PARALLELIZATION OF A FURNACE SIMULATION CODE FOR MULTI-CORE MACHINES

by

Ervin Domazet

B.S., Computer Engineering, Boğaziçi University, 2012

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2014

PARALLELIZATION OF A FURNACE SIMULATION
CODE FOR MULTI-CORE MACHINES

APPROVED BY:

Prof. Can Özturan (Thesis Supervisor)	.....
Prof. Oğuz Tosun	.....
Assoc. Prof. Ali Ecder	.....

DATE OF APPROVAL: 01.12.2014

ACKNOWLEDGEMENTS

First and foremost I offer my sincerest gratitude to my supervisor, Prof. Can Özturan, who has supported me throughout my thesis with his patience and knowledge while allowing me to work in my own way. I attribute the level of my Masters degree to his encouragement and effort and without him this thesis would not have been completed or written. One simply could not wish for a better or friendlier supervisor.

In addition, I express my warm thanks to Şişecam Chief Senior Researchers Mrs. Lale Önsel and and Mrs. Zeynep Eltutar for their support and guidance at Şişecam.

Besides my advisor and supporters from Şişecam, I would also like to thank Prof. Oğuz Tosun, and Assoc. Prof. Ali Ecder for serving on my thesis committee.

I would like to thank my family: my grandfather Abdurrahman Emuş, my parents Ergin Domazet and Güner Domazet, my brothers Furkan Domazet and Harun Domazet, and especally my friends Hakan Demir, Erdem Orman, Münir Sali, Affan Hasan, Ersin Iljazi, Cengiz Raşit and the ones that are in my heart for supporting me spiritually throughout my life.

Last but not the least, I would like to thank my darling, Nazife Yakup. She was always there cheering me up and stood by me through the good times and bad.

ABSTRACT

PARALLELIZATION OF A FURNACE SIMULATION CODE FOR MULTI-CORE MACHINES

Şişecam uses Fortran programs that implement mathematical models for designing glass furnaces and for planning optimized operational criteria. The mathematical model for glass furnaces employs revised version of Patankar and Spalding's Semi-Implicit Method for Pressure-Linked Equations method called SIMPLER. The main objective of this thesis is to parallelize the sequential modelling programs using Open Multi-Processing (OpenMP) for shared-memory multi-core architectures. The sequential program uses Tri-Diagonal Matrix Algorithm (TDMA) for solving the resulting linear systems of equations. Sequential code is analyzed by profiling in order to locate sections that have long running times. Additionally, another linear system solver, Stone's Strongly Implicit Procedure (SIP), is integrated to the program. Radiation calculations and linear system solvers which are the most computational intensive parts of the program are parallelized. The performance of the parallelized code is compared that of the sequential code. Results obtained on an Intel 8-core Xeon system show that speed-ups of roughly four times are possible.

ÖZET

FIRIN SİMULASYONU PROGRAMININ ÇOK ÇEKİRDEKLİ BİLGİSAYARLAR İÇİN PARALELLEŞTİRİLMESİ

Fırın tasarım ve işletme kriterlerinin belirlenmesinde Şişecam matematiksel modelleri Fortran yazılımları kullanmaktadır. Matematiksel modellerin temelini Patankar ile Spalding'in revize edilmiş olan SIMPLER metodu oluşturmaktadır. Bu tezin ana hedefi seri modelleme programının OpenMP ile paylaşım bellekli ve çok çekirdekli işlemciler içeren sistemler için paralelleştirilmesidir. Seri çalışan kod doğrusal denklemler sistemlerini çözmek için TDMA algoritmasını kullanmaktadır. Seri kod çalıştırılarak yoğun zaman alan kısımlar tespit edilmiştir. Ek olarak, doğrusal denklemler sistemi çözücüsü olan SIP algoritması da programa entegre edilmiştir. Seri kodun en yoğun kısımları olan Radyasyon hesaplaması ile doğrusal denklemler çözücüleri paralelleştirilmiştir. Paralelleşen kod, seri çalışan kod ile karşılaştırılmıştır. Intel 8-core Xeon sisteminde elde edilen sonuçlar, ortalama dört kat hızlanmanın mümkün olduğunu göstermiştir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . viii
LIST OF TABLES . . . xi
LIST OF ACRONYMS/ABBREVIATIONS . . . xiii
1. INTRODUCTION . . . 1
1.1. Contributions of the Thesis . . . 1
2. BACKGROUND WORK . . . 3
2.1. Overview of the Glass Furnace Simulation Model . . . 3
2.1.1. The SIMPLER Algorithm . . . 4
2.1.2. Radiation Calculations . . . 6
2.2. Overview of Solvers Used in Glass Furnace Simulation . . . 6
2.2.1. Tri-Diagonal Matrix Algorithm (TDMA) . . . 7
2.2.2. Strongly Implicit Procedure (SIP) . . . 8
2.3. Overview of Parallel Processing . . . 9
2.3.1. Parallel Performance . . . 11
2.4. Previous Work on Parallelization of SIMPLER Method . . . 12
3. PARALLELIZATION METHODOLOGY . . . 14
3.1. Locating Computationally Intensive Parts in the Sequential Code . . . 14
3.2. Loop Structures of the Computationally Intensive Parts . . . 18
3.2.1. 3D Radiation - Main Calculation . . . 19
3.2.2. TDMA . . . 21
3.2.3. SIP . . . 22
3.2.4. Analysis of the Computationally Intensive Parts after Parallelisation . . . 25
4. TESTS AND RESULTS . . . 30
4.1. Comparison Of Linear System Solvers . . . 30
4.2. Efficiency and Scaling Analysis . . . 34

4.3. Effects of Compiler Optimizations and OpenMP Loop Scheduling Policies 34
4.4. Effect of Problem (Grid) Size on Speedup 41
5. CONCLUSIONS 45
REFERENCES 46

LIST OF FIGURES

Figure 2.1. 2D staggered grid representation. 3

Figure 2.2. Glass furnace grid with uneven grid spacing. 4

Figure 2.3. High-level structure of the Glass Furnace Simulation program. . . 5

Figure 2.4. High-level structure of the Radiation Calculation by Using FVM. . 6

Figure 2.5. TDMA line-by-line methodology for two-dimensional grid. 7

Figure 2.6. High-level structure of the TDMA solver. 8

Figure 2.7. High-level structure of the SIP solver. 9

Figure 2.8. Different computer architectures. 10

Figure 3.1. Distribution of timings through an iteration of main part of Glass Furnace Simulation. 16

Figure 3.2. Average timing distribution across different parts of the Calculate Coeffficients part. 16

Figure 3.3. Timing distribution of Radiation part. 18

Figure 3.4. Timing distribution of 3D Radiation part. 18

Figure 3.5. Loop structures of Main Calculation part of 3D Radiation. 20

Figure 3.6. Schematic view of parallelisation of Main calculation part of Radiation. Starting from each corner, simultaneous processing on theta, phi, x, y and z direction. 21

Figure 3.7. Loop structures of TDMA among different blocks for a single iteration. 23

Figure 3.8. Simultaneous execution of 2D blocks, for a single iteration of TDMA. 24

Figure 3.9. Loop structures of modules related to SIP for a single iteration. . 24

Figure 3.10. Schematic view of pipeline parallel processing. Blocks with equal colors (or letters) are calculated simultaneously. 25

Figure 3.11. Distribution of timings among the single iteration of parallel Main Program. 26

Figure 3.12. Average timing distributions across different parts of parallel Calculate Coefficients. 26

Figure 3.13. Timing distribution of parallel Radiation part of Calculate Coefficients. 28

Figure 3.14. Timing distribution of 3D Radiation part. 29

Figure 4.1. Convergence plots for sequential code with TDMA and SIP solvers with no optimization flags (until error is reduced to 2%). 32

Figure 4.2. Parallel convergence curves for linear TDMA and SIP solvers (with no optimization flags - O0). 32

Figure 4.3. Temperature distribution for computation using TDMA solver (with no optimization flags - O0). 33

Figure 4.4. Temperature distribution for computation using SIP solver (with no optimization flags - O0). 33

Figure 4.5. Total computation times for TDMA and SIP solvers with no optimization flags - O0 (until error is reduced to 2%). 37

Figure 4.6. Speed-up analysis for TDMA and SIP solvers with no optimization flags - O0 (until error is reduced to 2%). 38

Figure 4.7. Speedup Analysis for Intel compiler optimizations. 39

Figure 4.8. Speedup analysis of Intel's O2 compiler optimization relative to O0. 41

Figure 4.9. Running times of program with different solvers as the input size increases, where Radiation is calculated on every iteration. 44

Figure 4.10. Running times of program with different solvers as the input size increases, where Radiation is calculated on every fifth iteration. . . 44

LIST OF TABLES

Table 3.1. Input Specifications for the Sequential Code. 15

Table 3.2. Sequential Glass Furnace Simulation Profile for 5 Iterations. 15

Table 3.3. Profile Of Sequential Calculate Coefficients part. 17

Table 3.4. Profile of sequential Radiation part. 17

Table 3.5. Sequential profile of 3D Radiation part. 19

Table 3.6. Comparison of loop merging with the current parallelized form for 50 iterations, by using TDMA solver. 22

