just like the natural rise of (CRS) and (CCS) as an optimal substitute to (COO), compressed versions of (BCOO) can be also established and derived. The vectors that stores the spatial coordinates of each block are further compressed column or row wise and substituted by a suitable pointer arrays; leading to Blocked Compressed Row Storage (BCRS) and Blocked Compressed Column Storage (BCCS) respectively. Despite the huge performance benefits offered by Blocking techniques, it has been reported to result in more than 70% performance degradation if not utilized properly [66].

When it comes to the general purpose massively parallel machines (GPU's), and besides the memory bottleneck problem associated with sparse matrix vector multiplications, there exists additional constrains to achieve better machine utilization. For example, processing many short rows will make loop overhead dominates the computational aspect [64, 69]. Various rows lengths lead to load imbalance and indirect device memory access degrades performance. As a result, and despite the huge advantage of (CRS) and it's derived forms of handling any sparsity pattern, the fact that those techniques require separate vectors to store indices will give rise to more memory transactions and hence limiting performance. Moreover, and although some sparse storage schemes access the stored coefficient matrix contiguously, they suffer from irregular access to the multiplicand vector x [70, 71]. Therefore, if the matrix structure is known priory, specific optimizations could be exploited and a great boost in performance could be achieved if the right representation scheme was chosen properly.

In an attempt to exploit the diagonal structure that resembles wide range simulation problems, [72] implemented a blocked version of the diagonal format. In their

representation, they are aiming at alleviating the overhead of storing unnecessary diagonal inter-elements zeros by defining a new data structure that holds the elements of interests according to predefined degree of freedom (DOF) criteria. [73] introduced a tool to model, profile and predict the performance of sparse matrix vector multiplication (SpMV) on GPUs. Based on the modeling and analysis of a given problem, they designed a dynamic and optimal domain and matrix specific (SpMV) kernel and reported obtaining optimal solution compared to similar kernels offered by NVIDIA. In his thesis, [74] extensively analyzed the performance of PETSc [75] GPU implementation with various sparse matrix storage mechanisms, while [76] studied memory efficiency implications on sparse matrix operations and introduced a new storage scheme. In his Variable Dual Compressed Blocks (VDCB) format, and besides memory manipulation, he divides the original matrix into a number of variable-sized sub-matrices with a bitmap that points to the presence of a non-zero element. He tested his implementation on FPGA and reported good bandwidth gain for various test cases.

The issue of sparse matrix vector multiplication has been extensively studied in the past when CUDA was first introduced. Two famous highly cited papers in [77] and [78]. The reader is referred to [55, 64, 66, 79-81] For more in depth review of sparse matrices on CPU, Multicore and Many-Core devices, their representations and comparisons, [62, 82] for studies dedicated to diagonal matrices, and [66, 67, 72, 83] for blocking restructuring techniques.

2.4 Review of Linear Solver Libraries

Over the past decades, researchers from all around the world have kept developing multipurpose computational libraries and tools that aid advancing their research by reducing programming overhead and facilitating rapid deployment and testing of their ideas. When it comes to linear solvers and computational modeling, [84] have not only listed and categorized dozens of those libraries but also organized them into sections along with links to their website.

Eigen[85, 86] is an excellent and reliable sequential library that provides headers to perform various linear algebra routines. The library has been developed to take advantage of object oriented C++ and its expression templates, features an easily declared, directly accessed matrix and vector data structures. Eigen is flexible and enables easily integrated functions, code reuse and abstraction while maintaining good performance by supporting various optimizations like explicit vectorization, loop unrolling and static memory allocation. The open source library is released under MPL219, is supported by many compilers and has been successfully deployed in many interdisciplinary projects ranging from simple extensions, mobile applications to demanding simulations. A list of those projects is listed on the library main page. Eigen supports both dense and sparse matrix functions with neatly organized, in depth class documentation and test examples. The library also supports multi-threading using OpenMP [10] and if available the Intel Math Kernel (MKL) library [87].

¹⁹ <https://www.mozilla.org/MPL/2.0/>

When it comes to GPUs, NVIDIA provides a CUDA Sparse Matrix library (cuSPARSE) for manipulating and operating on sparse matrices [88]. The library provides a collection of basic linear algebra functions that are called from C++ programs. They reported around 8x faster performance gain over their direct competitor Math Kernel Library (MKL) offered by Intel [87]. The library has been used extensively by the researchers as it provides fast and reliable performance with ease of programming and development effort. For example [89] utilize it to implement (ILU) and Cholesky factorization for iteratively solving linear systems, while [90] used it to accelerate the modeling of deformation of soft tissue using (FEM). [91] made use of the library to boost image segmentation implementation; and [92] apply it for image reconstruction.

Similar to cuSPARSE, the CUSP library [93] provides a wrapper for many functions in cuSPARSE.. It was designed solely to take advantage of the intensive computational aspect of the massively parallel NVidia's GPUs. It is released under the Apache 2.0 open source license. The CUSP library is an inevitable starting point for CUDA developers writing parallel scientific computing applications. The library not only provides abstraction and easy to call cuSPARSE and cuBLAS [94] routines, but also reports good performance. Moreover, the developed applications can be smoothly integrated with THRUST library [95] to enable fast prototyping. CUSP could be used directly by including the associated interface files, and provides dozens of graph algorithms and sparse linear algebra routines easily deployed with many available sparse storage schemes and preconditioners.

PARALUTION [96] supports dozens of well-organized and easily deployed methods for performing plenty of sparse matrix linear algebra routines. Not only it supports various parallel hardware architecture [CPU, NVIDIA GPU, AMD GPU, Xeon Phi (MIC)], but it can also be configured to run on various operating systems and use various plugins. The library comes with useful ready to run examples, and the online documentation provides class hierarchies and in depth implementation details. This open source project exploits object oriented programming paradigm in C++, taking advantage of code reuse, inheritance, clarity, maintainability and abstraction. It is released under GPLv3.20. The library implements various sparse storage schemes with neat functions to convert among them. The list of the provided linear solvers along with the available preconditioners is quite intense as well. Although vectors defined under this library can be easily accessed directly, matrices are not. The issue that has been discussed in the user manual along with some suggested solutions. The library generic implementation and independence, facilitates fast prototyping and testing. Nevertheless, this comes with a price of degrading performance as was tested in our simulator.

²⁰ <http://www.gnu.org/licenses/gpl-3.0.html>

CHAPTER 3

COMPUTATIONAL MODELS, EXPERIMENTATIONS

AND RESULTS

This chapter describes the computational aspect of the simulator. The first section starts by presenting the general operations for both the forward and the inverse models. Section two provides an analytical study for selecting a suitable linear solver for both reservoir problems. Section three then presents an exhaustive experimental evaluation of two nominated parallel solvers in section two applied for matrices extracted from our developed simulator. We concluded that although GMRES is a widely used solver for sequential reservoir simulation; BiCGSTAB with proper preconditioning provides faster performance on parallel machines. In section four and with the focus on the reservoir resulting matrix specific structure, we experiment the issue of parallel sparse matrix vector multiplication on Hepta-Diagonal Matrices. After that, we make use of the famous operation merging trick to attempt implementing a faster version of BiCGSTAB algorithm. Finally, section six extends the implementation presented in section five to implement a same parallel solver but dedicated for multiple right hand side matrices encountered in reservoir history matching.

3.1 Computational Model for Reservoir Simulation

Figure 8 describes the whole history matching process from computational perspective. The process starts by the forward simulation model by assuming certain reservoir parameters and repeatedly estimating the values of other state variables. At the end of this process, the system then starts the inverse model based on the final retained values of the estimated state variables. The inverse model leads to the computation of the sensitivity matrix that is eventually used in history matching. The sensitivity matrix could be obtained by either the forward sensitivity approach or the adjoint state method. Regardless of the followed method, obtaining sensitivity matrix requires the solution of a linear system with multiple RHS. To reduce the dimensionality of the system, some reduction techniques are optionally utilized. The details for each individual step are described next.

3.1.1 The Computational Model of the Forward Simulation Scheme

Without loss of generality, Figure 9 shows how the final assembled system for small grid dimension looks like.

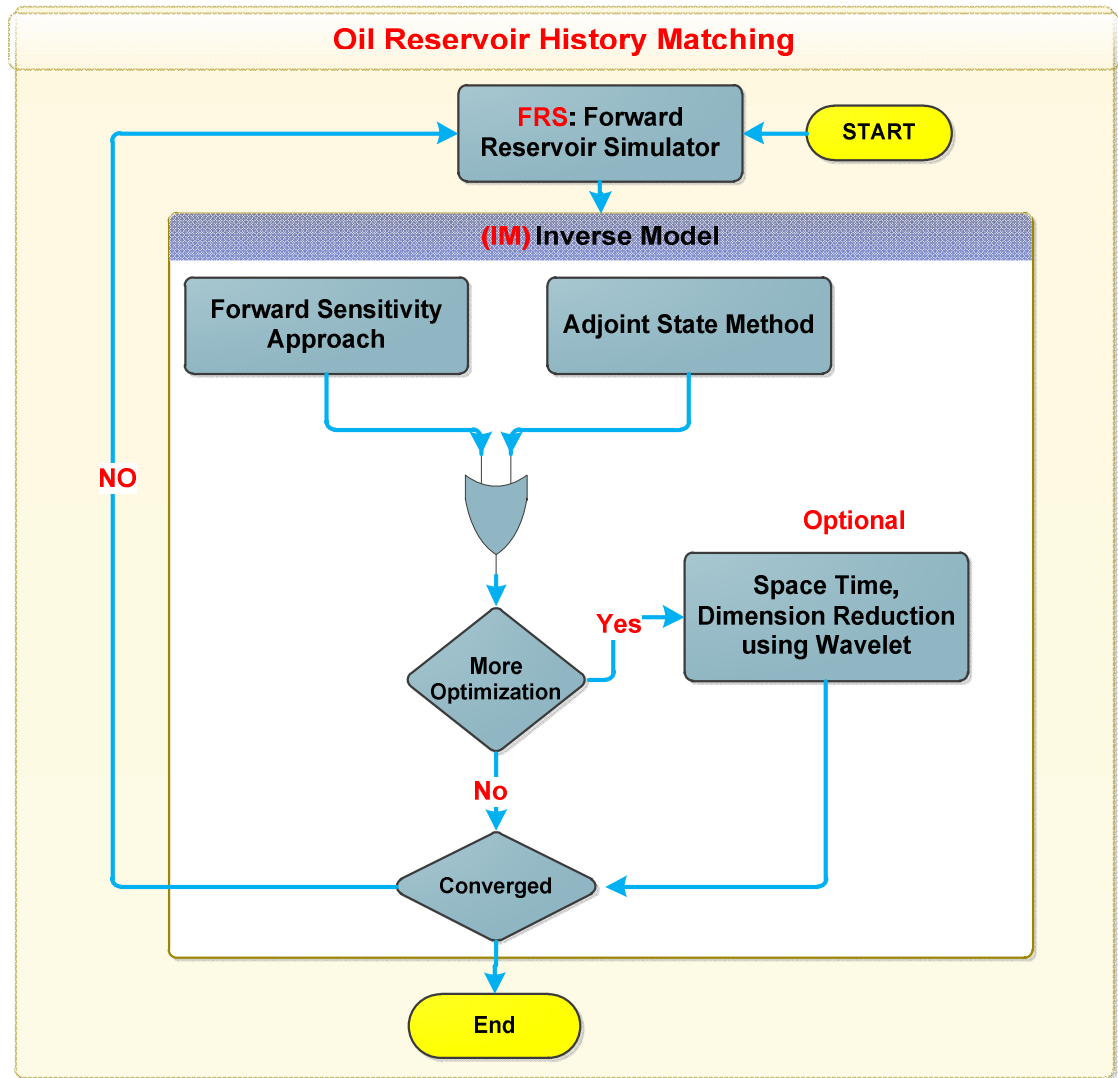


Figure 8: The Computational Model for Oil Reservoir History Matching

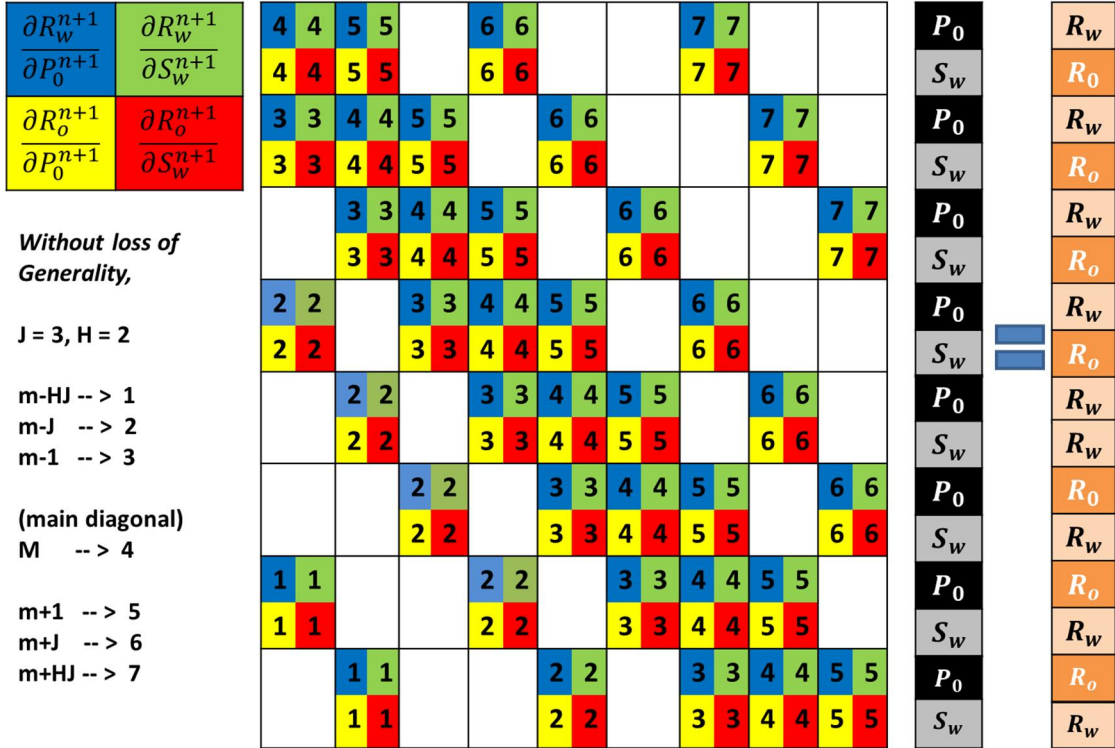


Figure 9: Sample snapshot of the assembled linear system for FSR, (J, I and H): is the maximum number of steps in the z, x and y directions, respectively

For a 3D problem and two simulated phases discretized using Finite volume method, it is clear that the maximum number of non-zero elements at each row is 14. Let m be the total number of grid cubes, then the size of the Jacobian matrix is $(2m \times 2m)$, and the total number of non-zero elements are at most $(14 \times 2m)$. As a result, the fraction of non-zero elements in the system is less than

$$\frac{14 \times 2m}{2m \times 2m} = \frac{7}{m}$$

It is obvious that special care should be taken when selecting a suitable solver for implementation, especially for practical dimensions ($m = 10^6$), as most of the operations on the zero elements are not necessary and should be avoided to reduce the computational complexity.

Not only the Jacobian matrix is sparse, but also it is unsymmetrical, ill-conditioned and has a special Hepta-Diagonal structure. That thing also influences the selection or implementation of any linear solver. More details on this computational model could be found in the Appendix.

3.1.2 The Computational Model of the Inverse Simulation Scheme

Forward Sensitivity Approach

The forward sensitivity process starts right after completing the whole forward reservoir simulation, for a given time step and for the initially assumed reservoir parameters. It repeats until the estimated parameters are matched. At this stage, we aim at simultaneously solving the system.

$$A_{2m \times 2m} \times X_{2m \times m} = B_{2m \times m}$$

$$B = \{b_1, b_2, \dots, b_m\} \text{ \& } m \text{ is the total number of grid blocks}$$

The coefficient matrix A is the same Jacobian matrix obtained in the forward model that is blocked Hepta-Diagonal and sparse. It is also ill-conditioned and unsymmetrical. B is a combination between diagonal matrix and other hepta diagonal sparse matrices. Figure 10, demonstrates the computational aspect of this approach. The figure shows four modules used to formulate the linear system with multiple right hand sides (RHS) that are described as follows:

- **Module 1:** Computing the partial Derivatives (Or Jacobian matrix (J) for the last retained values of (P_o & S_w) from current iteration. The matrix structure and characteristics are the same as the previous one in the forward model.
- **Module 2:** blocked diagonal matrix (D) of size ($2m \times 2m$) that represent the mass accumulation for each phase Eq. (1.25) and calculated using Eq.(1.29)
- **Module 3:** matrix (Y) of size ($2m \times m$). It is constructed by analytically taking the derivative of the residual equations (R_o & R_w) with respect to the perturbed parameter (α). This matrix is also sparse blocked-hepta diagonal.
- **Module 4:** The sensitivity matrix, denoted by S, of size ($2m \times m$) , initially zero and updated at every time iteration. The assembled system solves the system of multiple RHS for new S.

The final assembled system will have the form as presented in Eq. (1.27). Again this is computationally very expensive for practical reservoir dimensions and special care should be taken when choosing a suitable solver. One characteristic of Equation (1.27) is that the matrix J is the same for all the independent multiple right-hand-side vectors and the huge cost of either its factorization or preconditioning is alleviated by its repetitive utilization in the solution. Nevertheless, if the number of parameters of the system is very huge, the (RHS) size will also be very huge and the computational time of solving the previous system becomes prohibitive. Moreover, the forward sensitivity approach computes the sensitivities of the state variables at all grids leading to redundant calculations as sensitivities at well locations are the only ones required in the solution of the inverse problem.

An alternative that addresses the last two limitations is the adjoint-state method in which the computational time depends mainly on the number of data to be matched. The adjoint method also has the property of directly computing the sensitivities of well variables at well locations only.

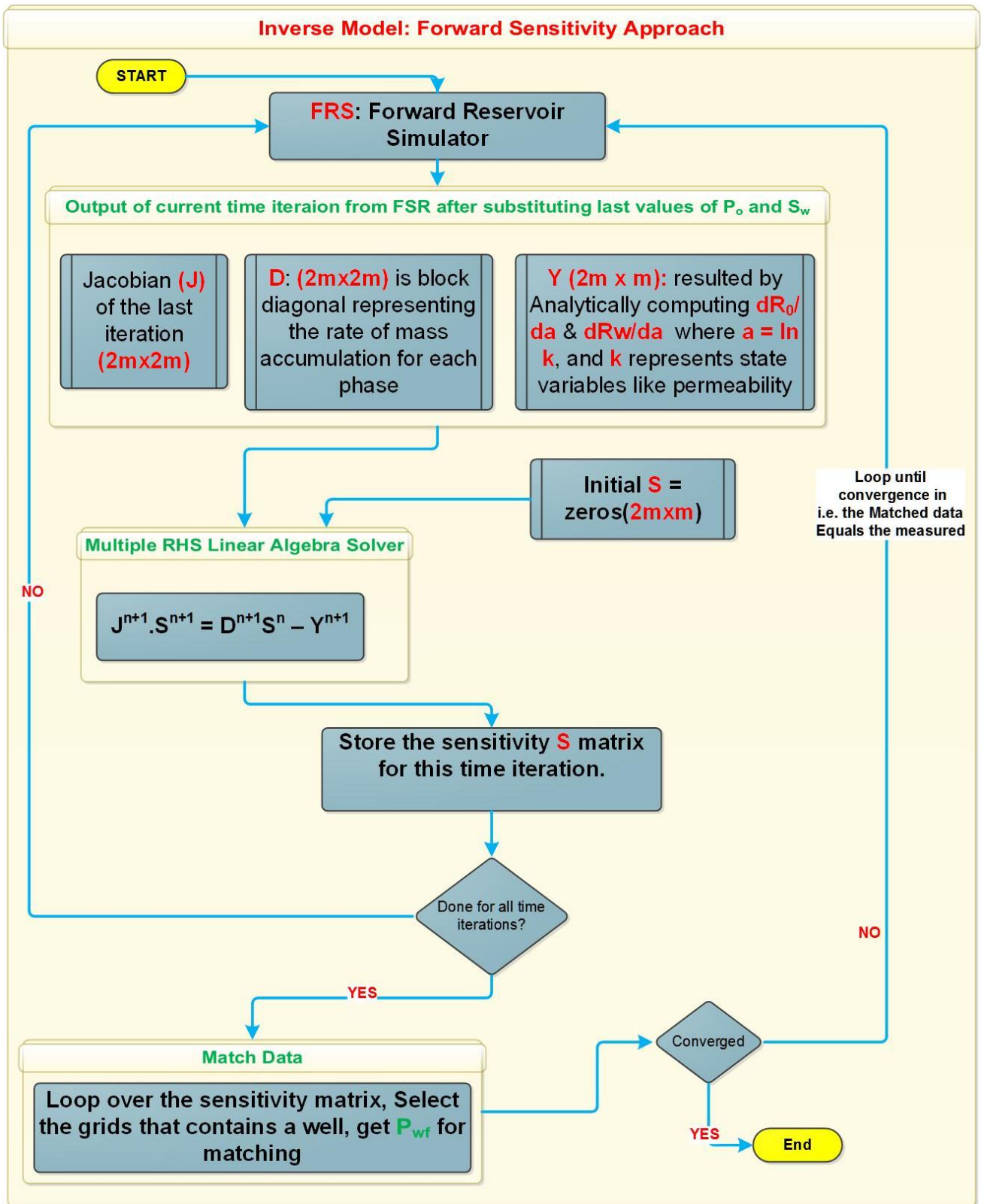


Figure 10: Inverse Model: Forward Sensitivity Approach

Adjoint Sensitivity Approach

From computational perspective, the goal is again to solve a system with many (RHS) in order to get the sensitivity matrix that is used later for matching data. Unlike the forward sensitivity method, the width of the (RHS) in this case is the actual data to be matched rather than the number of parameters to be estimated. Moreover, and unlike the previous approach, the adjoint method requires the forward reservoir simulator (FRS) to complete all its time iterations, and to store some needed data, like the Jacobian for all iterations. After that, and starting from the last time step, the assembled system of multiple (RHS) is solved in a backward substitution manner [Eq. (1.48)] and the sensitivity matrix is built at each backward step. The whole computational model is better explained in Figure 11. The J & D elements that constitute the multiple (RHS) are the same as the ones described in module 1 and 2 in the forward sensitivity approach, except we are taking the transpose of the Jacobian matrix. $\left(\frac{\partial \eta}{\partial \bar{u}^n}\right)$ contains the derivatives of the data to be matched with respect to state variables (P_o & S_w). $\vec{\lambda}^n$ is initially the adjoint variable resulted by adjoining the constrains to the data to be matched. Let k: number of data to be matched. Then $\vec{\lambda}^n$ is of size $2m \times k$, while $\left(\frac{\partial \eta}{\partial \bar{u}^n}\right)$ is of size $k \times 2m$.

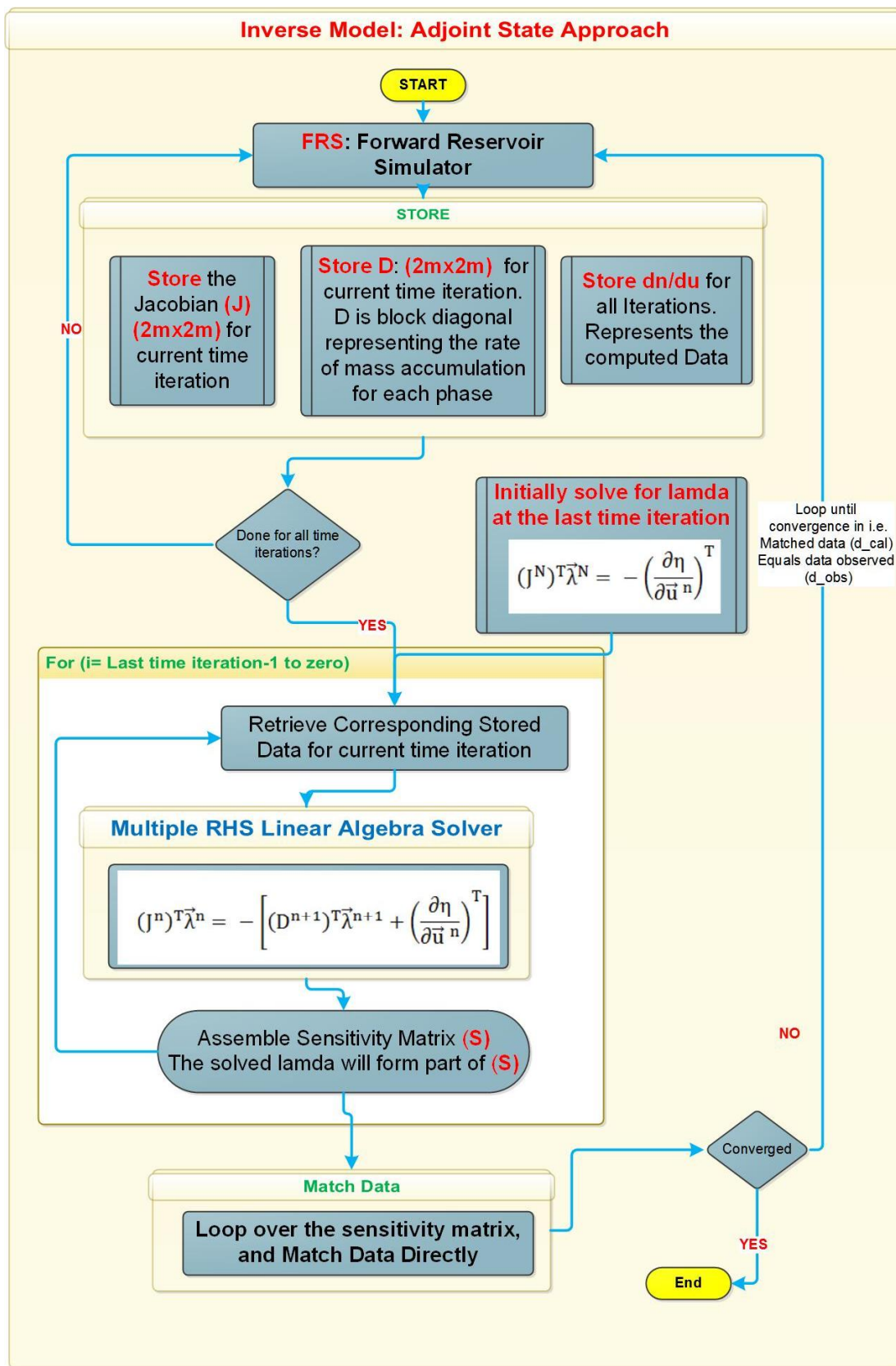


Figure 11: The computational model for the Adjoint Sensitivity Approach

3.2 Analytical Parallel Linear Solver Selection

The finite volume discretization of flow equations that governs two-phase oil water reservoir behavior in the forward modeling will yield a sparse system having ill-conditioned unsymmetrical coefficient matrix with Hepta-diagonal profile and 2×2 block representing each entry. Moreover, the inverse problem requires solving either the same matrix, forward sensitivity approach, or its transpose in the case of adjoint sensitivity approach, with multiple right hand side.

With the goal of writing a parallel code to speed up the computational process for our black-oil simulator, this section reviews four applicable preconditioned Krylov methods of interest. Our final selection will depend on analyzing the concurrency of each algorithm as well as the usually addressed issues of storage, accuracy and convergence. Given that perspective, the following iterative solvers will be nominated for further study: The Generalized Minimum Residual Method (GMRES) by Saad and Schultz [47], The Bi-Conjugate Gradient Method (BiCG) by Fletcher [48], the quasi-minimal residual method (QMR) by Freund and Nachtigal [49], and finally, the Bi-Conjugate Gradient Stabilized (Bi-CGSTAB) by Van der Vorst [50].

The rationale behind selecting the above four for further analysis is three folded. First, the coefficient matrix (A) of interest has certain properties that put further restrictions on any selection. Because of its very large dimensions and the sparsity pattern, direct methods will be excluded because of their reported memory demands. Moreover, as (A) is ill-conditioned, stationary iterative methods will not be considered because of issues related

to convergence. Finally, since (A) is unsymmetrical, some Krylov based methods dedicated for symmetrical systems will not be taken into account. Second, to study the variation among the same classification class, we chose to include QMR and BiCG as representatives of the Petrov–Galerkin approach. For interclass comparison we study BiCGSTAB from the hybrid camp and the famous GMERS for its desirable reported stability from the minimum norm residual methods. Third, as sound parallel implementation will eventually serve in reducing the overall execution time or enabling larger problems to be handled with the same sequential time, the selected solver should have high degree of data independency regardless of the amount of work involved. Moreover, the selected algorithms should be in harmony with the target parallel architecture as the later imposes additional constrains.

With different permutations and various scaling, Krylov subspace methods share common operations ranging from an embarrassingly parallel tasks like sparse matrix vector multiplication, to norm calculations, dot product as well as vector updates, Table 3. It is worth mentioning such a table is constructed with relaxed but unified assumptions and its only purpose is to give a general overview. The estimation of the *Required Steps to Complete Tasks in Parallel* is established by assuming infinite processing and memory resources, zero communication penalty and by neglecting all other overhead. Without loss of generality, in reduction example like vector multiplication and assuming a matrix dimension of size (N) , at least $\text{Log}(N)$ steps are needed before producing the final answer [97]. Moreover, matrix vector multiplication can be seen as the process of performing (N) independent reductions. Assuming that every worker will be responsible for calculating one element in the resulting vector by processing its corresponding row and column, then

each calculation will require $\log(N)$ steps. Finally and given the above assumptions, vector update is done instantly in one step. On the other hand the estimation of the available parallel work is established as follows: the reduction operation requires N processing elements at the beginning, $N/2$ in the next iteration, followed by $N/4$ and so on. In other words, the total work that could be completed in parallel could be calculated as:

$$\begin{aligned}
 TotalWork_{parallel} &= N + \frac{N}{2} + \frac{N}{4} + \dots + 2 + 1 \\
 &= N \left(1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{N} \right).
 \end{aligned}$$

Between the above parentheses is a Harmonic Series with its sum equals $\ln(N) + Euler\ Macheloni\ Constant$. Therefore and for the reduction case,

$$TotalWork_{parallel} = N \cdot \text{Log}(N)$$

Table 3: The main tasks the constitute Krylov Linear Solvers with an anticipated associated parallel complexities

Operation	Required Steps to complete Operations in Parallel (Infinite Resources)	Estimated Available Parallel Work	Example
Work Sharing	$O(\text{Log}(N))$	$N^2 \text{Log}(N)^{21}$	Matrix-vector multiplication. (Ax)
Solving a sparse preconditioned linear system	$O(N)$ assumed	N^2 (assumed)	$My = v$
Reduction	$O(\text{Log}(N))$	$N \text{Log}(N)$	Vector multiplication. $(v.v)$
Vector Update	$O(1)$	N	$\omega = y \pm v$
Vector Scaling	$O(1)$	N	$\omega = \alpha v$
Scalar operation	$O(1)$	1	$y = \alpha \pm b$ or $y = \alpha \times b$

3.2.1 The Generalized Minimum Residual Method (GMRES)

GMRES identifies x_k for which the Euclidean norm $\|b - Ax_k\|_2$ is minimal over the Krylov subspace generated by A and r_0 . As much as the method is well known for its robustness [37, 45, 47], it is also characterized by demanding large resources as the computation proceeds. The method is based on the Arnoldi-modified Gram-Schmidt procedure to build orthogonal basis of the Krylov subspace [56] and produces an upper Heisenberg matrix before finally the approximated solution is computed. [55] showed that if the coefficient matrix A is positive definite, then GMRES algorithm converges for any dimension of the considered Krylov space. To address storage issues, restarted versions

²¹ N for the all the rows, and $N \cdot \text{LOG}(N)$ for every reduction in a row

were introduced where only intermediate results are used in order to compute the next m iterations as initial data after the already accumulated data are erased [56]. The challenge remains in picking up suitable value of such (m) as its value is problem specific and bad choice may either result in unnecessary slow convergence or even failing of convergence. One realization of the algorithm as presented by [45] is shown in Algorithm 1. To obtain the final solution, GMRES requires solving an upper triangular system after applying some plane rotations. Let (N) be the matrix leading dimension and (m) the restart value. Besides storing the original matrix, we notice the need for a long recurrence²² for computing the Arnoldi iteration. Moreover, GMRES needs to store five main arrays of size (N); they are namely (r , v_1 , ω , y and x). Moreover an array of size ($N * m$) is needed to store v_{i+1} . As a result, and by ignoring spaces required to store scalars or vectors of small sizes compared to (N) like the space needed to store (H), the minimal total storage required by GMRES is:

$$STORE_{GMRES} = \text{Space to Store the Original Matrix} + N(5 + m).$$

²² Dependencies needed by subsequent iterations

```

1    $x^{(0)}$  is an initial guess
2   for  $j = 1, 2, \dots$ 
3       Solve  $r$  from  $Mr = b - Ax^{(0)}$ 
4        $v^{(1)} = \frac{r}{\|r\|_2}$ 
5        $S := \|r\|_2 e_1$ 
6       for  $i = 1, 2, \dots, m$ 
7           Solve  $\omega$  from  $M\omega = b - Av^{(i)}$ 
8           for  $k = 1, \dots, i$ 
9                $h_{k,i} = (\omega, v^{(k)})$ 
10               $\omega = \omega - h_{k,i} \cdot v^{(k)}$ 
11          end
12           $h_{i+1,i} \leftarrow \|\omega\|_2$ 
13           $v^{(i+1)} = \omega / h_{i+1,i}$ 
14          Apply  $J_1, \dots, J_{i-1}$  on  $(h_{1,i}, \dots, h_{i+1,i})$ 
15          Construct  $J_i$ , acting on  $i^{\text{th}}$  and  $(i+1)$ st component
16          of  $h_{\cdot,i}$ , such that  $(i+1)$ st component of  $J_i h_{\cdot,i}$  is 0
17           $s := J_i s$ 
18          if  $s(i+1)$  is small enough then (UPDATE  $(\tilde{x}, i)$  and quit)
19      end
20      UPDATE  $(\tilde{x}, m)$ 
21      Check convergence; continue if necessary
22  end
23  In this scheme UPDATE  $(\tilde{x}, i)$  replaces the following computations:
24  Compute  $y$  as the solution of  $Hy = \tilde{s}$ , in which the upper  $i \times i$  triangular
25  part of  $H$  has  $h_{i,j}$  as its elements (in least squares sense if  $H$  is singular),  $\tilde{s}$ 
26  represents the first  $i$  components of  $s$ 
26   $\tilde{x} = x^{(0)} + y_1 v^{(1)} + y_2 v^{(2)} + \dots + y_i v^{(i)}$ 
27   $s^{(i+1)} = \|b - A\tilde{x}\|_2$ 
28  if  $\tilde{x}$  is an accurate enough approximation then quit
29  else  $x^{(0)} = \tilde{x}$ 

```

Algorithm 1: Preconditioned GMRES (m) Method as presented in [45]

The main transactions per iteration in GMRES Algorithm, can be approximated as follows: $(2m + i + 2)$ reduction operations²³, $(i + 2)$ vector updates and $(m + 1)$ matrix vector multiplications. Although the sequence of operations in GMRES Algorithm, could be mapped directly to efficient parallel GPU kernels, the algorithm itself poses an inherent sequential behavior [98].

3.2.2 The Bi-Conjugate Gradient Method (BiCG)

As a generalization for the famous CG solver and following the Petrov–Galerkin approach, BiCG identifies x_k for which the residual $b - Ax_k$ is orthogonal to some other suitable k -dimensional subspace [37]. By utilizing both the original coefficient matrix A , and its transpose A^T , the BiCG method aims at generating two CG-like sequences of vectors that are mutually orthogonal to be used to update the residual as well as the search direction [45]. As the method may either breakdown and because of the reported irregular convergence behavior [35, 45], other approaches such as QMR and Bi-CGSTAB were suggested as a replacement. The general algorithm [45] for this process is shown in Algorithm 2.

BiCG method consists of a series of sparse matrix vector multiplications as well as vector updates. The implementation of the algorithm is straight forward. By comparing its sequence of operations with the previously shown one in GMRES, we expect BiCG to scale better and to consume less storage. Nevertheless, due to its failure conditions and to account for the case where A^T is not present, we will consider one of its enhancements.

²³ Every matrix vector multiplication operation has an embedded reduction task

1. Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$.
2. Choose $\tilde{r}^{(0)}$ (for example, $\tilde{r}^{(0)} = r^{(0)}$),
3. **for** $l = 1, 2, \dots$
4. Solve $Mz^{(i-1)} = r^{(i-1)}$
5. Solve $M^T \tilde{z}^{(i-1)} = \tilde{r}^{(i-1)}$
6. $\rho_{i-1} = z^{(i-1)T} \tilde{r}^{(i-1)}$
7. **if** $\rho_{i-1} = 0$, **method fails**
8. **if** ($l = 1$)
9. $p^{(i)} = z^{(i-1)}$
10. $\tilde{p}^{(i)} = \tilde{z}^{(i-1)}$
11. **else**
12. $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$
13. $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
14. $\tilde{p}^{(i)} = \tilde{z}^{(i-1)} + \beta_{i-1} \tilde{p}^{(i-1)}$
15. **endif**
16. $q^{(i)} = A.p^{(i)}$
17. $\tilde{q}^{(i)} = A^T.\tilde{p}^{(i)}$
18. $\alpha_i = \frac{\rho_{i-1}}{\tilde{p}^{(i)T} q^{(i)}}$
19. $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
20. $r^{(i)} = r^{(i-1)} + \alpha_i q^{(i)}$
21. $\tilde{r}^{(i)} = \tilde{r}^{(i-1)} + \alpha_i \tilde{q}^{(i)}$
22. Check convergence; continue if necessary
23. **end**

Algorithm 2: Preconditioned BiCG Method as presented in [45]

Besides the space needed to store the original matrix, the algorithm needs ten auxiliary vectors of size N to process data. Those are $(r, \check{r}, z, \check{z}, p, \check{p}, q, \check{q}, b$ and $x)$. As a result the minimal total storage required by BiCG is

$$\text{STORE}_{\text{BiCG}} = \text{Space to Store the Original Matrix} + 10N.$$

For the main transaction per iteration, there are four reduction operations, five vector updates (lines: 10, 11, 18, 19, 20), and two matrix vector multiplication.

3.2.3 The Bi-Conjugate Gradient Stabilized Method (BiCGSTAB)

Bi-CGSTAB can be seen as a product of BiCG algorithm and repeated application of GMRES algorithm of degree one [45]. In that sense, the operation with A^T is transformed to another polynomial in A . The convergence is smoother and may even be faster than BiCG [35]. A preconditioned version of BiCGSTAB as presented by [45] is shown in Algorithm 3.

Similar to BiCG, the main operations of the algorithm consists of sparse matrix vector multiplication as well as vector updates and dot products. The sequential implementation is also straight forward. BiCGSTAB makes use of ten vectors to complete its computation in addition to the original matrix storage. We can identify $(x, b, r, \check{r}, p, \check{p}, v, s, \check{s}$ and $t)$. As a result the minimal total storage required by BiCGSTAB is

$$\text{STORE}_{\text{BiCGSTAB}} = \text{Space to Store the Original Matrix} + 10N$$

For every iteration, we can identify seven reduction operations, four vector updates, and finally two matrix vector products.

1. Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$.
2. Choose $\tilde{r}^{(0)}$ (for example, $\tilde{r}^{(0)} = r^{(0)}$),
3. **for** $l = 1, 2, \dots$
4. $\rho_{l-1} = \tilde{r}^T r^{(l-1)}$
5. **if** $\rho_{l-1} = 0$, **method fails**
6. **if** ($l = 1$)
7. $p^{(l)} = r^{(l-1)}$
8. **else**
9. $\beta_{l-1} = (\rho_{l-1} / \rho_{l-2}) (\alpha_{l-1} / \omega_{l-1})$
10. $p^{(l)} = r^{(l-1)} + \beta_{l-1} (p^{(l-1)} - \omega_{l-1} v^{(l-1)})$
11. **endif**
12. Solve $M\tilde{p} = p^{(l)}$
13. $v^{(l)} = A.\tilde{p}$
14. $\alpha_l = \rho_{l-1} / \tilde{r}^T v^{(l)}$
15. $s = r^{(l-1)} - \alpha_l v^{(l)}$
16. Check norm of s ;
17. **if** small enough: set $x^{(l)} = x^{(l-1)} + \alpha_l \tilde{p}$
18. Solve $M\tilde{s} = s$
19. $t = A.\tilde{s}$
20. $\omega_l = t^T s / t^T t$
21. $x^{(l)} = x^{(l-1)} + \alpha_l \tilde{p} + \omega_l \tilde{s}$
22. $r^{(l)} = s - \omega_l t$
23. Check convergence; continue if necessary
24. For continuation it is necessary that $\omega_l \neq 0$
25. **end**

Algorithm 3: Preconditioned BiCGSTAB Method as presented in [45]

3.2.4 The Quasi-Minimal Residual Method (QMR)

With almost similar computational cost and parallelization properties as BiCG, QMR was designed originally to avoid the irregular convergence behavior as well as one of the two breakdown situations of BiCG by solving a reduced tridiagonal system in a least-square sense [45]. QMR uses a look-ahead variant of nonsymmetrical Lanczos process to generate basis vectors that is induced by matrix A and can be implemented using short recurrences [49]. Beside the smooth convergence property compared to BiCG, it is possible to obtain error bounds for QMR similar to the standard bounds for GMRES [49]. Algorithm 4, shows a preconditioned version of OMR as presented by [45].

Again besides the storage space for the input matrix, QMR demands a minimum of sixteen additional vectors to find the solution vector and residual. Those are mainly $(b, r, x, v, \check{v}, w, \check{w}, y, \check{y}, z, \check{z}, q, p, \check{p}, s$ and $d)$. Therefore, the minimal total storage required by QMR is

$$\text{STORE}_{\text{QMR}} = \text{Space to Store the Original Matrix} + 16N$$

When it comes to the main operations within an iteration, we can identify seven reductions, eight vector updates, and finally two matrix vector products.

1. Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
2. $\check{v}^{(1)} = r^{(0)}$; solve $M_1 y = \check{v}^{(1)}$; $\rho = \|y\|_2$
3. Choose $\check{w}^{(1)}$, for example $\check{w}^{(1)} = r^{(0)}$
4. solve $M_2^T z = \check{w}^{(1)}$; $\xi_1 = \|z\|_2$
5. $\gamma_0 = 1$; $\eta_0 = -1$
6. for $i = 1, 2, \dots$
7. if $\rho_i = 0$ or $\xi_i = 0$ **method fails**
8. $v^{(i)} = \check{v}^{(i)}/\rho_i$; $y = y/\rho_i$
9. $w^{(i)} = \check{w}^{(i)}/\xi_i$; $z = z/\xi_i$
10. $\delta_i = z^T y$; if $\delta_i = 0$ **method fails**
11. solve $M_2 \check{y} = y$
12. solve $M_1^T \check{z} = z$
13. if $i = 1$
14. $p^{(1)} = \check{y}$; $q^{(1)} = \check{z}$
15. else
16. $p^{(i)} = \check{y} - (\xi_i \delta_i / \varepsilon_{i-1}) p^{i-1}$
17. $q^{(i)} = \check{z} - (\rho_i \delta_i / \varepsilon_{i-1}) q^{i-1}$
18. endif
19. $\check{p} = Ap^{(i)}$
20. $\varepsilon_i = q^{(i)T} \check{p}$; if $\varepsilon_i = 0$ **method fails**
21. $\beta_i = \varepsilon_i / \delta_i$; if $\beta_i = 0$ **method fails**
22. $\check{v}^{(i+1)} = \check{p} - \beta_i v^{(i)}$
23. solve $M_1 y = \check{v}^{(i+1)}$
24. $\rho_{i+1} = \|y\|_2$
25. $\check{w}^{(i+1)} = A^T q^{(i)} - \beta_i w^{(i)}$
26. solve $M_2^T z = \check{w}^{(i+1)}$
27. $\xi_{i+1} = \|z\|_2$
28. $\theta_i = \rho_{i+1} / \gamma_{i-1} \beta_i$; $\gamma_i = 1 / \sqrt{1 + \theta_i^2}$; if $\gamma_i = 0$ **method fails**
29. $\eta_i = -\eta_{i-1} \rho_i \gamma_i^2 / (\beta_i \gamma_{i-1}^2)$
30. if $i = 1$
31. $d^{(1)} = \eta_1 p^{(1)}$; $s^{(1)} = \eta_1 \check{p}$
32. else
33. $d^{(i)} = \eta_i p^{(i)} + (\theta_{i-1} \gamma_i)^2 d^{(i-1)}$
34. $s^{(i)} = \eta_i \check{p} + (\theta_{i-1} \gamma_i)^2 s^{(i-1)}$
35. endif
36. $x^{(i)} = x^{(i-1)} + d^{(i)}$
37. $r^{(i)} = r^{(i-1)} + s^{(i)}$
38. Check convergence; continue if necessary
39. end

Algorithm 4 :
Preconditioned
QMR Method as
presented in [45]

3.2.5 Linear Solver Selection Based Tradeoffs

We aim at selecting a solver that suits the most our implemented oil reservoir simulator. Despite their tremendous flavors and the dozens of available implementations nicely summarized in [99], picking up a universal and efficient parallel sparse linear solver is very challenging as many mutually interacting and application specific factors stands in the way.

Table 4 summarizes the obtained analysis of the storage requirement for the four nominated solvers. It can be seen that both BiCG and BiCGSTAB demands the least storage among the four solvers if practical restart values are used in GMRES.

Table 4: The Storage Requirement for the four solvers

Linear Solver	Storage Requirement
GMRES(m)	$matrix + N(5 + m)$
BiCG	$matrix + 10N$
BiCGSTAB	$matrix + 10N$
QMR	$matrix + 16N$

One can also anticipate the parallel behavior of an algorithm from the number of required reductions. Usually, the more the reductions, the longer the sequential steps to be followed and roughly the lower the scalability of an algorithm. Let m be the restart number in GMRES and Iter: the number of required iterations, Table 5 lists the needed reductions for the nominated algorithms based on previous analysis.

Table 5: Number of reductions in the four nominated algorithms

Algorithm	Number of Reductions
GMRES	$\left(2m + 2 + \sum_{i=1}^m i \right) * \text{Iter}$
BiCG	$4 * \text{Iter}$
BiCGSTAB	$7 * \text{Iter}$
QMR	$7 * \text{Iter}$

Table 6 summarizes the main transactions per iteration that are utilized by every solver. Such operations could be transformed later to efficient GPU parallel kernels. As those algorithms are composed of the same operations but with different ordering and counts, they all poses good degree of data parallelism. Generally speaking and according to Amdahl's Law [100], an algorithm with fewer number of transactions would be more scalable and faster. However, this is true and should only be interpreted per iteration. The overall speed of a given Krylov solver is subjected to many other factors including the utilized preconditioner, the matrix condition number and the convergence characteristics.

Table 6: Summary of the number of main transactions within an iteration

	Reduction (Dot Product)	Matrix Vector Product	Vector Update $y = \alpha x + \beta y$
GMRES(m)	$2m + i + 2$	$i + 2$	$m + 1$
BiCG	4	2	5
BiCGSTAB	7	2	4
QMR	7	2	8

The convergence of Krylov subspace methods depends on the spectral properties of both the coefficient and preconditioned matrix. Convergence comparison can be achieved by a numerical experiments with a clear stopping criteria and using an appropriate norm. The only convergence result for Krylov subspace method is given in the following theorem [101]. Similar inequality also hold for the remaining three solvers.

Theorem: Let $u^{(k)}$ denote the iterate generated after k steps of GMRES iteration, with residual $r^{(k)}$. If F is diagonalizable, that is, $F = V\Lambda V^{-1}$ where Λ is the diagonal matrix of eigenvalues of F and V is the matrix whose columns are the eigenvectors, then

$$\frac{\|r^{(k)}\|}{\|r^{(0)}\|} \leq \kappa(V) \min_{p_k \in \Pi_k, p_k(0)=1} \max_{\lambda_j} |p_k(\lambda_j)|,$$

where $\kappa(V) = \|V\| \|V^{-1}\|$ is the condition number of V .

Accuracy depends on how many iterations the solver is allowed to perform. Theoretically, for all these solvers, Cayley-Hamilton theorem states that the exact solution (100% accuracy) is obtained in at most N iterations where N the size of the matrix is.

The following points summarize our selection criteria for the linear solver:

- First we exclude the famous GMRES algorithm, as it demands lot of computational resources. A suitable preconditioner will be utilized to improve the convergence with other solvers.

- We also exclude BiCG, as it poses some related convergence problems not to mention the existence of an enhancement with similar parallel behavior which also belong to the same class of BiCG.
- To choose between QMR and BiCGSTAB we further try to analyze concurrency profile

3.2.6 The Study of Concurrency Profile for BiCGSTAB and QMR

Independent of the number of available processors, assignment or orchestration, the concurrency profile shows the number of tasks that could be performed concurrently in a given time [102]. It could be derived and constructed by plotting the number of available operations at every level in the dependency graph versus the level number. Figure 12 and Figure 13. The number of available concurrent operations per iteration, in the preconditioned version of the two solvers is shown in Table 7. Estimated available parallel work was taken from Table 3.

Table 7: The number of available concurrent operations in QMR and BiCGSTAB. N is matrix leading dimension

Level	QMR	BiCGSTAB
1	$N + N^2 \text{Log}(N)$	$N + N^2 \text{Log}(N)$
2	$2N$	N
3	$2N^2$	$N \text{Log}(N)$
4	$2N \text{Log}(N)$	1
5	$4N$	$3N$
6	$2N^2 + N \text{Log}(N)$	N^2
7	$3N$	$N^2 \text{Log}(N)$
8	$N^2 \text{Log}(N)$	$N + \text{Log}(N)$
9	$N \text{Log}(N)$	$2N$
10	1	$N \text{Log}(N)$
11	$3N + N^2 \text{Log}(N)$	N^2
12	$2N^2$	$N^2 \text{Log}(N)$
13	$2N \text{Log}(N)$	$2N \text{Log}(N)$
14	$2N$	$5N$
15	1	
16	1	
17	$4N$	
18	$2N$	

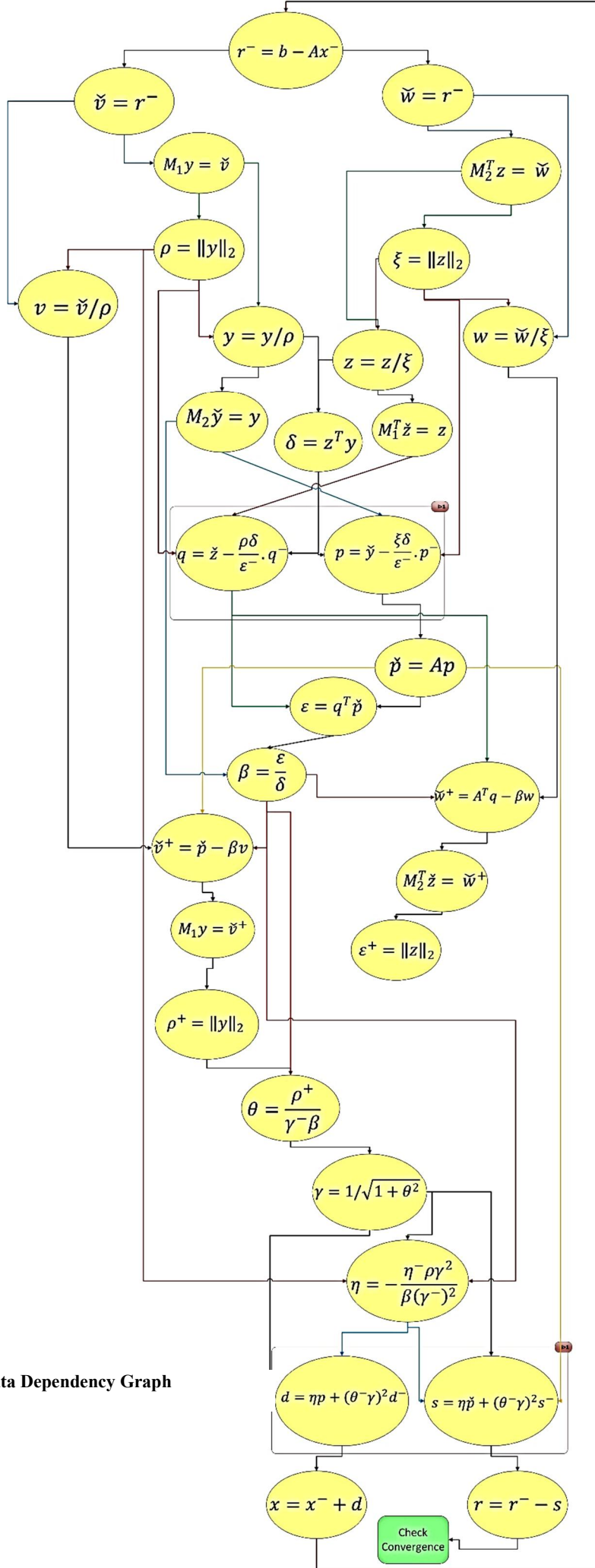


Figure 12: QMR Data Dependency Graph

BICG STAB
Data
Dependency
Graph

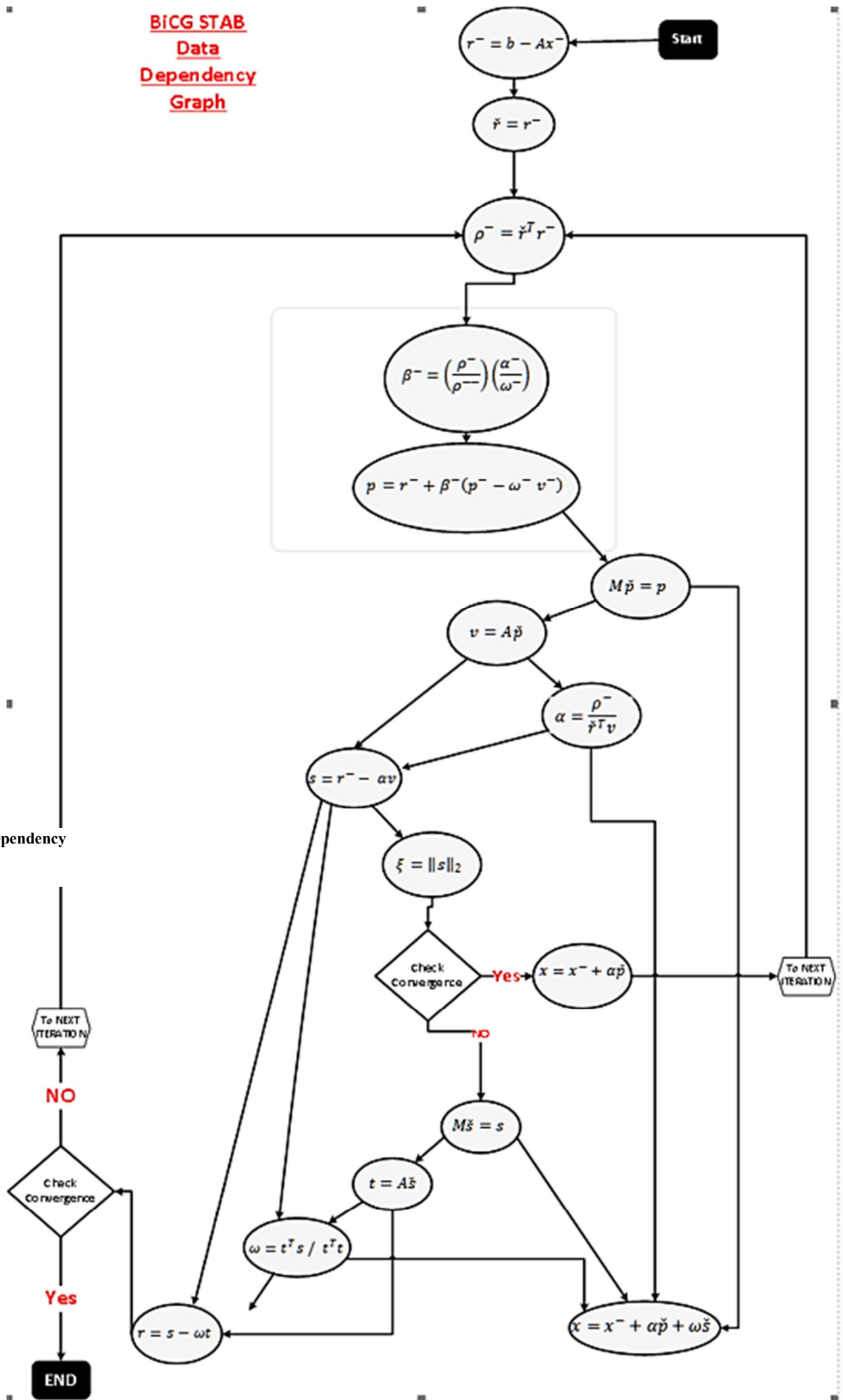


Figure 13: QMR Data Dependency Graph

The resulting plots of concurrency graphs are shown in Figure 14 and Figure 15. The total amount of computational work is then calculated by estimating the area under the resulting constructed line segments. [102] has showed if the previous assumptions were considered and if unlimited number of processors were utilized, then the maximum achievable speed up is less than or equal the value of average parallelism²⁴. Despite our relaxed assumptions, especially for the parallel amount of work involved in solving the preconditioned system, the average parallelism in BiCGSTAB algorithm is slightly higher than its counterpart in QMR algorithm.

²⁴ Average Parallelism is calculated by dividing the area bounded by the line segments in the dependency graph over the horizontal access extent.

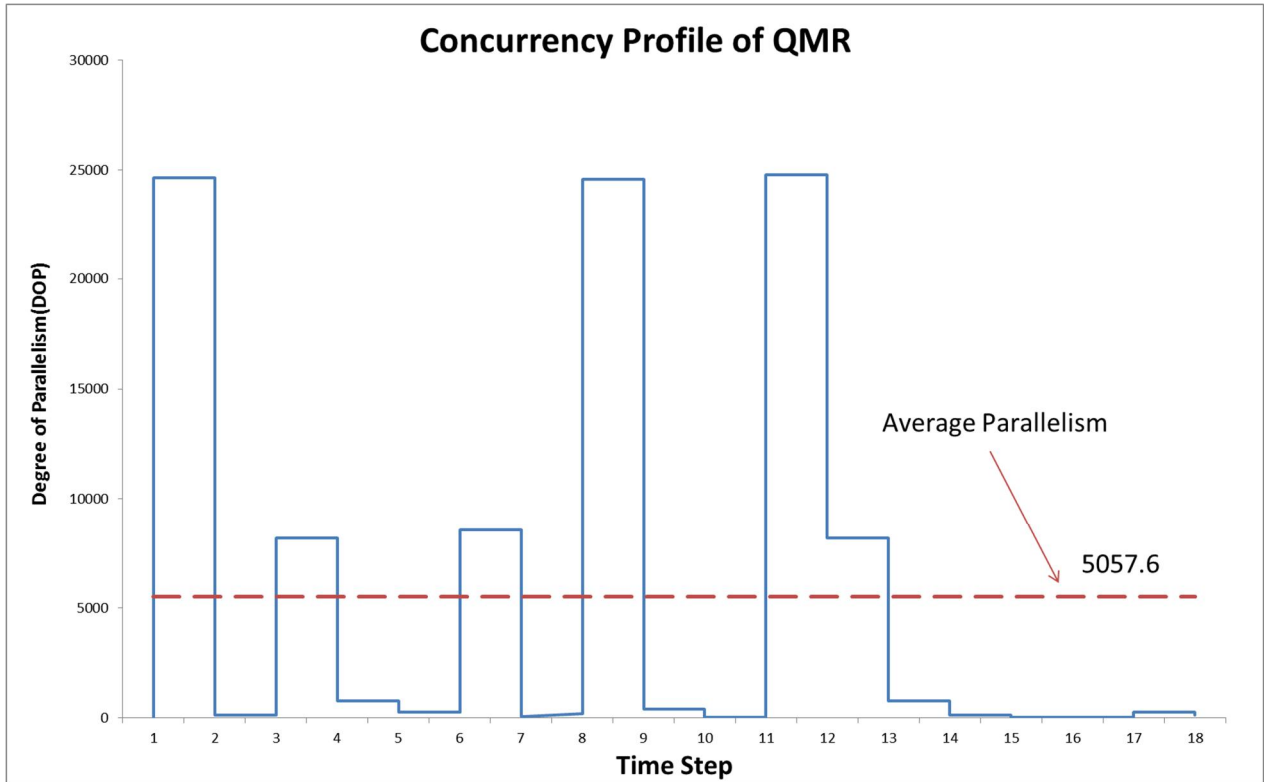


Figure 14: Concurrency Profile of QMR with N=64

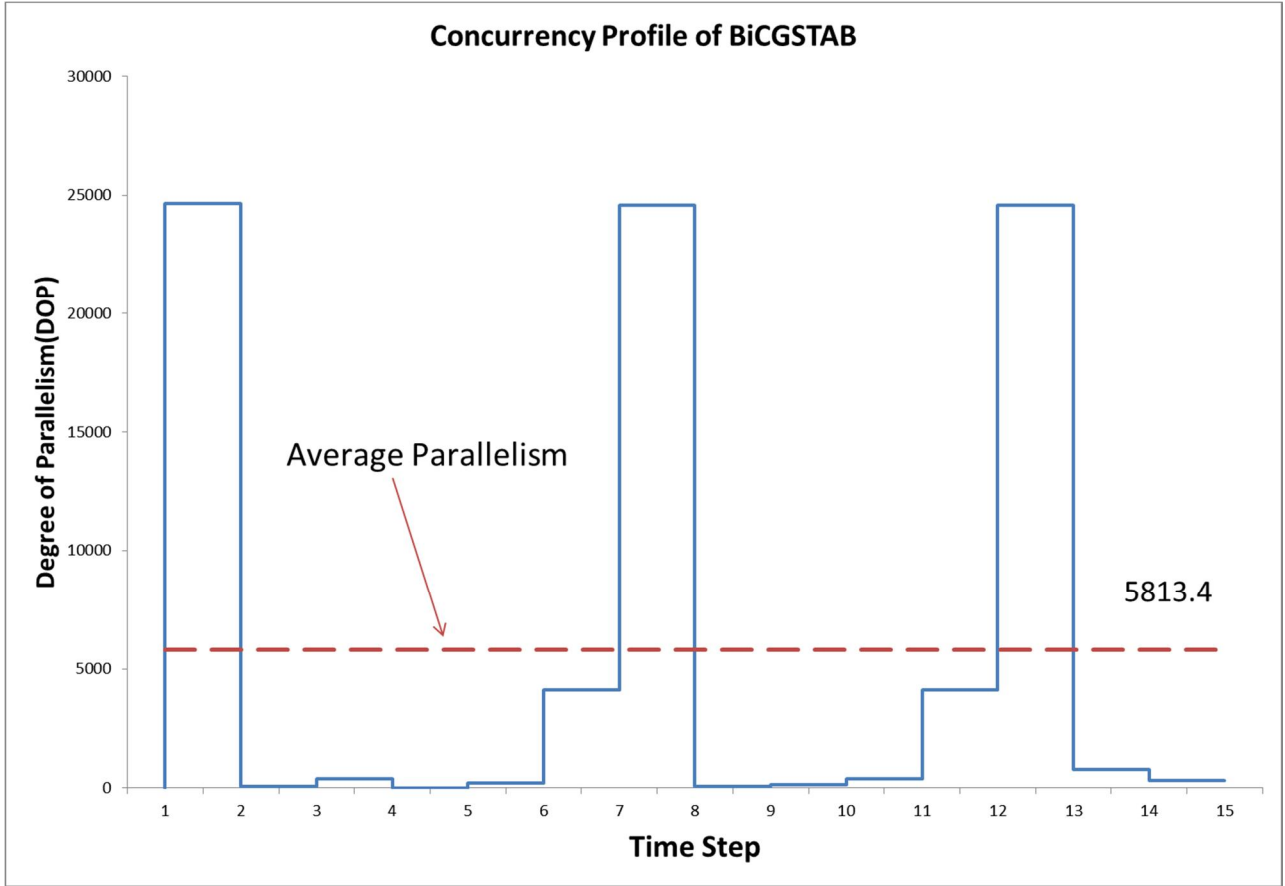


Figure 15: Concurrency Profile of BiCGSTAB with N=64

Next we attempt to further exploit the dependency graphs Figure 12 and Figure 13 as well as the constructed concurrency profiles Figure 14 and Figure 15, for comparing the QMR and BiCGSTAB Algorithms in terms of their estimated parallel cost. We start by identifying the span of each algorithm²⁵, Figure 16 and Figure 17. We then associate every link in the path with a cost function based on the perspective required steps to complete its operations in parallel when assuming Infinite Resources, as was demonstrated in Table 3. To summarize that in numbers, Table 8 shows the quantification of the estimated parallel cost when a matrix of leading dimension $N = 1024$, while Figure 18, plots the acquired results.

Table 8: Estimated parallel cost based on the perspective required steps to complete its operations in parallel when assuming Infinite Resources

Algorithm	Estimated required steps to complete operations in parallel for the SPAN	Summary of Parallel Cost	Estimated Parallel Cost for N = 1024
QMR	$LOG(N) + 1 + N + LOG(N)$ $+1 + N + 1 + LOG(N)$ $+ LOG(N) + 1 + 1 + N$ $+ LOG(N) + 1 + 1 + 1 + 1 + 1$	$5Log(N) + 3N + 10$	3132
BiCGSTAB	$LOG(N) + 1 + LOG(N) + 1 + 1$ $+N + LOG(N) + LOG(N) + 1$ $+LOG(N) + N + LOG(N)$ $+ 2LOG(N) + 1$	$8Log(N) + 2N + 4$	2132

²⁵ The span: is the longest serial path of the algorithm

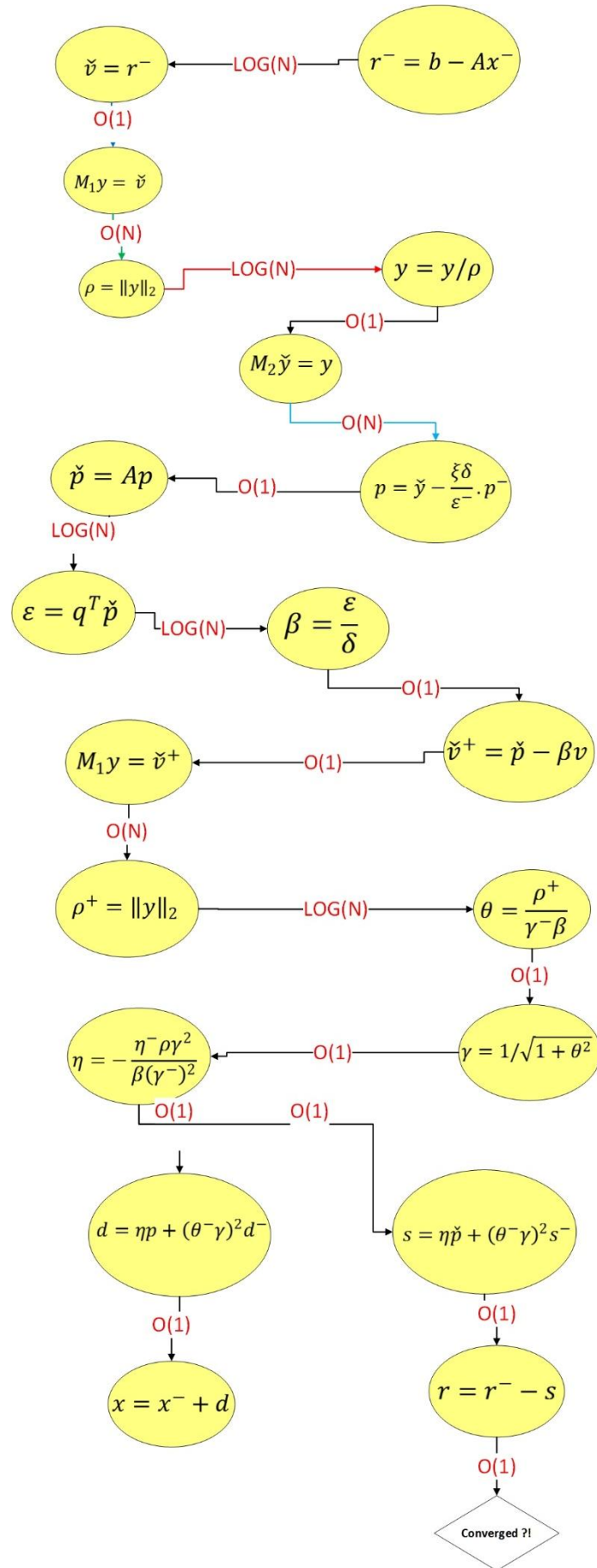


Figure 16: QMR Span

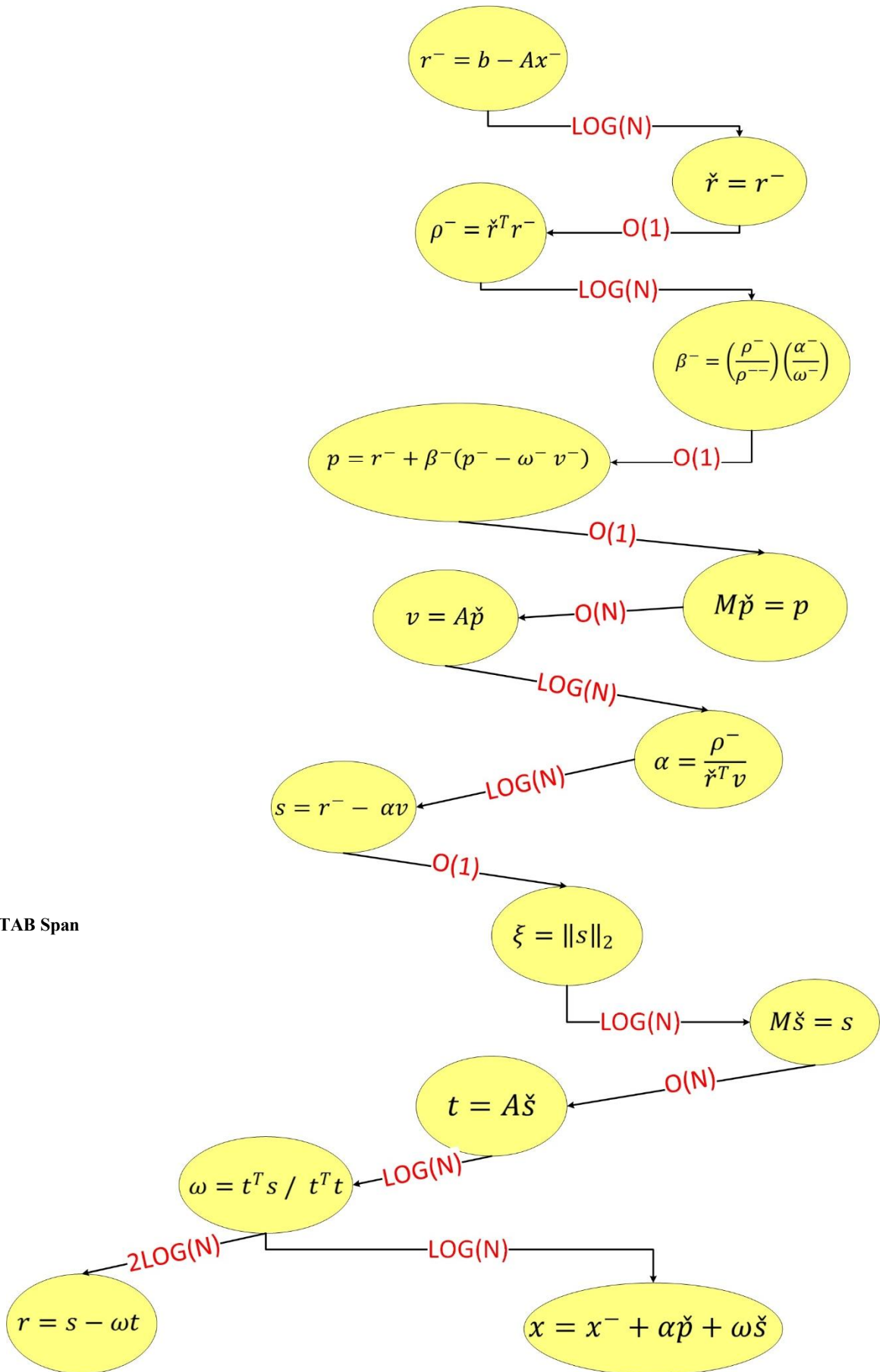


Figure 17: BiCGSTAB Span

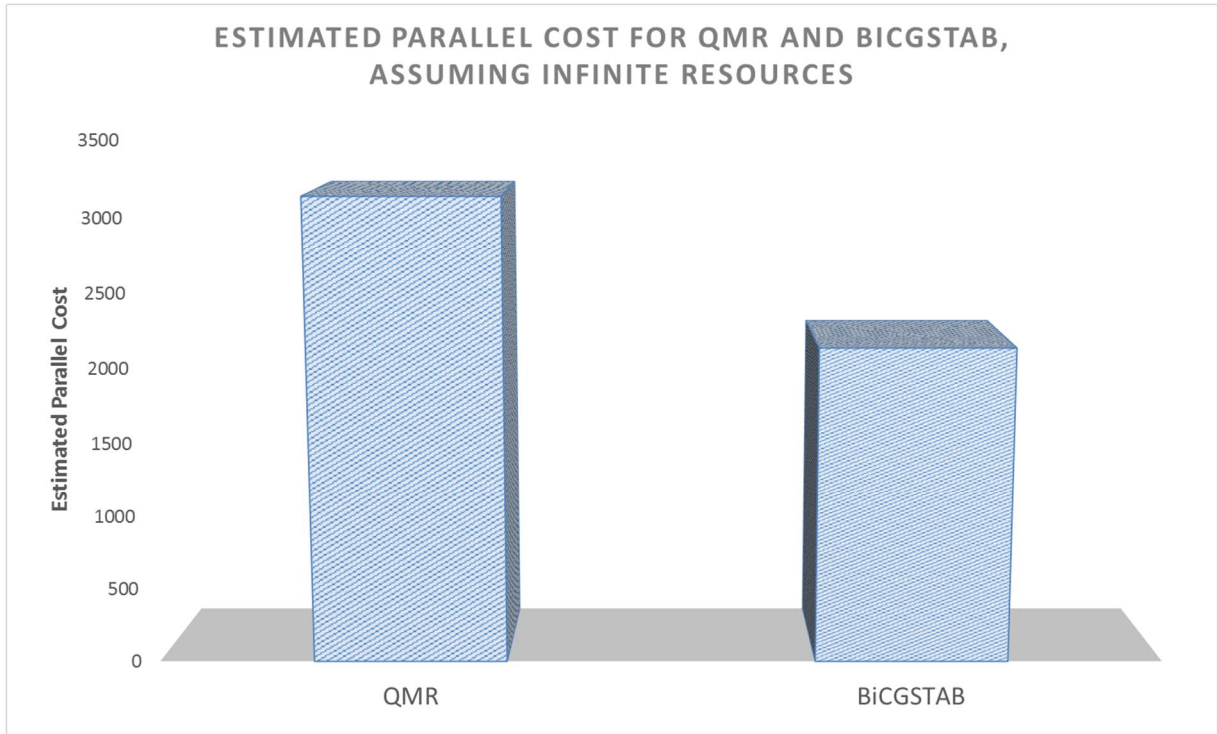


Figure 18: A comparison between the estimated parallel cost based on the perspective required steps for QMR and BiCGSTAB Algorithms, with matrix leading dimension $N = 1024$. The smaller the parallel cost, the better.

To enable a more compact realization of the parallel cost and the associated algorithmic complexity, we introduce a new universal graphical abstraction model which conserves data dependency. When applied to Krylov subspace methods and without loss of generality, the method takes the following steps:

1. Construct a data dependency graph for the selected algorithm (DDG).
2. The related tasks are grouped into functions that can be efficiently invoked from optimized library. For example, Intel Math Kernel Library [87] and CUBLAS [94]
3. Analyze the parallel complexity of each of the grouped tasks. The work flow of Krylov solvers consists of known building blocks detailed Table 3
4. Associate each edge in the dependency graph with a weight equals the estimated parallel cost, Table 3
5. Extract the span, the longest path of the algorithm.
6. Construct the Abstract Parallel Complexity Graph (APCG) by converting every node in the span to an abstract node (AN) represented by a box whose width is proportional to the estimated parallel cost in step 3. The order of the operations should remain preserved.

The construction of the Abstract Parallel Complexity Graph (APCG) is mainly based on functional decomposition with its associated temporal dependency, which in turns yields limited scalability according to Amdahl's Law. Nevertheless, that decomposition and the resulted (APCG) is not only a primary step in the analysis procedure that guides implementing an optimized parallel code, but also servers the following advantages:

1. Presents a global abstract visual analytical way for comparing various parallel algorithms.
2. Structuring functional parallelism to better make use of parallel design patterns to support automatic parallelism.
3. Shedding light on the limitations associated with certain algorithms and their inevitable serial behavior.
4. Optimizing parallel programs by pipelining the consecutive task groups, discovering common patterns and controlling the granularity levels by single or hierarchical merge of two or more nodes. Useful parallel patterns could be found in [6, 9]

Figure 19, shows the constructed graph for both BiCGSTAB and QMR. General speaking, the parallel computational work inside any algorithm is achieved by either a single thread or many cooperating threads. The associated presented earlier complexities $[O(n), O(\text{Log}(n)), O(1)]$ will depend on the operational context throughout the program execution. We will call the flow of data from operations requiring $O(1)$ to another $O(1)$ a linear operation. A scatter, one to many, operation takes place from tasks requiring $O(1)$ followed by either $O(\text{Log}(n))$ or $O(n)$ operations. The opposite is a reduction, many to one, operation. The final combination is the broadcast, many to many, that mimics the flow of data from operations with $O(\text{Log}(n))$ or $O(n)$ to other operations of either $O(\text{Log}(n))$ or $O(n)$. An example of a linear operation is vector update followed by vector scaling. One optimization is to merge such operations together so that they are performed by the same worker. Other examples could be constructed in the same way.