Table 3.7. Parallel Glass Furnace Simulation Profile for 5 Iterations. 25

Table 3.8. Profile of parallel Calculate Coefficients. 27

Table 3.9. Profile of parallel Radiation part of Calculate Coefficients. 27

Table 3.10. Parallel profile of 3D Radiation part. 28

Table 3.11. Outputs of Sequential & Parallel program when approaching 500'th iter. 29

Table 4.1. Running Times and Speedup for the Parallel Program for 8 Threads for X = 82, Y = 86 and Z = 28 nodes (with no optimization flag -O0). 31

Table 4.2. Performance of solvers on sequential runs with no optimization flags(until error is reduced to 2%). 31

Table 4.3. Performance of Solvers on Parallel Runs (with no optimization flags - O0). 32

Table 4.4. Efficiency and Speedup analysis for TDMA solver (with no optimization flags - O0). 35

Table 4.5. Efficiency and Speedup analysis for SIP solver (with no optimization flags - O0). 36

Table 4.6. Running Time and Speedups of the Program when Executed Using Compiler Optimizations. 40

Table 4.7. Input specifications for three data sets that have roughly 200K, 1M and 2M nodes. 42

Table 4.8. Running times for varying data parameters, when using TDMA solver for 100 iteration. Radiation is calculated on every and every fifth iteration. 43

Table 4.9. Running times for varying data parameters, when using SIP solver for 100 iteration. Radiation is calculated on every and every fifth iteration. 43

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
API	Application Programming Interface
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
FVM	Finite Volume Method
K	Thousand
M	Million
MPI	Message-Passing Interface
OpenMP	Open Multi-Processing
SIMPLER	Semi-Implicit Method for Pressure Linked Equations Revised
SIP	Strongly Implicit Procedure
TDMA	Tri-Diagonal Matrix Algorithm

1. INTRODUCTION

Şişecam uses programs for mathematical models to carry out furnace design and operational criteria. These programs were developed using the Fortran programming language and have been used within Şişecam since 1990s. During 2000-2002, studies were carried out to parallelize Şişecam's modelling programs by using Message Passing Interface (MPI) [1]. The existing MPI program splits the domain only on the x-direction. The implementation in [1] also requires the problem domain and the input to be partitioned manually which makes the use of the program difficult.

Şişecam codes have complex structure and sensitive numerical operations. If parallelization is to be done for distributed memory machines by using message passing libraries, the domain, operations and the data structures need to be decomposed which in turn necessitate complete redesign and reimplementation of the whole code from the scratch. As a result, such an approach is not attractive. Instead, an approach that uses shared memory model and allows incremental parallelization without requiring major code redesign and implementation is preferable. OpenMP [2] can facilitate such an an approach and, hence, is used in this thesis.

1.1. Contributions of the Thesis

Since OpenMP works on shared memory model, starting from the original sequential program and without changing the data structures, incrementally locating the loops that have long running times and then parallelizing them, will reduce the risk of introducing mistakes while requiring less effort in order to speed up the code. Thus, the main contribution of this thesis are:

(i) Profiling of Şişecam codes to locate computationally intensive parts,

(ii) Parallelization of Şişecam glass furnace model by OpenMP [2] for systems employing shared-memory and multi-core architectures. The overhead due to the message passing among processors will also be lowered significantly in this ap-

proach.

(iii) Integration of a multi-threaded version of another linear system solver (namely, Stone's Strongly Implicit Procedure (SIP) [3, 4] solver) into Şişecam codes.

(iv) Performance assessment of the new parallel code.

This thesis begins by presenting background information on the Glass Furnace Simulation Model in Chapter 2. In the same chapter, two linear system solvers that are employed are presented together with an overview of previous work. Additionally, a general information about parallel processing is given. Chapter 3 describes the employed parallelization methodology. In Chapter 4, we present results of various tests performed with the parallelized program and assess the performance of the parallelization. Finally, in Chapter 5, the thesis is concluded and some possible future work are discussed.

2. BACKGROUND WORK

2.1. Overview of the Glass Furnace Simulation Model

The codes that are developed in the area of mathematical modeling of combustion chambers involve sophisticated physical models and numerical solution techniques. The related models consists of non-linear partial differential equations. The solution method that is used in the Glass Furnace Simulation is based on SIMPLER (Revised version of Semi-Implicit Method for Pressure Linked Equations) [5] method.

In order to simulate a furnace whose dimensions are defined, firstly, the domain needs to be discretized with a grid [4]. A staggered grid is generated by two types of lines; i.e solid and dashed as shown in Figure 2.1. Intersection of solid lines form a basic nodes. Intersection of a solid line and dashed line form a velocity nodes, whereas the intersection of two dashed lines form pseudo-nodes. Scalar variables are stored in basic nodes, whereas velocity components are stored at velocity nodes. Note that grid spacings need not be regular. This fact is shown in Figure 2.2.

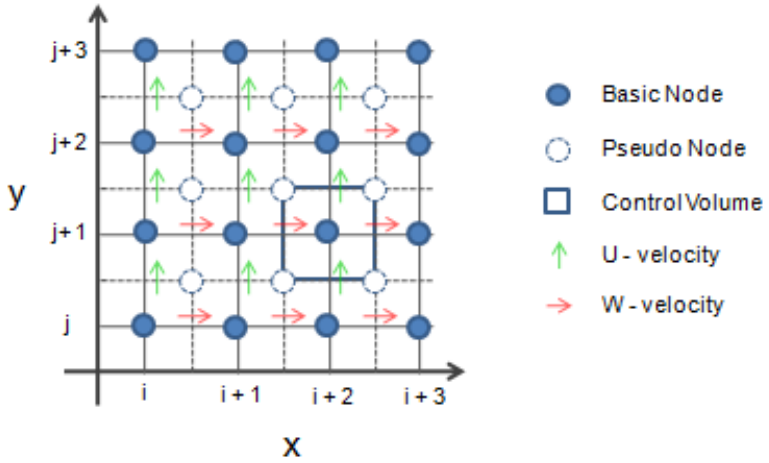


Figure 2.1. 2D staggered grid representation.

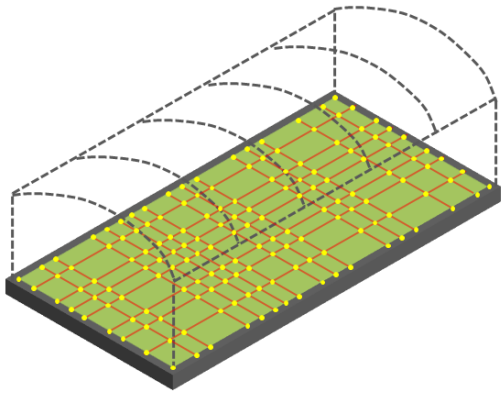


Figure 2.2. Glass furnace grid with uneven grid spacing.

2.1.1. The SIMPLER Algorithm

High level view of the Glasss Furnace Simulation program is illustrated in Figure 2.3. The program reads a finite difference grid and initializes the variables. Afterwards, it starts a loop that computes the solution iteratively. In a single iteration, all stages of the SIMPLER algorithm are executed. In every stage with the exception of radiation, the resulting linear systems of equations are solved by either Tri-Diagonal Matrix Algorithm (TDMA) [5,6] based or Stone's SIP linear system solvers. The following are calculated in stages that make up the used SIMPLER algorithm:

(i) Pressure
(ii) U-V-W Velocity
(iii) Pressure Correction
(iv) Turbulent Kinetic Energy Calculation
(v) Dissipation Rate Of Turbulance
(vi) Mixture Fraction
(vii) Species Fraction
(viii) Enthalpy
(ix) Soot Mass Fraction
(x) Radiation

Details of the each of these stages are given in [1].