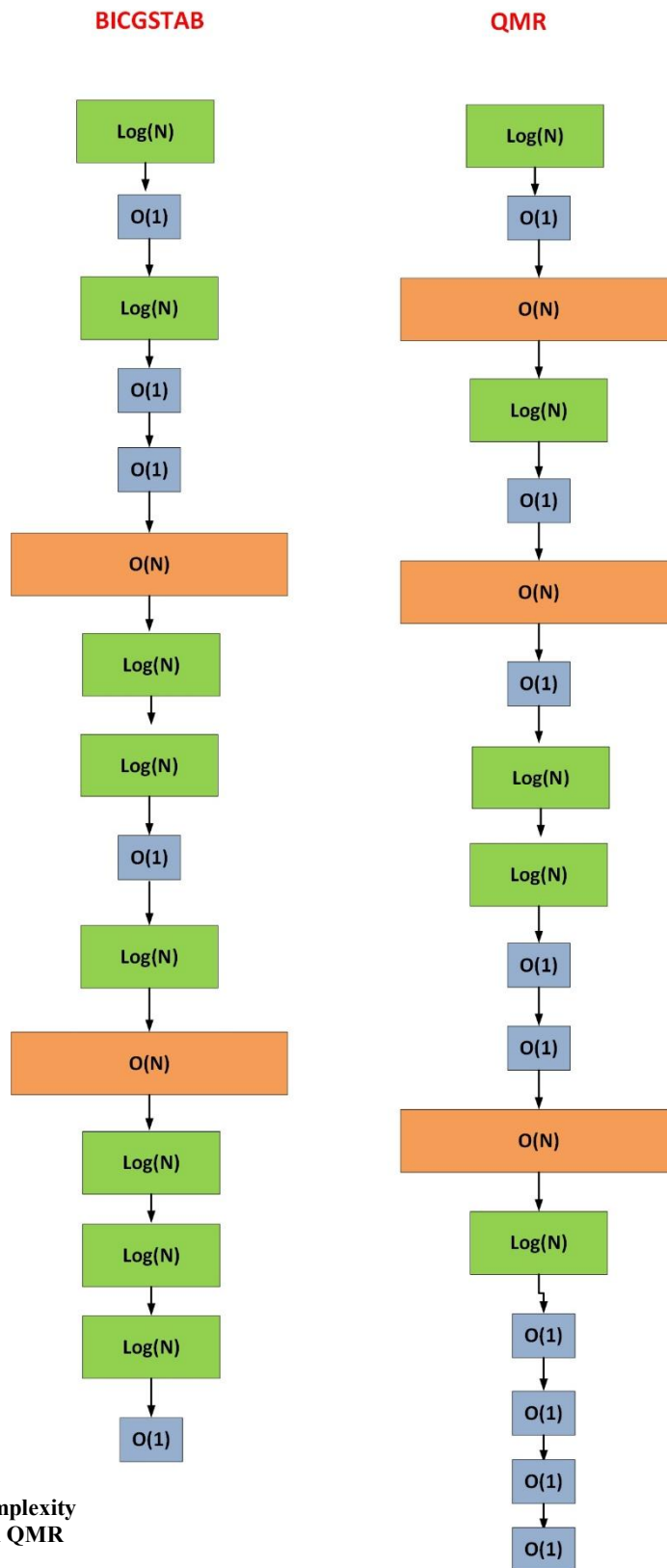


Figure 19: The Abstract Parallel Complexity Graph (APCG) for BiCGSTAB and QMR

For example, referring to Figure 19, the possibilities of two extracted consecutive patterns are presented in Table 9, along with their associated interpretations. Similarly, other n-way patterns could also be constructed by grouping three or more consecutive patterns. The first combinations of a tri-pattern are in Table 10.

Table 9: Two consecutive patterns possibilities for tasks representing Krylov Solvers

$O(1)$ $\text{Log}(N)$	one to many (scatter)
$\text{Log}(N)$ $O(1)$	Many to one (gather)
$O(N)$ $O(1)$	Many to one
$O(1)$ $O(N)$	One to Many
$O(N)$ $\text{Log}(N)$	Many to many (broadcast)
$\text{Log}(N)$ $O(N)$	Many to many (broadcast)
$O(N)$ $O(N)$	Many to many (broadcast)

Table 10: Three consecutive patterns possibilities for tasks representing Krylov Solvers

$O(1)$ $O(1)$ $O(1)$	Linear
$O(1)$ $\text{Log}(N)$ $O(1)$	One-many-one (scatter-gather)
$O(1)$ $O(1)$ $\text{Log}(N)$	One-one-many (linear-scatter)
$O(1)$ $O(1)$ $O(N)$	One-one-many (linear-scatter)
$O(1)$ $O(N)$ $O(1)$	One-many-one (scatter-gather)
$O(1)$ $O(1)$ $O(N)$	One-one-many (linear-scatter)
$O(1)$ $\text{Log}(N)$ $O(N)$	One-many-many (scatter-broadcast)
$O(1)$ $O(N)$ $\text{Log}(N)$	One-many-many (scatter-broadcast)
$O(1)$ $\text{Log}(N)$ $O(N)$	One-many-many (scatter-broadcast)

As the span, the sequential path, of BiCGSTAB is shorter than the span of QMR, we may expect BiCGSTAB to scale better than QMR. Nevertheless, this is somehow a relaxed conclusion as lot of other factors may take place, one of which was discussed above about the combination of some operations to create a shorter path. For instance, if we merged all tasks taking $O(1)$ with either its predecessor or successor, then QMR will have a shorter span than BiCGSTAB!

Based on the above discussion, we decide to select BiCGSTAB, as the chosen solver to be parallelized and incorporated in our parallel reservoir simulator. A support to this choice will be further verified via experimentations.

3.3 Experimental Parallel Linear Solver Selection

Objectives:

- Examining how the parallel execution time of various already implemented parallel iterative linear solvers in CUSP library is affected with various sparse storage mechanisms.
- To get an initial insight about the solver that suits our developed reservoir simulator.

Experimental Setup and Conditions:

- Five large matrix samples at different time iterations of the forward reservoir simulator have been extracted and their condition number was measured Figure 9.
- Each sample represents a 3-D structured grid with (2×2) block entries distributed in a Hepta-diagonal fashion as resulted from finite volume discretization.
- Matrix representing Sample_0 is assembled at the first time iteration of the simulator, and its coefficients are a combination of various reservoir parameters (permeability, compressibility ...), oil pressure values P_o and water saturation levels S_w .

- As the simulation time proceeds, elements composing the coefficient matrix changes as both P_o and S_w get updated and other samples are extracted.
- Tests were performed on a node in an HPC cluster offered by the Information Technology Center at KFUPM featuring a Xeon E5-2680 10-Core, 2.8 GHz (Dual-processor) and Tesla k20x GPU [103], Table 12 . A Comparison of different compute capabilities for GPU Architecture is presented in [103].

Table 11: Condition number for various samples of the reservoir simulator

Matrix Dimension	[120,000 x 120,000]
Avg. Number of Non-Zeros	[1512800]
Sampling Time	Condition Number
0	1.279E+05
31	1.112E+06
62	1.873E+06
93	3.548E+06
124	4.708E+06

Table 12: TESLA K20X GPU ACCELERATOR²⁶

Specifications	Tesla K20X
Generic SKU reference	699-22081-0200-xxx
Chip	GK110
Package size GPU	45 mm × 45 mm 2397-pin S-FCBGA
Processor clock	732 MHz
Memory clock	2.6 GHz
Memory size	6 GB
Memory I/O	384-bit GDDR5
Memory configuration	24 pieces of 64M × 16 GDDR5 SDRAM
Display connectors	None
Power connectors	<ul style="list-style-type: none"> • 8-pin PCI Express power connector • 6-pin PCI Express power connector
Board power	235 W
Idle power	25 W
Thermal cooling solution	Passive heat sink

Table 13: Comparison of different compute capabilities for GPU Architecture²⁷

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2 ¹⁶ -1	2 ¹⁶ -1	2 ³² -1	2 ³² -1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

²⁶ <http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf>

²⁷ <http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf>

Method:

- We start first by examining how the parallel execution time of different solvers is behaving with different sparse storage mechanisms.
- We use the CUSP implemented CUDA versions of the solvers. Each matrix in Table 11, and with a given storage format were tested across various restarted versions of GMRES (5, 15, 50, 1000) and BiCGSTAB.
- To speed up convergence, we made use of the available Bridson approximate inverse preconditioner that reduces the fill-in and improves convergence via reordering elements in coefficient matrix [104].
- Each experiment was repeated ten times and the average as well as some statistics were reported, Figure 20 to Figure 24. T1 to T10 represents the recorded time for every experiment.
- Each of the previous samples was examined using four available different sparse storage mechanisms: Compressed Row Storage (CSR), ELLPACK (ELL), Hybrid (HYB), and Coordinate Format (COO).

Results and Discussion

Figure 25 to Figure 30 show the results of plotting the execution time for different matrix storage schemes at different samples drawn from our simulator and for the two mentioned preconditioned iterative linear solvers. Every Sample plot is accompanied with another semi-log plot that shows the relative residual per-iteration with a minimum²⁸ tolerance

²⁸ It may also reach 1e-7 or 1e-8 depending on the matrix sample

value of $1e - 6$. Let i be the iteration number, and the residual $Res = \|r - Ax\|_2$, then the relative-residual is calculated as $Rel_{Res} = \log_{10}(res(i) / res(0))$. The following are observed and concluded:

- Solver convergence is independent of the utilized storage scheme. However, the solver execution time is.
- Even for the same matrix structure but with different data values, it is difficult to specify an absolute storage scheme that gives the best performance time. For example, in Sample_0 and for all solvers, COO outperforms HYB. This is not the case for $GMRES(5)$ in Sample_93. This could be attributed to the utilized preconditioner that approximate the inverse by exploiting the reordering property to minimize the fill in [93] [104] [105].
- As time step in our reservoir advances, more iterations would be needed for reaching an accepted convergence level. This is clearly seen in the relative error plot as it is steeper in early reservoir samples. Compare for instance the relative error in Sample_0 and Sample 93. The previous behavior is due to an increase in the condition number of the assembled system as the time advances; the thing that in turns require more iteration to converge.
- A proper restarted version of $GMRES(m)$ may be shown to outperform BiCGSTAB for different storage formats. However, automatic identification of an optimal restart value is not possible. Moreover, and although $GMRES(m)$ enjoys a smoother convergence behavior shown in the relative residual plot, it demands lot of storage space.

Condition Number		1.279E+05		Matrix_Sample		0		Conf. Coeff: 1.96											
Solver	Sparsity	Parallel Execution Time										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
		T1	T2	T3	T4	T5	T6	T7	T8	T9	T10								
GMRES(5)	HYP	1.75	1.73	1.74	1.73	1.77	1.74	1.77	1.74	1.77	1.74	1.748	0.016	0.010	1.758	1.738	1.770	1.730	0.040
	ELL	1.74	1.73	1.77	1.75	1.77	1.74	1.76	1.73	1.74	1.73	1.746	0.016	0.010	1.756	1.736	1.770	1.730	0.040
	CSR	1.63	1.62	1.7	1.62	1.6	1.59	1.59	1.62	1.59	1.59	1.615	0.034	0.021	1.636	1.594	1.700	1.590	0.110
	COO	1.69	1.69	1.73	1.69	1.74	1.69	1.75	1.68	1.75	1.72	1.713	0.028	0.017	1.730	1.696	1.750	1.680	0.070
GMRES(15)	HYP	1.83	1.79	1.73	1.72	1.72	1.72	1.72	1.74	1.71	1.74	1.742	0.038	0.024	1.766	1.718	1.830	1.710	0.120
	ELL	1.75	1.73	1.72	1.74	1.76	1.75	1.75	1.75	1.75	1.74	1.744	0.012	0.007	1.751	1.737	1.760	1.720	0.040
	CSR	1.64	1.62	1.63	1.62	1.65	1.64	1.63	1.63	1.62	1.62	1.630	0.011	0.007	1.637	1.623	1.650	1.620	0.030
	COO	1.67	1.71	1.68	1.72	1.68	1.68	1.67	1.66	1.63	1.66	1.676	0.025	0.016	1.692	1.660	1.720	1.630	0.090
GMRES(50)	HYP	1.8	1.77	1.75	1.75	1.77	1.76	1.74	1.76	1.78	1.78	1.766	0.018	0.011	1.777	1.755	1.800	1.740	0.060
	ELL	1.77	1.76	1.76	1.77	1.75	1.76	1.78	1.8	1.75	1.75	1.765	0.016	0.010	1.775	1.755	1.800	1.750	0.050
	CSR	1.65	1.69	1.65	1.66	1.65	1.66	1.64	1.65	1.7	1.66	1.661	0.019	0.012	1.673	1.649	1.700	1.640	0.060
	COO	1.75	1.69	1.67	1.69	1.68	1.67	1.68	1.72	1.7	1.76	1.701	0.032	0.020	1.721	1.681	1.760	1.670	0.090
GMRES(1000)	HYP	1.94	1.9	1.88	1.9	1.94	1.89	1.89	1.94	1.89	1.89	1.906	0.024	0.015	1.921	1.891	1.940	1.880	0.060
	ELL	1.89	1.91	1.87	1.88	1.92	1.88	1.91	1.88	1.89	1.88	1.891	0.017	0.010	1.901	1.881	1.920	1.870	0.050
	CSR	1.77	1.78	1.77	1.78	1.8	1.77	1.78	1.79	1.77	1.79	1.780	0.011	0.007	1.787	1.773	1.800	1.770	0.030
	COO	1.81	1.82	1.81	1.85	1.82	1.84	1.82	1.84	1.8	1.81	1.822	0.016	0.010	1.832	1.812	1.850	1.800	0.050
BiCGSTAB	HYP	1.87	1.81	1.81	1.85	1.81	1.86	1.81	1.8	1.85	1.81	1.828	0.026	0.016	1.844	1.812	1.870	1.800	0.070
	ELL	1.78	1.76	1.78	1.7	1.71	1.75	1.71	1.73	1.72	1.71	1.735	0.030	0.019	1.754	1.716	1.780	1.700	0.080
	CSR	1.69	1.64	1.72	1.7	1.66	1.66	1.67	1.71	1.67	1.71	1.683	0.027	0.017	1.700	1.666	1.720	1.640	0.080
	COO	1.73	1.73	1.73	1.72	1.73	1.77	1.74	1.84	1.74	1.76	1.749	0.035	0.022	1.771	1.727	1.840	1.720	0.120

Figure 20: Average Parallel Execution Times for Sample_0

Condition Number		1.112E+06		Matrix_Sample		31		Conf. Coeff: 1.96											
Solver	Sparsity	Parallel Execution Time										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
		T1	T2	T3	T4	T5	T6	T7	T8	T9	T10								
GMRES(5)	HYP	2.98	3.03	3.08	2.99	3.03	2.97	3	3.06	3.06	3.08	3.028	0.041	0.026	3.054	3.002	3.080	2.970	0.110
	ELL	2.75	2.75	2.72	2.84	2.73	2.79	2.7	2.77	2.71	2.8	2.756	0.044	0.027	2.783	2.729	2.840	2.700	0.140
	CSR	2.72	2.62	2.77	2.68	2.62	2.79	2.68	2.74	2.65	2.81	2.708	0.069	0.043	2.751	2.665	2.810	2.620	0.190
	COO	2.99	2.95	2.98	3.05	3.08	2.95	2.98	3	3.06	3.08	3.012	0.051	0.032	3.044	2.980	3.080	2.950	0.130
GMRES(15)	HYP	2.54	2.64	2.59	2.53	2.63	2.62	2.54	2.53	2.53	2.59	2.574	0.045	0.028	2.602	2.546	2.640	2.530	0.110
	ELL	2.49	2.41	2.41	2.48	2.43	2.54	2.49	2.5	2.47	2.56	2.478	0.051	0.031	2.509	2.447	2.560	2.410	0.150
	CSR	2.46	2.4	2.37	2.38	2.36	2.36	2.47	2.39	2.37	2.36	2.392	0.041	0.025	2.417	2.367	2.470	2.360	0.110
	COO	2.51	2.7	2.54	2.68	2.58	2.54	2.53	2.6	2.49	2.49	2.566	0.074	0.046	2.612	2.520	2.700	2.490	0.210
GMRES(50)	HYP	2.68	2.61	2.62	2.6	2.65	2.61	2.6	2.62	2.6	2.68	2.627	0.032	0.020	2.647	2.607	2.680	2.600	0.080
	ELL	2.54	2.55	2.59	2.57	2.54	2.6	2.55	2.59	2.57	2.5	2.560	0.030	0.019	2.579	2.541	2.600	2.500	0.100
	CSR	2.51	2.45	2.5	2.49	2.44	2.44	2.49	2.44	2.45	2.43	2.464	0.030	0.019	2.483	2.445	2.510	2.430	0.080
	COO	2.56	2.68	2.61	2.58	2.63	2.59	2.58	2.63	2.59	2.59	2.604	0.035	0.022	2.626	2.582	2.680	2.560	0.120
GMRES(1000)	HYP	4.58	4.6	4.63	4.62	4.61	4.62	4.58	4.62	4.62	4.6	4.608	0.018	0.011	4.619	4.597	4.630	4.580	0.050
	ELL	4.5	4.55	4.55	4.51	4.53	4.5	4.52	4.52	4.52	4.56	4.526	0.021	0.013	4.539	4.513	4.560	4.500	0.060
	CSR	4.43	4.48	4.42	4.42	4.41	4.44	4.42	4.43	4.47	4.44	4.436	0.023	0.014	4.450	4.422	4.480	4.410	0.070
	COO	4.52	4.51	4.46	4.54	4.55	4.52	4.5	4.55	4.57	4.55	4.527	0.032	0.020	4.547	4.507	4.570	4.460	0.110
BiCGSTAB	HYP	2.58	2.5	2.68	2.52	2.51	2.62	2.51	2.51	2.51	2.6	2.554	0.062	0.039	2.593	2.515	2.680	2.500	0.180
	ELL	2.45	2.45	2.49	2.48	2.48	2.46	2.44	2.44	2.45	2.44	2.458	0.019	0.012	2.470	2.446	2.490	2.440	0.050
	CSR	2.4	2.4	2.37	2.37	2.36	2.48	2.49	2.43	2.4	2.39	2.409	0.045	0.028	2.437	2.381	2.490	2.360	0.130
	COO	2.62	2.56	2.67	2.6	2.52	2.53	2.59	2.52	2.51	2.53	2.565	0.053	0.033	2.598	2.532	2.670	2.510	0.160

Figure 21: Average Parallel Execution Times for Sample_31

Condition Number		1.873E+06		Matrix_Sample		62		Conf. Coeff: 1.96											
Solver	Sparsity	Parallel Execution Time										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
		T1	T2	T3	T4	T5	T6	T7	T8	T9	T10								
GMRES(5)	HYP	3.57	3.43	3.46	3.55	3.47	3.44	3.57	3.49	3.44	3.58	3.500	0.061	0.038	3.538	3.462	3.580	3.430	0.150
	ELL	3.27	3.25	3.25	3.24	3.25	3.26	3.26	3.24	3.25	3.26	3.253	0.009	0.006	3.259	3.247	3.270	3.240	0.030
	CSR	3.08	3.08	3.11	3.08	3.13	3.17	3.17	3.2	3.19	3.08	3.129	0.050	0.031	3.160	3.098	3.200	3.080	0.120
	COO	3.49	3.53	3.62	3.52	3.5	3.64	3.48	3.67	3.64	3.49	3.558	0.075	0.047	3.605	3.511	3.670	3.480	0.190
GMRES(15)	HYP	2.97	3.09	2.9	2.92	2.88	2.98	2.92	2.97	2.89	2.99	2.951	0.063	0.039	2.990	2.912	3.090	2.880	0.210
	ELL	2.76	2.76	2.76	2.7	2.7	2.7	2.72	2.77	2.73	2.79	2.739	0.033	0.021	2.760	2.718	2.790	2.700	0.090
	CSR	2.73	2.66	2.68	2.68	2.74	2.65	2.7	2.69	2.61	2.71	2.685	0.039	0.024	2.709	2.661	2.740	2.610	0.130
	COO	2.88	2.82	2.85	2.8	2.86	2.79	2.86	2.96	2.85	2.93	2.860	0.053	0.033	2.893	2.827	2.960	2.790	0.170
GMRES(50)	HYP	2.96	2.98	3	3.04	2.95	2.97	3.05	3.08	3.04	3.08	3.015	0.049	0.030	3.045	2.985	3.080	2.950	0.130
	ELL	2.89	2.82	2.87	2.82	2.91	2.91	2.83	2.87	2.83	2.85	2.860	0.035	0.022	2.882	2.838	2.910	2.820	0.090
	CSR	2.8	2.85	2.8	2.86	2.78	2.8	2.78	2.79	2.81	2.73	2.800	0.037	0.023	2.823	2.777	2.860	2.730	0.130
	COO	2.96	2.99	2.99	3.03	2.93	2.94	2.99	3.04	2.92	2.95	2.974	0.041	0.025	2.999	2.949	3.040	2.920	0.120
GMRES(100)	HYP	5.93	5.93	5.93	5.95	5.99	5.99	5.96	5.99	6.06	5.95	5.968	0.041	0.025	5.993	5.943	6.060	5.930	0.130
	ELL	5.87	5.9	5.86	5.85	5.84	5.89	5.83	5.85	5.88	5.87	5.864	0.022	0.014	5.878	5.850	5.900	5.830	0.070
	CSR	5.77	5.75	5.79	5.84	5.75	5.8	5.77	5.77	5.74	5.73	5.771	0.032	0.020	5.791	5.751	5.840	5.730	0.110
	COO	5.92	5.89	5.85	5.87	5.87	5.86	5.86	5.88	5.88	5.86	5.874	0.020	0.012	5.886	5.862	5.920	5.850	0.070
BiCGSTAB	HYP	3	2.96	2.96	3.05	3.08	3.09	2.97	2.96	2.96	2.96	2.999	0.054	0.033	3.032	2.966	3.090	2.960	0.130
	ELL	2.75	2.88	2.89	2.77	2.76	2.82	2.88	2.83	2.71	2.78	2.807	0.063	0.039	2.846	2.768	2.890	2.710	0.180
	CSR	2.72	2.8	2.74	2.81	2.73	2.72	2.75	2.86	2.79	2.86	2.778	0.054	0.034	2.812	2.744	2.860	2.720	0.140
	COO	2.86	2.97	2.93	2.87	2.95	2.96	2.84	2.94	2.87	2.89	2.908	0.047	0.029	2.937	2.879	2.970	2.840	0.130

Figure 22: Average Parallel Execution Times for Sample_62

Condition Number		3.548E+06		Matrix_Sample		93		Conf. Coeff: 1.96											
Solver	Sparsity	Parallel Execution Time										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
		T1	T2	T3	T4	T5	T6	T7	T8	T9	T10								
GMRES(5)	HYP	4.27	4.25	4.23	4.37	4.37	4.4	4.39	4.33	4.27	4.34	4.322	0.062	0.039	4.361	4.283	4.400	4.230	0.170
	ELL	3.86	3.78	3.84	3.79	3.84	3.93	3.94	3.79	3.91	3.79	3.847	0.061	0.038	3.885	3.809	3.940	3.780	0.160
	CSR	3.89	3.77	3.86	3.75	3.86	3.79	3.87	3.72	3.84	3.72	3.807	0.065	0.040	3.847	3.767	3.890	3.720	0.170
	COO	4.42	4.37	4.37	4.37	4.55	4.45	4.47	4.36	4.38	4.47	4.421	0.063	0.039	4.460	4.382	4.550	4.360	0.190
GMRES(15)	HYP	3.56	3.64	3.57	3.61	3.55	3.65	3.6	3.53	3.71	3.59	3.601	0.054	0.033	3.634	3.568	3.710	3.530	0.180
	ELL	3.29	3.29	3.27	3.3	3.41	3.47	3.4	3.37	3.35	3.37	3.352	0.064	0.040	3.392	3.312	3.470	3.270	0.200
	CSR	3.2	3.22	3.23	3.26	3.4	3.23	3.24	3.31	3.32	3.34	3.275	0.064	0.040	3.315	3.235	3.400	3.200	0.200
	COO	3.59	3.62	3.6	3.55	3.63	3.64	3.73	3.65	3.6	3.66	3.627	0.049	0.030	3.657	3.597	3.730	3.550	0.180
GMRES(50)	HYP	3.66	3.59	3.68	3.59	3.53	3.56	3.51	3.58	3.6	3.57	3.587	0.052	0.032	3.619	3.555	3.680	3.510	0.170
	ELL	3.48	3.41	3.49	3.45	3.44	3.35	3.36	3.37	3.44	3.46	3.425	0.050	0.031	3.456	3.394	3.490	3.350	0.140
	CSR	3.35	3.32	3.32	3.41	3.32	3.37	3.34	3.33	3.32	3.33	3.341	0.029	0.018	3.359	3.323	3.410	3.320	0.090
	COO	3.57	3.54	3.64	3.57	3.56	3.62	3.55	3.54	3.59	3.53	3.571	0.036	0.022	3.593	3.549	3.640	3.530	0.110
GMRES(1000)	HYP	9.46	9.51	9.55	9.61	9.5	9.44	9.4	9.37	9.32	9.38	9.454	0.090	0.056	9.510	9.398	9.610	9.320	0.290
	ELL	9.31	9.36	9.22	9.27	9.3	9.28	9.37	9.38	9.31	9.26	9.306	0.052	0.032	9.338	9.274	9.380	9.220	0.160
	CSR	9.14	9.16	9.15	9.15	9.14	9.26	9.18	9.27	9.21	9.28	9.194	0.057	0.035	9.229	9.159	9.280	9.140	0.140
	COO	9.43	9.38	9.38	9.45	9.4	9.42	9.41	9.39	9.41	9.41	9.408	0.022	0.014	9.422	9.394	9.450	9.380	0.070
BiCGSTAB	HYP	3.46	3.4	3.56	3.47	3.43	3.34	3.37	3.56	3.46	3.41	3.446	0.073	0.045	3.491	3.401	3.560	3.340	0.220
	ELL	3.21	3.17	3.16	3.16	3.16	3.16	3.17	3.15	3.29	3.23	3.186	0.045	0.028	3.214	3.158	3.290	3.150	0.140
	CSR	3.09	3.13	3.14	3.15	3.17	3.14	3.15	3.15	3.23	3.2	3.155	0.038	0.024	3.179	3.131	3.230	3.090	0.140
	COO	3.27	3.31	3.36	3.34	3.31	3.31	3.34	3.46	3.31	3.4	3.341	0.055	0.034	3.375	3.307	3.460	3.270	0.190

Figure 23: Average Parallel Execution Times for Sample_93

Condition Number		4.708E+06		Matrix_Sample		124		Conf. Coeff: 1.96											
Solver	Sparsity	Parallel Execution Time										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
		T1	T2	T3	T4	T5	T6	T7	T8	T9	T10								
GMRES(5)	HYP	5.19	5.18	5.13	4.99	4.99	5.06	5.02	5.1	4.98	5.02	5.066	0.079	0.049	5.115	5.017	5.190	4.980	0.210
	ELL	4.4	4.41	4.41	4.53	4.47	4.41	4.57	4.46	4.42	4.59	4.467	0.072	0.044	4.511	4.423	4.590	4.400	0.190
	CSR	4.47	4.52	4.4	4.41	4.46	4.52	4.45	4.43	4.43	4.52	4.461	0.046	0.028	4.489	4.433	4.520	4.400	0.120
	COO	5.46	5.1	5.24	5.32	5.19	5.13	5.28	5.29	5.28	5.28	5.257	0.102	0.063	5.320	5.194	5.460	5.100	0.360
GMRES(15)	HYP	3.77	3.71	3.69	3.69	3.78	3.7	3.76	3.75	3.77	3.87	3.749	0.055	0.034	3.783	3.715	3.870	3.690	0.180
	ELL	3.55	3.45	3.45	3.54	3.5	3.47	3.48	3.69	3.62	3.63	3.538	0.084	0.052	3.590	3.486	3.690	3.450	0.240
	CSR	3.53	3.47	3.45	3.56	3.45	3.37	3.42	3.41	3.39	3.38	3.443	0.063	0.039	3.482	3.404	3.560	3.370	0.190
	COO	3.77	3.75	3.82	3.68	3.8	3.68	3.82	3.74	3.68	3.69	3.743	0.058	0.036	3.779	3.707	3.820	3.680	0.140
GMRES(50)	HYP	4.11	4.1	4.14	4.12	4.07	4.1	4.08	4.17	4.21	4.19	4.129	0.047	0.029	4.158	4.100	4.210	4.070	0.140
	ELL	4.02	3.9	3.92	3.84	3.88	3.91	3.93	3.96	4.01	3.9	3.927	0.056	0.035	3.962	3.892	4.020	3.840	0.180
	CSR	3.9	3.82	3.81	3.84	3.82	3.83	3.88	3.83	3.83	3.93	3.849	0.040	0.025	3.874	3.824	3.930	3.810	0.120
	COO	4.04	4.05	4.09	4.19	4.2	4.15	4.19	4.08	4.19	4.18	4.136	0.064	0.040	4.176	4.096	4.200	4.040	0.160
GMRES(1000)	HYP	12.8	13	13	13.1	12.8	12.8	12.7	12.8	12.9	13.1	12.888	0.138	0.086	12.974	12.802	13.100	12.730	0.370
	ELL	12.8	12.9	12.8	12.7	12.7	12.6	12.9	12.8	12.9	12.8	12.792	0.082	0.051	12.843	12.741	12.920	12.640	0.280
	CSR	12.7	12.9	12.7	12.6	12.6	12.7	12.6	12.8	12.7	12.8	12.711	0.090	0.056	12.767	12.655	12.890	12.600	0.290
	COO	12.7	12.8	12.8	13.1	12.8	12.7	12.8	12.7	12.7	12.8	12.782	0.107	0.066	12.848	12.716	13.060	12.690	0.370
BiCGSTAB	HYP	4.25	4.2	4.2	4.19	4.22	4.27	4.14	4.14	4.18	4.16	4.195	0.043	0.027	4.222	4.168	4.270	4.140	0.130
	ELL	3.54	3.69	3.69	3.54	3.6	3.64	3.56	3.49	3.62	3.53	3.590	0.069	0.043	3.633	3.547	3.690	3.490	0.200
	CSR	3.71	3.58	3.74	3.59	3.59	3.67	3.59	3.76	3.59	3.75	3.657	0.077	0.048	3.705	3.609	3.760	3.580	0.180
	COO	3.76	3.75	3.76	3.77	3.77	3.74	3.75	3.71	3.69	3.77	3.747	0.027	0.017	3.764	3.730	3.770	3.690	0.080

Figure 24: Average Parallel Execution Times for Sample_124

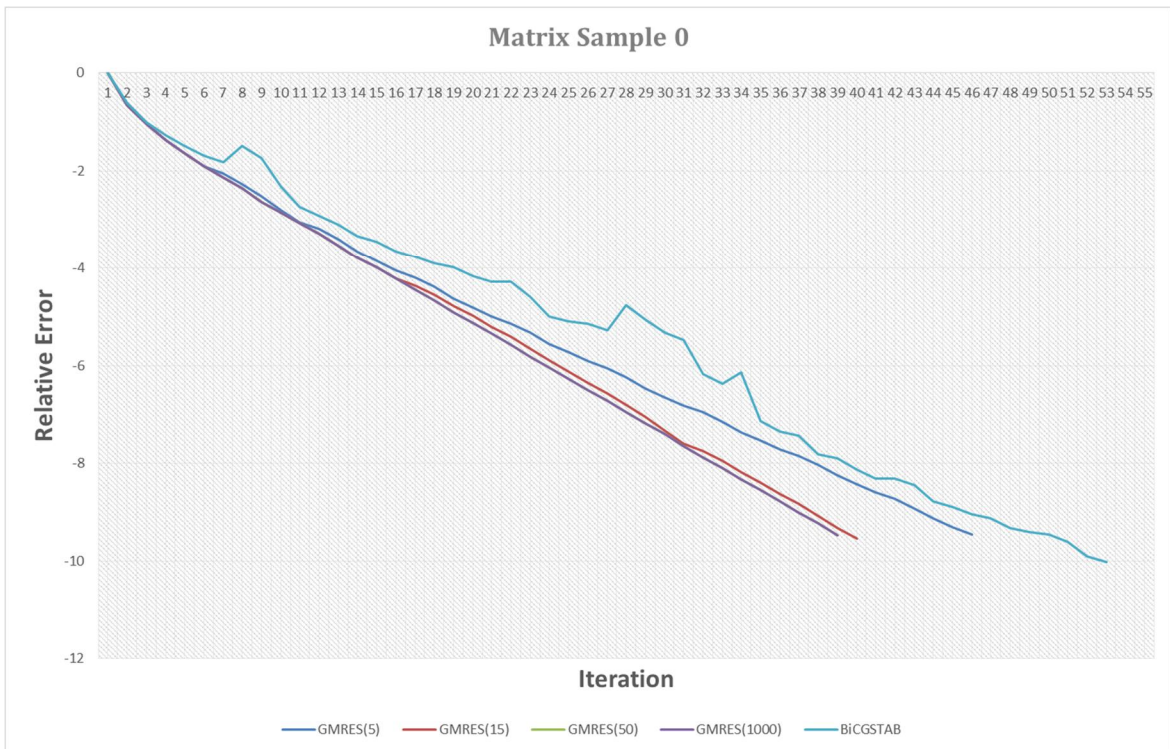
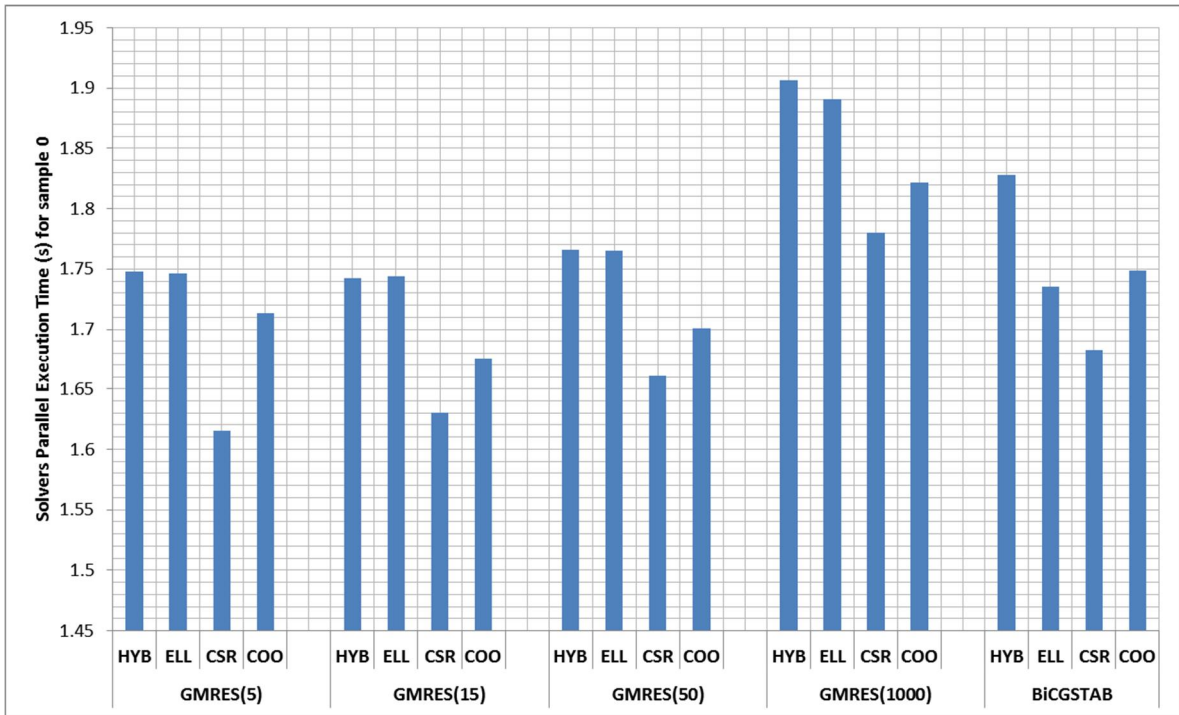


Figure 25: Solvers Parallel Execution time for various storage formats with relative residual semi-log plot. Sample_0. Convergence is independent of the utilized storage scheme

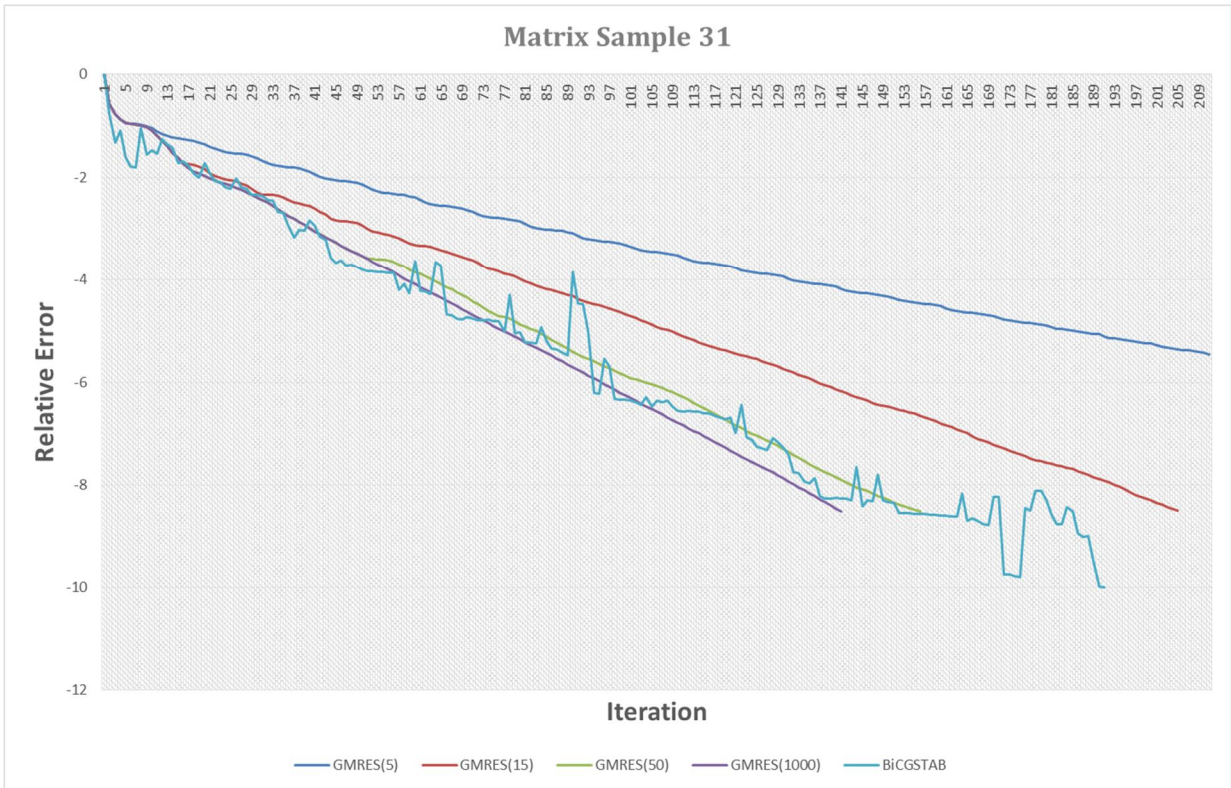
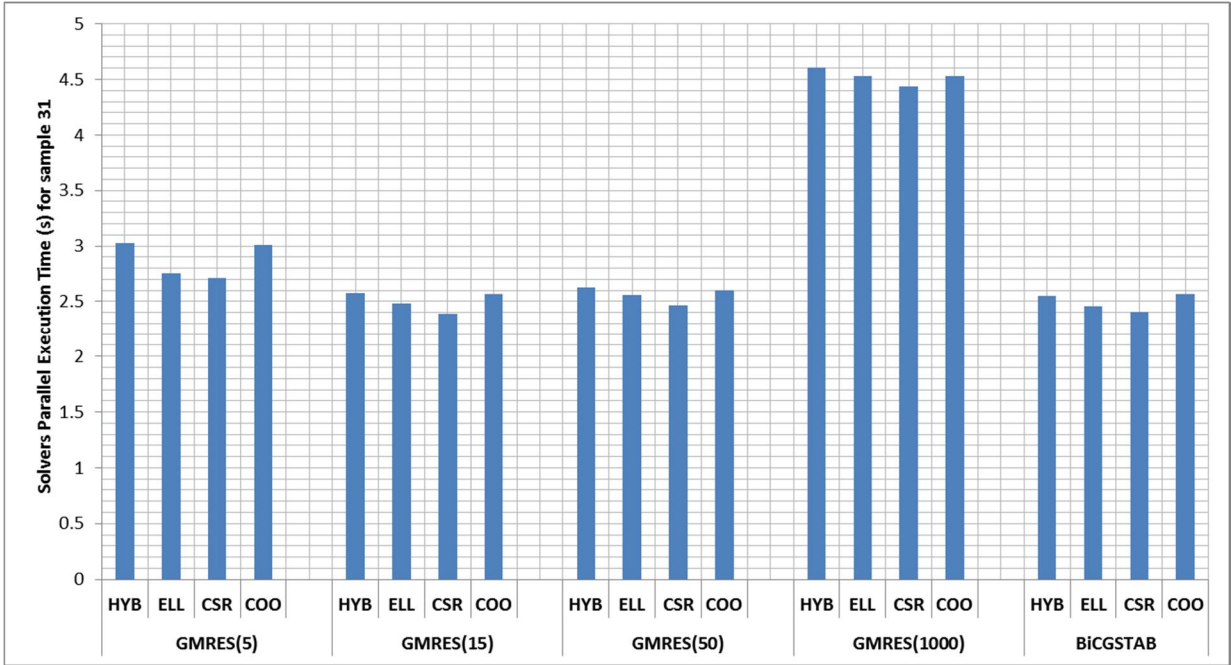


Figure 26: Solvers Parallel Execution time for various storage formats with relative residual semi-log plot. Sample_31. Convergence is independent of the utilized storage scheme

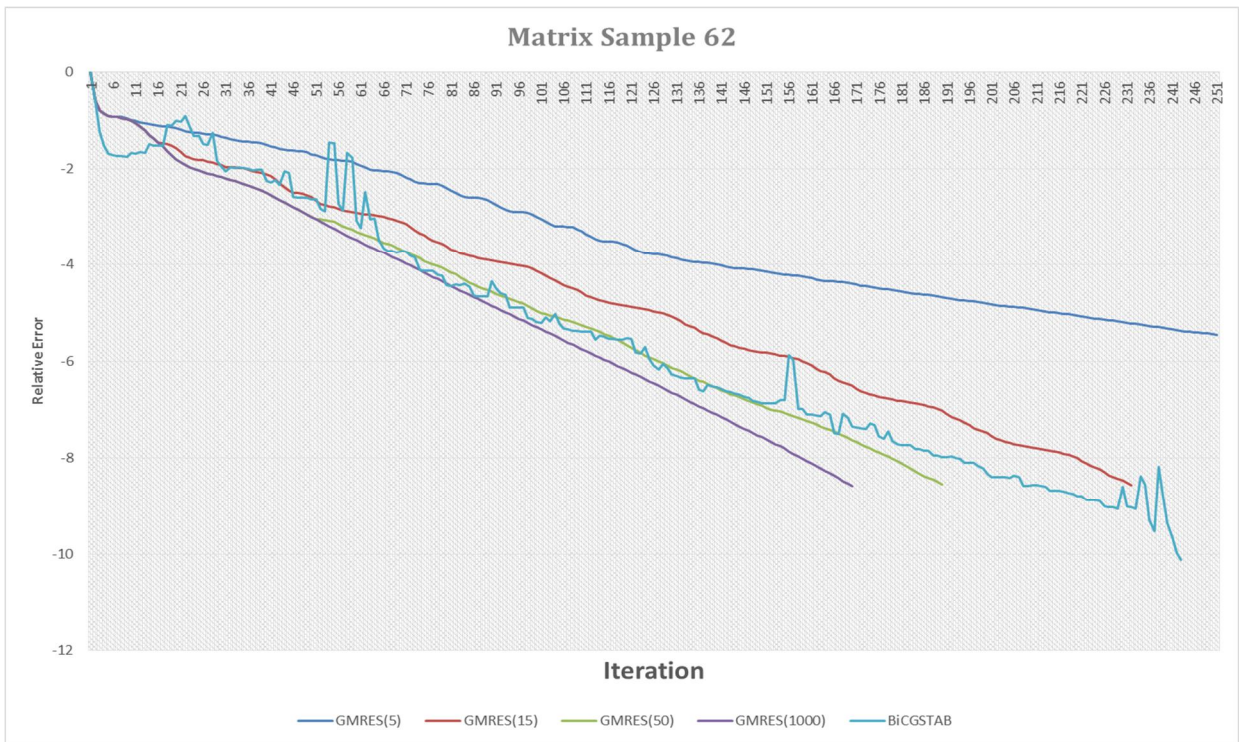
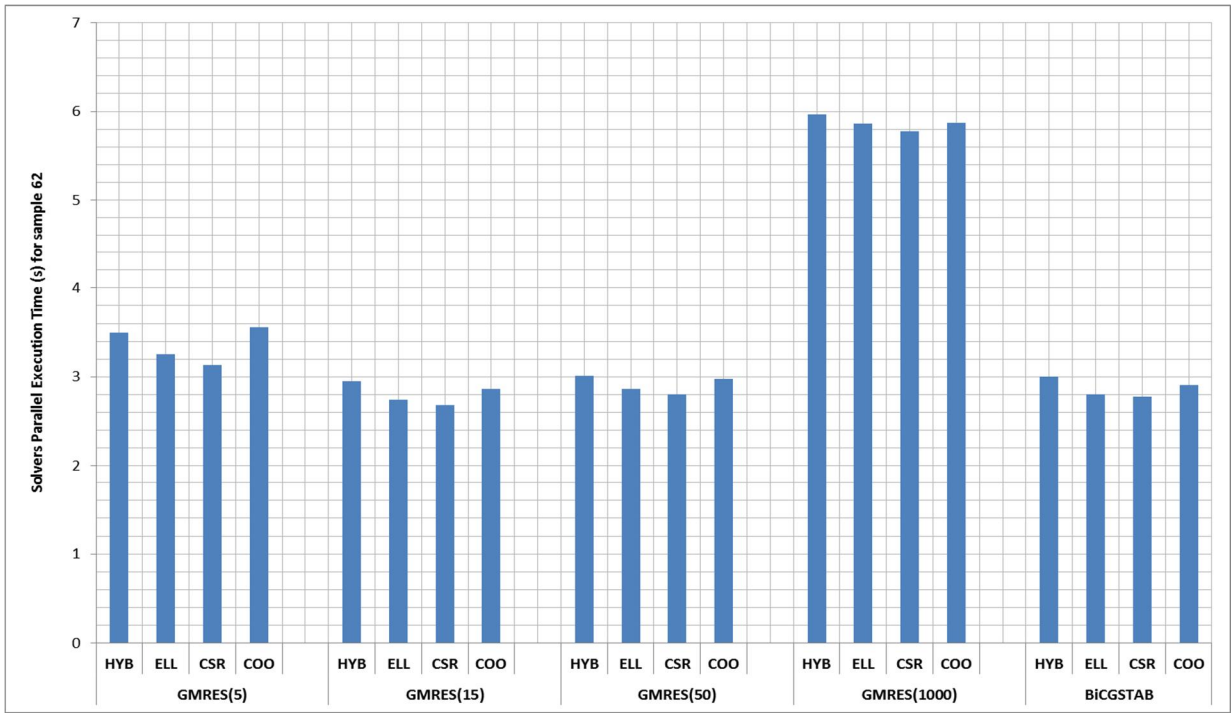


Figure 27: Solvers Parallel Execution time for various storage formats with relative residual semi-log plot. Sample_62. Convergence is independent of the utilized storage scheme.

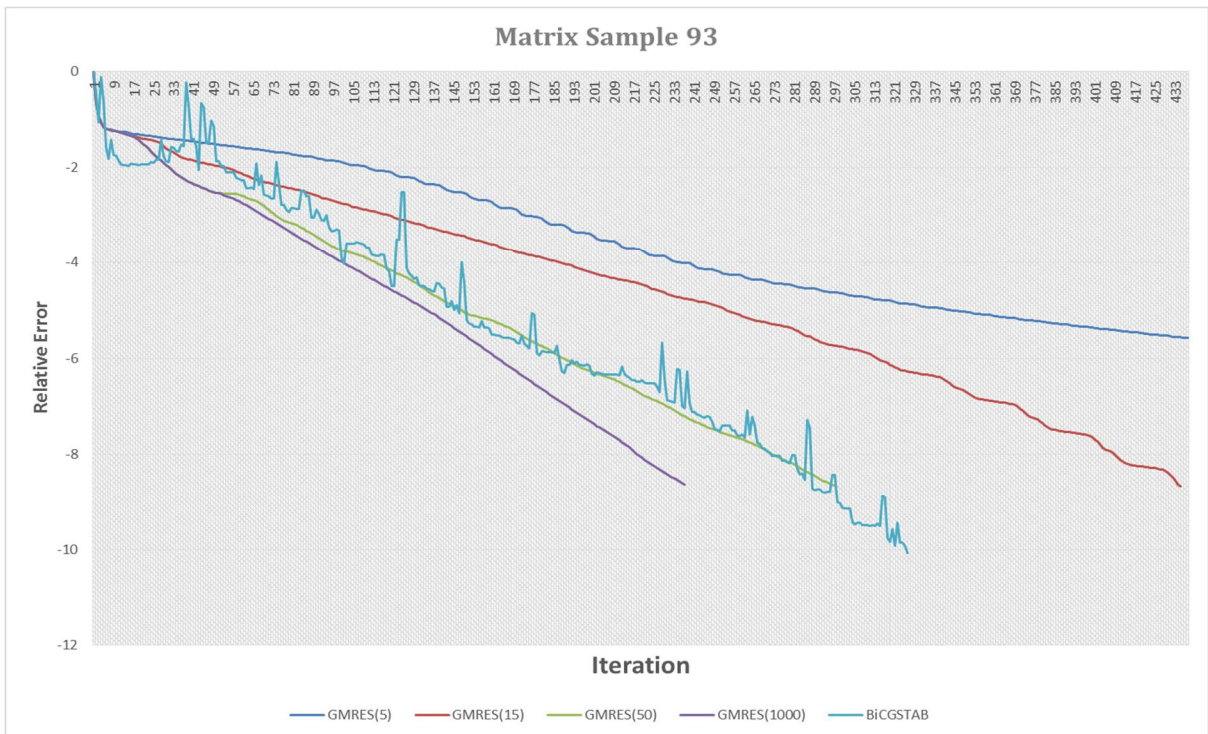
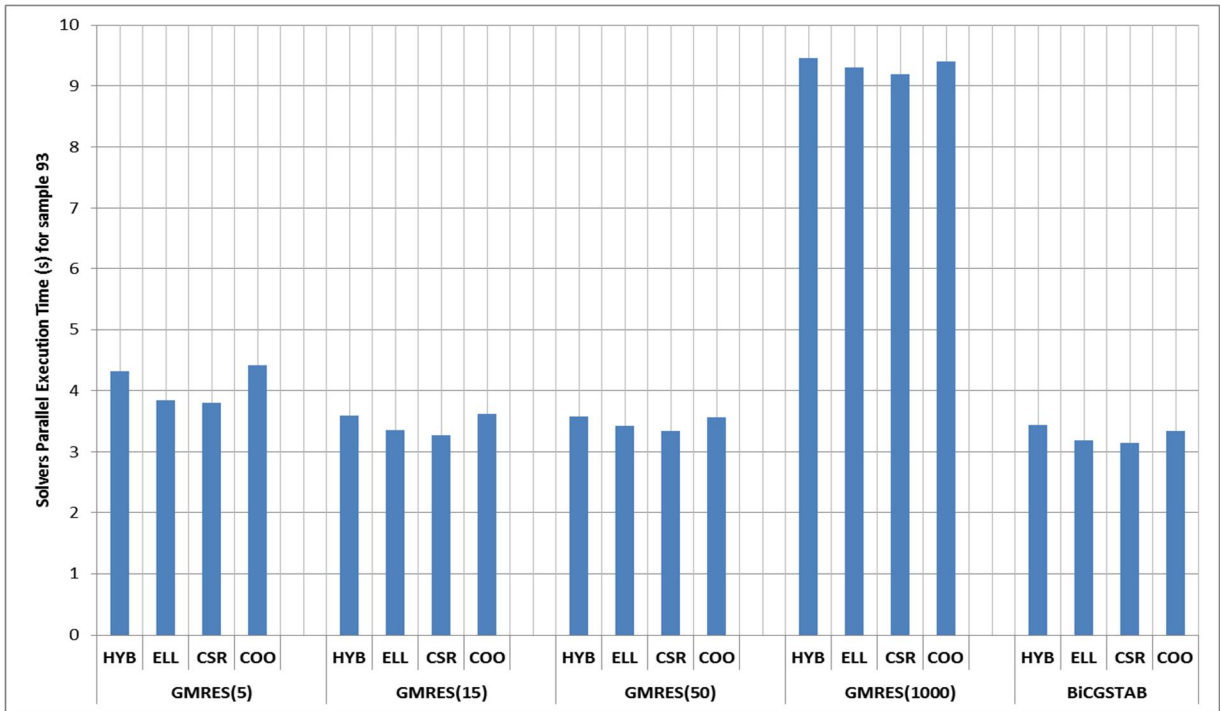


Figure 28: Solvers Parallel Execution time for various storage formats with relative residual semi-log plot. Sample_93. Convergence is independent of the utilized storage scheme.

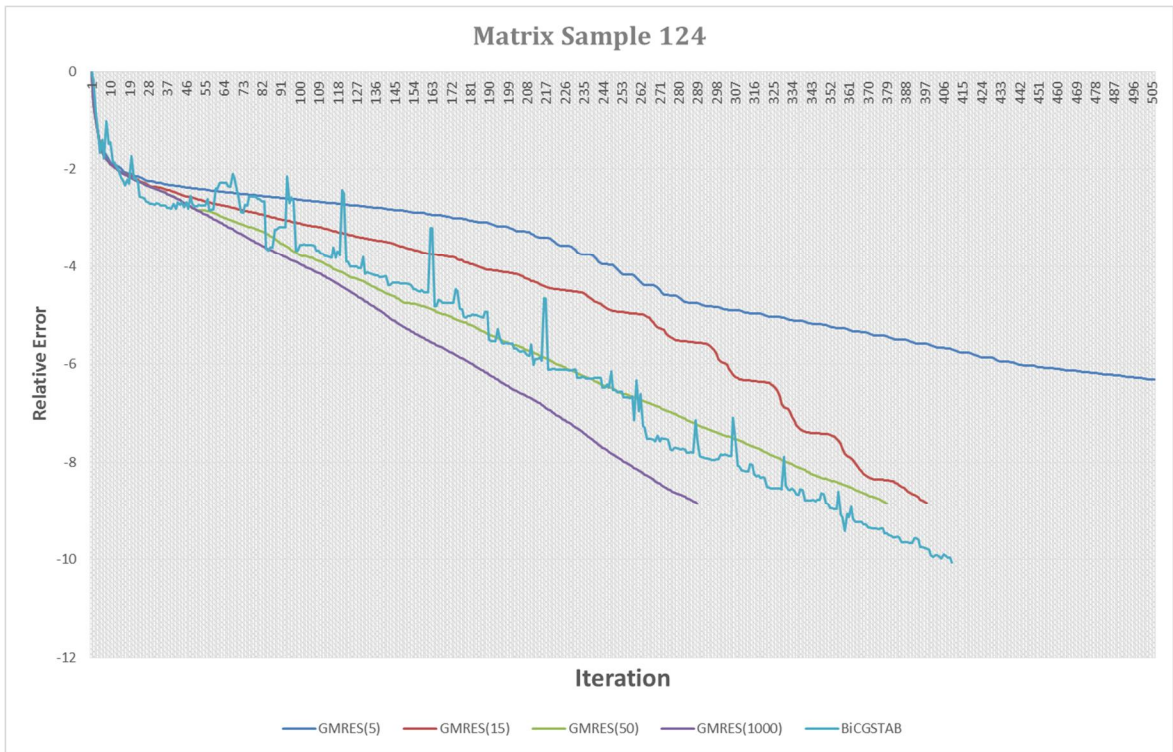
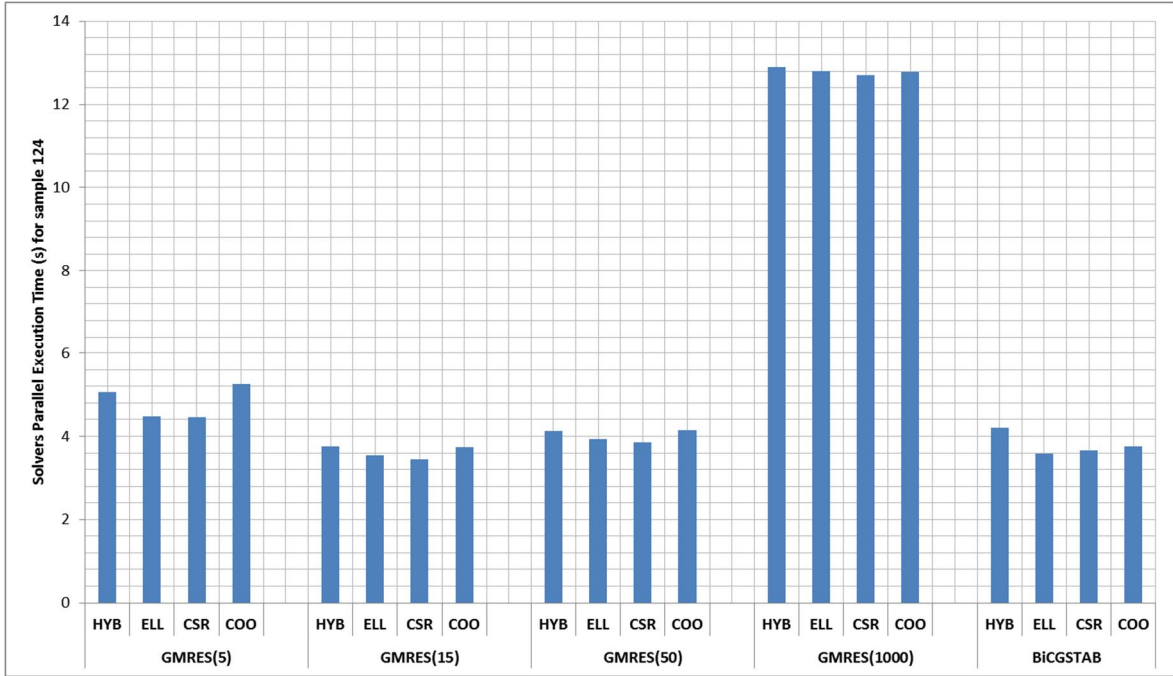


Figure 29: Solvers Parallel Execution time for various storage formats with relative residual semi-log plot. Sample_124. Convergence is independent of the utilized storage scheme.

Figure 30 shows the execution time of BiCGSTAB Solver for all considered samples with different storage schemes. The following could be further established and concluded:

- With the use of suitable preconditioner, BiCGStab convergence to the right solution at minimal execution time compared to GMRES.
 - The larger the matrix sample, the higher the condition number and the longer it takes to converge.
 - BiCGSTAB with CSR storage scheme outperformed others from Samples_0 to Sample_93. It came second in Sample_124.
 - Given the above experimental conditions, a suitable preconditioner and the set of storage schemes we studied, BiCGStab with CSR is considered a suitable tradeoff that solves our reservoir simulation problem. Although this selection represents a sub-optimal answer for purely Hepta-Diagonal Systems, it paves the way for supporting wide range of more interesting simulation conditions²⁹.
 - Interested readers in special optimal Blocked Hepta-Diagonal Storage format and its application to Sparse Matrix Vector Multiplication as well as extensive comparisons with other formats may refer to SG_DIA scheme presented in [72].
- The following section sheds more light on that issue.

²⁹ These includes utilizing unstructured meshes, different discretization or when using multi-well completion method.

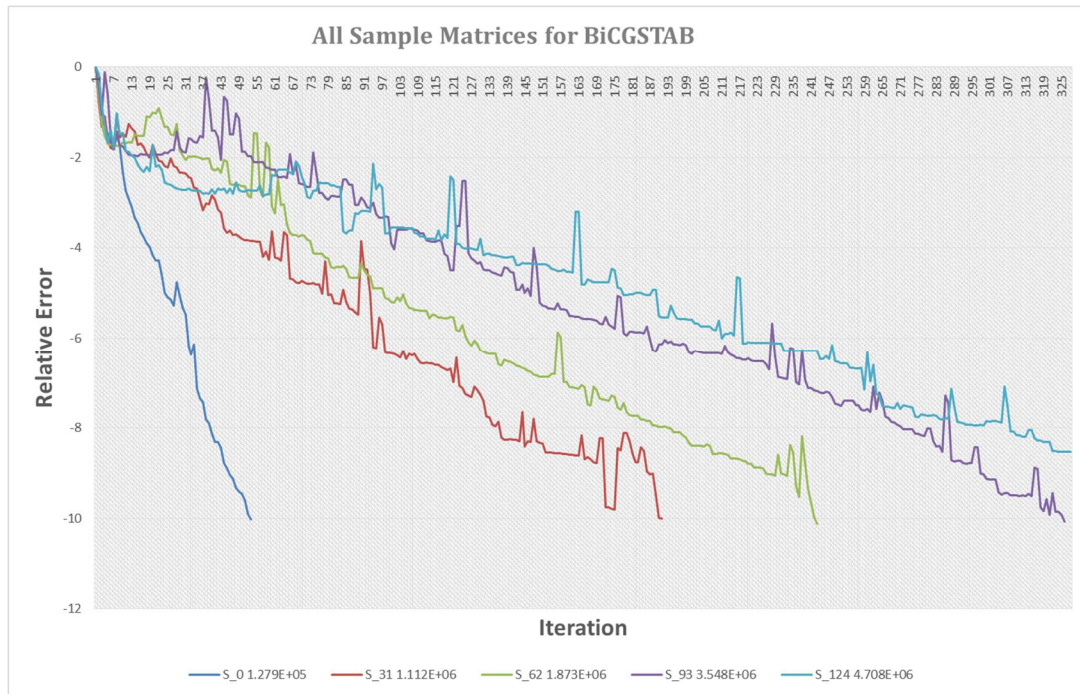
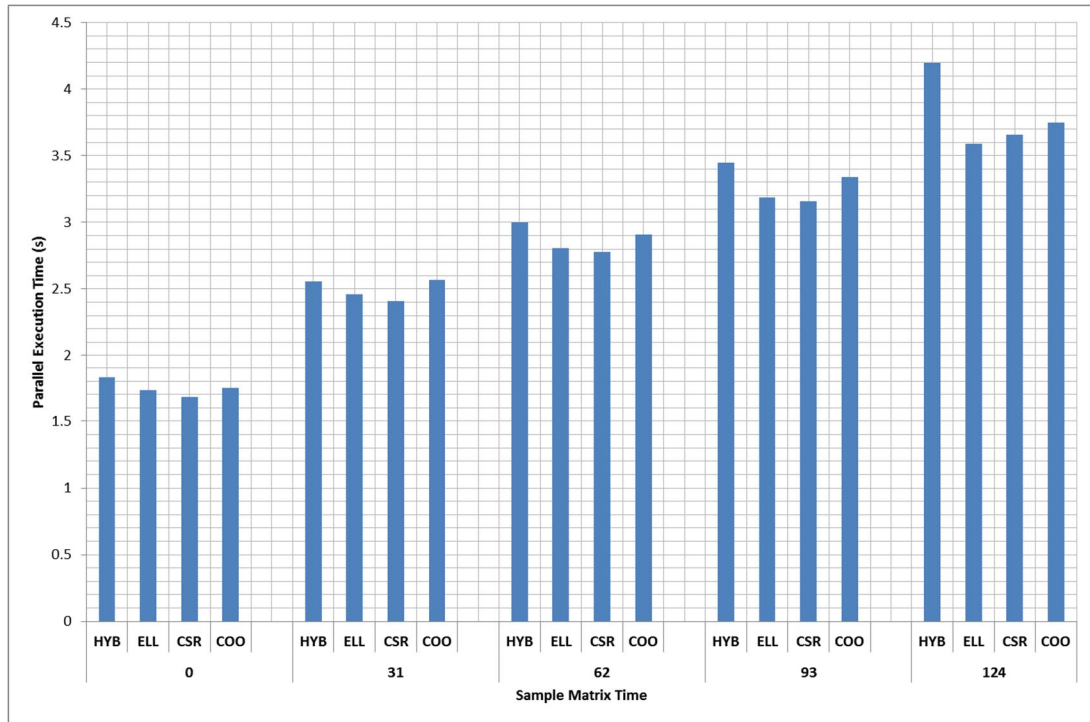


Figure 30: BiCGSTAB Parallel Execution time for various storage formats with relative residual semi-log plot. All Extracted Sample. Convergence is independent of the utilized storage scheme

3.4 Special Case: Sparse Matrix Vector Multiplication for Hepta-Diagonal Matrices

Objectives:

Examining the effect of different sparse storage schemes on the overall parallel execution time and comparing them for sparse Matrix-Vector Multiplication operation (SpMV) over Hepta-Diagonal Sample Matrices

Experimental Setup and Conditions:

- Six sample matrices with variable sizes that resemble elements distribution in the developed FRS have been considered.
- Each sample represents a large matrix with (2×2) block entries distributed in a Hepta-diagonal fashion. The rest of elements are zeros. Figure 9
- Tests were performed on a node in an HPC cluster offered by the Information Technology Center at KFUPM featuring a Xeon E5-2680 10-Core, 2.8 GHz (Dual-processor) and Tesla k20x GPU [103], Table 12 . A Comparison of different compute capabilities for GPU Architecture is presented in [103].

Method

- We use the CUSP implemented CUDA versions of Matrix-Vector Multiplication. We further implemented SG_DIA found in [72].

- The execution time of Sparse Matrix Vector Multiplication was examined using five different sparse storage mechanisms; four of which were provided by CUSP library: Compressed Row Storage (CSR), ELLPACK (ELL), Hybrid (HYB), and Coordinate Format (COO), and the last one is the implementation for the blocked diagonal format SG_DIA found in [72].
- Each experiment was repeated a number of times and the average parallel executed time was recorded.

Results and Discussion

Figure 31 demonstrates the average execution time of SpMV for various increasingly related matrix sizes for every utilized sparse storage schemes, while Figure 32 shows the execution time of the previous experiment by varying storage format across a given matrix leading dimension. The following can be concluded:

- Just as expected, for all matrix dimensions and due to its ability to exploit the reservoir matrix structure, (SG_DIA) outperformed all other schemes. This is more prominent when comparing it to (COO) as the latest enjoyed the most indirect addressing problem presented earlier.
- Moreover, as (ELL) is somehow close to (SG_DIA), and as the former has already been developed to suite sparse matrix vector multiplications on GPUs, it is then no wonder that (ELL) comes second in performance.
- Just like other formats, and although (SG_DIA) group multiple memory transactions, it suffers from the described earlier short row problem.

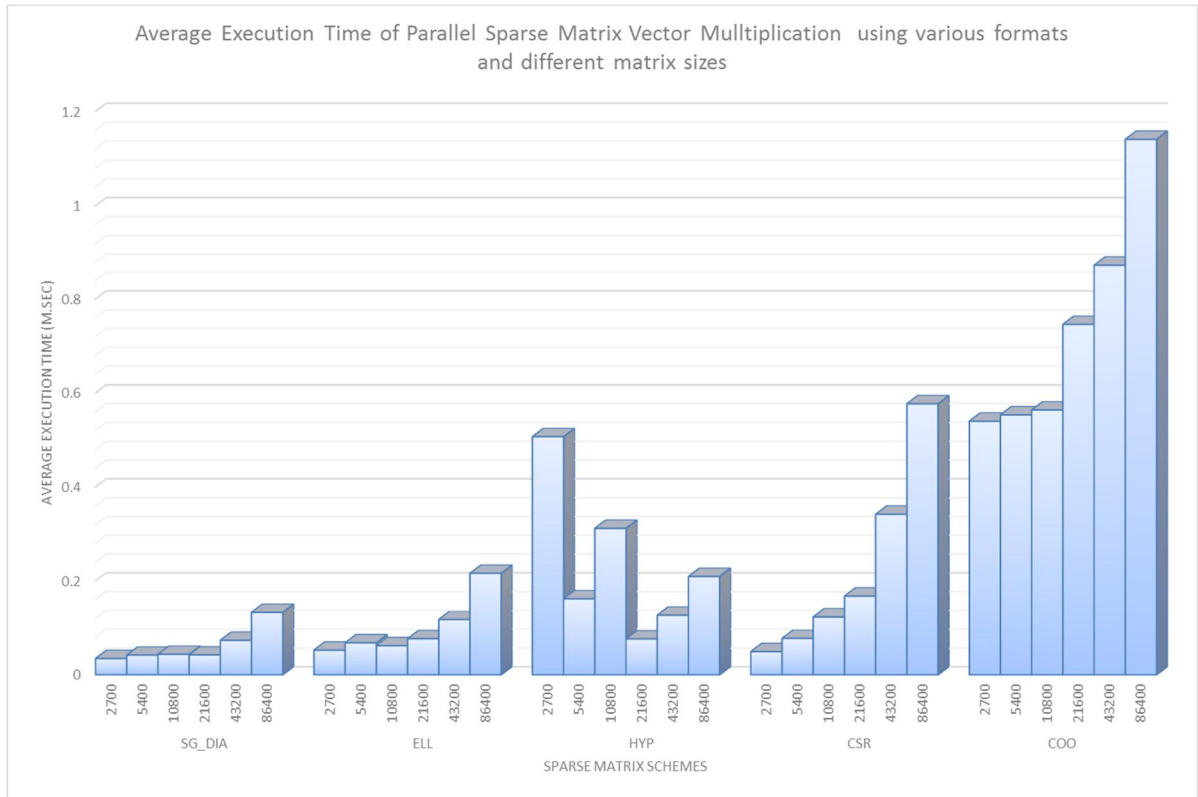


Figure 31: The average execution time of SpMV for various storage schemes and different related matrix dimensions. Here the input size has been studied within each scheme separately.

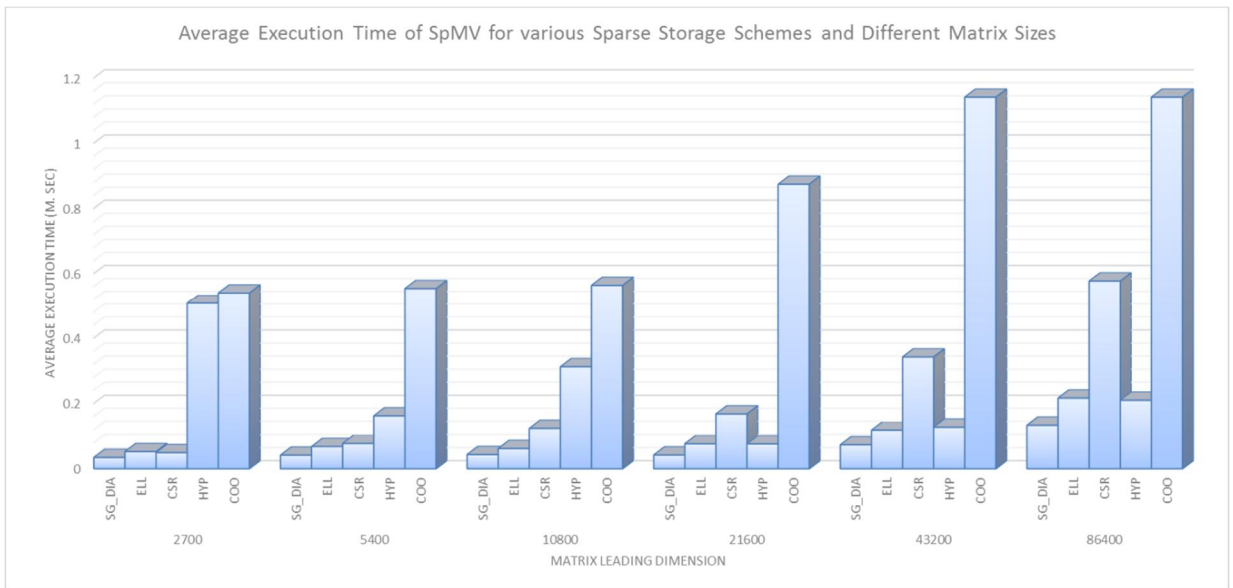


Figure 32: The average execution time of SpMV for various storage schemes and different related matrix dimensions. The focus here is see how each storage scheme behaves for a given matrix dimension.

3.5 Parallel Implementation of the selected Linear Solver for Matrices with Single (RHS)

3.5.1 Introduction

The goal of parallel programming is to provide tools and techniques for either solving big problems faster or to run larger instances of the given problem for the same time interval that was used to execute its serial counterpart. Exposing application concurrency refers to the art of breaking down the main problem into independent logical tasks³⁰ that could be later executed in parallel after mapping them to corresponding physical processing elements. It is then no wonder that restructuring the problem to exploit any available concurrency is indeed the first mandatory step before implementing any serial algorithm using a suitable parallel programming environment. The process for finding concurrency starts by a decomposition step performed on program data and the associated tasks. It is followed by an analysis step where the decomposed parts are grouped, ordered, or share their data [9].

Figure 33 shows the established Data Dependency Graph (DDG) of BiCGSTAB Algorithm, highlights concurrent operations, and demonstrates detailed tasks according to Table 3. The parallel pattern is directly inferred from the arrows that express data flow direction. For example, before vector s is correctly computed, α , v , and r^- should be available.

³⁰ A task is a sequence of instructions that operate together as a group.

BiCGSTAB
Data
Dependency
Graph

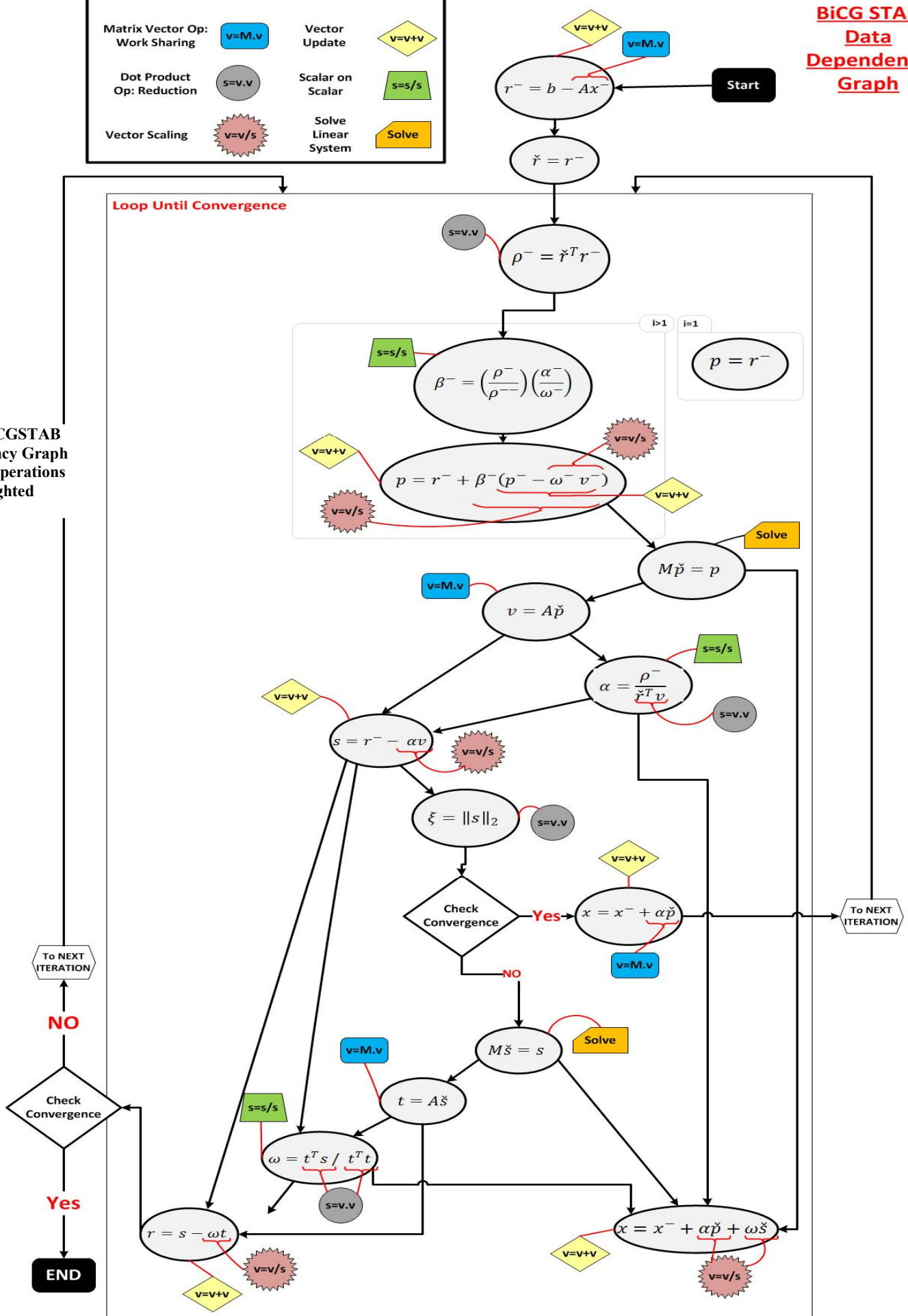
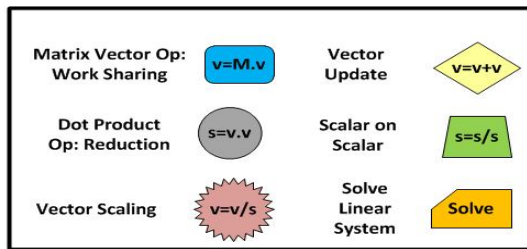


Figure 33: BiCGSTAB Data Dependency Graph (DDG), main operations are highlighted

3.5.2 Merging Operations³¹

Although the algorithm seems to complete its constituting tasks in sequence, various optimizations could be established to enable better parallel behavior [6]. One of which is based on the observation that various operations could be merged together taking advantage of both the commutative and associative properties of real numbers. This will allow different workers to continue to evaluate the next line of the algorithm without causing data hazards by accumulating partial results that could be later merged to form the complete solution. This is opposed to the other approach of establishing a barrier that forces thread synchronization after completing every operation. For example, and because of the dependency shown in Figure 33, one way of computing the sequence of operations extracted from BiCGSTAB Algorithm, shown in Figure 34, could be by first assigning multiple workers to perform the reduction operation, then they wait until everyone finishes its assigned job. After that, a single worker computes the scalar value at line 7, before they cooperate again to compute lines: 8. It is worth mentioning that Figure 34 presents an abstract symbolic view for how the calculations flow. After all, it is well known that performing a reduction operation in CUDA requires N cooperating threads with $\log N$ steps!

³¹ It is worth mentioning that, a similar trick has been utilized in [106] H. Anzt, S. Tomov, P. Luszczek, I. Yamazaki, J. Dongarra, and W. Sawyer, "Accelerating Krylov Subspace Solvers on Graphics Processing Units."

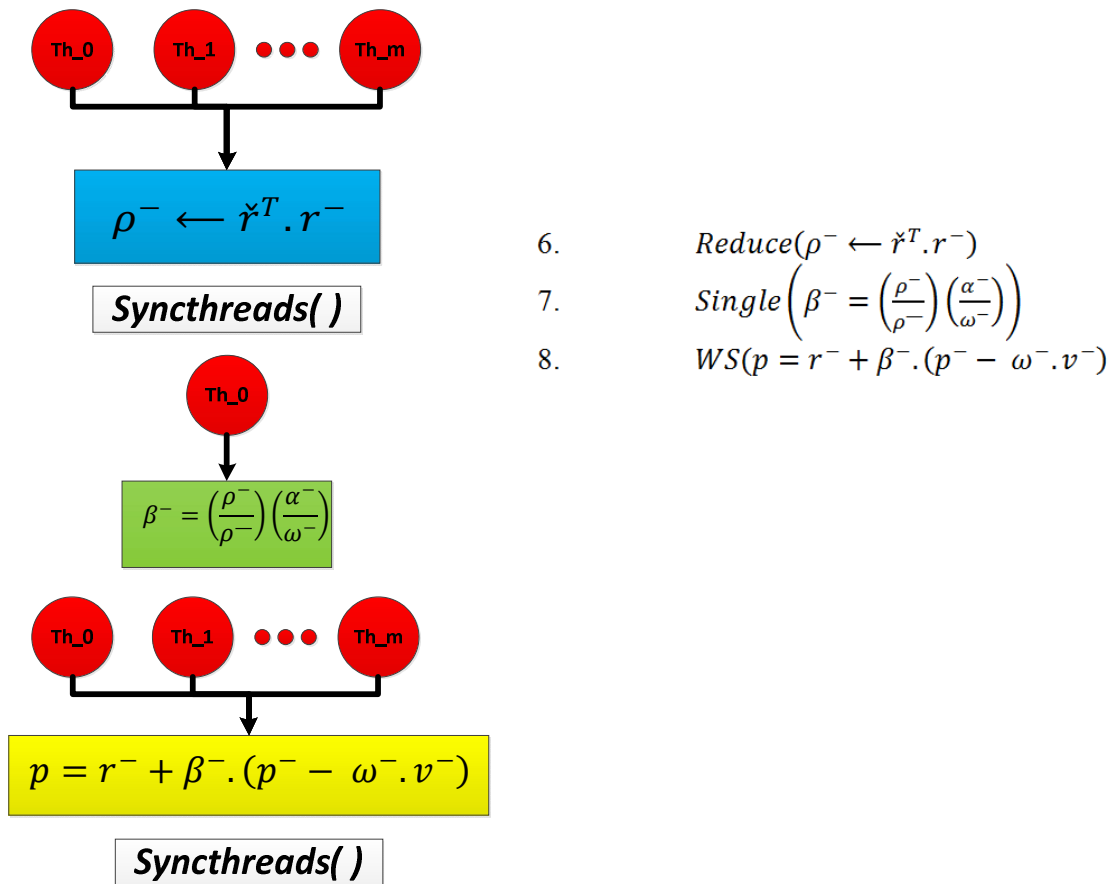


Figure 34: The normal flow for various threads cooperating to compute sequence of operations in BiCGSTAB Algorithm

At first glance and given the above dependencies, any CUDA developed parallel implementation of BiCGSTAB solver seems to be bounded in terms of both bandwidth and computation. This is due to the fact that, the nature of operations composing BiCGSTAB algorithm demands many memory loads with minimal computations performed on the loaded data i.e. computing resources spend the majority of the time busy waiting for data to be fetched. With the aim of reducing bandwidth pressure and increasing data locality, a split and merge strategy was adapted. Without loss of generality, the previous snippet of the algorithm shown in Figure 34, could be implemented as follows:

Let $J := \text{total number of elements of vector } r$.