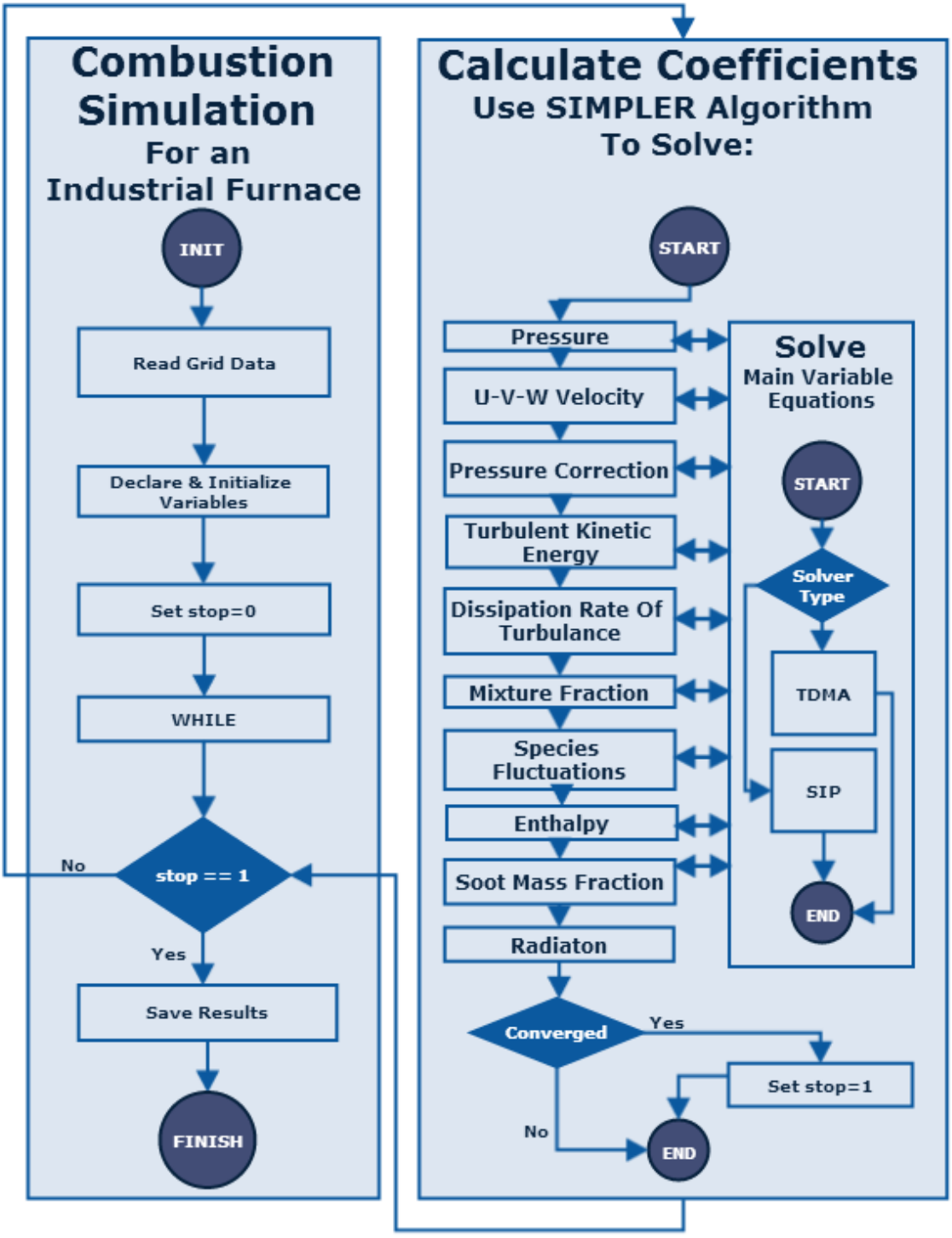


Figure 2.3. High-level structure of the Glass Furnace Simulation program.

The differential forms of the equations which are presented in [1] are discretised by the FVM and solved using an iterative procedure that employs radiation calculations and a linear system solver. TDMA and SIP are used as solvers in order to solve the

systems that are obtained from FVM. The equations related to the radiation phase are solved by the Finite Volume Method (FVM) [7–9].

After the radiation stage, if the convergence criteria is not met, the program continues to iterate. If convergence occurs, the program saves the final status of the furnace and finishes execution.

2.1.2. Radiation Calculations

In radiation phase, the differential equations are discretized by the Finite Volume Method. Previous work [1] describes the algorithmic details of the FVM and radiation calculations. Figure 2.4 shows a high level view of the radiation calculations.

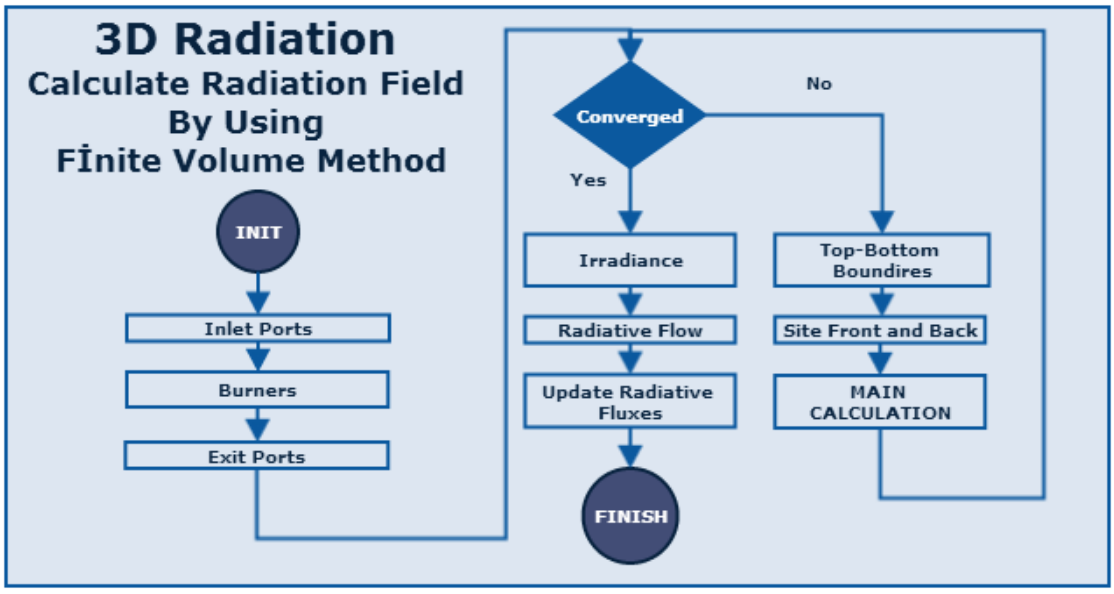


Figure 2.4. High-level structure of the Radiation Calculation by Using FVM.

2.2. Overview of Solvers Used in Glass Furnace Simulation

Solution techniques for linear systems can be broadly categorized as direct and iterative methods. In Şişecam codes iterative solvers based on TDMA and SIP are used. These are reviewed in more detail below.

2.2.1. Tri-Diagonal Matrix Algorithm (TDMA)

An algorithm for rapidly solving tri-diagonal systems has been developed by Thomas [6]. This algorithm is called Thomas algorithm or the Tri-Diagonal Matrix Algorithm (TDMA). TDMA is widely used in Computational Fluid Dynamics (CFD) programs. It is computationally inexpensive and requires minimal amount of storage. TDMA is a direct method intended to be used for one-dimensional problems. To solve multi-dimensional problems, TDMA can be applied iteratively in a line-by-line fashion.

Figure 2.5 illustrates this on a two-dimensional problem. In this particular figure, sequence of lines are chosen first in the x direction, then in the y direction, called the *sweep directions*. TDMA is applied along these lines to solve the system. In this manner, line-by-line process will run until solution converges [10].

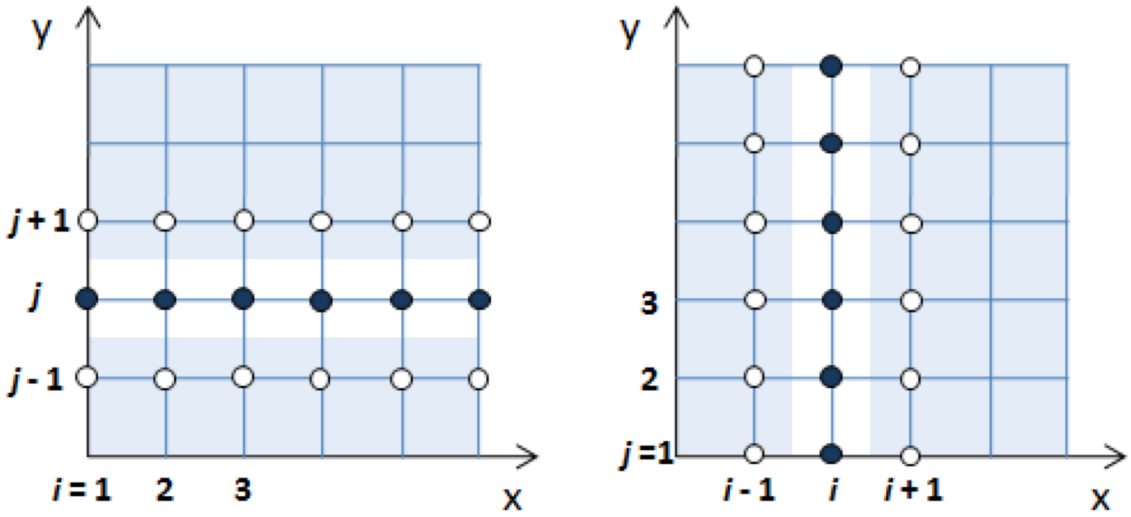


Figure 2.5. TDMA line-by-line methodology for two-dimensional grid.

Inside the Glass Furnace Simulation program, three-dimensional form of TDMA is used. Iterative procedure of two dimensional forms the basis for the three dimensional problem. TDMA is applied line by line in every plane on the domain. Figure 2.6 depicts how the TDMA solver is used inside the three-dimensional Glass Furnace Simulation program.

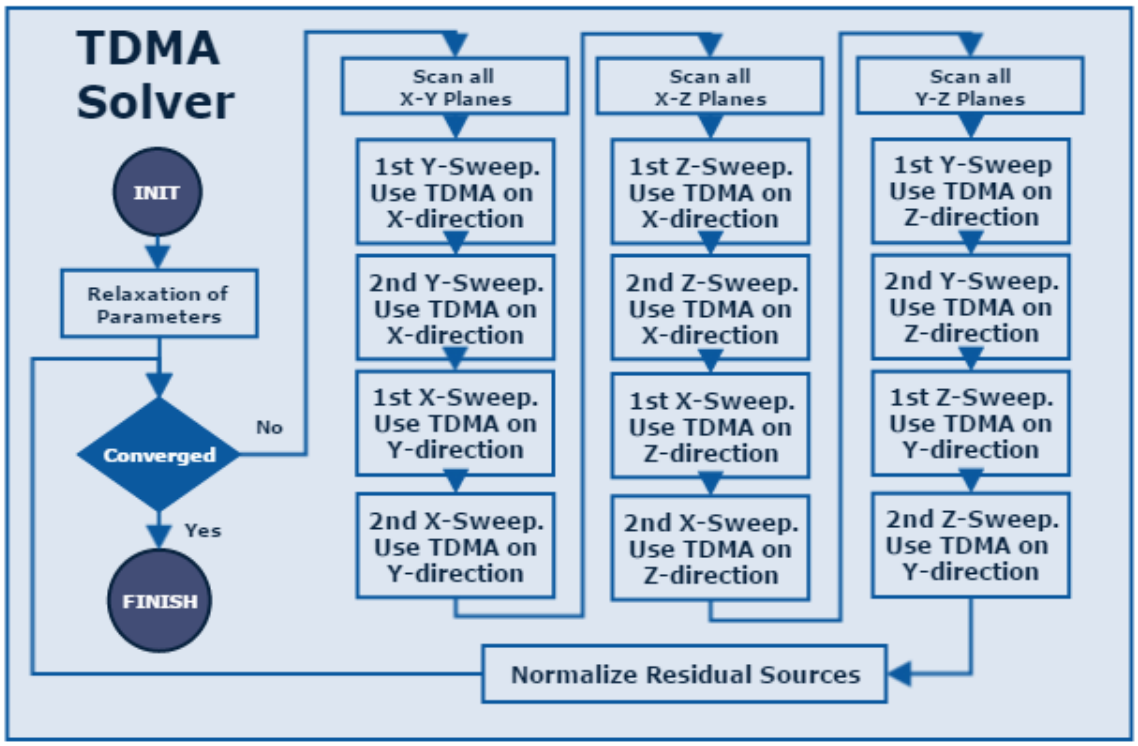


Figure 2.6. High-level structure of the TDMA solver.

2.2.2. Strongly Implicit Procedure (SIP)

The SIP algorithm is an iterative method that employs incomplete LU decomposition as a preconditioner. It was developed by Stone [11] in the U. S. in 1968. It is widely used in CFD. The following steps are executed when SIP is used as a solver.

(i) Perform incomplete LU decomposition
(ii) Residual Calculation
(iii) Forward Substitution
(iv) Backward Substitution
(v) Go back to (ii), until convergence.

Figure 2.7 shows a high level view of the SIP solver.

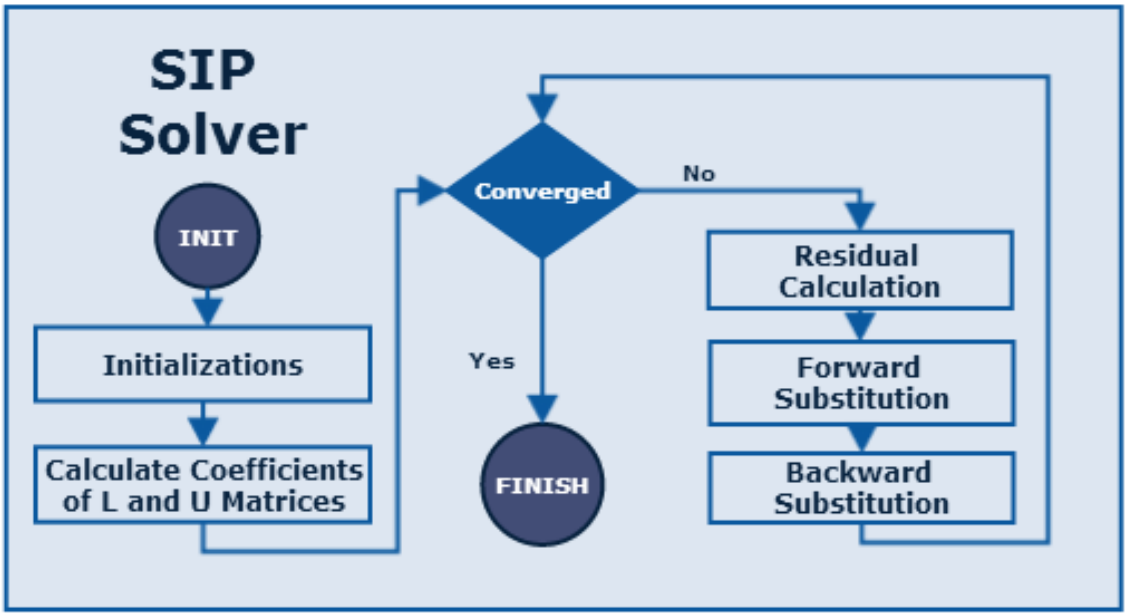


Figure 2.7. High-level structure of the SIP solver.

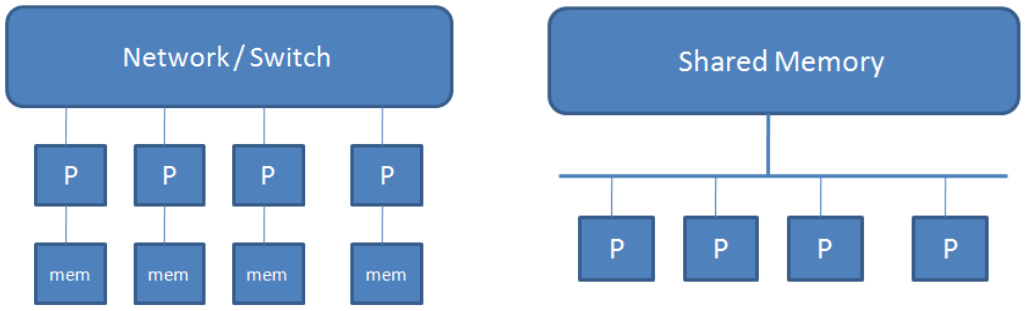
2.3. Overview of Parallel Processing

The main objective of parallel programming is to improve performance of programs. In parallel programs, the work is partitioned into a number of parts so that they can be run concurrently. The parallel architectures can be broadly categorized into (*i*) distributed and (*ii*) shared memory systems.

In distributed memory architectures (Figure 2.8a), each processor has its own local memory, and the communication is achieved by passing messages among processors. MPI (Message Passing Interface) [12] and PVM (Parallel Virtual Machine) [13] are two well known message passing libraries. MPI is currently the most widely used library on clusters.

In shared memory architectures (Figure 2.8b), shared memory locations can be accessible by all the processors. In other words, a single shared address space exists. Shared memory programming are seemingly more convenient to use than distributed memory systems which require the use of lower level send/receive type message passing.

However, shared memory systems require expensive locks to be used to access critical regions of memory [14]. Some of the well known libraries for shared programming are OpenMP(Open Multi-Processing) [2], TBB(Threading Building Blocks) [15] and PThreads(POSIX Threading Interface) [16].



(a) Distributed memory architecture (b) Shared memory architecture

Figure 2.8. Different computer architectures.

OpenMP is an Application Programming Interface (API), developed and defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for shared memory parallel applications. The API supports C/C++ and Fortran on a wide variety of architectures.

In OpenMP iteration space of loops in parallel regions are partitioned into chunks. Iteration chunks are then scheduled to threads for execution in parallel. OpenMP provides a number of loop scheduling methods that can be optionally chosen and controlled by the programmer. Four different loop scheduling types (kinds) can be provided to OpenMP as listed below. The optional parameter (chunk), when specified, must be a positive integer.

- *Static-Balanced*: This method divides the loop iterations into equally sized chunks.
- *Dynamic*: Loop iterations are divided into specified chunks and free threads are scheduled to execute these iteration chunks.
- *Guided*: This is similar to dynamic with an exception that chunk size is variable. It starts with large chunks, and then decreases chunk size (exponentially) to 1 or a user specified value.

- *Auto*: The decision is done by the compiler at the startup phase of the program.

We have used the integrated version of Intel Fortran Compiler for developing the parallel code. The compiler provides optimization techniques in order to improve the performance [17] of compiled programs. The effect of various optimization flags on the parallelized program will be discussed in section 4.3. The list of optimization flags is given below.

- `O0`: no optimization is done.
- `O1` - *Optimize for Maximum Speed & Minimum Size*: Mainly the compiler creates smallest code size. It is useful on large server/database applications.
- `O2` - *Optimize for Maximum Speed*: Stronger optimization techniques are applied when compared to O1. Vectorization is used in order to maximize speed.
- `O3` - *Optimize for Maximum Speed by Considering More High Level Aggressive Optimizations*: Even Stronger optimization techniques than O2 are used, such as: memory-access optimizations, loop unrolling and so on. Optimized for applications that have intensive floating point operations.