ρ is then calculated as

$$\rho = \sum_{j=1}^J r_j \cdot \check{r}_j^T$$

$$\rho = r_0 \cdot \check{r}_0^T + r_1 \cdot \check{r}_1^T + \dots + r_J \cdot \check{r}_J^T$$

$$\rho = \rho_0 + \rho_1 + \rho_2 + \dots + \rho_J$$

Let every worker (thread) operate on one element of vector r & \check{r} , multiply them and store the result in the corresponding indexed location in the resulted ρ vector. Rather than finishing up the computation and finding the reduced value of ρ , i.e. summing the values over all indices, each worker continue to the next line of the algorithm, and calculates its corresponding $(\beta_0, \beta_1, \beta_2, \dots, \beta_J)$, where total β is

$$\beta = \rho_0 \left(\frac{\alpha}{\omega \cdot \rho_{\text{prev}}} \right) + \rho_1 \left(\frac{\alpha}{\omega \cdot \rho_{\text{prev}}} \right) + \dots + \rho_J \left(\frac{\alpha}{\omega \cdot \rho_{\text{prev}}} \right)$$

Or

$$\beta = \beta_0 + \beta_1 + \beta_2 + \dots + \beta_j$$

As a result, instead of having one thread loading two values of vectors r & \check{r} , multiply the value and store the result back, the kernel proceeds with calculating the corresponding partial β value using some other constants that has been already brought to shared memory and broadcasted to all threads within the block. In other words, increasing the computational intensity per memory operations.

Again, the same logic applies when calculating the P as it requires the value of β to be available priority. One work around that we adopted is to make every thread computes the reduced value of β that has been already accumulated in (`inter_blk_Beta`) vector by the aid of some other preloaded shared scalars before finally computing the final value of P and storing it back to global memory.

$$p_j = r_j + (p_j - \omega^- \cdot v_j) \sum_{m=1}^m \beta_m.$$

Similarly (r & v). This leads the following relation:

$$\sum_{j=0}^J p_j = \sum_{j=0}^J r_j + \sum_{i=1}^m \beta_i \sum_{j=0}^J (p_j - \omega^- \cdot v_j)$$

Without loss of generality, the following Figure 35, shed more light about such possible merge.

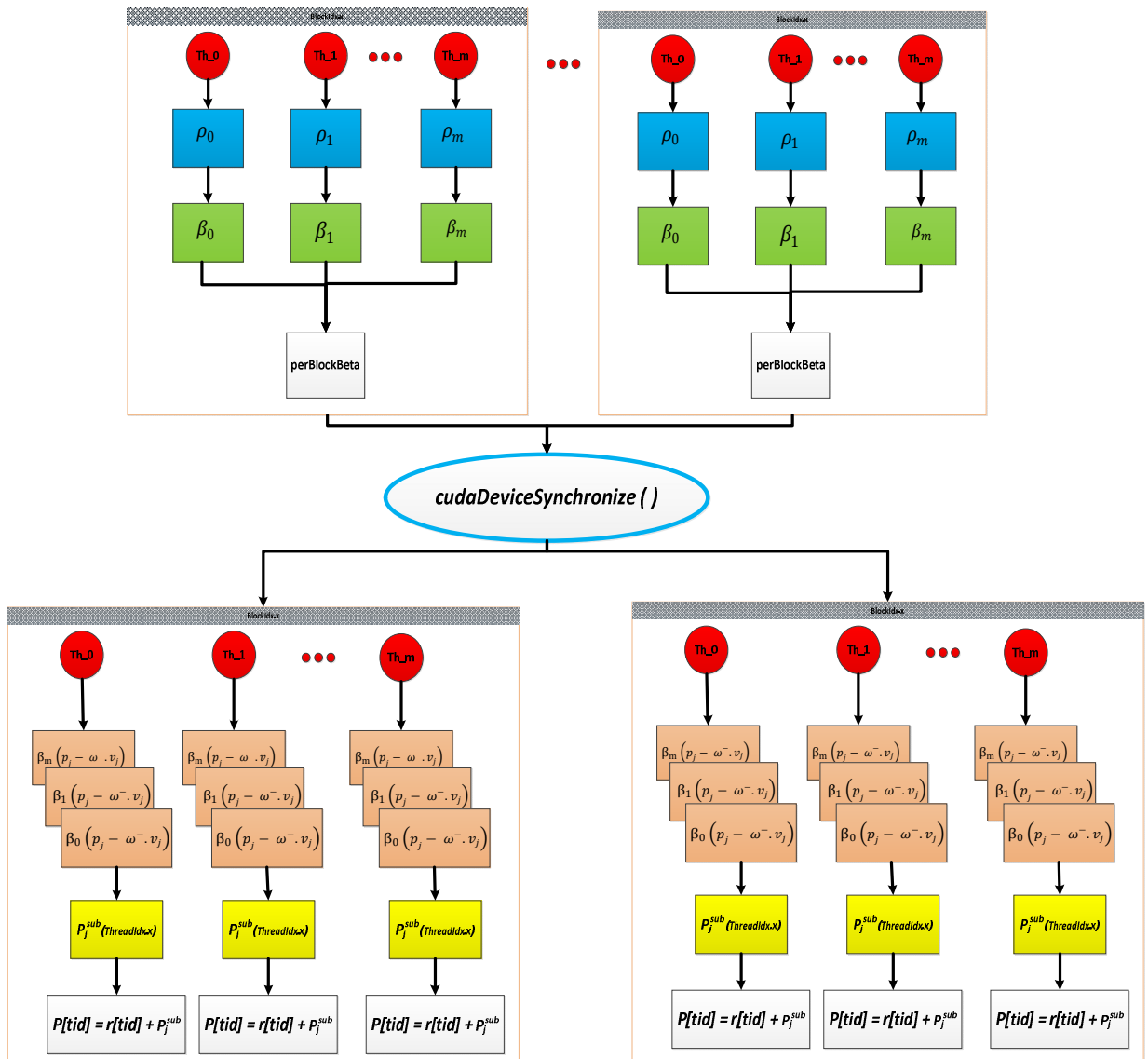


Figure 35: One possibility for merging arithmetic operations of the snippet of BiCGSTAB code, shown in Figure 34

The same trick was applied when computing vectors α and s at lines: 14 and 15, updating x and r from lines: 21 to 22, Algorithm 3. This strategy is easily extended to include the preconditioner as well as the matrix vector multiplication that follows.

The following code snippet shows kernel implementation for partial values of $(\rho, \beta$ and $P)$,
CODE 1.

CODE 1: GPU Kernels for computing rho, beta and P

```
__global__ void per_BLK_Rho_Beta(double *r_tld, double *r,
double *vector_Beta, double *vector_rho, double *global_Alpha,
double *global_rho1 ,int data_size) {
=====

    unsigned int Index = threadIdx.x;
    __shared__ double shared_Constants[3]; // this will make
    use of the broadcast property in shared memory all threads will
    read either first, second or third word in the bank and the
    returned value will be broadcast

    if(Index == 0 ){
        shared_Constants[0]= *global_rho1;
    }
    if(Index == 32 ){
        shared_Constants[1]= *global_Alpha;
    }
    if(Index == 64 ){
        shared_Constants[2]= global_Omega;
    }
    __syncthreads();
    //Allocating shared memory for intra (within) block
    reduction: Intra Blk
    __shared__ double Intra_Blkg_rho[threadsWithinBlock];
    __shared__ double Intra_Blkg_Beta[threadsWithinBlock];

    double rho_1 = shared_Constants[0]; double alpha =
    shared_Constants[1]; double omega = shared_Constants[2];
    double current_rho=0;

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    Intra_Blkg_rho[Index] = 0; Intra_Blkg_Beta[Index] = 0;

    while (tid < data_size ){

        current_rho = r_tld[tid] * r[tid]; // partial rho:
        rho_0, rho_1, rho_2

        Intra_Blkg_rho[Index] += current_rho;
        Intra_Blkg_Beta[Index] += (current_rho/ rho_1) * (alpha
        / omega);

        tid += blockDim.x * gridDim.x;
    }
    __syncthreads();

    if(Index < threadsWithinBlock ){
        UnrolledBlockReduce(Index,
        Intra_Blkg_Beta, Intra_Blkg_rho, threadsWithinBlock);
    }
}
```

```

    }
    __syncthreads();

    //Thread 0 from each block will write the resulted per
    block reduced rho to global memory
    if (Index == 0 ) {

        vector_Beta[blockIdx.x] = Intra_Blkg_Beta[0];
        vector_rho[blockIdx.x] = Intra_Blkg_rho[0];
    }
}

__global__ void compute_P(double *p, double *r, double *r_tld,
double *v, double *vector_Beta, double *vector_rho,int
data_size){

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int Index = threadIdx.x;

    __shared__ double omega;
    if(Index ==0)
        omega =global_Omega; // let th0 of every block brings
omega and share it with threads in a block

    // step_1: Bring vector beta to shared memory

    __shared__ double Inter_Blkg_Beta[blocksPerGrid];
    __shared__ double Inter_Blkg_Rho[blocksPerGrid];

    if( Index < blocksPerGrid){
        // very optimal if blocks is 32 as it will give only
one memory transaction
        Inter_Blkg_Beta[Index]= vector_Beta[Index];
        Inter_Blkg_Rho[Index]= vector_rho[Index];
    }

    __syncthreads();

    // operate on shared memory
    __shared__ double p_Sh[threadsPerBlock];
    __shared__ double v_Sh[threadsPerBlock];

    double current_Beta, current_Beta1, current_Beta2,
current_Beta3, current_Beta4, current_Beta5, current_Beta6,
current_Beta7;
    double p_next, p_next1, p_next2, p_next3, p_next4, p_next5,
p_next6, p_next7 ;

    while (tid < data_size ){

```

```

p_next = 0; p_next1 = 0; p_next2 = 0; p_next3 = 0;
p_next4 = 0; p_next5 = 0; p_next6 = 0; p_next7 = 0;

p_Sh[Index] = p[tid];
v_Sh[Index] = v[tid];

for(int i=0; i<blocksPerGrid;i+=8){

    current_Beta = Inter_Blk_Beta[i];
    current_Beta1 = Inter_Blk_Beta[i+1];
    current_Beta2 = Inter_Blk_Beta[i+2];
    current_Beta3 = Inter_Blk_Beta[i+3];
    current_Beta4 = Inter_Blk_Beta[i+4];
    current_Beta5 = Inter_Blk_Beta[i+5];
    current_Beta6 = Inter_Blk_Beta[i+6];
    current_Beta7 = Inter_Blk_Beta[i+7];

    p_next += current_Beta * (p_Sh[Index]- omega *
v_Sh[Index]);
    p_next1 += current_Beta1 * (p_Sh[Index]- omega *
v_Sh[Index]);
    p_next2 += current_Beta2 * (p_Sh[Index]- omega *
v_Sh[Index]);
    p_next3 += current_Beta3 * (p_Sh[Index]- omega *
v_Sh[Index]);
    p_next4 += current_Beta4 * (p_Sh[Index]- omega *
v_Sh[Index]);
    p_next5 += current_Beta5 * (p_Sh[Index]- omega *
v_Sh[Index]);
    p_next6 += current_Beta6 * (p_Sh[Index]- omega *
v_Sh[Index]);
    p_next7 += current_Beta7 * (p_Sh[Index]- omega *
v_Sh[Index]);

    }
    p[tid] = r[tid] + p_next + p_next1 + p_next2 + p_next3
+ p_next4 + p_next5+ p_next6 + p_next7 ;

    tid += blockDim.x * gridDim.x;

}
}

```

GPU devices feature a number of memory types that are characterized by their speed and scope. In addition to operations' merging, the previous kernels feature the following optimizations:

- Intensive use of shared memory and making use of its broadcast property.
- Loop unrolling to further increase computation intensity.
- A call to optimized implemented reduction kernel (`UnrolledBlockedReduce()`) based on various recommendations reported in literature [97, 107].
- Makes use of *Asynchronous* data transfer between host and device by utilizing Pinned Memory and streams. Kepler GK110 introduced a HyperQ mechanism that supports 32 hardware managed connections for communication between host and device. That improvement has a direct impact on increasing device utilization as multiple processors on the CPU could initiate work on a single GPU at the same time [15].
- Host and kernel execution overlap: when possible, the original code was restructured in a way that a call to device kernel is followed by many calls to host functions. By default, kernel launch is asynchronous or non-blocking. So while the GPU is busy, the host performs some other computations. If used properly, this mix, combined with streaming has great impact on performance.

To preserve dependency when sharing thread results, synchronization was enforced by exiting every related kernel and launching another one. Whenever necessary, a call to `cudaDeviceSync()` after kernel launch was initiated.

To elaborate more and without loss of generality, for calculating the values of ρ and β , each thread in CODE 1, loads to shared memory part of the global (r) and (\tilde{r}) vector, multiply the corresponding value using shared memory vector called *intra-blk-rho*, and accumulate partial sum before finally storing the final result to another vector in global memory called *inter_blk_rho*. Now and as the algorithm states, vector inter-block-rho is read by another kernel to either continue subsequent operations or got reduced on the host. In other words, whenever necessary, every block reduces given data through partial accumulation of the results, writes it to global memory and then the final reduction is done on the host by reading the reduced data by all blocks. This two steps synchronization is necessary as GPU devices do not allow data to be shared among blocks.

The convergence is checked from the host side at the end of each iteration. One optimization could be to skip the check for some iterations. However, this requires some prior anticipation of the number of expected iterations needed before converging to the right solution. Kernels constituting this program are shown in the appendix.

3.5.3 Experiments and Comparisons

Objectives:

To examine the speed up obtained after merging some operations in a CUDA implementation of BiCGStab Algorithm.

Experimental Setup and Conditions:

- A number of large matrix sets with variable sizes are extracted from our developed FRS. The dimension has been chosen to double the previous one starting from (10800 x 10800) and up to (921600 x 921600)
- Each sample represents a 3-D structured grid with (2 x 2) block entries distributed in a Hepta-diagonal fashion as resulted from finite volume discretization.
- Tests were performed on a node in an HPC cluster offered by the Information Technology Center at KFUPM featuring a Xeon E5-2680 10-Core, 2.8 GHz (Dual-processor) and Tesla k20x GPU [103], Table 12 . A Comparison of different compute capabilities for GPU Architecture is presented in [103].

Method

- Two parallel versions of the BiCGSTAB were programmed. The first one was solely based on calling cuBLAS and cuSPARSE routines (BiCGSolver_Lib) and the other utilizes the ideas and optimizations mentioned above (BiCGSolver_Merged). See the appendix for the two programs.

- In both cases and since we are solely interested in evaluating the speedup that results from merging, we decided to utilize CSR storage schemes.
- It is worth mentioning that other specialized schemes like SG_DIA [72], will definitely produce a better overall performance to both implementations and for the considered testing matrices.
- In both cases, we utilized the ILU preconditioner offered by cuSPARSE library. Again, using other advanced preconditioners will definitely have better overall performance results.

Program Tuning

Obtaining the best performance out of CUDA-Based parallel programs is beholden by many design choices that in many cases are contradicting in nature. To manage this and to help programmers tune their applications according to their desired performance goals, NVidia provided a number of tools including the visual profiler and occupancy calculator.

It is always intimidating to utilize more resources that grant higher throughput like registers, but unfortunately that comes with the price of limiting concurrency. After all, one key aspect at which GPU devices achieve their Tera-flop performance is through latency hiding. When a given warp³² stalls because of unavailable data and while these data being fetched from global memory, other warps are context-switched and scheduled for

³² A warp is a group of 32 consecutive threads within a block scheduled to be executed by the CUDA multiprocessor.

execution with zero penalty. Similarly, when a block stalls for any reason, other blocks are switched in by the scheduler. As a result, a smart selection for the number of blocks to be executed as well as the number of threads used by each block is mandatory for any successful exploitation of GPU device capabilities for achieving higher performance.

Each streaming multiprocessor (SMX) in Kepler GK110 supports a maximum of 65536 registers, 16 blocks, 2048 threads and 64 warps. Forcing CUDA kernel to use registers for variables may be achieved by explicitly using scalar variables and via loop unrolling. However and as mentioned above, using more registers will hinder performance as it limits the number of launched blocks. For example, assume we are using 256 threads each uses 100 32-bit registers (50 double private variables). Then each block will demand $256 \times 100 = 25600$ registers. As a result, the maximum number of blocks that can be launched is calculated by dividing the maximum number of registers supported by each SMX over the utilized registers or $(65536/25600) = 2$ blocks. This means utilizing only 12.5 % of the maximum blocks allowed per SMX!

In a similar way and although its latency is almost 100x lower than uncached global memory latency³³, the exorbitant use of shared memory may also limit the pledged device performance. If 48 KB of shared memory is to be used among 8 blocks, then each block should utilize a maximum of 6 KB shared memory! Moreover, to prompt for higher bandwidth utilization, shared memory is distributed into concurrently accessed, equally sized 32 4-Bytes logical banks each with bandwidth of 64 bits per clock cycle. Memory bank conflict degrades shared memory performance by serializing bank accesses and

³³ <http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/>

occurs when multiple simultaneous requests by different threads are made to the same bank. Therefore, whenever shared memory is utilized, the associated variables should be placed under scrutiny to avoid possible bank conflicts. To enable better optimization when double precision variables are used, device bank size should be configured to be 8 bytes instead of the default on.

In our developed program, we started with a given number of blocks & threads; and empirically tuned their figures until the least execution time was obtained for kernels launching 128 blocks and 256 threads per block. As stated before, the motivation behind lies in the observation that usually but not necessarily [108], the higher the occupancy ratio, the more attainable performance.

Moreover and by running the visual profiler, we studied memory bandwidth utilization, how instruction and memory latency limit the performance by analyzing stalls, compute resources as well as other offered suggestions. We then came up with a list of optimizations that we later manually addressed. These include: overlapping communication and computations, utilizing streams for data transfer, minimizing the number of used registers, tiling and memory coalescing and others. Table 16.

It is worth mentioning that, there has been several attempts for designing auto-tuning applications that automatically aim at adjusting several CUDA parameters. Interested readers may consult [109-114]

Results and Discussion

The following Figure 36, presents the data along with some useful statistics while Figure 37, presents a double-log plot for the average execution time for both solvers. It can be seen that there is an order of magnitude speedup gain between the two implementations. It is clear that this merging technique utilizes more space than the usual calculation but it poses the following advantages that contributed to this drop in execution time:

- Increasing work intensity per thread.
- Efficient utilization of resources by allowing data reuse through shared memory.
- Better throughput utilization by reducing global memory transactions.
- Less power consumption due to reducing the loads from global memory.

BiCGSTAB Function Calls Implementation											Conf. Coeff: 1.96							
Coeff. Mat. Leading Dim.	Parallel Execution Time(msec)										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10								
	10800	526.6	527	528.6	528	523.2	523.7	522.8	522.5	520.9								
28800	839.9	844.7	844.1	839.6	840.9	840.9	837.2	831.6	833.3	857	840.920	7.028	4.356	845.276	836.564	857.000	831.600	25.400
57600	1368.8	1373.4	1344.7	1356.8	1340.6	1338.5	1250.1	1341	1341.6	1345.2	1340.070	33.935	21.033	1361.103	1319.037	1373.400	1250.100	123.300
115200	2229.6	2229.4	2213.5	2203.8	2230.8	2412.5	2252.2	2245.7	2386.2	2254	2265.770	72.407	44.878	2310.648	2220.892	2412.500	2203.800	208.700
230400	3924.2	4012.4	3959.6	3988.8	3954.4	3912.5	3962.8	3953.8	3912.9	3871.4	3945.280	40.950	25.381	3970.661	3919.899	4012.400	3871.400	141.000
460800	7300.5	7346.8	7061.9	7369	7317.3	7253.7	7326.6	7375.4	7364.5	7204.4	7292.010	97.263	60.284	7352.294	7231.726	7375.400	7061.900	313.500
921600	13425.3	12863.5	13855.4	13746.3	13335.3	12875.7	13584.6	13465.2	13602.7	13835.8	13458.980	354.246	219.564	13678.544	13239.416	13855.400	12863.500	991.900

BiCGSTAB Merging Operations Implementation											Conf. Coeff: 1.96							
Coeff. Mat. Leading Dim.	Parallel Execution Time(msec)										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10								
	10800	68.6	68.4	68.3	71.8	68.0	68.5	68.5	68.5	68.5								
28800	139.6	139.6	139.7	139.5	139.5	139.7	139.7	139.6	139.8	139.6	139.63	0.09	0.06	139.69	139.57	139.80	139.50	0.30
57600	273.6	273.7	273.8	273.7	273.7	273.7	273.9	273.7	273.8	273.8	273.74	0.08	0.05	273.79	273.69	273.90	273.60	0.30
115200	399.9	399.2	399.9	399.1	401.3	401.6	399.7	399.5	399.6	399.9	399.97	0.83	0.52	400.49	399.45	401.60	399.10	2.50
230400	515.7	515.6	515.7	515.7	515.7	515.7	516.2	516.2	516.1	518.5	516.11	0.87	0.54	516.65	515.57	518.50	515.60	2.90
460800	852.7	852.7	853.3	857.9	854.8	853.1	855.6	856.1	855.2	853.0	854.44	1.76	1.09	855.53	853.35	857.90	852.70	5.20
921600	1139.1	1138.3	1137.5	1141.9	1140.7	1139.7	1140.7	1143.4	1138.8	1143.9	1140.40	2.14	1.33	1141.73	1139.07	1143.90	1137.50	6.40

Figure 36: Average Parallel Execution time for the two versions of the implemented solvers

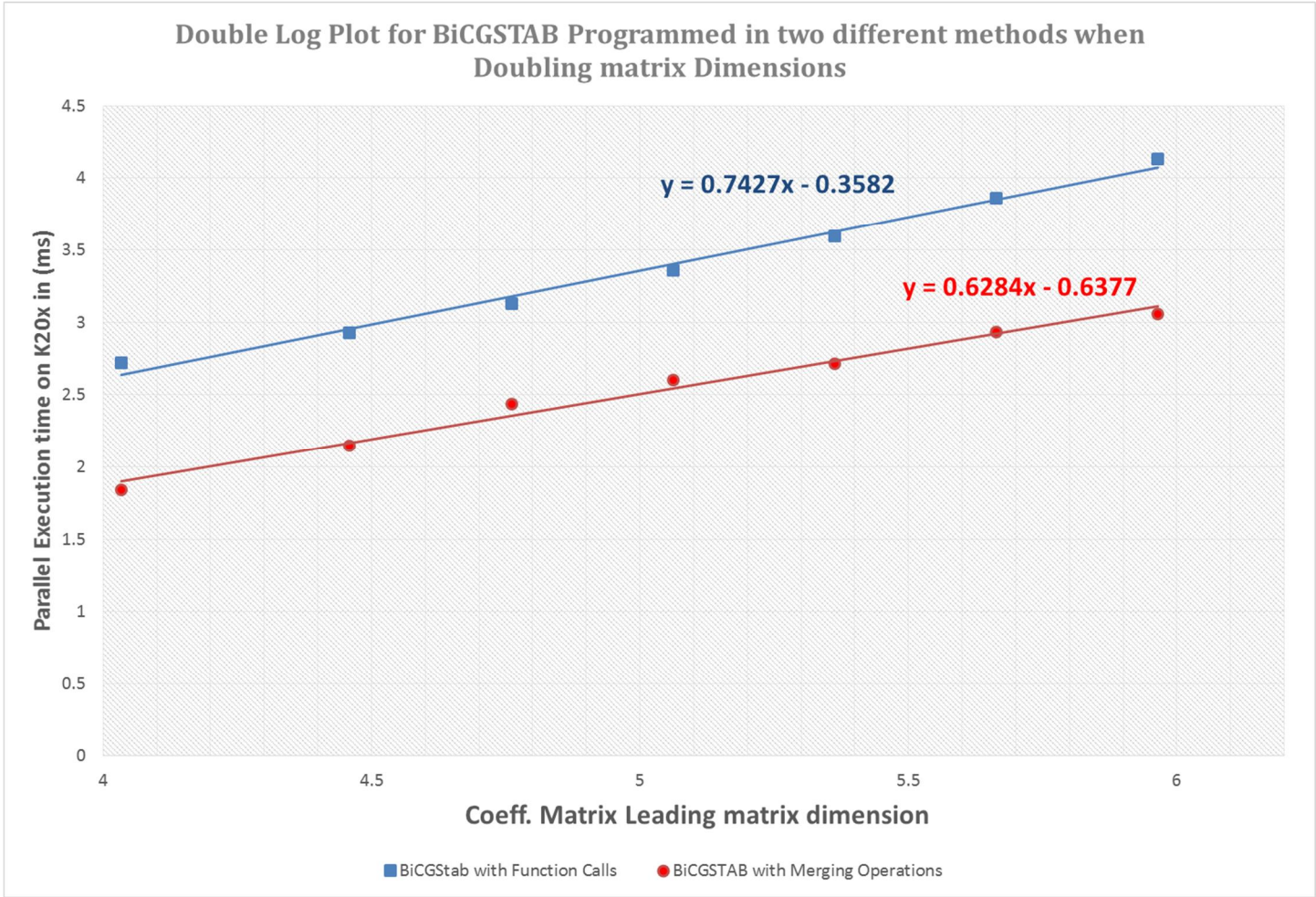


Figure 37: Average Parallel Execution time for the two versions of the implemented solvers

Next we analyze the performance flops given the various kernels that composes our implementation of BiCGStab_Merged for samples extracted from the reservoir. The following table shows the count of their multiply-add operations as well as the computed GFLOPS/s while Figure 38, plots the computed GFLOPS/s for various matrix dimensions.

Table 14: Performance FLOPS for the kernels constituting the BiCGSTAB merged implementation

Number of operations	19	20	13	10	23	4
Kernel Name	Reduced Omega	per_blk Omega	compute S	per_blk alpha	Compute_P	per_blk rho_beta
Vector Size						
10800	7.89	12.71	8.26	6.75	1.38	2.06
28800	10.32	33.88	19.71	19.20	25.48	5.01
57600	13.85	60.63	28.80	30.32	38.96	7.94
115200	14.69	96.00	35.66	42.67	42.74	12.45
230400	15.18	121.26	41.03	56.20	46.08	16.17
460800	15.28	146.29	44.70	69.82	53.26	20.03

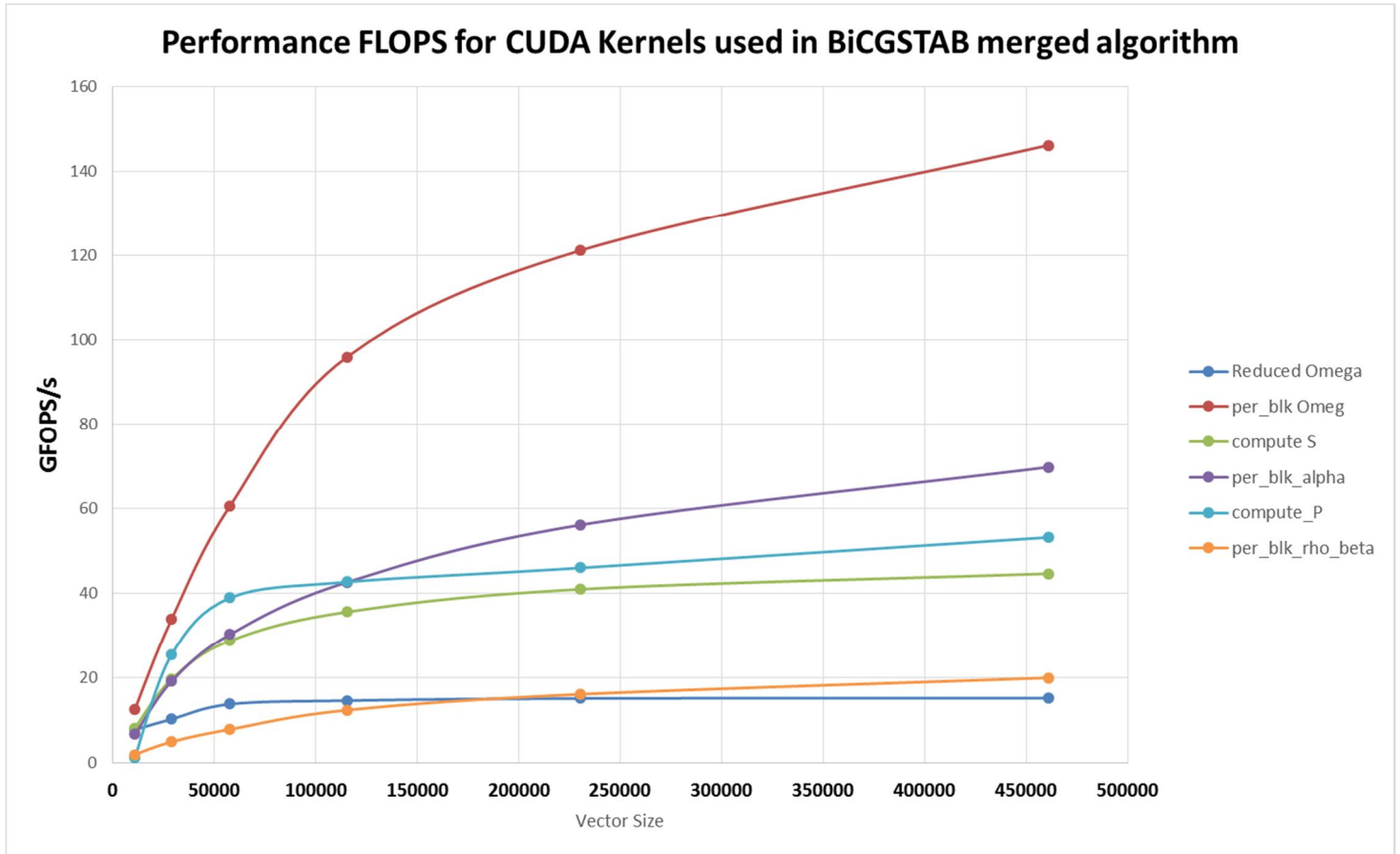


Figure 38: GFLOPS/s for the kernels used to program the BiCGSTAB merged for various matrix dimensions

It is clear that all kernels are memory bandwidth bounded because the algorithm itself does not demand reuse of loaded matrix elements. For that reason, and with the aim of achieving more performance, we focused on bandwidth optimization and heavily utilized shared memory and registers to increase the intensity of computation per memory operations. Despite this huge lag between the performance plotted in Figure 38 to the device peak performance (1.31 teraflops for double precision), the performance of our implemented BiCGSTAB-merged is comparable to the one suggested and implemented in [106].

3.6 Parallel Implementation of the selected Linear Solver for Matrices with Single (RHS)

3.6.1 Introduction and Motivation

This section describes our attempt for implementing a parallel Krylov based subspace solver designed to solve a linear system with multiple right hand sides (MRHS). Based on previously mentioned considerations, we modify the past implementation of BiCGSTAB to suit the problem at hand. Solving a linear system with multiple right hand sides is required by our simulator when doing history matching. All vectors in the right hand side matrix are independent; the thing that prompts and motivates experimenting three important ideas.

One would be tempted to utilize direct methods and find the inverse of the coefficient matrix (A) as the decomposition will be done once and repeated for all MRHS. The famous approach would be sparse LU factorization with pivoting that requires $O(n^3)$ complexity. However, as our initial matrix is Hepta-sparse and as the dimension of the matrix is very huge, we would not be able to afford the high storage demand required by this approach. After all, the inverse of a sparse matrix is a full matrix.

We were also tempted to insert an outer loop over any version of our implemented parallel BiCGSTAB and repeat the whole solver thing until we finish all vectors in MRHS. This is indeed an easy and naïve solution and takes advantage of the previously implemented parallel solver and produces right results. Nevertheless, this approach does not take

advantage of various optimization opportunities that has been raised because of this data independent MRHS.

The third solution that we will adapt, utilizes a cuSPARSE library call designed specifically to solve MRHS systems along with dynamic parallelism in order to implement a fast BiCGSTAB dedicated to solving MRHS problems. The next pages explain the idea more and show performance results. For details on various functions provided by cuSPARSE and various examples for different basic linear algebra operations at different levels please see [88]. The functions utilized from the library package are provided as a black box. Nevertheless, we can anticipate and guess many optimizations utilized. Those include: eliminating some common operations or results which are required for solving with each RHS and maximizing data reuse via the use of shared memory and registers.

Recent releases of CUDA supports a process through which a kernel may invoke another kernel. The new functionality jargoned by Dynamic Parallelism³⁴ not only synchronizes kernel execution, but also enables wide range of applications, including recursive calls, to be implemented. A parent kernel that is executed simultaneously with other parent kernels on various SM's, could invoke other child kernels that also demands its share on those SM's, may also evolve to become parents and invoke other child kernels and so on. This indeed creates an extra overhead over the programmer's scheduler and requires him to keenly utilize available resources to tune the application for better performance. Nonetheless, Dynamic Parallelism creates more parallelization opportunities as GPU

³⁴ Adapted from
http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf

hardware is more involved in the optimization process. The fact that a parent kernel halts waiting for his child to complete before it resumes execution enables both parties to implicitly synchronize their actions and exchange Data without CPU participation [115].

3.6.2 Implementation Strategy

The complete BiCGSTAB code was written using cuBLAS, cuSPARSE function calls as well as Thrust. The tasks and operations in the algorithm were mapped to suitable functions. The synchronization between various algorithm operations was done depending on either the implicit barrier provided by those function calls, the implicit synchronization point created between a parent and a child processes as described earlier, or an explicit call to *cudaDeviceSynchronize()* API call after kernel invocation. Convergence check is done at the host side at the end of each iteration by reading convergence flags passed from the device side. Same optimizations as the one presented in (section 3.4.2, P138) have been utilized. Four kernels were developed (*compute_X*, *compute_S*, *compute_alpha*, *compute_P*); they all have the same logic. Without loss of generality, details and explanation is given for one of them Figure 39, *compute_alpha* Kernel.

The kernel parameters are vectors stored in global memory and passed by reference. After setting up a global thread ID, the kernel initializes handles for cuBLAS routines. In their turn, those routines call implicitly other kernels in order to finish up the computation. An offset is assigned to pick up the right data portion that is passed as a parameter to each cuBLAS function. Since there is no need for data reuse, only registers were utilized. In the end, the cuBLAS destroy event is called. Performing the computation in this manner will

enable each thread (represented by its global ID) to operate on one complete vector of the MRHS. Whenever necessary, tiling could be implemented to handle larger vector dimensions. Kernels constituting this program are shown in the appendix.

```

__global__ void compute_alpha_MRS(double *r_tld, double *v, double *rho_values_Vector,
    double *alpha_values_Vector, int N, int RHS_Width){

    int Multipl_RHS_Size = N * RHS_Width;
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int offset;

    double myTemp, alpha_value;

    cublasHandle_t blasHandle;
    cublasStatus_t status = cublasCreate(&blasHandle);

    while (tid < RHS_Width ){

        offset = tid * N;

        cublasDdot(blasHandle, N, &r_tld[0]+offset, 1, &v[0]+offset, 1, &myTemp); // ( r_tld'*v )
        alpha_value = rho_values_Vector[tid] / myTemp; // alph = rho / ( r_tld'*v )
        alpha_values_Vector[tid]=alpha_value;

        //cout <<"rho.. " << rho_values_Vector[ind] <<"\t r_tld*v.. "
        //<< myTemp<<"\tAlpha.. " <<alpha_value << endl;
        tid += blockDim.x * gridDim.x;
    }
    cublasDestroy(blasHandle);
}

```

Figure 39: The Kernel Function for compute_alpha

3.6.3 Performance Evaluation

The testing was performed on samples with different sizes, extracted from our reservoir simulator, representing a 3-D structured grid with (2×2) block entries and distributed in a Hepta-diagonal fashion. As explained before, the coefficients are a combination of various reservoir parameters (permeability, compressibility ...), oil pressure values P_o and water saturation levels S_w . When simulation time proceeds, elements composing the coefficient matrix changes as both P_o and S_w get updated. Tests were performed on a node in an HPC cluster offered by the Information Technology Center at KFUPM featuring a Xeon E5-2680 10-Core, 2.8 GHz (Dual-processor) and Tesla k20x GPU that features a 6 GB memory [103]. The tests were repeated for different MRHS dimensions ranging from 32 vectors and up to 2048. *Device Allocation Fail* flag is raised whenever CudaMalloc function fails to execute because of exceeding the size of global memory inside the GPU. Figure 40 shows the data along with some statistics while Figure 41, plots the results of the average execution time of the implemented parallel BiCGSTAB_MRHS for different matrix dimensions and various MRHS widths.