2.3.1. Parallel Performance

In parallel processing, ideally a program can be made P times faster if P processors are used when compared to a single processor. However, this is not always observed. Let :

- n : denote the input size,
- P : denote the number of processors,
- $T^*(n)$: denote the complexity of the best sequential algorithm,
- $T_P(n)$: Time complexity of the parallel algorithm when run on P processors,
- $T_1(n)$: Time complexity of the parallel algorithm when run on 1 processor.

The speed-up is defined as:

$$S_P(n) = \frac{T^*(n)}{T_P(n)} \tag{2.1}$$

Note that in this thesis, we will take the complexity of the best sequential algorithm to be the original sequential program, i.e. $T^*(n) = T_1(n)$. Another metric that is useful to assessing the utilization and the idleness of the processors is the efficiency which is defined as:

$$E_P(n) = \frac{T_1(n)}{P \cdot T_P(n)} \tag{2.2}$$

2.4. Previous Work on Parallelization of SIMPLER Method

Parallelization of SIMPLER method is considered in [18–21]. In [18], high numerical efficiency is achieved by using the nonlinear multigrid method and the parallel solution is based on message passing strategy. [19], considers the development of a parallel algorithm on a cluster of worstations, uses domain decomposition method for decomposing the problem domain into smaller subdomains. In [20], the SIMPLE based Single-Grid algorithm is improved by using the Multi-Grid method. Lastly, in [21] a parallel implementation of SIMPLE for simulating mixed convective flow is described.

During 2000-2002, studies were carried out to parallelize Şişecam's modelling programs by using Message Passing Interface (MPI) [1]. In this work, the grid is partitioned in the x-direction and each MPI process is assigned a sub-grid. TDMA and SIP are used as solvers. SIP is shown to be more eficient than TDMA. The results show that the speedup obtained on 8 processors is approximatelly 5.85.

Additionally, an alternative parallelization technique is also used in the previous work in which radiation field calculations are performed on different processors. Due

to high messaging overhead, this method was not shown to be effective.

One of the main disadvantage of the implementation in [1] is that, the input problem domain and the data structures need to be partitioned manually. This process currently makes the use of the program difficult. However, this manual partitioning process can be automated easily by writing a program.

3. PARALLELIZATION METHODOLOGY

In order to parallelize the sequential codes, the following subtasks are carried out:

(i) Conversion of sequential program from Fortran 77 to Fortran 90,
(ii) Integration of the SIP solver into the sequential code,
(iii) Analysis of the sequential code by performing various runs in order to locate functions/loops that have long running times,
(iv) Dependency analysis of time consuming loop,
(v) Reorganization of loops based on dependencies to make them more suitable for parallel execution,
(vi) OpenMP based multi-threaded parallelization,
(vii) Performance study of various loop iteration scheduling policies such as static, dynamic and guided (exponentially reducing chunks),
(viii) Performance studies of the parallel TDMA and SIP linear system solvers.

Sequential code is written in Fortran 77 format. As a starting point we converted it to Fortran 90 in Subtask 1. In Subtask 2, routine taken from [22] is integrated to the sequential code. After SIP integration, the overall code is profiled in order to locate parts that have long running times in Subtask 3. In Subtask 4, dependency analysis is performed on the most time consuming parts. In Subtask 5 some parts are reorganized in terms of dependencies so they can run concurrently. In Subtask 6 these parts are parallelized using OpenMP. Finally, in Subtask 7 and 8 performance is studied by employing different loop iteraton scheduling policies on TDMA and SIP linear solvers.

3.1. Locating Computationally Intensive Parts in the Sequential Code

We profile the sequential code in order to locate its computationally intensive parts. Sequential program runs iteratively until a convergence criterion is satisfied. For profiling, we run over the sequential program for 5 iterations. As an input, we use a file called *merged.dat* which contains the parameters listed in Table 3.1:

Table 3.1. Input Specifications for the Sequential Code.

Parameter	**Description**	**Value**
NII	Total # of nodes in the x-direction	82
NJJ	Total # of nodes in the y-direction	86
NKK	Total # of nodes in the z-direction	28
NPRT	Total # of inlet ports	9
NEXT	Total # of inlet ports	9
NBRN	Total # of burners	30
NPP	Total # of dependent variables to be solved	12
NMAX	The maxium of [NII, NJJ AND NKK]	86
NCL	Total # of control volumes in horizontal direction	40
NFF	Total # of angles in the phi-direction	10
NTT	Total # of angles in the theta-direction	6

While executing the program, we measure timings of all the functions and subroutines and observe that the repeated parts consume nearly 99% of the total time. The timing statistics are given Table 3.2 and Figure 3.1. As seen here, only the repeated Calculate Coefficients part of the program consumes the most time. The other parts (i.e Initialization Phase and etc.) are only executed once (even though the program itself runs for 5 iterations), and does not affect the overall performance of the program.

Table 3.2. Sequential Glass Furnace Simulation Profile for 5 Iterations.

Part	**Sub Parts**	**Avg Exec Time(sec)**	**# of Iters**
Main Program		25.162161	
	Calculate Coefficients	24.721543	5
	Initialization Phase	0.240593	1
	Save Data	0.200025	1

This means that Calculate Coefficients part needs to be analyzed and parallelized, since it makes up 99% of the total execution time. We apply the same profiling procedure to the main part, and list the obtained results in Table 3.3.

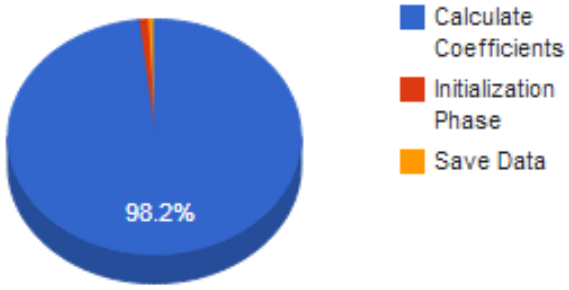


Figure 3.1. Distribution of timings through an iteration of main part of Glass Furnace Simulation.

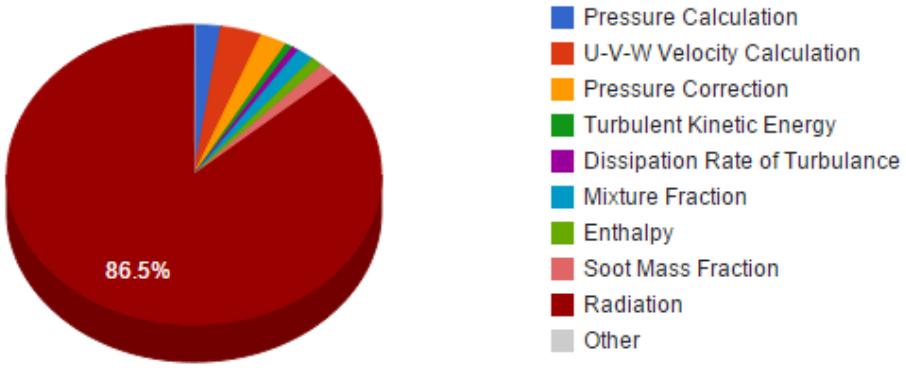


Figure 3.2. Average timing distribution across different parts of the Calculate Coeffficients part.

Figure 3.2 shows the average distribution of timings obtained for different sections of the Calculate Coeffficients part. It can be observed from Table 3.3 and Figure 3.2, that Radiation part of the Calculate Coeffficients consumes the largest portion of the execution time, or in other words 86.5% of the total time.

Before analyzing the Radiation part, lets analyze the other parts that consume the 13.5% portion. In all of the parts of Calculate Coefficients except Radiation, equations are solved using the TDMA or the SIP solver. We observe that on average ~70-75% of the execution time of all parts except that of Radiation, are spent in the TDMA or SIP solver parts respectively.

Table 3.3. Profile Of Sequential Calculate Coefficients part.

Part	**Sub Part**	**Avg Exec Time(sec)**	**# of Iters**
Calculate Coefficients		24.721543	5
	Pressure Calculation	0.571232	5
	U-V-W Velocity Calculation	0.874788	5
	Pressure Correction	0.571702	5
	Turbulent Kinetic Energy	0.160829	5
	Dissipation Rate of Turbulance	0.149432	5
	Mixture Fraction	0.367302	5
	Species Fraction	0.023145	5
	Enthalpy	0.268378	5
	Soot Mass Fraction	0.358505	5
	Radiation	21.376228	5

We also profile inner part of Radiation in more detail. Table 3.4 and Figure 3.3 presents the results obtained from profiling. The most time consuming part of the Radiation is the 3D Radiation part which forms i.e 99% of the time. Absorb part is responsible for the portion that is less than 1%. When 3D Radiation part is analyzed further, it is seen that it consumes the most time. This fact is shown in Figure 3.4, and the sequential profile of 3D Radiation is listed in Table 3.5.

Table 3.4. Profile of sequential Radiation part.

Part	**Sub Part**	**Avg. Exec. Time(sec)**	**# of Iters**
Radiation		21.376228	
	Absorb	0.208417	5
	3D Radiation	21.167811	5

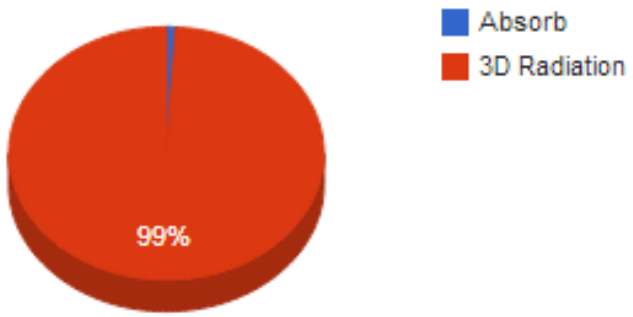


Figure 3.3. Timing distribution of Radiation part.

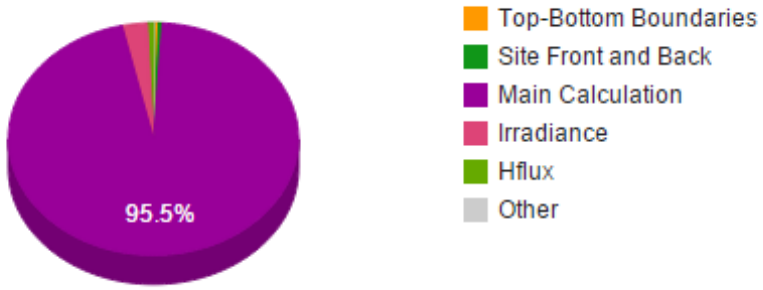


Figure 3.4. Timing distribution of 3D Radiation part.

3.2. Loop Structures of the Computationally Intensive Parts

From the analysis of both sequential programs, it can be concluded that there are 2 main parts on the program, that affect substantially the performance of the program. The most important one is the Main Calculation part of the 3D Radiation, and the second one is the linear equation solver (TDMA and SIP). In a single iteration, there are several calls to the Main Calculation until the radiation calculation algorithm converges. As a result parallelization of these parts can potentially increase the performance.

There are also other functions that can be parallelized. Approximately there are 40 such parts. However, we do not show their profile, since their effect on the performance will be relatively low.

Table 3.5. Sequential profile of 3D Radiation part.

Part	**Sub Part**	**Avg Exec Time(sec)**	**# of Iters**
3D Radiation		21.167811	
	Inlet Ports	0.000004	5
	Burners	0.000010	5
	Exit Ports	0.000003	5
	Top-Bottom Boundaries	0.012659	5 x ~10 (per iter)
	Site Front and Back	0.009102	5 x ~10 (per iter)
	Main Calculation	2.090310	5 x ~10 (per iter)
	Irradiance	0.061405	5
	Hflux	0.015062	5
	Radiative Flow	0.000011	5
	Update Radiative Fluxes	0.000049	5

3.2.1. 3D Radiation - Main Calculation

Main Calculation is the most time consuming part of calculation of radiation field. Figure 3.5 shows the inner structure of Main Calculation. There is a compute intensive six-level nested loop structure. This six-level nested loop structure is executed several times in a single iteration of Calculate Coefficients part, until the radiation algorithm converges. Hence, this part needs to be parallelized.

The first level-loop performs a total of 8 iterations. These iterations correspond to the 8 corners of the numerical grid (which represents the furnace). On every iteration, starting from a corner, two level-loops are executed which are responsible for combinations of angles in the theta and phi-directions. The innermost three-level loops iterate through the x,y and z-direction of the grid. Paralellisation of this part can be based on the outermost loop, by starting simultaneous execution from the corners. One issue that may arise is that, the outermost loop has a fixed, 8, number of iterations. If we employ more than 8 threads, some threads will be idling, which will decrease

efficiency of the parallel method.

```
– CORNERS OF THE GRID –
DO ( level - 1 ) NN=1,8
  – ANGLES IN THE THETA-DIRECTION –
  DO ( level - 2 ) LL=2, NTT-1
    – ANGLES IN THE PHI-DIRECTION –
    DO ( level - 3 ) MM=2, NFF-1
      – NODES IN THE Z-DIRECTION –
      DO ( level - 4 ) K=2, NKK
        – NODES IN THE Y-DIRECTION –
        DO ( level - 5 ) J=2, NJJ
          – NODES IN THE X-DIRECTION –
          DO ( level - 6 ) I=2, NII
            ......
          END DO
        END DO
      END DO
    END DO
  END DO
END DO
```

Figure 3.5. Loop structures of Main Calculation part of 3D Radiation.

Main Calculation part of 3D Radiation is parallelized using OpenMP directives. Parallelization is done on level-1 loop) by executing each corner simultaneously. Figure 3.6 shows the simultaneous execution starting from each corner of the numerical grid.

We make test runs in order to see if the produced outputs of the parallel program matches the ones with the sequential output. The results show that they actually do. When we run on 8 cores, the speedup of Main Calculation part is almost 6. This is a major speedup that will boost the overall speedup of the whole program.

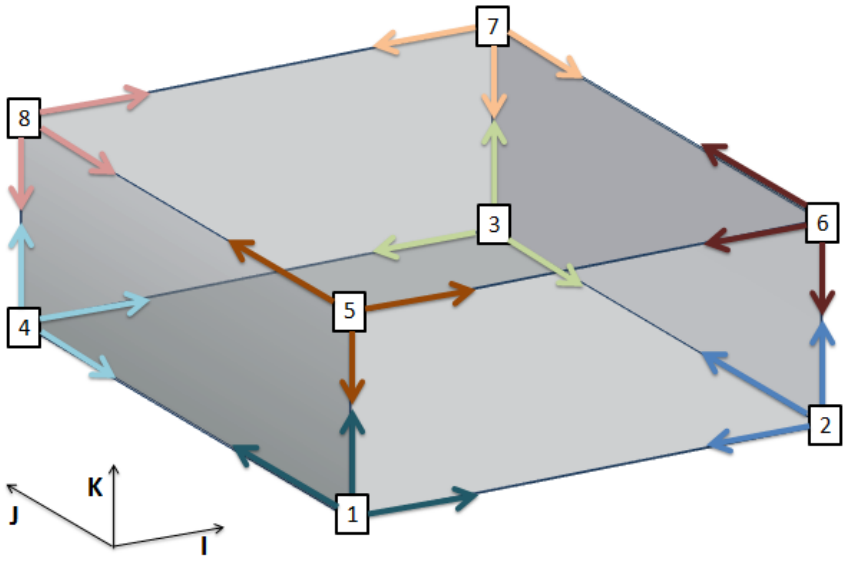


Figure 3.6. Schematic view of parallelisation of Main calculation part of Radiation. Starting from each corner, simultaneous processing on theta, phi, x, y and z direction.

Additionally, since the level-1 loop of this part is fixed with 8 iterations, when the program is run with a higher number of threads, no more speed-up can be obtained. To alleviate this problem, another approach that can be taken is to merge level-2 (theta-direction) and level-3 (phi-direction), and execute them in parallel while executing level-1(corners) sequentially. This approach is tested with more than 8 threads. Table 3.6 makes a comparison between the loop merging method and the aforementioned parallelization form method. It can be concluded that loop merging is more efficient than the current approach, especially when there are more than 8 threads.

3.2.2. TDMA

Figure 3.7 presents the inner structure of a single iteration involving TDMA. The solver scans all planes on z, y and x direction respectively. The planes have no dependency between them. Blocks of planes among the same direction can be executed concurrently. Figure 3.8 illustrates concurrent block executions shown in Figure 3.7.

After the code is parallelized, we also analyze the parallel and sequential outputs,

Table 3.6. Comparison of loop merging with the current parallelized form for 50 iterations, by using TDMA solver.

Method	**Core**	**Avg Exec Time(sec)**
Loop Merging	12	127
Current		170
Loop Merging	8	217
Current		235
Loop Merging	1	1321
Current		1329

and observe that they are same. We have also done a level-2 parallelization for a single iteration on TDMA. However, the speedup obtained is lower than the speedup gained in level-1 parallelization.

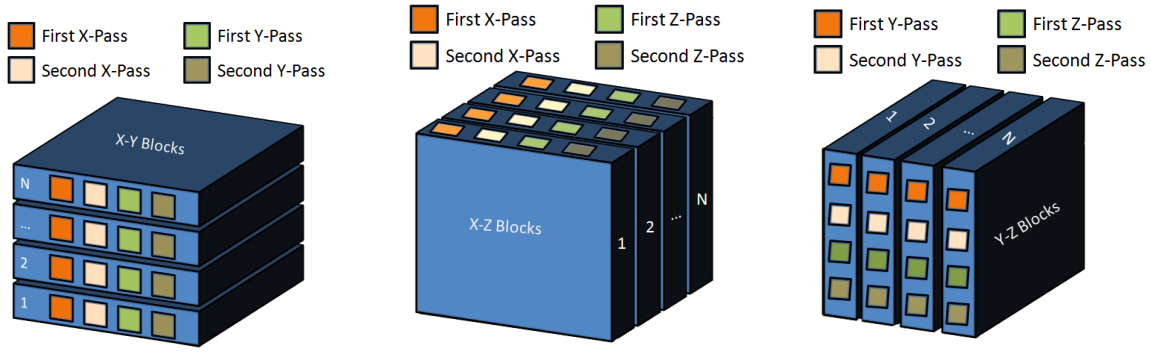
3.2.3. SIP

Figure 3.9 shows the inner structure of the modules related to SIP for a single iteration. The shown SIP solver modules have dependencies and cannot be parallelized directly. There is a need for rearrangement of the loop structure so that different parts can be executed in parallel.