BiCGSTAB for Multiple Right Hand Side (MRHS)											MRHS = 32			Conf. Coeff: 1.96				
Coeff. Mat. Leading Dim.	Parallel Execution Time(msec)										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10								
	2700	282.5	282.8	282.9	283.4	282.9	283.9	283.9	282.5	282.9								
5400	507.1	509.2	508.0	508.1	508.1	508.1	507.4	507.5	507.8	511.2	508.25	1.18	0.73	508.98	507.52	511.20	507.10	4.10
10800	983.9	983.8	983.5	969.1	970.4	971.4	971.4	971.5	971.4	971.6	974.80	6.21	3.85	978.65	970.95	983.90	969.10	14.80
21600	1518.2	1554.6	1524.1	1515.7	1514.5	1518.8	1527.4	1518.4	1528.2	1527.4	1524.73	11.66	7.23	1531.96	1517.50	1554.60	1514.50	40.10
43200	2426.2	2430.3	2431.2	2428.1	2427.2	2432.1	2434.2	2407.8	2425.2	2420.7	2426.30	7.56	4.69	2430.99	2421.61	2434.20	2407.80	26.40
86400	4719.2	4701.3	4728.3	4706.9	4718.6	4735.9	4748.1	4745.6	4760.9	4732.8	4729.76	18.78	11.64	4741.40	4718.12	4760.90	4701.30	59.60
BiCGSTAB for Multiple Right Hand Side (MRHS)											MRHS = 64			Conf. Coeff: 1.96				
2700	451.1	449.8	449.8	450.6	452.0	452.9	458.2	453.3	453.4	452.4	452.35	2.46	1.52	453.87	450.83	458.20	449.80	8.40
5400	795.9	794.7	794.0	794.8	793.6	800.4	798.1	795.6	794.0	793.3	795.44	2.24	1.39	796.83	794.05	800.40	793.30	7.10
10800	1592.5	1592.0	1594.0	1593.5	1610.4	1593.7	1596.1	1595.7	1592.9	1593.7	1595.45	5.41	3.35	1598.80	1592.10	1610.40	1592.00	18.40
21600	2504.9	2504.7	2510.9	2547.0	2520.2	2509.6	2512.1	2510.4	2547.3	2513.1	2518.02	15.95	9.89	2527.91	2508.13	2547.30	2504.70	42.60
43200	4079.6	4086.2	4094.8	4084.8	4087.9	4067.3	4087.1	4078.5	4067.7	4100.3	4083.42	10.56	6.54	4089.96	4076.88	4100.30	4067.30	33.00
86400	7996.9	7978.2	7969.5	7964.7	7970.2	7971.8	7988.9	7980.0	7993.8	7963.7	7977.77	11.94	7.40	7985.17	7970.37	7996.90	7963.70	33.20
BiCGSTAB for Multiple Right Hand Side (MRHS)											MRHS = 128			Conf. Coeff: 1.96				
2700	800.6	800.0	798.8	800.6	801.7	804.2	802.0	796.4	797.3	798.7	800.03	2.32	1.44	801.47	798.59	804.20	796.40	7.80
5400	1401.3	1405.0	1400.2	1402.5	1404.2	1400.6	1395.9	1403.1	1409.6	1403.5	1402.59	3.57	2.21	1404.80	1400.38	1409.60	1395.90	13.70
10800	2846.1	2851.0	2850.1	2854.8	2842.9	2885.3	2844.4	2878.5	2842.1	2844.2	2853.94	15.35	9.51	2863.45	2844.43	2885.30	2842.10	43.20
21600	4431.3	4437.0	4433.0	4428.3	4435.4	4449.9	4418.7	4423.4	4429.6	4423.4	4431.00	8.77	5.44	4436.44	4425.56	4449.90	4418.70	31.20
43200	7334.2	7344.2	7338.4	7330.0	7336.7	7345.0	7382.9	7404.2	7406.2	7341.4	7356.32	29.58	18.33	7374.65	7337.99	7406.20	7330.00	76.20
86400	14477.6	14492.6	14467.2	14471.9	14455.0	14475.0	14468.0	14483.0	14488.7	14485.0	14476.40	11.39	7.06	14483.46	14469.34	14492.60	14455.00	37.60

BiCGSTAB for Multiple Right Hand Side (MRHS)											MRHS = 256			Conf. Coeff: 1.96					
Coeff. Mat. Leading Dim.	Parallel Execution Time(msec)										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range	
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10									
	2700	1473.8	1474.5	1479.8	1485.7	1475.7	1472.6	1588.6	1547.2	1463.4									1515.6
5400	2540.8	2540.8	2542.0	2542.7	2537.7	2545.0	2543.0	2542.4	2535.5	2545.7	2541.56	3.09	1.92	2543.48	2539.64	2545.70	2535.50	10.20	
10800	5202.0	5202.6	5202.7	5254.0	5210.5	5199.0	5205.8	5243.4	5198.9	5200.8	5211.97	19.81	12.28	5224.25	5199.69	5254.00	5198.90	55.10	
21600	8175.4	8187.9	8198.6	8262.7	8183.2	8188.1	8189.8	8183.3	8180.2	8185.2	8193.44	25.10	15.56	8209.00	8177.88	8262.70	8175.40	87.30	
43200	13825.4	13783.2	13773.5	13750.9	13738.0	13753.8	13749.2	13774.0	13766.4	13763.6	13767.80	24.41	15.13	13782.93	13752.67	13825.40	13738.00	87.40	
86400	Device Allocation Fail																		
BiCGSTAB for Multiple Right Hand Side (MRHS)											MRHS = 512			Conf. Coeff: 1.96					
2700	2798.6	2853.2	2824.8	2806.1	2803.3	2810.9	2811.7	2805.9	2820.6	2807.1	2814.22	15.77	9.78	2824.00	2804.44	2853.20	2798.60	54.60	
5400	4820.1	4851.5	4850.4	4852.1	4828.7	4828.4	4845.3	4824.2	4825.5	4814.2	4834.04	14.31	8.87	4842.91	4825.17	4852.10	4814.20	37.90	
10800	9916.7	9935.1	9930.8	9934.9	9943.0	9922.6	9960.0	9959.4	9910.7	9934.5	9934.77	16.26	10.08	9944.85	9924.69	9960.00	9910.70	49.30	
21600	15660.1	15655.4	15688.7	15655.9	15618.4	15631.4	15650.8	15633.7	15658.5	16012.9	15686.58	116.27	72.06	15758.64	15614.52	16012.90	15618.40	394.50	
43200	Device Allocation Fail																		
BiCGSTAB for Multiple Right Hand Side (MRHS)											MRHS = 1024			Conf. Coeff: 1.96					
2700	5530.3	5483.6	5478.6	5482.3	5500.7	5492.1	5489.3	5502.4	5476.7	5489.5	5492.55	15.77	9.78	5502.33	5482.77	5530.30	5476.70	53.60	
5400	9450.7	9438.5	9458.6	9437.0	9441.8	9449.1	9439.4	9450.7	9452.3	9432.9	9445.10	8.26	5.12	9450.22	9439.98	9458.60	9432.90	25.70	
10800	19478.9	19507.1	19477.3	19475.3	19491.1	19415.3	19449.3	19470.5	19464.2	19500.9	19472.99	26.46	16.40	19489.39	19456.59	19507.10	19415.30	91.80	
21600	Device Allocation Fail																		
BiCGSTAB for Multiple Right Hand Side (MRHS)											MRHS = 2048			Conf. Coeff: 1.96					
2700	10802.4	10704.4	10740.6	10779.8	10839.1	10763.2	10733.9	10753.1	10779.9	10744.9	10764.13	38.16	23.65	10787.78	10740.48	10839.10	10704.40	134.70	
5400	18610.4	18482.0	18471.2	18545.3	18583.0	18621.7	18494.5	18498.4	18681.8	18559.0	18554.73	69.69	43.19	18597.92	18511.54	18681.80	18471.20	210.60	
10800	Device Allocation Fail																		

Figure 40: Data and some statistics for a version of BiCGSTAB that solves a system with MRHS. Whenever GPU memory cannot be allocated on the device, device allocation fail flag is raised

Double Log Plot for BiCGSTAB Solver for a Matrix with Variable Multiple Right Hand Side (MRHS) when Doubling Matrix Dimensions

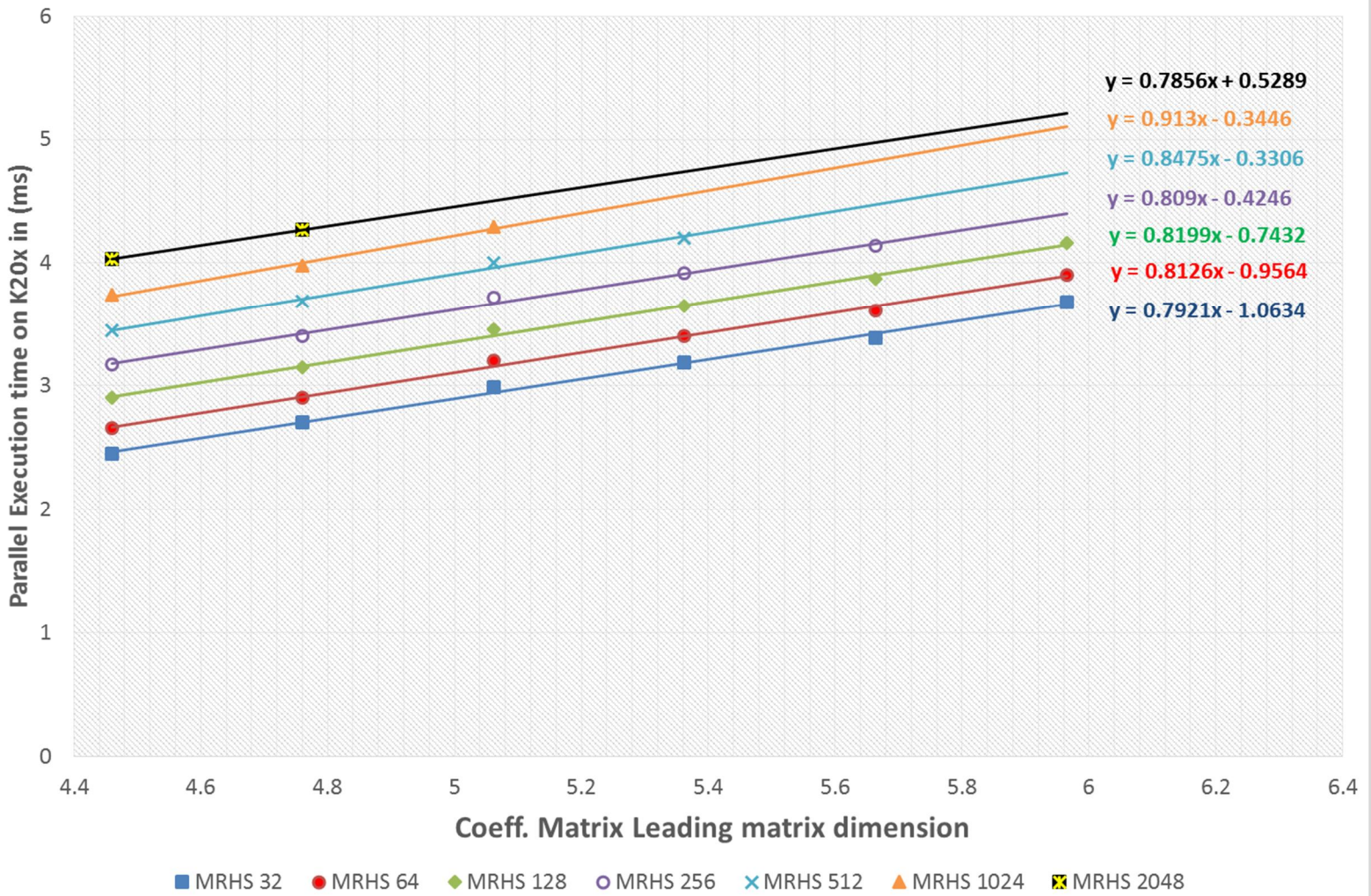


Figure 41: A double log plot for the average execution time of MRHS BiCGStab solver for various matrix dimensions and different MRHS widths.

The developed implementation of our BiCGStab_MRHS is composed of several kernels, shown in the appendix. The following Table 15, shows the count of their multiply-add operations as well as the computed GFLOPS/s for sample matrices extracted from the reservoir, while Figure 42, plots the computed GFLOPS/s for various dimensions. Without loss of generality, consider the kernel shown in Figure 39 that computes alpha. Besides the obvious count for multiplication and addition operations, the kernel computes a reduction task that has been shown to have an order of $(N \log(N))$ operations!

Table 15: Performance FLOPS for the kernels constituting the BiCGSTAB merged implementation

N $\times 10^3$	compute_p			compute_alpha			compute_S			compute_X		
	Op. Cont $\times 10^3$	Time (s)	GFLOP /s	Op. Cont $\times 10^3$	Time (sec)	GFLOP /s	Op. Cont $\times 10^3$	Time (sec)	GFLOP/ sec	Op. Cont $\times 10^3$	Time (sec)	GFLOP /s
2,7	40,5	2.85	0.04	32,4	1.95	0.04	37,8	3.79	0.03	72,9	3.91	0.05
5,4	86,4	3.42	0.14	70,2	2.35	0.16	81,0	4.44	0.10	156,6	4.76	0.18
10,8	183,6	4.94	0.40	151,2	3.61	0.45	172,8	5.72	0.33	334,8	6.39	0.57
21,6	388,8	6.62	1.27	324	5.67	1.23	367,2	9.32	0.85	712,8	9.16	1.68
43,2	820,8	4.97	7.14	691,2	2.57	11.62	777,6	8.13	4.13	151,2	6.47	10.1
86,4	172,8	6.86	21.77	146,8	2.93	43.30	164,1	10.79	13.15	319,6	9.38	29.4

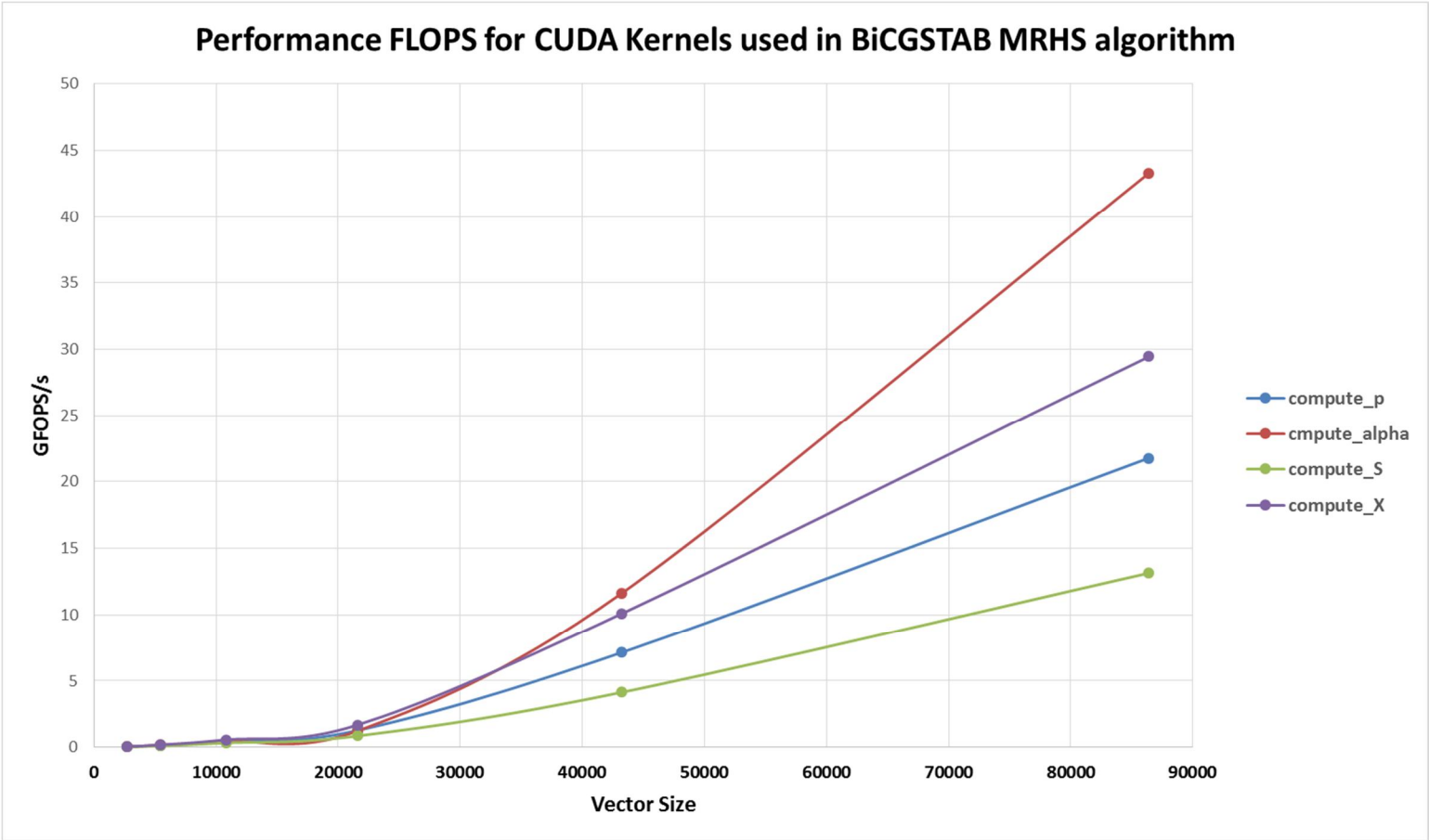


Figure 42: GFLOPS/s for the kernels used to program the BiCGSTAB merged for various matrix dimensions

Similar to the discussion in the previous section and as there is no reuse of loaded matrix elements, it is clear that the presented kernels feature computations that are memory and bandwidth bounded. The peak FLOPS of the device (1.31 teraflops for double precision) is much less than the results plotted in Figure 42. This is mainly because the measured execution time of each kernel is high. After all, GFLOPS/s is calculated as:

$$\frac{\text{GFLOPS}}{\text{s}} = \frac{\text{number of multiply or add operations} * \text{problem size}}{\text{Execution time} * 10^9}$$

The measured execution time shown in Table 15 was high because of the overhead associated with launching a kernel inside a kernel and managing the described earlier parent-child relation.

3.6.4 Concluding Remarks for this Section

Creating more parallelization opportunities by utilizing dynamic parallelism has been examined in light of implementing a parallel BiCGSTAB with multiple right hand sides MRHS. Such solvers play a key role in history matching applications and inverse problems in general. The utilized method is promising and can be further enhanced. Moreover, the same approach can be applied in the near future to other solvers like QMR and GMRES to compare performance.

CHAPTER 4

PARALLEL MODELING AND IMPLEMENTATION OF FORWARD RESERVOIR SIMULATION

4.1 The Parallel Model

The goal of parallel programming is to provide tools and techniques for either solving big problems faster or to run larger instances of the given problem for the same time interval that was used to execute their serial counterpart. Exposing application concurrency refers to the art of breaking down the main problem into independent logical tasks³⁵ that could be later executed in parallel after mapping them to corresponding physical processing elements. It is then no wonder that restructuring the problem to exploit any available concurrency is indeed first mandatory step before implementing any serial algorithm using a suitable parallel programming environment. The process for finding concurrency starts by a decomposition step performed on program data and the associated tasks. It is followed by an analysis step where the decomposed parts are grouped, ordered, or share their own data.

³⁵ A task is a sequence of instructions that operate together as a group.

Just as various complex algorithms and software modeling techniques have emerged as a necessity for developing large sequential applications, large scale massively parallel programs are in more demand for either making use of such techniques or even developing new aiding tools. This could be attributed to the observed fact that the life cycle of a parallel program is very long, error prone, complex and requires special attention to the underlying hardware resources [116]. Although exposing program concurrency may be achieved by developing and analyzing the dependency graph that in turns may be constructed in many ways [117], those methods are suited to express concurrency of computationally expensive algorithms or small scale systems.

As our reservoir simulator is more complicated, we tend to utilize more elegant methods from the software engineering general-purpose UML modeling [118, 119] which essentially provides standard graphs to visualize the design of large scale systems and their associated relations. Throughout the development process, we have constructed several related and complementary diagrams that describe the whole system from various design viewpoints to eventually aid in understanding and analyzing the parallel program.

While the Activity Diagram represents the behavioral part of the system, Deployment Diagram, also called Topology or Collaboration Diagram, shows the structural aspect and demonstrates how software and hardware work together [120]. The Deployment Diagram is usually the first recommended step in the modeling of traditional large scale parallel applications [116, 121]. The Activity Diagram shows the execution flow of the processes and what actions are performed to achieve an ultimate goal. In the context of parallel application modeling, this diagram provides means of representing communication,

synchronization and computational operations[116, 119]. Sequence Diagram as well as Communication or Collaboration Diagram, are also utilized to add another perspective to the behavioral description of the system. While the Sequence Diagram depicts dynamic system elements as they interact overtime, Collaboration Diagram also shows how system components are spatially related [122].

A quick glance at our sequential implementation of the reservoir simulator reveals and in a broader sense a number of write after write [7, 102] data hazards for each flow calculation. The issue has been resolved by giving off some space in order to create independent tasks. Instead of having one variable location being updated sequentially, multiple copies of the same variable have been allocated with proper renaming. Moreover, by refereeing back to the computational model of the developed forward simulator, Figure 51 and Figure 52 in the appendix, one can establish the associated corresponding detailed Activity Diagram, Figure 43. Without loss of generality, the concurrent operations of flow calculations from north to south are shown in Figure 44.

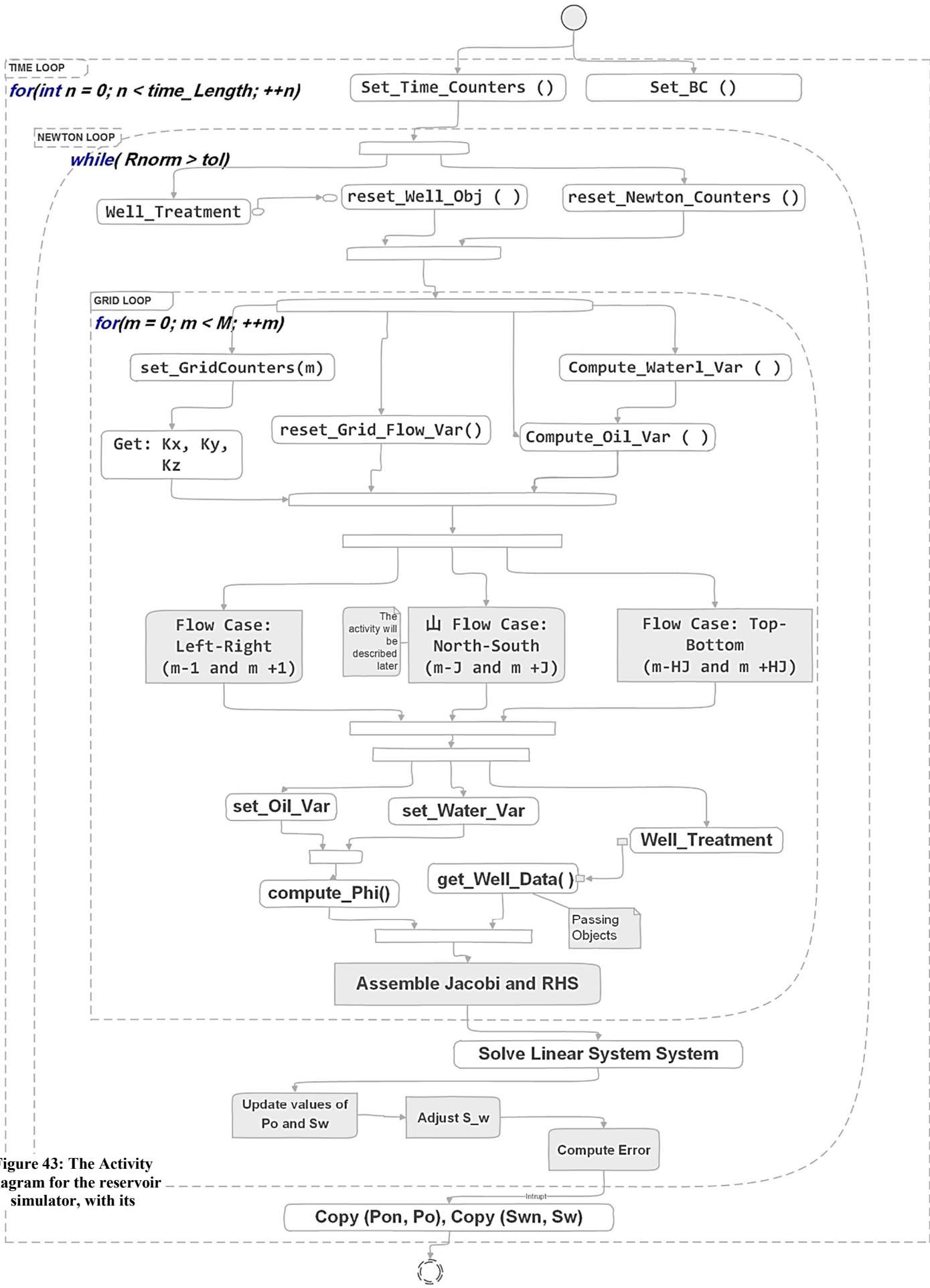


Figure 43: The Activity Diagram for the reservoir simulator, with its

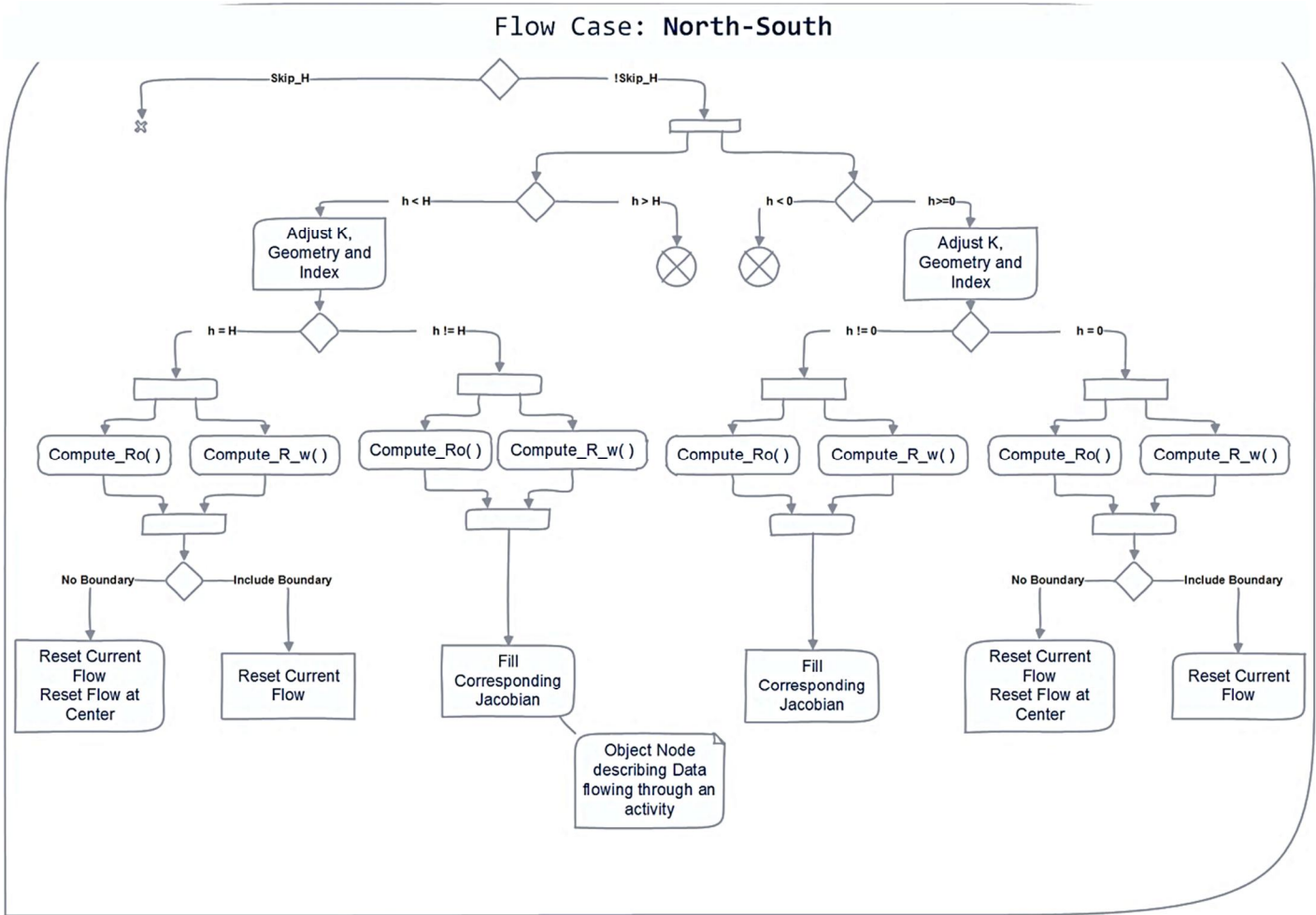


Figure 44: The Activity Diagram for a sample North-South flow calculation inside the Newton Iteration

As result, the following can be concluded about the matrix assembly stage³⁶:

- The system operates on large data structures. Basically large arrays that store $(P_o \ \&S_w, P_{on} \ \&S_{wn})$ values.
- Unlike the Newton and time loops, and if managed properly, the grid iterations are independent and do not carry dependency.
- The data portions of the arrays are read independently, for every flow direction.
- The update of the variables inside the array is done through multiple consecutive function calls.

The previous behavior and the established notes suggest that we start the parallelization process by data decomposition step over the large arrays and incorporate task decomposition whenever needed.

The process of data decomposition is about mapping a global index space into a task local index space [9]. It is associated with a granularity level³⁷ that determines the amount of data each chunk holds. The more the granularity gets smaller, the more independent tasks that are created and the more communication overhead to manage the dependencies among the resulting chunks is required. It has been suggested that a good data decomposition will poses the following characteristics [9]:

- It has to yield dependencies that scale at a lower dimension than the computational effort associated with each chunk; i.e. making chunks large enough so that the

³⁶ Before calling the linear solver

³⁷ A coarse-grained decomposition results in smaller number of large chunks which decrease communication overhead. A fine-grained decomposition, leads larger number of smaller chunks which facilitates load balancing and scheduling.

computational effort required to update data, offsets any resulted dependency overhead. Moreover, larger chunks will offer more flexibility when scheduling operations on the processors.

- Preserve load balancing among the execution elements. If not, then the speed at which the computation finishes will be haunted by the speed of the lowest process; i.e. the one with more work. This will be soon reflected on the overall performance that suffers as the problem being parallelized is scaled³⁸. After all, better scaling is achieved through the minimization data movement and reducing the serial bottlenecks³⁹ to the limit [6].

The analyzed concurrency pattern presents an additional force that influences the way tasks are mapped to processing elements. The simulator consists of multiple independent tasks⁴⁰ or weakly related tasks that share a common data structure as well as a sequence of tasks with a static and regular flow ordering pattern. When applicable the so called not true dependency was removed by suitable code transformations⁴¹. Moreover, a replication of the data structure was done when necessary. Whenever applicable, the whole program has been restructured to create more work with more potential concurrency. Also, optimized routines in Thrust library like reduction and their special data structure has been employed and utilized.

³⁸ This is achieved by either increasing resources or increasing problem dimensions.

³⁹ Such as exclusive-access mechanism such as locks, semaphores, or synchronization barriers

⁴⁰ In such a case, the focus will be on maximizing the efficiency of scheduling by ensuring load balancing

⁴¹ Some iterative expressions can be transformed into closed form expressions to remove any loop carried dependency.

Throughout the program execution, each function call can be thought of as a task⁴² which in turn may be composed of other tasks. Moreover, as the iterations in the most inner loop that spans all grid points are independent, each iteration, or even group of iterations, could be thought of as a separate task⁴³ that in turns operate on its assigned data portion. Again, the general rule of thumb lies in ensuring the creation of enough independent tasks that keep the processors busy. In CUDA terms, a global function will launch a number of thread blocks that handles specific portion of the input data. Threads in the associated blocks will then bring to shared or local memory necessary related data, calling any necessary device functions and operate on them.

The previous tasks could also be grouped in a way that makes it easier for managing dependency. The temporal dependency in the simulator loops puts further restrictions on data flow⁴⁴ and directly influences the way different tasks could be grouped. As mentioned earlier, the shared data arrays in (P_o & S_w), are solely read during matrix assembly stage. Before passing them to the next iteration they are modified and written back after solving the assembled ill conditioned unsymmetrical sparse linear system. The decomposed tasks utilize a shared data structure and their interaction is also synchronous as they occur at regular time intervals. Therefore, proper synchronization should also be introduced to avoid any race conditions.

⁴² In this case, this task decomposition is referred to as functional decomposition.

⁴³ This style of task-based decomposition leads to what is sometime called loop-splitting algorithms.

⁴⁴ This sequential flow could be exploited by pipelining.

Table 16, lists some utilized optimizations in the developed FRS code. More detailed information with examples could be found in [123, 124].

Table 16: List of utilized optimizations in the developed parallel FRS code

Target Optimizations	Details
Shared Memory Utilization	Intensive use of device shared memory and making use of its broadcast property to serve data among threads at a fast pace.
Titling	To handle large vectors, each thread at first load data into shared memory and performs the corresponding desired operation. It then stores the result back to global memory before another kernel take data accumulated in this new vector in global memory and continue operating on it.
Memory Coalescing	A warp can access a number of successive memory locations in a single transaction. Therefore, maximizing BW utilization.
Occupancy and Latency Hiding	Launching enough threads to keep resources busy.
Data Transfer	Minimize copying, and makes use of asynchronous data transfer between host and device by utilizing pinned memory and streams. Kepler GK110 introduces HyperQ mechanism that supports 32 hardware managed connections for communication between host and device. As a result, device

	utilization has been increased as multiple processors on the CPU could initiate work on a single GPU at the same time.
Overlap Communication and computation	Host and kernel execution overlap: when possible, the original code was restructured in a way that a call to device kernel is followed by a many calls to host functions. By default, kernel launch is asynchronous or non-blocking. So while the GPU is busy, the host computes part of the algorithm. If used properly, this mix, combined with streaming has great impact on performance.
Computation Intensity	Loop unrolling was utilized to further increase computation intensity.

4.2 Experiments and Comparisons

We implemented the previous described model and compare the obtained execution time with a serial version that makes use of Eigen library [85]. Correctness of results has been verified by comparing the output pressure values from the two programs for the given well distribution, see the Appendix for more details. Table 17, shows the execution time and the obtained speedup.

Table 17: The Execution time (ET) for serial and parallel FRS

Reservoir Dimension	X	Y	Z
	240	240	2
Coefficient Matrix Leading Dimension	230400		
Average Serial Execution Time (sec)	18693.25		
Average Parallel Execution Time (sec)	570.07		
Speed Up	33		

Next, Table 18 and Figure 43 demonstrate how the parallel execution time of the entire FRS varies when doubling reservoir dimension. The objective is two folded: First, to quantify the importance of the above obtained speedup shown in Table 17, and see what reservoir dimension is simulated in the same time used to produce results in the serial version. Second: to get an idea on how the developed parallel FRS scales when increasing problem size so that further optimizations could be implemented in subsequent work. For the sake of experimentations, only 25 wells were used.

Table 18: The parallel execution time of CUDA based FRS for various grid dimensions

Parallel Oil Reservoir Simulation											Conf. Coeff: 1.96							
Coeff. Mat. Leading Dim.	Parallel Execution Time(sec)										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10								
	10800	75.0	74.8	74.6	74.5	74.7	74.7	74.8	74.7	74.7								
28800	170.6	170.5	170.6	170.9	170.6	170.9	170.6	170.2	170.6	170.6	170.61	0.19	0.12	170.72	170.49	170.89	170.23	0.66
57600	259.4	258.6	258.9	258.9	258.7	259.0	258.7	258.7	259.0	259.0	258.89	0.24	0.15	259.04	258.74	259.41	258.57	0.84
115200	414.8	414.5	415.0	414.7	414.6	414.2	414.6	415.1	414.9	414.2	414.66	0.32	0.20	414.85	414.46	415.10	414.16	0.94
230400	570.3	570.2	569.5	570.7	570.0	569.6	570.9	570.1	569.9	569.6	570.07	0.47	0.29	570.35	569.78	570.92	569.50	1.42
460800	885.8	884.4	884.0	884.4	885.0	885.5	885.2	884.0	885.5	885.3	884.91	0.67	0.42	885.32	884.49	885.82	883.97	1.85
921600	1162.3	1165.9	1164.8	1165.9	1164.3	1165.5	1165.4	1164.2	1164.0	1165.2	1164.75	1.09	0.68	1165.43	1164.07	1165.93	1162.33	3.60
1843200	1879.6	1882.8	1881.5	1881.6	1882.5	1877.9	1881.9	1882.6	1881.0	1880.1	1881.14	1.53	0.95	1882.09	1880.19	1882.80	1877.94	4.86

In accordance with common observation on GPUs, data shows that the GPU simulation becomes more efficient with increasing model size. They reflect the fact that GPUs need a large amount of independent work to operate at maximum efficiency. The serial implementation of FRS took 311.55 minutes to solve a problem with 230,400 grids. On the other hand, the interpolated data from Figure 45, speculates that a problem with 18,873,402 grids could be solved in parallel in 311.55 minutes. In other words the CUDA parallel implementation of FRS enables solving an 82 times larger grid dimension, given the same time to produce results from the counterpart serial implementation.

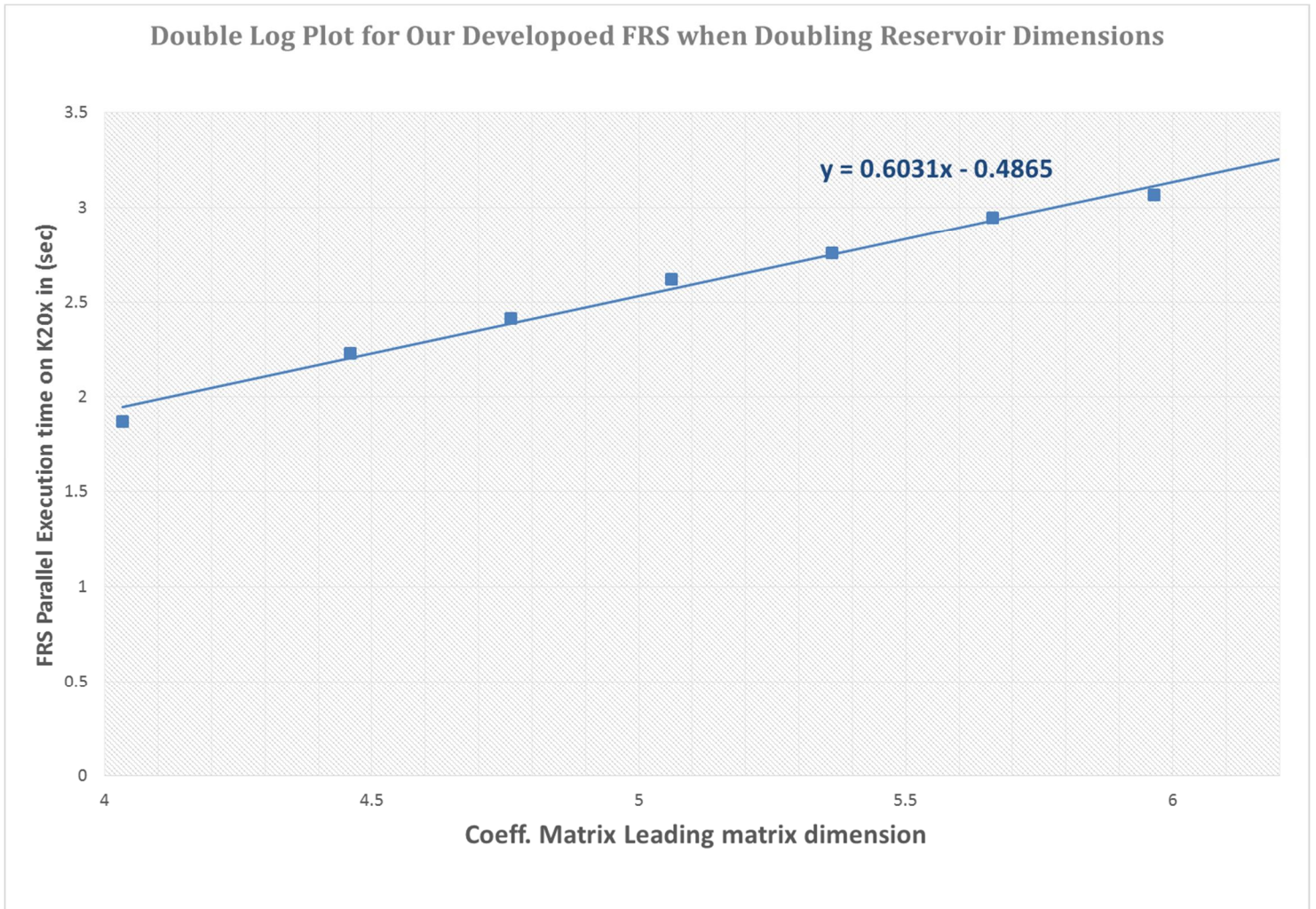


Figure 45: A double-log plot for the parallel execution time of our developed FRS for various geometries

4.3 The Parallel FRS Graphical User Interface (GUI)

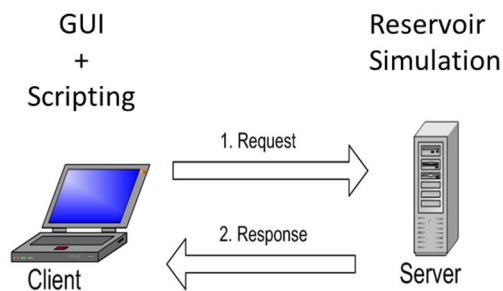
With the goal of deploying a real time version of the parallel simulator, a client-server application that is suitable for such heterogeneous configuration has been developed. The following technologies have been utilized:

Client Side

- HTML5, CSS, Java Script,
- (Shiny): A web application framework for R⁴⁵.

Middle Layer (R + scripting to communicate with the Server)

Server Side (Simulation Program (C++ and CUDA)).



The GUI enables basic control like setting reservoir dimensions as well as loading some configuration files. Following are some snapshots.

⁴⁵ <http://shiny.rstudio.com/>

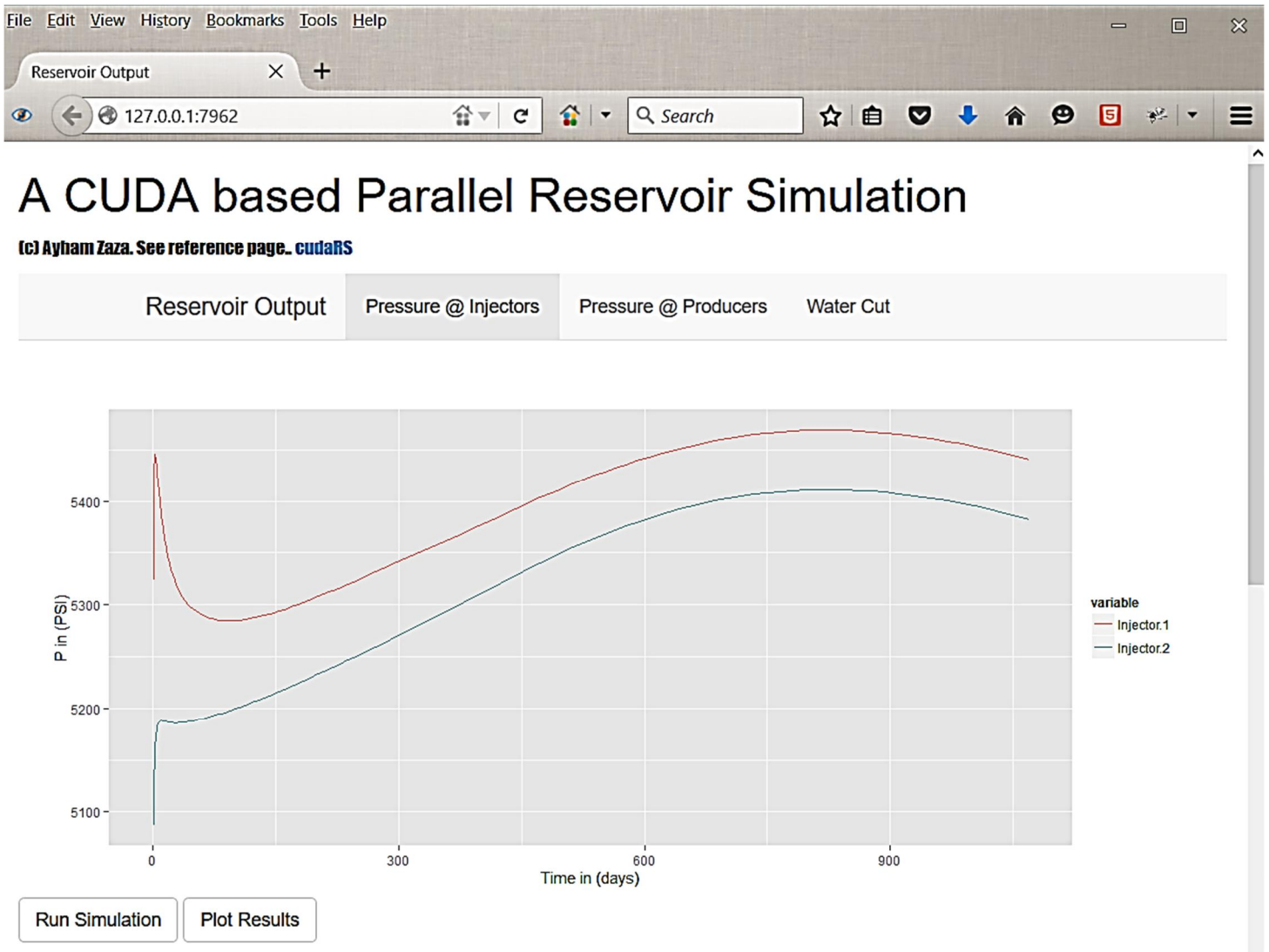
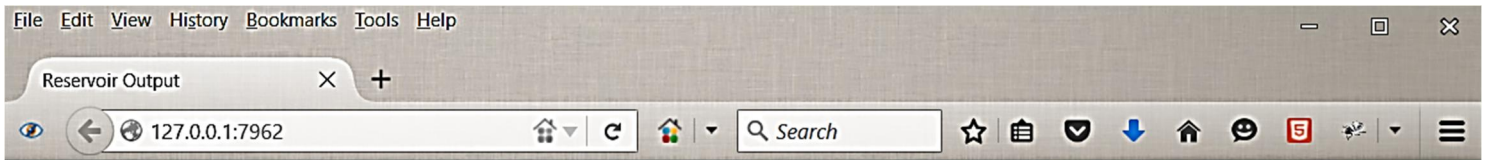


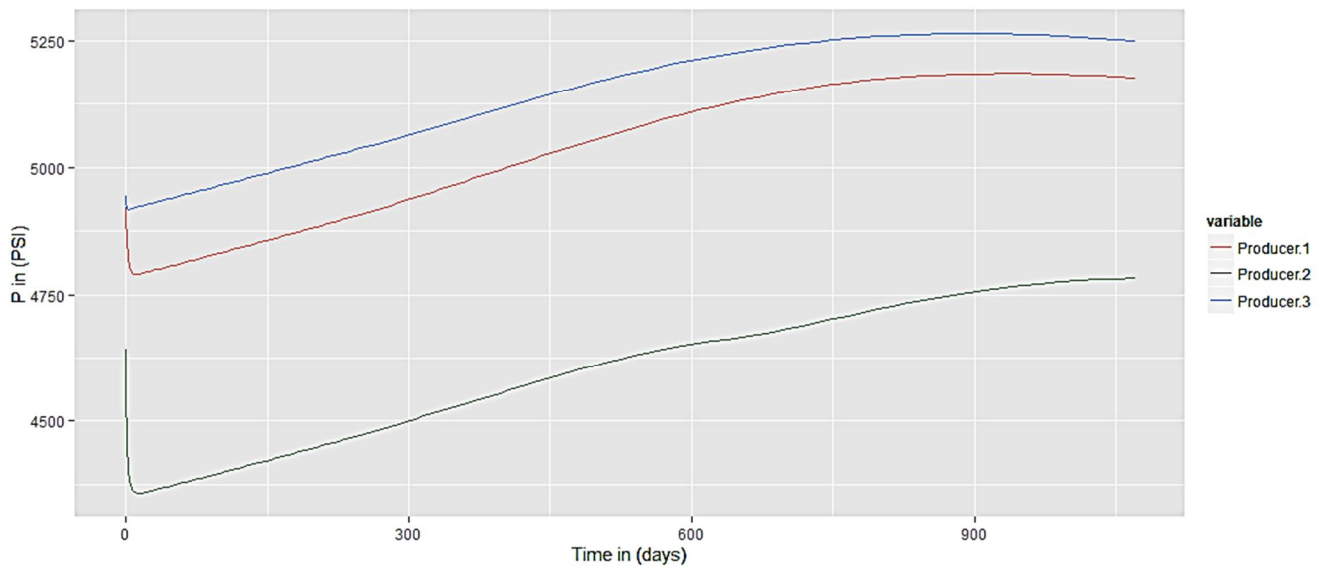
Figure 46: GUI Snapshot showing the resulting pressure at Injectors



A CUDA based Parallel Reservoir Simulation

(c) Ayham Zaza. See reference page. [cudaRS](#)

Reservoir Output Pressure @ Injectors **Pressure @ Producers** Water Cut



Run Simulation Plot Results

Figure 47: GUI Snapshot showing the resulting pressure at Producers

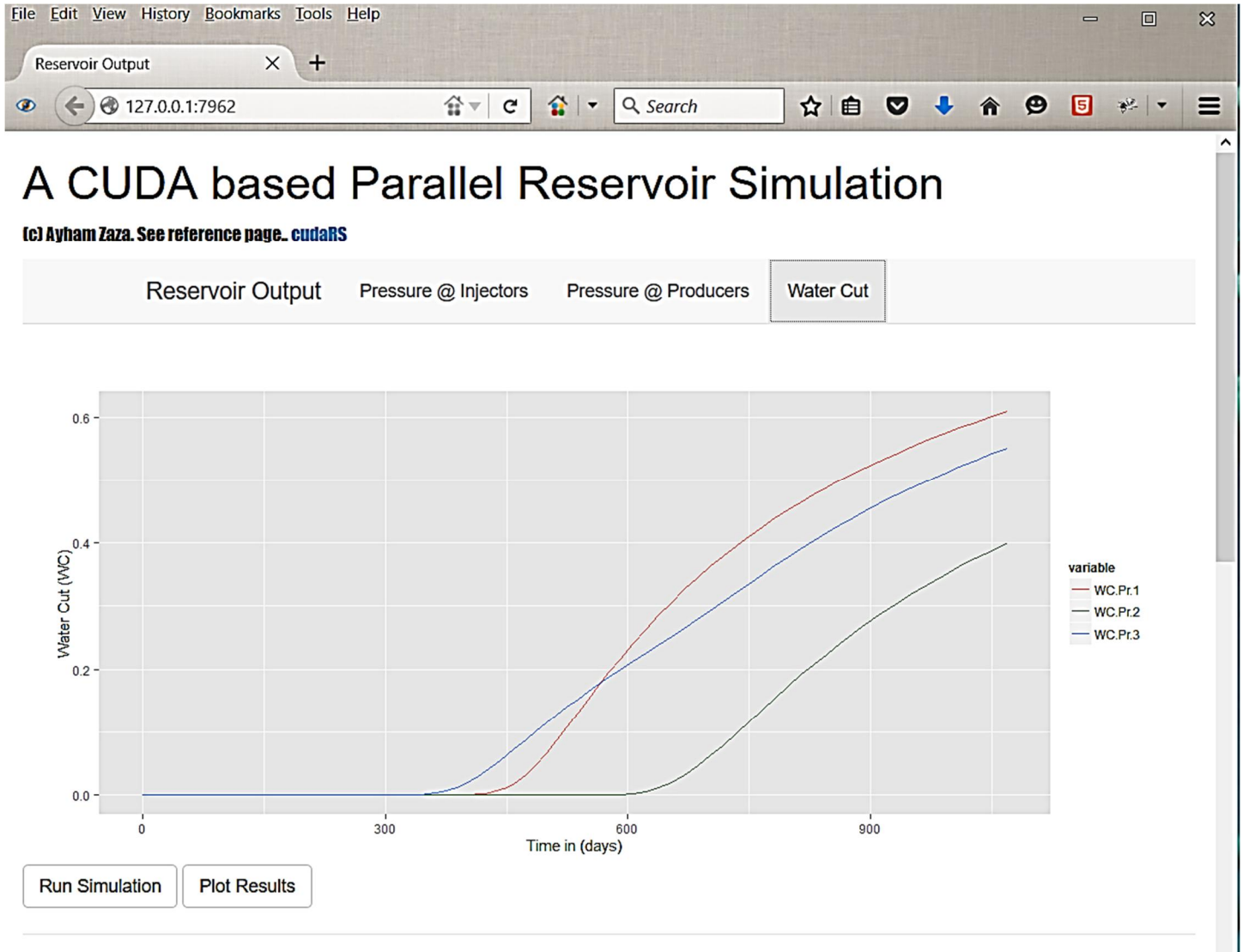


Figure 48: GUI Snapshot showing water cut values

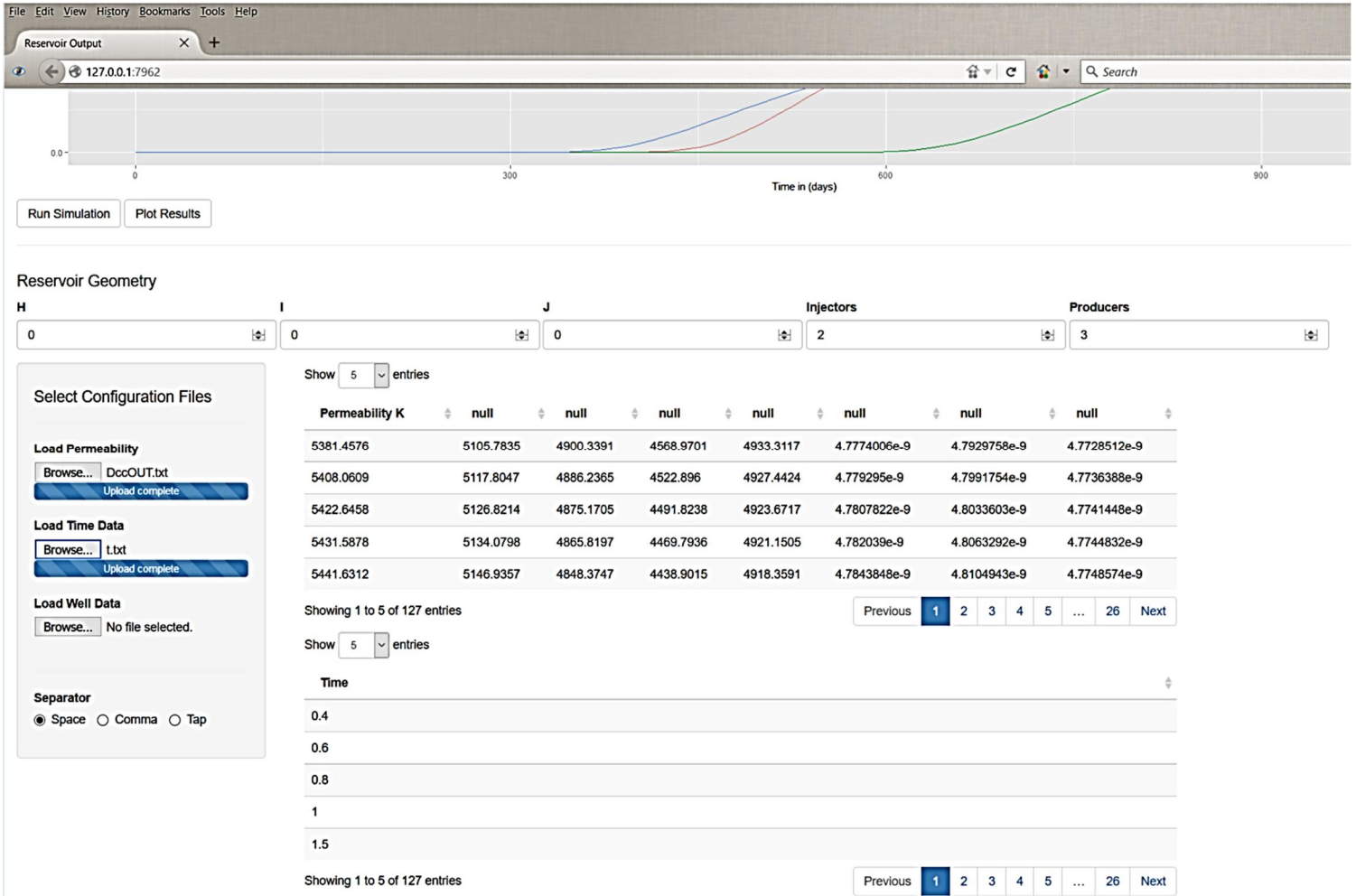


Figure 49: GUI Snapshot showing how data is loaded into the system

4.4 Concluding Remarks and Future Work

This work has studied and implemented a CUDA based parallel implementation for a flexible, two phase, 3D Forward Reservoir Simulation (FRS), and reviewed all related issues. Results show that CUDA parallel implementation of FRS enables solving an 81 times larger problem than the serial counterpart. Moreover, if accompanied by proper preconditioning, BiCGSTAB was shown to be a stable solver that could be incorporated in such simulations instead of the costly GMRES. This work is a founding stone for many interesting work to come. Future work includes imposing further optimizations on the CUDA program, MIC implementation, utilizing Multigrid preconditioners, OpenACC comparison, trying different solvers like QMR and others.

This page was intentionally been left blank

Appendix A

Work Completed Under Directed Research

A.1 Computational Model for Reservoir Simulation

The goal of Forward Reservoir Simulation (FRS) is to model fluid flow and mass transfer in porous media to eventually draw conclusions about the behavior of certain flow variables and well responses. Starting with initial values for pressure and saturation together with other reservoir parameters, (FRS) eventually produces new enhanced values of those state variables (P_0 and S_w) at different time steps given the initial reservoir properties Figure 50



Figure 50: General Scheme for Forward Reservoir Simulation

Figure 51, presents a general description of our developed FRS model that is utilized later to introduce the computation model. The Forward Model consists of three main iterations Figure 52, namely L1, L2 and L3 and optionally a fourth one L4.

- ***The outer most (Loop L1):*** is the temporal loop which repeats the simulation for different time steps that are usually measured in days.
- ***The middle iteration (Loop L2):*** is Newton iteration that achieves the linearization. During this iteration the resulted sparse linear system of the form $Ax = b$ is solved.
- ***The most inner one (Loop L3):*** is the spatial loop that visits all system grid cubes and form the corresponding non-linear system to be linearized, solved and refined during the middle iteration (L2)
- ***Optional (Loop L4):*** this loop is available if iterative methods are used to solve the linear system. Generally speaking, iterative methods are favored over direct methods for large sparse linear systems, because of their computational and storage efficiency. More details were presented in the survey in the preceding section.

In previous iterations, L2 accounts for around 67% of the computational complexity in the whole forward modeling process. After the discretization step, the system of non-linear algebraic equations for each phase is then written in terms of its corresponding residual equation R_o & R_w . The famous Newton Iteration achieves system linearization by repeatedly refining a nearby obtained approximation after solving a linear system with the Jacobian as the coefficient matrix. The Jacobian is obtained for each phase by deriving the

residual equation with respect to both P_o & S_w at each grid point and all its neighbors. It is worth mentioning that the condition number for the assembled Jacobian matrix ranges from around $1.279E+05$ in the beginning of the simulation time and reaches $4.708E+06$ at the end.

Forward Reservoir Simulation (Two phase oil-water model)

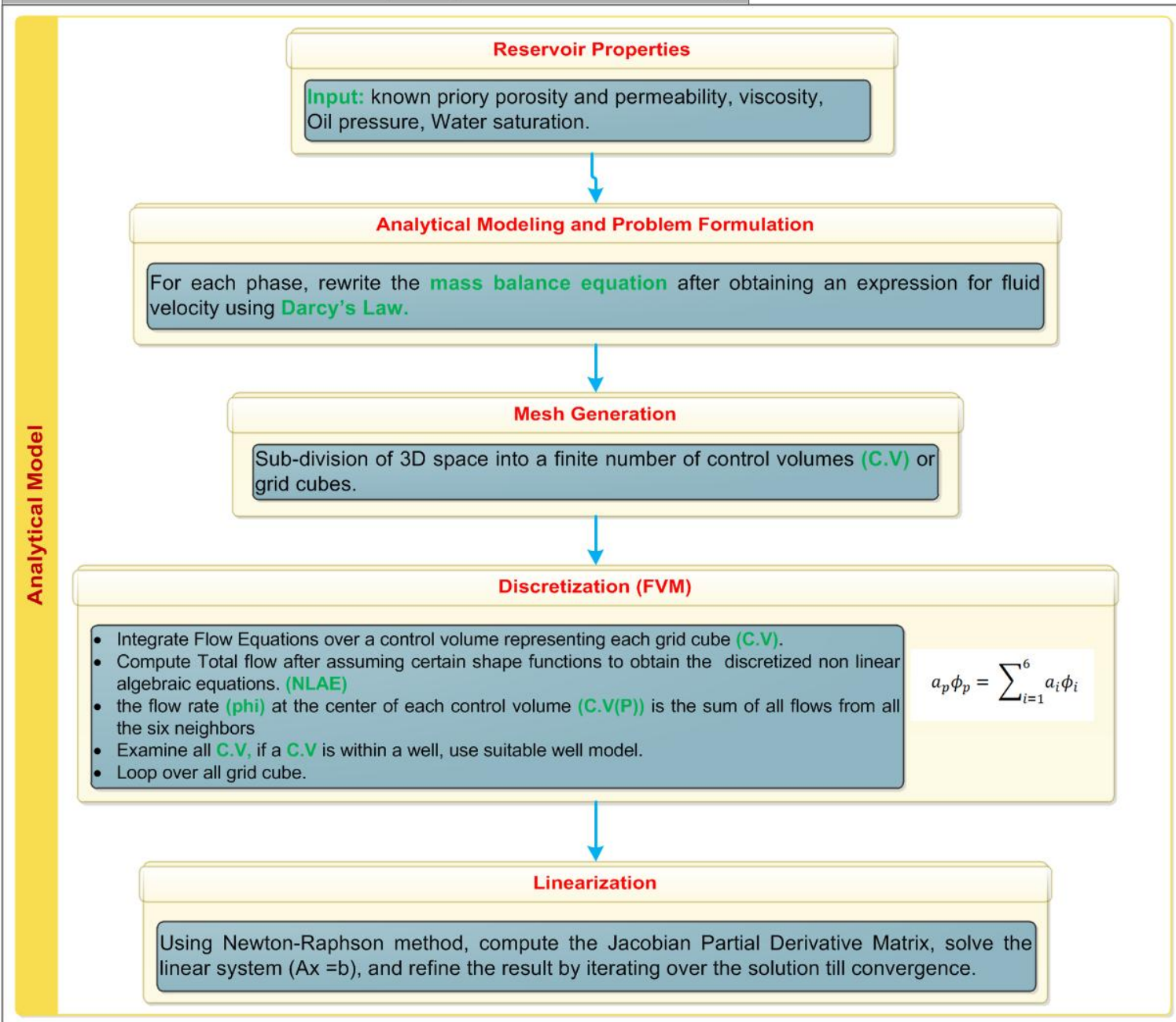


Figure 51: General Description for the Forward Reservoir Simulation Model

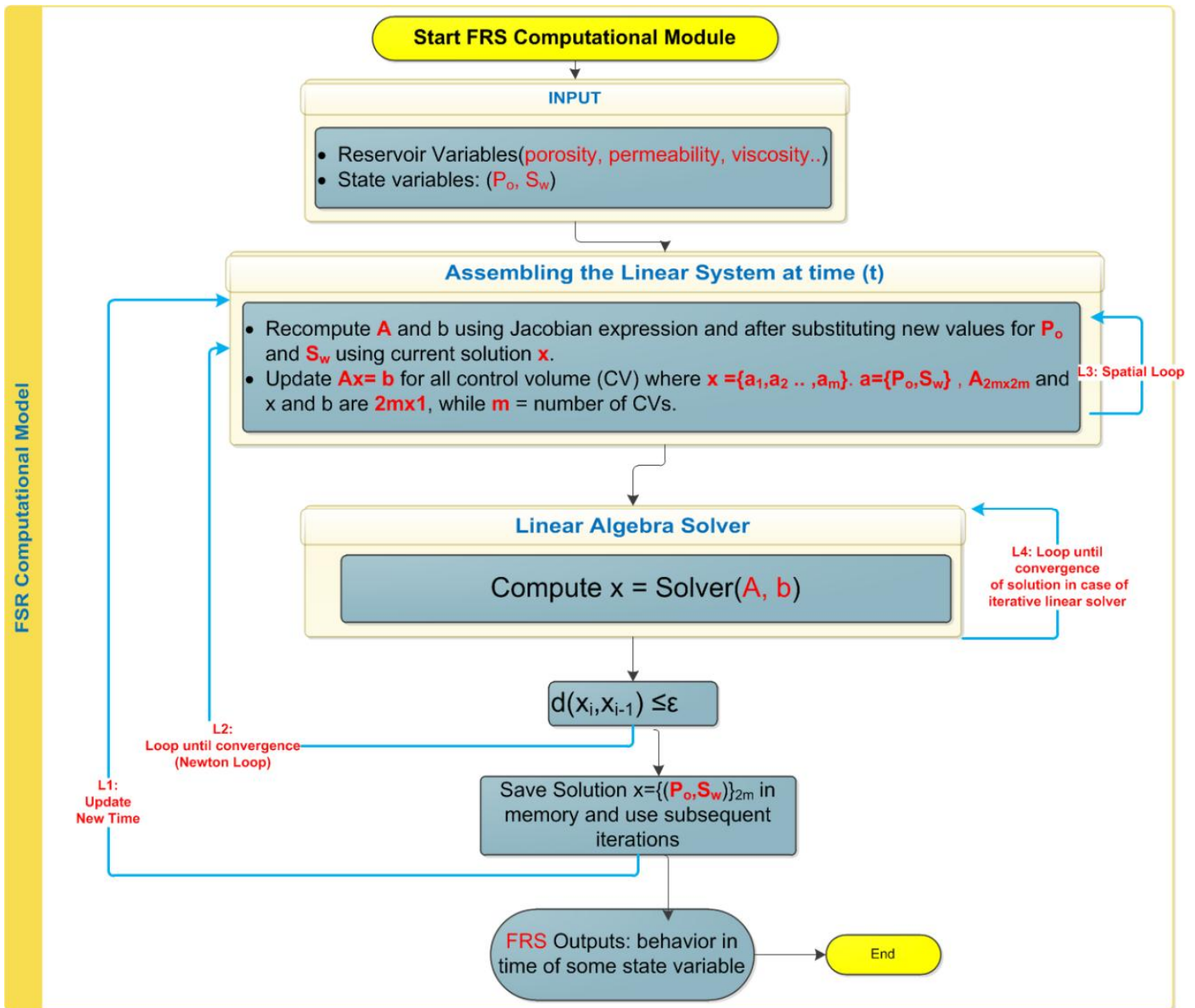


Figure 52: General Computational Scheme for the Forward Oil-Black model: When assembling the linear system. All grid points are visited. Newton Iteration repeatedly solves the system of linear equations formed in the grid iteration

A.2 Validating Reservoir Results

Validating the correctness of parallel program output was done in two stages. First, an already verified MATLAB code developed by [19, 20] that utilized a direct solver was compared against the implemented serial C++ program for small grid dimensions (20 x30x2). No flow boundary condition was initially assumed, six injectors with specified water rate and seven producers with specified total rate were utilized. The distribution of the wells is shown in Table 19, while Figure 53, shows the permeability map with the distribution of wells shown on the map.

Table 19: Well distribution for both the producer and the injector over grid space of (20 x 30 x 2)

X-Coor	Y-Coor	Z-Coor	Stb/day	P limit Psi
1	1	1	-550	7000
10	1	1	-850	7000
5	5	1	550	2000
1	10	1	350	2000
10	10	1	600	2000
1	20	1	-550	7000
10	20	1	-850	7000
5	15	1	500	2000
15	5	1	600	2000
20	1	1	-550	7000
20	10	1	650	2000
15	15	1	600	2000
20	20	1	-550	7000

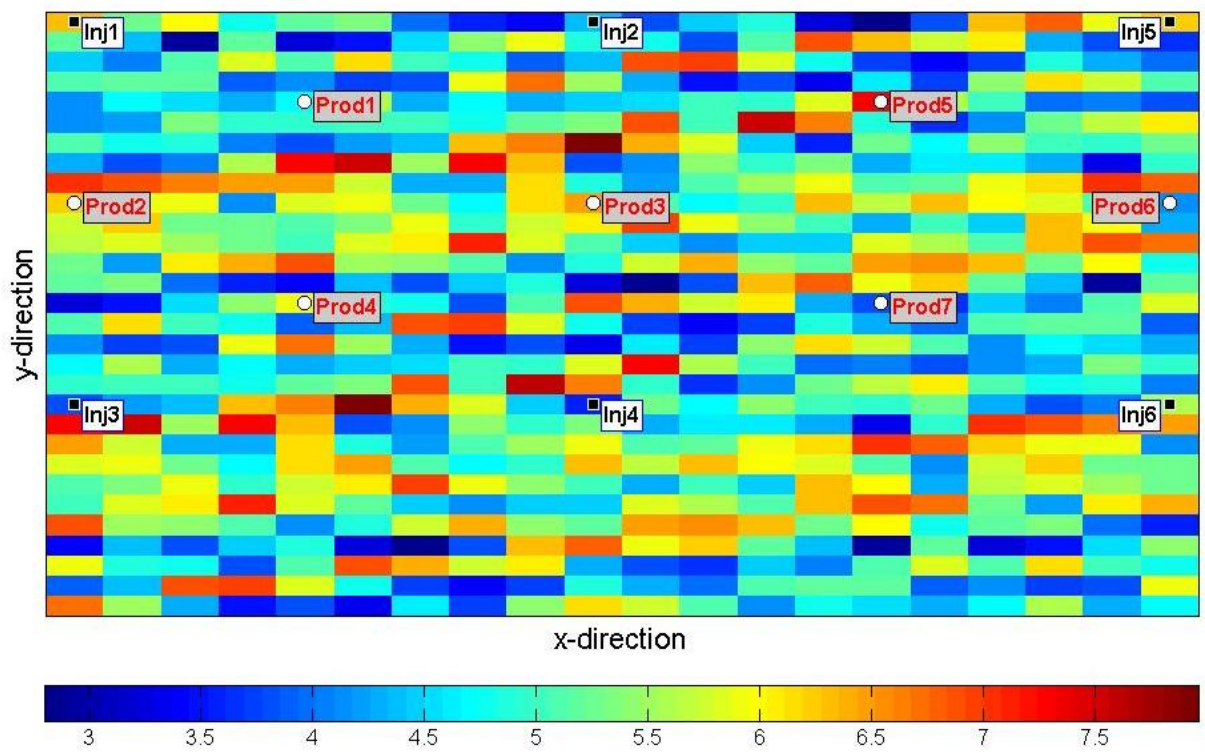


Figure 53: Permeability map for the utilized wells shown in Table 19

Figure 54 and Figure 55 demonstrate the two versions for the running simulator when the effect of capillary pressure is included and plot P_{wf} for the injectors and producers against a similar configuration where capillary was not included. Six injectors with specified water rate and seven producers with specified total rate were utilized. Next Figure 56 and Figure 57 show the running simulator when the constant pressure boundary condition is assumed from certain directions (m-HJ, m-J) with a value of 5000, no flow boundary condition is assumed for all other directions. Again six injectors with specified water rate and seven producers with specified total rate were utilized.

In the second verification phase we consider larger grid dimensions (240 x 240 x 2) and test the serial C++ code against our developed parallel version. As mentioned before, the serial version uses Egien library to provide implementation of the BiCGSTAB solver and the ILU preconditioner while our parallel program utilizes a program we wrote for BiCGSTAB code based on various related cuSPARSE and cuBLAS library calls.

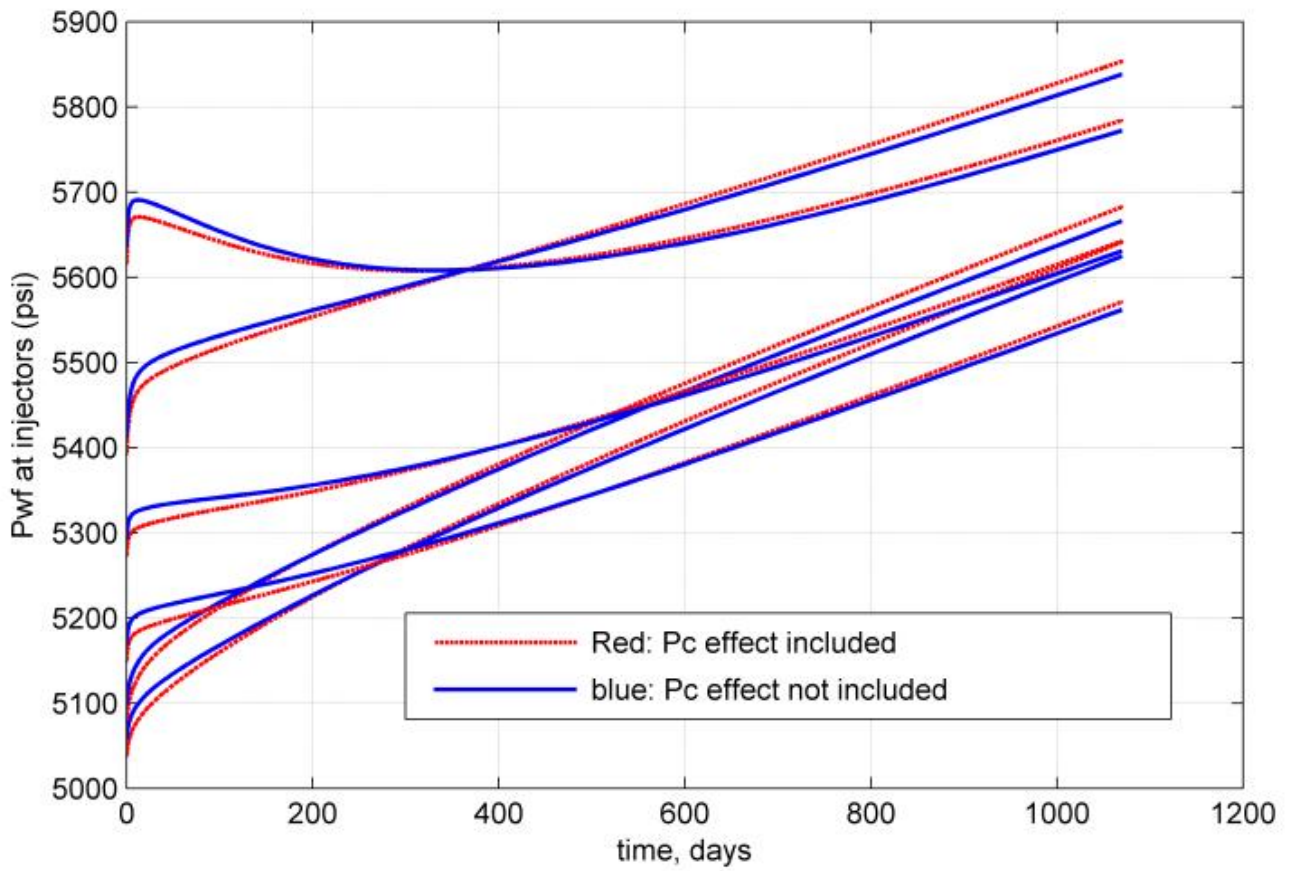


Figure 54: Pwf at Injectors, Pc is included, No Flow BC for 20*30*2, specified flow rate at injector

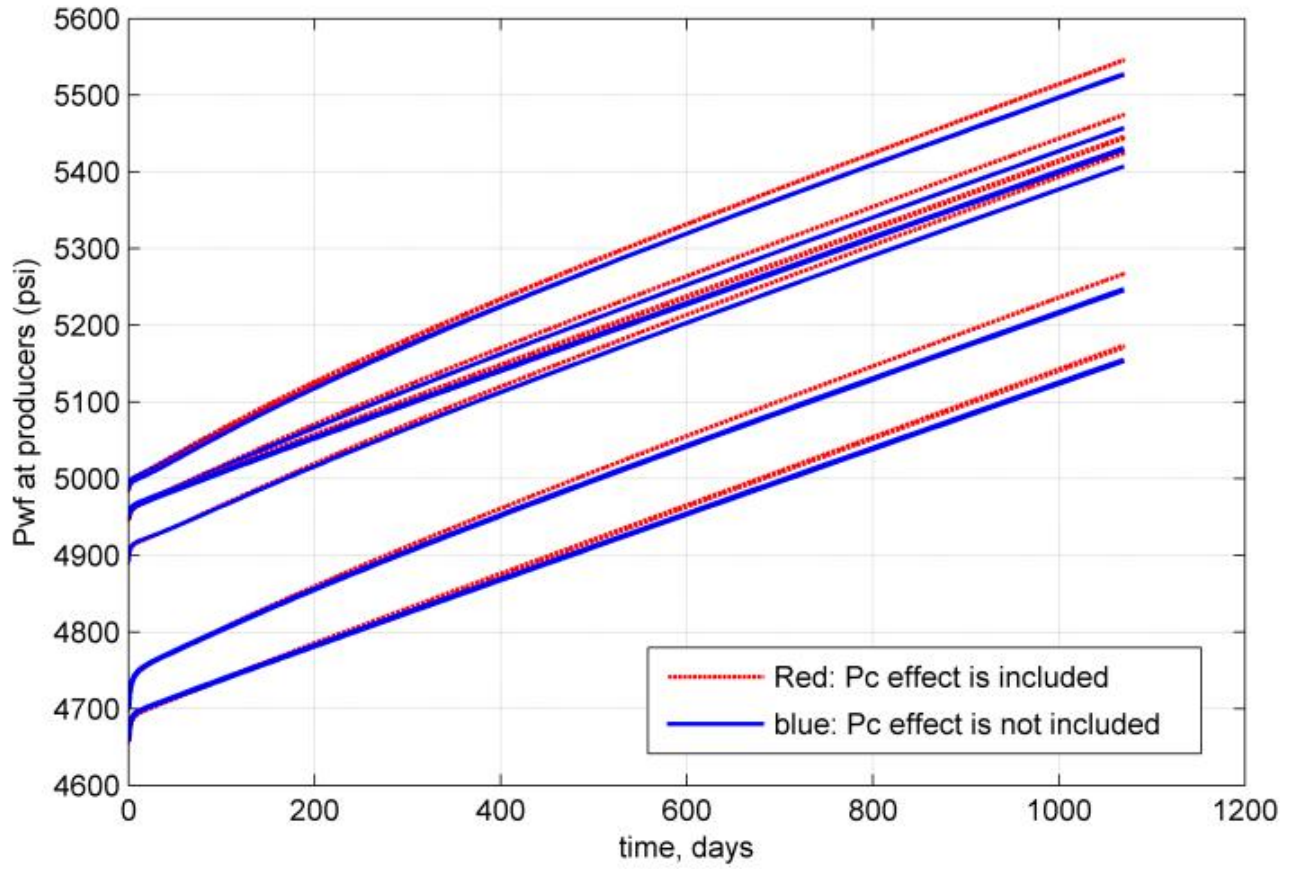


Figure 55: Pwf at Producers, Pc is included, No Flow BC for 20*30*2, specified total rate at producer