To parallelize a single SIP iteration, **Pipeline-Parallel-Processing** [23] methodology is used for all the parts of SIP that are given on Figure 3.9. The furnace is divided up into chunks of a fixed size as shown in Figure 3.10. Level - 2 (middle loop) is parallelized (see Figure 3.9). The key point of algorithm is that any chunk is calculated provided its left neighbour chunk is processed. That is to say, the blocks that have the same color in Figure 3.10 are calculated in parallel by different cores. There is some overhead due to the opening and closing phases of this pipeline, since some cores will have to wait for some time in an idle state. When the grid size is large enough, this overhead can be tolerated.

DO(level-1)K=2,NKK	**DO(level-1)**J=2,NJJ	**DO(level-1)**I=2, NII
– 1st Y-Sweep –	*– 1st Z-Sweep –*	*– 1st Y-Sweep –*
DO(level - 2)J=2,NJJ	DO(level - 2)K=2,NKK	DO(level - 2)J=2,NJJ
DO(level-3)I=NII,2	DO(level-3) I=2,NII	DO(level-3)K=2,NKK
(Use TDMA).	(Use TDMA).	(Use TDMA).
END DO	END DO	END DO
END DO	END DO	END DO
– 2nd Y-Sweep –	*– 2nd Z-Sweep –*	*– 2nd Y-Sweep –*
DO(level - 2)J=2,NJJ	DO(level - 2)K=NKK,2	DO(level - 2)J=NJJ,2
DO(level-3) I=2,NII	DO(level-3)I=NII,2	DO(level-3)K=NKK,2
(Use TDMA).	(Use TDMA).	(Use TDMA).
END DO	END DO	END DO
END DO	END DO	END DO
– 1st X-Sweep –	*– 1st X-Sweep –*	*– 1st Z-Sweep –*
DO(level - 2) I=2, NII	DO(level - 2) I=2,NII	DO(level - 2)K=2,NKK
DO(level-3)J=2,NJJ	DO(level-3)K=2,NKK	DO(level-3)J=2,NJJ
(Use TDMA).	(Use TDMA).	(Use TDMA).
END DO	END DO	END DO
END DO	END DO	END DO
– 2nd X-Sweep –	*– 2nd X-Sweep –*	*– 2nd Z-Sweep –*
DO(level - 2)I=NII,2	DO(level - 2)I=NII,2	DO(level - 2)K=NKK,2
DO(level-3)J=NJJ,2	DO(level-3)K=NKK,2	DO(level-3)J=NJJ,2
(Use TDMA).	(Use TDMA).	(Use TDMA).
END DO	END DO	END DO
END DO	END DO	END DO
END DO	**END DO**	**END DO**

Figure 3.7. Loop structures of TDMA among different blocks for a single iteration.



(a) X-Y blocks. (b) X-Z blocks. (c) Y-Z blocks.

Figure 3.8. Simultaneous execution of 2D blocks, for a single iteration of TDMA.

(1)–ILU Decomposition– DO (level - 1) K=2,NKK DO (level - 2) J=2,NJJ DO (level - 3) I=2,NII .... END DO END DO END DO	**(3)–Forward Substitution–** DO (level - 1) K=2,NKK DO (level - 2) J=2,NJJ DO (level - 3) I=2,NII .... END DO END DO END DO
(2)–Residual Calculation– DO (level - 1) K=2,NKK DO (level - 2) J=2,NJJ DO (level - 3) I=2,NII .... END DO END DO END DO	**(4)–Backward Substitution–** DO (level - 1) K=NKK,2 DO (level - 2) J=NJJ,2 DO (level - 3) I=NII,2 .... END DO END DO END DO

Figure 3.9. Loop structures of modules related to SIP for a single iteration.

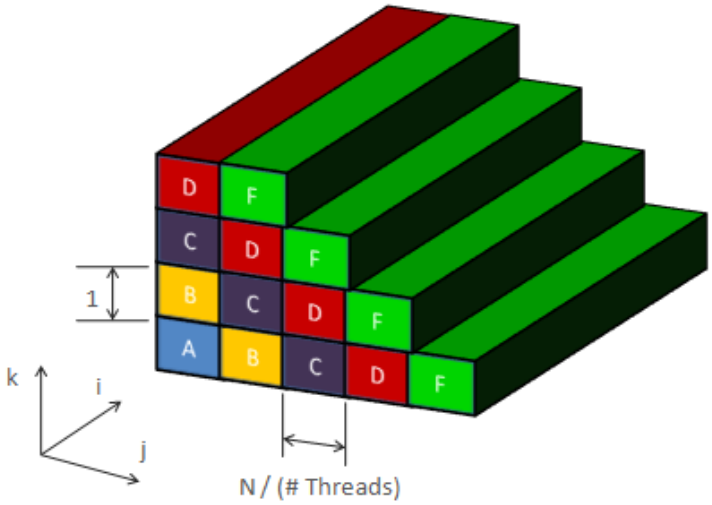


Figure 3.10. Schematic view of pipeline parallel processing. Blocks with equal colors (or letters) are calculated simultaneously.

3.2.4. Analysis of the Computationally Intensive Parts after Parallelisation

To analyze performance of parallelization, we run over the parallel program for 5 iterations on 8 cores. As an input, the file merged.dat with specifications in Table 3.1 is used.

While executing the program, we measure timings of all the functions and sub-parts that are under main program and calculate the speedups. Table 3.7 and Figure 3.11 shows the results obtained. Note that Calculate Coefficients part is the most important one, since it is repeated throughout the iterations.

Table 3.7. Parallel Glass Furnace Simulation Profile for 5 Iterations.

Part	**Sub Part**	**Avg Exec Time(sec)**	**# of Iters**	**Speedup on 8 Threads**
Main Program		4.739442		5.309098
	Calculate Coefficients	4.486279	5	5.510478
	Initialization Phase	0.089978	1	2.673909
	Save Data	0.163184	1	1.225762

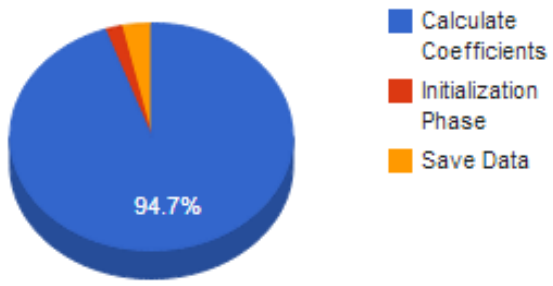


Figure 3.11. Distribution of timings among the single iteration of parallel Main Program.

Table 3.7 and Figure 3.11 concludes that there is a total speedup of approximately 5.51 in Calculate Coefficients part. Please also note that in all of these 5 iterations the Radiation is executed.

We apply the same analysis procedure to the parallel Calculate Coefficients part, and list the results in Table 3.8. Additionally, the Figure 3.12 illustrates the average distribution of timings across different parts.

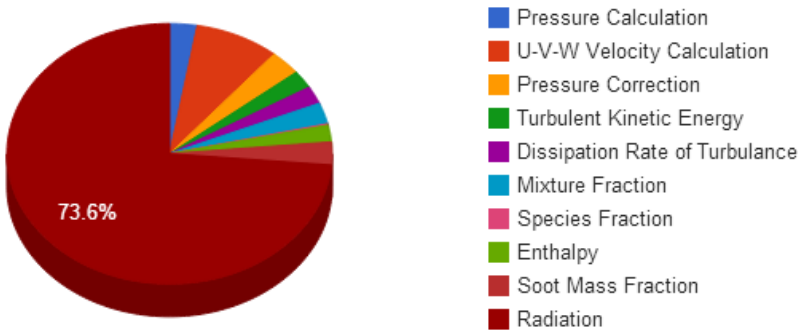


Figure 3.12. Average timing distributions across different parts of parallel Calculate Coefficients.

It can be observed from Table 3.8, that Radiation part of the Calculate Coefficients part has the highest speedup, ~6.47. Pressure Calculation and Pressure Correction parts have speedups ~4.66 and ~4.09 respectively. TDMA and SIP parts are not

Table 3.8. Profile of parallel Calculate Coefficients.