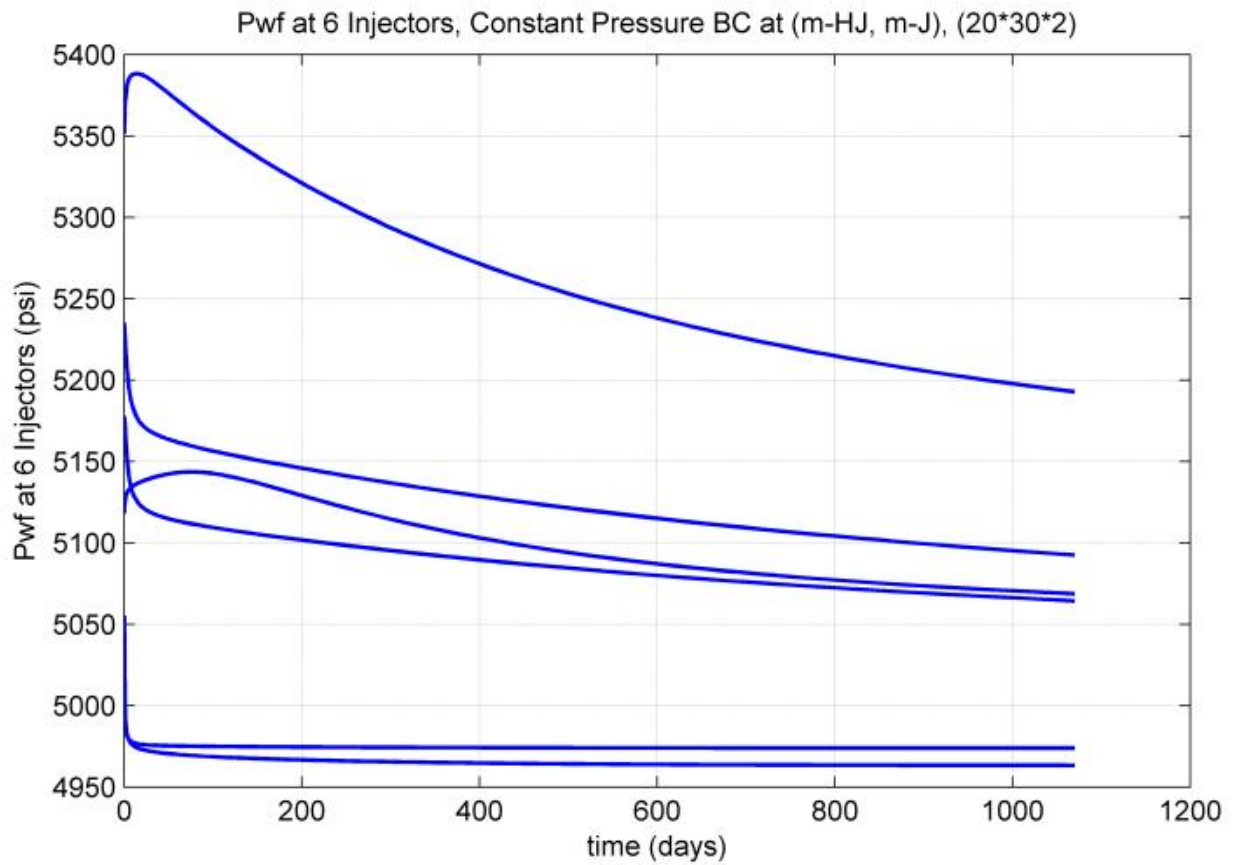


Figure 56: Pwf at Injectors. Constant Flow BC (5000psi) at m-J and m-HJ, No Flow BC for the rest. Water-oil reservoir of dimensions (20*30*2) and specified flow rate at 6 injectors

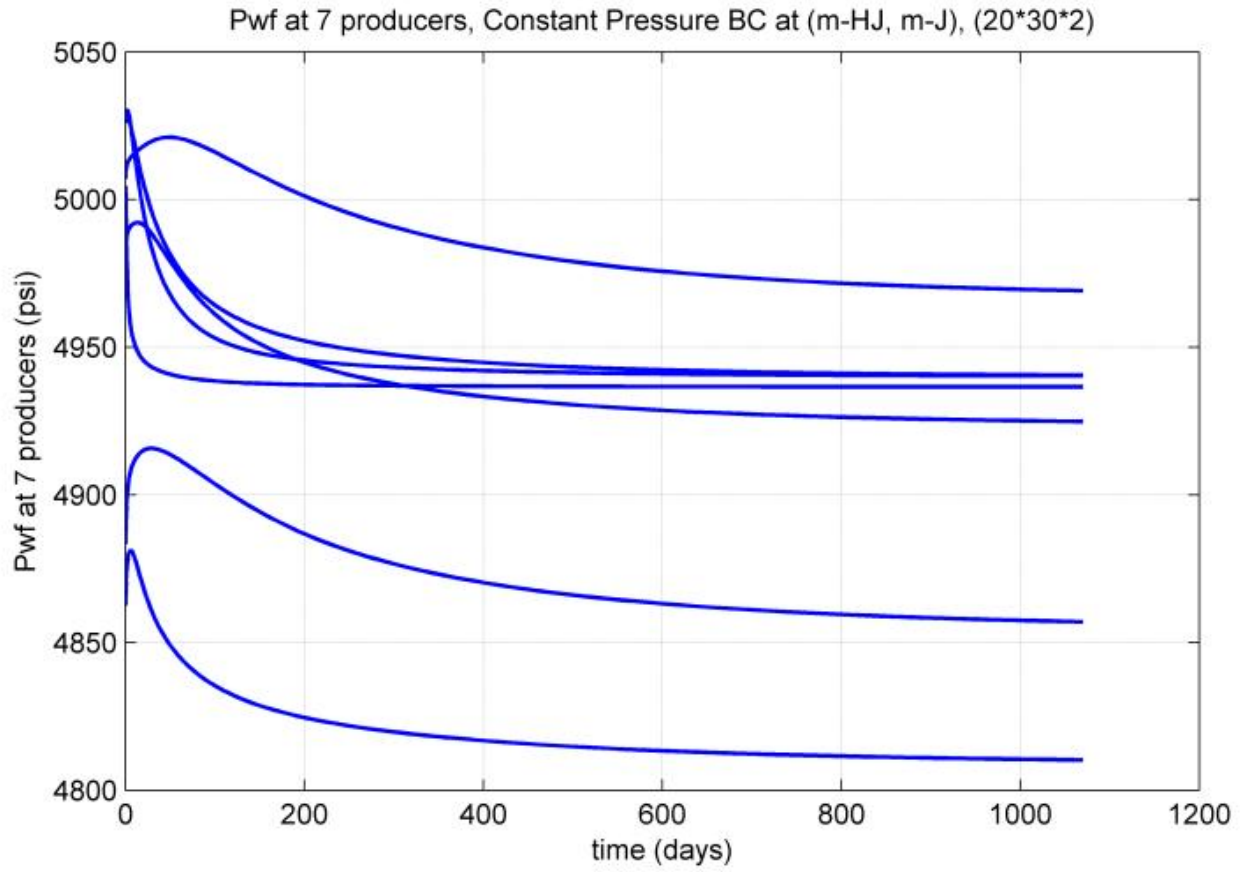


Figure 57: Pwf at Producers. Constant Flow BC (5000psi) at m-J and m-HJ, No Flow BC for the rest. Water-oil reservoir of dimensions (20*30*2) and specified total rate at 7 producers

Appendix B

CUDA Kernels Utilized in This Work

B.1 BiCGSTAB Merged Implementation

```
__global__ void reduced_Omega(double *vector_Neum, double
*vector_Deno, double *alpha_phat, double *x, double
*s_hat, double *t, double *r, double *s, int data_size ){

    __shared__ double Inter_Blck_Neum[blocksPerGrid];
    __shared__ double Inter_Blck_Deno[blocksPerGrid];

    __shared__ double shared_Omega;

    unsigned int Index = threadIdx.x;
    int tid = threadIdx.x + blockIdx.x * blockDim.x; //
global thread ID

    if( Index < blocksPerGrid){

        Inter_Blck_Neum[Index]= vector_Neum[Index];
        Inter_Blck_Deno[Index]= vector_Deno[Index];

        __syncthreads();

        UnrolledBlockReduce(Index, Inter_Blck_Neum,
Inter_Blck_Deno,blocksPerGrid);

    }

    __syncthreads(); // make thread 0 waits all others

    if(Index == 0){

        shared_Omega =
Inter_Blck_Neum[0]/Inter_Blck_Deno[0]; //broadcast from
shared memory
    }
}
```

```

        global_Omega = shared_Omega;
    }

    __syncthreads(); //make all threads, wait for thread
zero to come

    double omega = shared_Omega;

    while (tid < data_size){

        x[tid] = x[tid] + alpha_phat[tid] +
omega*s_hat[tid];
        r[tid]= s[tid]- omega*t[tid];

        tid += blockDim.x * gridDim.x;
    }
}

```

```

_global__ void per_Block_Omega(double *t,double *s, double
*vector_Neum, double *vector_Deno,double *alpha_phat,
double *r, double *s_hat, double *x ,int data_size){

    __shared__ double
Intra_Blck_Omega_Neu[threadsPerBlock];
    __shared__ double
Intra_Blck_Omega_Deno[threadsPerBlock];

    double current_t=0;

    int tid = threadIdx.x + blockIdx.x * blockDim.x; //
global thread ID
    unsigned int Index = threadIdx.x;

    Intra_Blck_Omega_Neu[Index] = 0;
Intra_Blck_Omega_Deno[Index] = 0;
    /* omega = ( t'*s) / ( t'*t ) */

    while (tid < data_size ){

        current_t = t[tid];

        Intra_Blck_Omega_Neu[Index] += current_t * s[tid];
        Intra_Blck_Omega_Deno[Index]+= current_t *
current_t;

        tid += blockDim.x * gridDim.x;
    }

    __syncthreads();

    if( Index < blocksPerGrid){
        UnrolledBlockReduce(Index, Intra_Blck_Omega_Neu,
Intra_Blck_Omega_Deno,threadsPerBlock);
    }
    __syncthreads();
//Write the resulted per block reduced rho to global
memory
    if (0 == Index) {

        vector_Neum[blockIdx.x] = Intra_Blck_Omega_Neu[0];
        vector_Deno[blockIdx.x] = Intra_Blck_Omega_Deno[0];
    }
}

```

```

__global__ void per_BLK_alpha(double *r_tld, double *v,
double *vector_rtld_v,int data_size) {

    __shared__ double Intra_Blkc_rtld_v[threadsPerBlock];

    int tid = threadIdx.x + blockIdx.x * blockDim.x; //
global thread ID
    unsigned int Index = threadIdx.x;

    double current_rtld_v =0; // r_tld[i]*v[i]

    while (tid < data_size ){

        current_rtld_v += r_tld[tid] * v[tid];
        tid += blockDim.x * gridDim.x;
    }

    if(Index < threadsPerBlock ){
        Intra_Blkc_rtld_v[Index] = current_rtld_v;
        __syncthreads();

        UnrolledBlockReduce(Index,
Intra_Blkc_rtld_v,threadsPerBlock);
    }
    __syncthreads();

    //Thread 0 from each block will write the resulted per
block reduced rho to global memory
    if (Index == 0 ) {
        vector_rtld_v[blockIdx.x] = Intra_Blkc_rtld_v[0];
    }
}

```

```

__global__ void compute_S(double *r, double *v, double *s,
double *p_hat, double *alpha_hat, double *vector_S
, double *global_Alpha, int data_size) {

    int tid = threadIdx.x + blockIdx.x * blockDim.x; //
global thread ID
    unsigned int Index = threadIdx.x;
    __shared__ double alpha_sh;

    if(Index == 0) alpha_sh = *global_Alpha;

    __shared__ double Intra_Blck_S[threadsWithinBlock]; // for
reduced S value
    double s_square = 0, s_value = 0;

    __syncthreads();

    double local_Alpha = alpha_sh;

    while (tid < data_size ){
        s_value = r[tid] - local_Alpha*v[tid] ; //
s[tid] = r[tid] - global_Alpha * v[tid];
        s_square += s_value *s_value;
        s[tid] = s_value;
        alpha_hat[tid] = local_Alpha*p_hat[tid];

        tid += blockDim.x * gridDim.x;
    }
    __syncthreads();

    if(Index < threadsPerBlock ){
        Intra_Blck_S[Index] = s_square;
        __syncthreads();

        UnrolledBlockReduce(Index,
Intra_Blck_S, threadsPerBlock);
    }

    __syncthreads();

    //Thread 0 from each block will write the resulted per
block reduced rho to global memory
    if (Index == 0 ) {
        vector_S[blockIdx.x] = Intra_Blck_S[0];
    }
}

```



```

__global__ void per_BLK_Rho_Beta(double *r_tld, double *r,
double *vector_Beta, double *vector_rho, double
*global_Alpha, double *global_rho1 ,int data_size) {

=====
//      INPUT:
//          r_tld, r: to perform dot product
//          cons_vec[4]: (rho_1, alpha, omega,
data_size )
//          -----
//          -----
//      OUTPUT:
//          vector_Beta, vector_rho: contains per
block reduced values of beta and rho
//
=====

unsigned int Index = threadIdx.x;
__shared__ double shared_Constants[3];

if(Index == 0 ){
    shared_Constants[0]= *global_rho1;
}
if(Index == 32 ){
    shared_Constants[1]= *global_Alpha;
}
if(Index == 64 ){
    shared_Constants[2]= global_Omega;
}
__syncthreads();

//Allocating shared memory for intra (within) block
reduction: Intra_Blck
__shared__ double Intra_Blck_rho[threadsWithinBlock];
__shared__ double Intra_Blck_Beta[threadsWithinBlock];

double rho_1 = shared_Constants[0]; double alpha =
shared_Constants[1]; double omega = shared_Constants[2];
double current_rho=0;

int tid = threadIdx.x + blockIdx.x * blockDim.x; //
global thread ID
Intra_Blck_rho[Index] = 0; Intra_Blck_Beta[Index] = 0;

```

```

while (tid < data_size ){

    current_rho = r_tld[tid] * r[tid]; // partial rho:
rho_0, rho_1, rho_2

    Intra_Blkc_rho[Index] += current_rho;
    Intra_Blkc_Beta[Index] += (current_rho/ rho_1) *
(alpha / omega);

    tid += blockDim.x * gridDim.x;
}
__syncthreads();

if(Index < threadsPerBlock ){
    UnrolledBlockReduce(Index,
Intra_Blkc_Beta, Intra_Blkc_rho, threadsPerBlock);
}

__syncthreads();

//Thread 0 from each block will write the resulted per
block reduced rho to global memory
if (Index == 0 ) {

    vector_Beta[blockIdx.x] = Intra_Blkc_Beta[0];
    vector_rho[blockIdx.x] = Intra_Blkc_rho[0];
}
}

```

```

__global__ void compute_P(double *p, double *r, double
*r_tld, double *v, double *vector_Beta, double
*vector_rho,int data_size){

    int tid = threadIdx.x + blockIdx.x * blockDim.x; //
global thread ID
    unsigned int Index = threadIdx.x;

    __shared__ double omega;
    if(Index ==0){
        // let th0 of every block brings omega and share it
with threads in a block
        omega =global_Omega;
    }

    // step_1: Bring vector beta to shared memory
    __shared__ double Inter_Blck_Beta[blocksPerGrid];
    __shared__ double Inter_Blck_Rho[blocksPerGrid];

    if( Index < blocksPerGrid){
        // very optimal if blocks is 32 as it will give
only one memory transaction
        Inter_Blck_Beta[Index]= vector_Beta[Index];
        Inter_Blck_Rho[Index]= vector_rho[Index];
    }
    __syncthreads();

    // operate on shared memory
    __shared__ double p_Sh[threadsPerBlock];
    __shared__ double v_Sh[threadsPerBlock];

    double current_Beta, current_Beta1, current_Beta2,
current_Beta3, current_Beta4, current_Beta5,
current_Beta6, current_Beta7;
    double p_next, p_next1, p_next2, p_next3, p_next4,
p_next5, p_next6, p_next7 ;

    //#pragma unroll
    while (tid < data_size ){

        p_next = 0; p_next1 = 0; p_next2 = 0; p_next3 = 0;
p_next4 = 0; p_next5 = 0; p_next6 = 0; p_next7 = 0;

        p_Sh[Index] = p[tid];
        v_Sh[Index] = v[tid];

```

```

//for(int i=0; i<blocksPerGrid;i++){
for(int i=0; i<blocksPerGrid;i+=8){

    current_Beta = Inter_Blk_Beta[i];
    current_Beta1 = Inter_Blk_Beta[i+1];
    current_Beta2 = Inter_Blk_Beta[i+2];
    current_Beta3 = Inter_Blk_Beta[i+3];
    current_Beta4 = Inter_Blk_Beta[i+4];
    current_Beta5 = Inter_Blk_Beta[i+5];
    current_Beta6 = Inter_Blk_Beta[i+6];
    current_Beta7 = Inter_Blk_Beta[i+7];

    p_next += current_Beta * (p_Sh[Index]- omega
* v_Sh[Index]);
    p_next1 += current_Beta1 * (p_Sh[Index]-
omega * v_Sh[Index]);
    p_next2 += current_Beta2 * (p_Sh[Index]-
omega * v_Sh[Index]);
    p_next3 += current_Beta3 * (p_Sh[Index]-
omega * v_Sh[Index]);
    p_next4 += current_Beta4 * (p_Sh[Index]-
omega * v_Sh[Index]);
    p_next5 += current_Beta5 * (p_Sh[Index]-
omega * v_Sh[Index]);
    p_next6 += current_Beta6 * (p_Sh[Index]-
omega * v_Sh[Index]);
    p_next7 += current_Beta7 * (p_Sh[Index]-
omega * v_Sh[Index]);
}

    p[tid] = r[tid] + p_next + p_next1 + p_next2 +
p_next3 + p_next4 + p_next5+ p_next6 + p_next7 ;

    tid += blockDim.x * gridDim.x;
}
}

```

B.2 BiCGSTAB for MRHS System

```
__global__ void compute_alpha_MRS(double *r_tld, double *v, double *rho_values_Vector,
    double *alpha_values_Vector, int N, int RHS_Width){

    int Multipl_RHS_Size = N * RHS_Width;
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int offset;

    double myTemp, alpha_value;

    cublasHandle_t blasHandle;
    cublasStatus_t status = cublasCreate(&blasHandle);

    while (tid < RHS_Width ){

        offset = tid * N;

        cublasDdot(blasHandle, N, &r_tld[0]+offset, 1, &v[0]+offset, 1, &myTemp); // ( r_tld'*v )
        alpha_value = rho_values_Vector[tid] / myTemp; // alph = rho / ( r_tld'*v )
        alpha_values_Vector[tid]=alpha_value;

        //cout <<"rho.. " << rho_values_Vector[ind] <<"\t r_tld*v.. "
        //<< myTemp<<"\tAlpha.. " <<alpha_value << endl;
        tid += blockDim.x * gridDim.x;
    }
    cublasDestroy(blasHandle);
}
```

```

__global__ void compute_S_MRS(double *v, double *s, double *x, double *p_hat,
    double *alpha_values_Vector, int *snorm_pass_flag, int N, int RHS_Width){

    int Multipl_RHS_Size = N * RHS_Width;
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int offset;

    double alpha, nalpha, snrm2;
    cublasHandle_t blasHandle;
    cublasStatus_t status = cublasCreate(&blasHandle);

    while (tid < RHS_Width ){

        offset = tid * N;

        // s = r - av;
        alpha = alpha_values_Vector[tid];
        nalpha = -1*alpha;
        cublasDaxpy(blasHandle, N, &nalpha, &v[0]+offset, 1, &s[0]+offset, 1);
        cublasDnrm2(blasHandle, N, &s[0]+offset, 1, &snrm2);

        if (snrm2 < TOL) {
            //early convergence
            cublasDaxpy(blasHandle, N, &alpha, &p_hat[0]+offset, 1, &x[0]+offset, 1);
            // if all the vectors passed the test, then we can break
            snorm_pass_flag[tid] =1; // this should be shared
            //break;
        }
        tid += blockDim.x * gridDim.x;
    }
    cublasDestroy(blasHandle);
}

```

```

__global__ void compute_P_MRS(double *r, double *r_tld, double *v, double *p, double *rho_values_Vector,
    double *rho1_values_Vector, double *alpha_values_Vector, double *omega_values_Vector, int N, int RHS_Width){

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int offset;

    double rho_value, beta, nomega;    double doubleOne = 1.0;
    cublasHandle_t blasHandle;    cublasStatus_t status = cublasCreate(&blasHandle);

    while (tid < RHS_Width ){

        offset = tid * N;
        //rho = r_tld'*r
        cublasDdot(blasHandle, N, &r_tld[0]+offset, 1, &r[0]+offset, 1, &rho_value);
        if (rho_value == 0.0) { printf("rho is zero\n "); break;}

        rho_values_Vector[tid] = rho_value;
        //cout << rho_value << " .. at .. " << ind << " " << offset << endl;

        beta = (rho_values_Vector[tid] / rho1_values_Vector[tid]) * (alpha_values_Vector[tid] / omega_values_Vector[tid]);
        // cout << rho_value << " .. at .. " << ind << " " << offset << endl;

        nomega = -1*omega_values_Vector[tid];

        cublasDaxpy(blasHandle, N, &nomega, &v[0]+offset, 1, &p[0]+offset, 1);
        cublasDscal(blasHandle, N, &beta, &p[0]+offset, 1); // scale p by beta
        cublasDaxpy(blasHandle, N, &doubleOne, &r[0]+offset, 1, &p[0]+offset, 1); //p = r + beta*( p - omega*v )

        tid += blockDim.x * gridDim.x;
    }
}

```

```

__global__ void compute_X_MRS(double *t, double *p_hat, double *s, double *s_hat, double *x,
    double *alpha_values_Vector, double *omega_values_Vector, int N, int RHS_Width){

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int offset;

    cublasHandle_t blasHandle; cublasStatus_t status = cublasCreate(&blasHandle);

    double omega, nomega, alpha, myTemp1, myTemp2 ;

    while (tid < RHS_Width ){

        offset = tid * N;
        // omega = ( t'*s) / ( t'*t )
        cublasDdot(blasHandle, N, &t[0]+offset, 1, &s[0]+offset, 1, &myTemp1);
        cublasDdot(blasHandle, N, &t[0]+offset, 1, &t[0]+offset, 1, &myTemp2);
        omega = myTemp1 / myTemp2;
        alpha = alpha_values_Vector[tid];
        omega_values_Vector[tid]=omega;
        //cout <<"omega.. " << omega_values_Vector[ind] << endl;
        // -----
        // Update approximation.
        // -----
        nomega = -omega;
        cublasDaxpy(blasHandle, N, &alpha, &p_hat[0]+offset, 1, &x[0]+offset, 1);
        cublasDaxpy(blasHandle, N, &omega, &s_hat[0]+offset, 1, &x[0]+offset, 1); // x = x + alph
        cublasDaxpy(blasHandle, N, &nomega, &t[0]+offset, 1, &s[0]+offset, 1);
        tid += blockDim.x * gridDim.x;
    }
}

```


References

- [1] A. Thompson and G. R. Bowen, "Parallelisation of an oil reservoir simulation," in *High-Performance Computing and Networking*. vol. 1067, H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, Eds., ed: Springer Berlin Heidelberg, 1996, pp. 20-28.
- [2] P. Pacheco, *An introduction to parallel programming*: Elsevier, 2011.
- [3] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, 1995, pp. 24-36.
- [4] S. U. Khan, L. Wang, and A. Y. Zomaya, *Scalable Computing and Communications: Theory and Practice*, 2013.
- [5] R. W. Shonkwiler and L. Lefton, *An Introduction to Parallel and Vector Scientific Computation* vol. 41: Cambridge University Press, 2006.
- [6] M. McCool, J. Reinders, and A. Robison, *Structured parallel programming: patterns for efficient computation*: Elsevier, 2012.
- [7] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*: Elsevier, 2012.
- [8] S. P. Midkiff, "Automatic parallelization: an overview of fundamental compiler techniques," *Synthesis Lectures on Computer Architecture*, vol. 7, pp. 1-169, 2012.
- [9] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for parallel programming*: Pearson Education, 2004.
- [10] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, pp. 46-55, 1998.
- [11] M. Snir, *MPI--the Complete Reference: The MPI core* vol. 1: MIT press, 1998.
- [12] C. Campbell and A. Miller, *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*: Microsoft Press, 2011.
- [13] R. Rahman, *Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers*: Apress, 2013.
- [14] I. X. P. C. I. Set, "Architecture Reference Manual," *Intel Corp., September*, 2012.
- [15] (2012). *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. Available: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-ArchitectureWhitepaper.pdf>
- [16] J. Abou-Kassem, S. M. F. Ali, and M. R. Islam, *Petroleum reservoir simulation a basic approach*. Houston, TX: Gulf Pub. Co., 2006.
- [17] Z. E. HEINEMANN. (2005). *FLUID FLOW IN POROUS MEDIA*. Available: <http://edces.netne.net/files/HEINEM~1.PDF>

- [18] I. V. Minin and O. V. Minin, *Computational Fluid Dynamics Technologies and Applications*. [S.l.]: INTECH, 2011.
- [19] A. A. Awotunde, "Relating time series in data to spatial variation in the reservoir using wavelets," Ph.D. Thesis, Department of Energy Resource Engineering, Stanford University, 2010.
- [20] A. A. Awotunde and R. N. Horne, "An improved adjoint-sensitivity computations for multiphase flow using wavelets," *SPE J*, vol. 17, pp. 402-417, 2012.
- [21] C. Mattax and R. Dalton, "Reservoir Simulation: Society of Petroleum Engineers, Henry L," *Doherty Series, Monograph*, vol. 13, p. 172, 1990.
- [22] D. S. Oliver, A. C. Reynolds, and N. Liu, *Inverse theory for petroleum reservoir characterization and history matching*. Cambridge: Cambridge University Press, 2008.
- [23] N. Sun, N.-Z. Sun, M. Elimelech, and J. N. Ryan, "Sensitivity analysis and parameter identifiability for colloid transport in geochemically heterogeneous porous media," *WATER RESOURCES RESEARCH*, vol. 37, pp. 209-222, 2001.
- [24] L. Chu, A. C. Reynolds, and D. S. Oliver, "Computation of Sensitivity Coefficients for Conditioning the Permeability Field to Well-Test Pressure Data," *In situ.*, vol. 19, p. 179, 1995.
- [25] W. Yeh, "Variational sensitivity analysis, data requirement and parameter identification in a leaky aquifer system," *Water Resour. Res. Water Resources Research*, vol. 26, pp. 1927-1938, 2000.
- [26] S. L. L. Petzold, "Adjoint sensitivity analysis for time-dependent partial differential equations with adaptive mesh refinement," *Journal of Computational Physics*, vol. 198, pp. 310-325, 2004.
- [27] I. Daubechies, *Ten lectures on wavelets* vol. 61: SIAM, 1992.
- [28] D. Peaceman, *Fundamentals of numerical reservoir simulation*. Amsterdam; New York: Elsevier Scientific Pub. Co., 1977.
- [29] A. Khalid and S. Antonín, *Petroleum reservoir simulation*. London: Applied Science Publishers, 1979.
- [30] C. Zhangxin, H. Guanren, and M. Yuanle, *Computational methods for multiphase flows in porous media*. Norwich, NY: Knovel, 2006.
- [31] J. Blazek, *Computational fluid dynamics principles and applications*. Amsterdam; San Diego: Elsevier, 2005.
- [32] R. Lewis, N. Wynne, and K. N. Seetharamu, *Fundamentals of the finite element method for heat and fluid flow*. Hoboken, NJ: Wiley-Interscience, 2005.
- [33] H. Lomax, T. H. Pulliam, and D. W. Zingg, *Fundamentals of computational fluid dynamics*. Berlin; New York: Springer, 2001.
- [34] T. Jiyuan, Y. G. Heng, and L. Chaoqun, *Computational fluid dynamics a practical approach*. Boston: Butterworth-Heinemann, 2008.
- [35] Y. Saad and H. A. van der Vorst, "Iterative solution of linear systems in the 20th century," *Journal of Computational and Applied Mathematics*, vol. 123, pp. 1-33, 11/1/ 2000.
- [36] R. Mehmood and J. Crowcroft, "Parallel iterative solution method for large sparse linear equation systems," University of Cambridge, Computer Laboratory 650, 2005.

- [37] H. A. Van der Vorst, *Iterative Krylov methods for large linear systems* vol. 13: Cambridge University Press, 2003.
- [38] M. H. Gutknecht, "A brief introduction to Krylov space methods for solving linear systems," in *Frontiers of Computational Science*, ed: Springer, 2007, pp. 53-62.
- [39] H. M. Markowitz, "The elimination form of the inverse and its application to linear programming," *Management Science*, vol. 3, pp. 255-269, 1957.
- [40] J. Scott, "Sparse direct methods: An introduction," in *Electronic Structure and Physical Properties of Solids*, ed: Springer, 2000, pp. 401-415.
- [41] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, pp. 354-356, 1969.
- [42] J. Dongarra, V. Eijkhout, and P. Luszczek, "Recursive approach in sparse matrix LU factorization," *Scientific Programming*, vol. 9, pp. 51-60, 2001.
- [43] A. Kamthane, *Programming in C, 2/e*: Pearson Education India, 2011.
- [44] D. S. Watkins, *Fundamentals of matrix computations* vol. 64: John Wiley & Sons, 2004.
- [45] R. Barrett, *Templates for the solution of linear systems : building blocks for iterative methods*. Philadelphia: SIAM, 1994.
- [46] G. H. Golub and C. F. Van Loan, *Matrix computations* vol. 3: JHU Press, 2012.
- [47] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, vol. 7, pp. 856-869, 1986.
- [48] R. Fletcher, "Conjugate gradient methods for indefinite systems," in *Numerical Analysis*, ed: Springer, 1976, pp. 73-89.
- [49] R. W. Freund and N. M. Nachtigal, "QMR: a quasi-minimal residual method for non-Hermitian linear systems," *Numerische Mathematik*, vol. 60, pp. 315-339, 1991.
- [50] H. A. Van der Vorst, "Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems," *SIAM Journal on scientific and Statistical Computing*, vol. 13, pp. 631-644, 1992.
- [51] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*: Clarendon Press Oxford, 1986.
- [52] T. A. Davis, *Direct methods for sparse linear systems*: Siam, 2006.
- [53] A. Gupta, "Recent advances in direct methods for solving unsymmetric sparse systems of linear equations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, pp. 301-324, 2002.
- [54] M. Benzi, "Preconditioning techniques for large linear systems: a survey," *Journal of Computational Physics*, vol. 182, pp. 418-477, 2002.
- [55] Y. Saad, *Iterative methods for sparse linear systems*. Philadelphia: SIAM, 2003.
- [56] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *Parallel iterative algorithms: from sequential to grid computing*: CRC Press, 2007.
- [57] V. Eijkhout, *LAPACK working note 50 : distributed sparse data structures for linear algebra operations*. Knoxville, Tenn.: University of Tennessee, Computer Science Dept., 1992.

- [58] Y. Saad, "Krylov subspace methods on supercomputers," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, pp. 1200-1232, 1989.
- [59] G. V. Paolini and G. R. Di Brozolo, "Data structures to vectorize CG algorithms for general sparsity patterns," *BIT Numerical Mathematics*, vol. 29, pp. 703-718, 1989.
- [60] E. Montagne, "An optimal storage format for sparse matrices," *Information Processing Letters Information Processing Letters*, vol. 90, pp. 87-92, 2004.
- [61] A. Ekambaram and E. Montagne, "An alternative compressed storage format for sparse matrices," in *Computer and Information Sciences-ISCIS 2003*, ed: Springer, 2003, pp. 196-203.
- [62] L. Yuan, Y. Zhang, X. Sun, and T. Wang, "Optimizing Sparse Matrix Vector Multiplication Using Diagonal Storage Matrix Format," in *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, 2010, pp. 585-590.
- [63] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone, "Performance optimization and modeling of blocked sparse kernels," *International Journal of High Performance Computing Applications*, vol. 21, pp. 467-484, 2007.
- [64] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *The Journal of Supercomputing*, vol. 50, pp. 36-77, 2009.
- [65] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *High Performance Computing and Communications*, ed: Springer, 2005, pp. 807-816.
- [66] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *Computational Science and Engineering, 2009. CSE'09. International Conference on*, 2009, pp. 247-256.
- [67] P. Stathis, S. Vassiliadis, and S. Cotofana, "A hierarchical sparse matrix storage format for vector processors," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003, p. 8 pp.
- [68] D. Langr, I. Simecek, P. Tvrđík, T. Dytrych, and J. P. Draayer, "Adaptive-blocking hierarchical storage format for sparse matrices," in *Computer Science and Information Systems (FedCSIS), 2012 Federated Conference on*, 2012, pp. 545-551.
- [69] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance optimizations and bounds for sparse matrix-vector multiply," in *Supercomputing, ACM/IEEE 2002 Conference*, 2002, pp. 26-26.
- [70] R. Geus and S. Röllin, "Towards a fast parallel sparse matrix-vector multiplication," in *PARCO*, 1999, pp. 308-315.
- [71] E.-J. Im and K. A. Yelick, *Optimizing the performance of sparse matrix-vector multiplication*: University of California, Berkeley, 2000.
- [72] J. Godwin, J. Holewinski, and P. Sadayappan, "High-performance sparse matrix-vector multiplication on GPUs for structured grid computations," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, 2012, pp. 47-56.

- [73] P. Guo, L. Wang, and P. Chen, "A Performance Modeling and Optimization Analysis Tool for Sparse Matrix-Vector Multiplication on GPUs," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, pp. 1112-1123, 2014.
- [74] P. Kumbhar, "Performance of PETSc GPU Implementation with Sparse Matrix Storage Schemes," 2011.
- [75] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, *et al.*, "PETSc Users Manual Revision 3.4," ed, 2013.
- [76] S. Jain, "Memory efficiency implications on sparse matrix operations," THE UNIVERSITY OF NORTH CAROLINA AT CHARLOTTE, 2014.
- [77] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," Nvidia Technical Report NVR-2008-004, Nvidia Corporation 2008.
- [78] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, p. 18.
- [79] R. Shahnaz, A. Usman, and I. R. Chughtai, "Review of storage techniques for sparse matrices," in *9th International Multitopic Conference, IEEE INMIC 2005*, 2005, pp. 1-7.
- [80] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Citeseer, 2003.
- [81] S. Xu, W. Xue, and H. X. Lin, "Performance modeling and optimization of sparse matrix-vector multiplication on NVIDIA CUDA platform," *The Journal of Supercomputing*, vol. 63, pp. 710-721, 2013.
- [82] X. Sun, Y. Zhang, T. Wang, X. Zhang, L. Yuan, and L. Rao, "Optimizing spmv for diagonal sparse matrices on gpu," in *Parallel Processing (ICPP), 2011 International Conference on*, 2011, pp. 492-501.
- [83] P. Stathis, S. Cotofana, and S. Vassiliadis, "Sparse matrix vector multiplication evaluation using the bbcs scheme," in *in Proc. of 8th Panhellenic Conference on Informatics*, 2001.
- [84] *Freely Available Software for Linear Algebra*. Available: <http://www.netlib.org/utk/people/JackDongarra/la-sw.html>
- [85] B. Jacob and G. Guennebaud. (2012). *Eigen C++ template library for linear algebra: Matrices, vectors, numerical solvers, and related algorithms (3.2.2 ed.)*. Available: http://eigen.tuxfamily.org/index.php?title=Main_Page
- [86] G. Guennebaud and B. Jacob, "Eigen," ed, 2012.
- [87] M. Intel, "Intel Math Kernel Library," ed, 2007.
- [88] C. NVIDIA, "CUSPARSE library," *NVIDIA Corporation, Santa Clara, California*, 2011.
- [89] M. Naumov, "Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS," *Nvidia white paper*, 2011.
- [90] B. Yang, X. Chen, X. Y. Liao, M. L. Zheng, and Z. Y. Yuan, "FEM-Based Modeling and Deformation of Soft Tissue Accelerated by CUSPARSE and CUBLAS," *Advanced Materials Research*, vol. 671, pp. 3200-3203, 2013.
- [91] K. He, S. X.-D. Tan, E. Tlelo-Cuautle, H. Wang, and H. Tang, "A new segmentation-based GPU-accelerated sparse matrix-vector multiplication," in

- Circuits and Systems (MWSCAS), 2014 IEEE 57th International Midwest Symposium on*, 2014, pp. 1013-1016.
- [92] L. A. Flores, V. Vidal, P. Mayo, F. Rodenas, and G. Verdú, "CT Image Reconstruction Based on GPUs," *Procedia Computer Science*, vol. 18, pp. 1412-1420, 2013.
- [93] N. Bell and M. Garland, "Cusp library, 2012," ed.
- [94] C. Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, 2008.
- [95] J. Hoberock and N. Bell, "Thrust: A parallel template library," *Online at <http://thrust.googlecode.com>*, vol. 42, p. 43, 2010.
- [96] D. Lukarski, "PARALUTION project," ed, 2012.
- [97] M. Harris, "Optimizing parallel reduction in CUDA," *NVIDIA Developer Technology*, vol. 2, 2007.
- [98] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, pp. 443-466, 2013.
- [99] J. Dongarra, "Freely available software for linear algebra on the web," *URL: <http://www.netlib.org/utk/people/JackDongarra/la-sw.html> (april, 2003)*, 2000.
- [100] Y. Shi, "Reevaluating amdahl's law and gustafson's law," *Computer Sciences Department, Temple University (MS: 38-24)*, 1996.
- [101] H. C. Elman, D. J. Silvester, and A. J. Wathen, *Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics*: Oxford University Press, 2014.
- [102] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*: Gulf Professional Publishing, 1999.
- [103] I. T. Center. *HPC*. Available: <http://hpc.kfupm.edu.sa/NewHPC/Resources.html>
- [104] R. Bridson and W.-P. Tang, "Ordering, anisotropy, and factored sparse approximate inverses," *SIAM Journal on Scientific Computing*, vol. 21, pp. 867-882, 1999.
- [105] M. Benzi and M. Tuma, "A comparative study of sparse approximate inverse preconditioners," *Applied Numerical Mathematics*, vol. 30, pp. 305-340, 1999.
- [106] H. Anzt, S. Tomov, P. Luszczek, I. Yamazaki, J. Dongarra, and W. Sawyer, "Accelerating Krylov Subspace Solvers on Graphics Processing Units."
- [107] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73-82.
- [108] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU Technology Conference, GTC*, 2010, p. 16.
- [109] M. Tillmann, T. Karcher, C. Dachsbacher, and W. F. Tichy, "Application-independent Autotuning for GPUs," in *PARCO*, 2013, pp. 626-635.
- [110] Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning GEMM for GPUs," in *Computational Science-ICCS 2009*, ed: Springer, 2009, pp. 884-892.
- [111] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, p. 30.

- [112] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Innovative Parallel Computing (InPar)*, 2012, 2012, pp. 1-10.
- [113] P. Guo and L. Wang, "Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus," in *Computational and Information Sciences (ICCIS), 2010 International Conference on*, 2010, pp. 1154-1157.
- [114] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, vol. 38, pp. 391-407, 2012.
- [115] S. Jones, "Introduction to dynamic parallelism," in *GPU Technology Conference Presentation S*, 2012.
- [116] Y. Perez-Riverol and R. V. Alvarez, "A UML-based Approach to Design Parallel and Distributed Applications," *arXiv preprint arXiv:1311.7011*, 2013.
- [117] F. Gebali, *Algorithms and parallel computing* vol. 84: John Wiley & Sons, 2011.
- [118] S. Pllana and T. Fahringer, "Modeling Parallel Applications with UML," in *15th International Conference on Parallel and Distributed Computing Systems (PDCS 2002)*, 2002, pp. 19-21.
- [119] H. Gomaa, "Designing concurrent, distributed, and real-time applications with UML," in *Proceedings of the 23rd International Conference on Software Engineering*, 2001, pp. 737-738.
- [120] J. B. Warmer and A. G. Kleppe, "The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series)," 1998.
- [121] S. Pllana and T. Fahringer, "UML based modeling of performance oriented parallel and distributed applications," in *Simulation Conference, 2002. Proceedings of the Winter*, 2002, pp. 497-505.
- [122] S. S. Alhir, *Learning Uml*: " O'Reilly Media, Inc.", 2003.
- [123] C. NVIDIA, "Cuda c best practices guide," ed, 2014.
- [124] N. I.-L. A. Gray and A. Sjöström, "Best Practice mini-guide accelerated clusters. Using General Purpose GPUs," ed, 2014.

Vitae

Name :Ayham Horiah Zaza

Nationality :Syrian

Date of Birth :11/10/1983

Email :dr.ayham@zoho.com

Address :KFUPM, Main Campus. Dhahran, Saudi Arabia

Academic Background :I hold a B.Sc. in Computer Engineering, a Master Degree M.Sc. in Medical Physics and a PhD in Computer Science and Engineering all done at KFUPM .I have developed several software systems to automate some radiotherapy QA routines at hospitals. I have also attempted developing an intelligent software for an automated early cervical cancer detection. Besides my interest in Intelligent Computing and Medical Image Processing (Image Registration), I have developed a data parallel simulator for multi-phase fluid flow in oil reservoir. Since July 2013, I have become a Keystone fellow after participating in a creative program for idea translation and business development. I am always enthusiastic at taking part in any multidisciplinary work where I can discover my own innovation and creativity to meet the future needs. <http://www.ccse.kfupm.edu.sa/~ayham/about.html>