Part	**Sub Part**	**Avg Exec Time(sec)**	**# of Iters**	**Speedup on 8 Threads**
Calculate Coefficients		4.486279	5	5.510478
	Pressure Calculation	0.122641	5	4.657763
	U-V-W Velocity Calculation	0.376012	5	2.326489
	Pressure Correction	0.139821	5	4.088813
	Turbulent Kinetic Energy	0.099487	5	1.616582
	Dissipation Rate of Turbulance	0.099232	5	1.505885
	Mixture Fraction	0.117683	5	3.121115
	Species Fraction	0.008225	5	2.814153
	Enthalpy	0.093552	5	2.868753
	Soot Mass Fraction	0.127131	5	2.819964
	Radiation	3.302496	5	6.472749

included here, since, all of the parts of Calculate Coefficients except Radiation uses them. On the other hand, lets investigate the speedups related to the Radiation part: The details are listed on Table 3.9, and the timing distribution is illustrated in Figure 3.13.

Table 3.9. Profile of parallel Radiation part of Calculate Coefficients.

Part	**Sub Part**	**Avg Exec Time(sec)**	**# of Iters**	**Speedup on 8 Threads**
Radiation		3.302496	5	6.472749
	Absorb	0.019190	5	10.860874
	3D Radiation	3.283306	5	6.447103

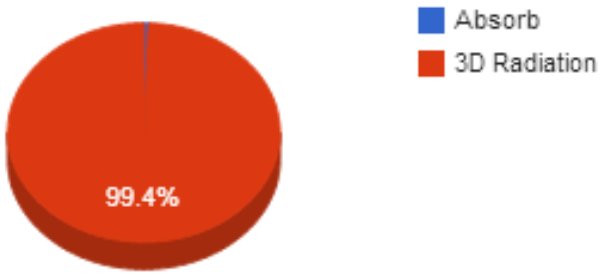


Figure 3.13. Timing distribution of parallel Radiation part of Calculate Coefficients.

Once again, the speedup of 3D Radiation part clearly seen. Lets see the speedup related to the inner part Speedup related to the inner part of 3D Radiation are listed on Table 3.10. The time distribution is illustrated in Figure 3.14.

Table 3.10. Parallel profile of 3D Radiation part.

Part	**Sub Part**	**Avg Exec Time(sec)**	**# of Iters**	**Speedup on 8 Threads**
3D Radiation		3.283306		6.447103
	Inlet Ports	0.000009	5	0.441506
	Burners	0.000001	5	10.485760
	Exit Ports	0.000006	5	0.503316
	Top-Bottom Boundaries	0.002077	5 x ~10 (per iter)	6.094547
	Site Front and Back	0.001594	5 x ~10 (per iter)	5.709924
	Main Calculation	0.350209	5 x ~10 (per iter)	5.968750
	IrradianceE	0.015662	5	3.920647
	Hflux	0.010984	5	1.371274
	Radiative Flow	0.000004	5	2.695896
	Update Radiative Flux	0.000047	5	1.043253

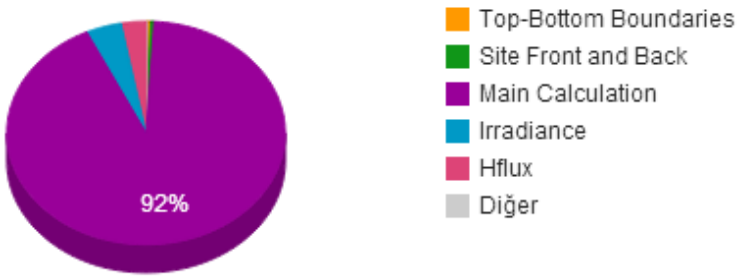


Figure 3.14. Timing distribution of 3D Radiation part.

Finally, we note that when we run the program with the input specified in Table 3.1, we get almost the same results. Table 3.11 shows the results of the last 5 iterations after a long run of 500 iterations for both sequential and parallel program on 8 threads.

Table 3.11. Outputs of Sequential & Parallel program when approaching 500'th iter.

ISTP	SSUM	HSUM	I	J	K	P	TEMP	O2
Sequential Output								
496	1.42E-04	4.13E+01	3	31	6	0.00E+00	1.50E+03	1.36E-01
497	1.18E-04	4.13E+01	3	22	6	0.00E+00	1.50E+03	1.36E-01
498	1.07E-04	4.13E+01	2	12	6	0.00E+00	1.50E+03	1.36E-01
499	1.04E-04	4.13E+01	3	22	6	0.00E+00	1.50E+03	1.36E-01
500	1.19E-04	5.28E+01	3	40	6	0.00E+00	1.50E+03	1.36E-01
Parallel Output								
496	1.16E-04	4.13E+01	3	31	6	0.00E+00	1.50E+03	1.36E-01
497	8.94E-05	4.13E+01	3	40	6	0.00E+00	1.50E+03	1.36E-01
498	8.42E-05	4.13E+01	3	22	6	0.00E+00	1.50E+03	1.36E-01
499	1.03E-04	4.13E+01	3	22	6	0.00E+00	1.50E+03	1.36E-01
500	1.18E-04	5.28E+01	3	40	6	0.00E+00	1.50E+03	1.36E-01

4. TESTS AND RESULTS

We test the parallelized code on the following hardware and software platforms:

- Intel Xeon E5430, 2.66 Ghz, 64 bit, 6M L2 Cache, 2 socket (4 cores in each of the sockets, totally 8 cores), 8GB of RAM
- Intel Fortran Compiler
- Unix Operating System

In the Table 4.1, we list the average running times when the program is executed with the input data given in Table 3.1. TDMA and SIP are used as linear solvers. The furnace that is modelled for the current work is a end-fired container furnace. Its dimensions are 16m length, 9.4m width and 3m crown height. This table tells that, time taken by the code that employs the SIP solver is shorter for each iteration as well as all for all iterations. If we make a speedup analysis, we can observe the following: when radiation is calculated on every iteration, the overall speedup with TDMA and SIP solvers will be approximately 5.44 and 5.37 respectively. On the other hand, when the radiation is calculated at every 5 iteration, the overall speedup with TDMA and SIP solvers will be approximately 4.30 and 3.61 respectively. Note when radiation computation is performed at every iteration, higher speed-up will result. This is due to the fact that radiation offers more parallelism and hence when its share in the overall computation is higher.

4.1. Comparison Of Linear System Solvers

In Table 4.2 the results of runs for sequential code with TDMA and SIP are given. The run is carried out until error is reduced to 2%. The plot given in Figure 4.1 shows that code with both solvers follow nearly the same convergence history. For the same test, the results for parallel runs are given in Table 4.3. Similar to that of the sequential case, the convergence history shown in Figure 4.2 follows the same path in both solvers. The results of Table 4.2 show that the code with the SIP solver is faster.

Table 4.1. Running Times and Speedup for the Parallel Program for 8 Threads for X = 82, Y = 86 and Z = 28 nodes (with no optimization flag - O0).

Solver Type	**Radiation on every X iteration**	**#of iterations**	**#of threads**	**Overall time (sec)**	**Time Per Iteration (sec) / (iter)**	**Speedup**
TDMA	1	100	1	2121	21.21	1
			8	390	3.9	5.44
	5	500	1	3650	7.3	1
			8	848	1.7	4.3
SIP	1	100	1	1889	18.89	1
			8	352	3.52	5.37
	5	500	1	2574	5.15	1
			8	714	1.43	3.61

Table 4.2. Performance of solvers on sequential runs with no optimization flags(until error is reduced to 2%).

Solver Type	**Core No.**	**Iteration No.**	**Single iteraton time (sec/iter)**	**Total Time(min)**
TDMA	1	4925	6.66	546.5
SIP	1	4950	4.57	376.7

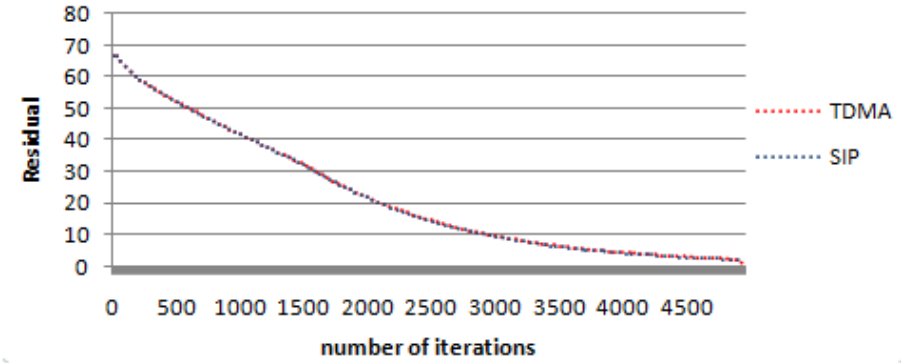


Figure 4.1. Convergence plots for sequential code with TDMA and SIP solvers with no optimization flags (until error is reduced to 2%).

Table 4.3. Performance of Solvers on Parallel Runs (with no optimization flags - O0).

Solver Type	**Core No.**	**Iteration No.**	**Single iteraton time (sec/iter)**	**Total Time(min)**	**Speedup comp. to seq. version**
TDMA	8	4910	1.59	130.2	4.2
SIP	8	4925	1.33	109.1	3.45

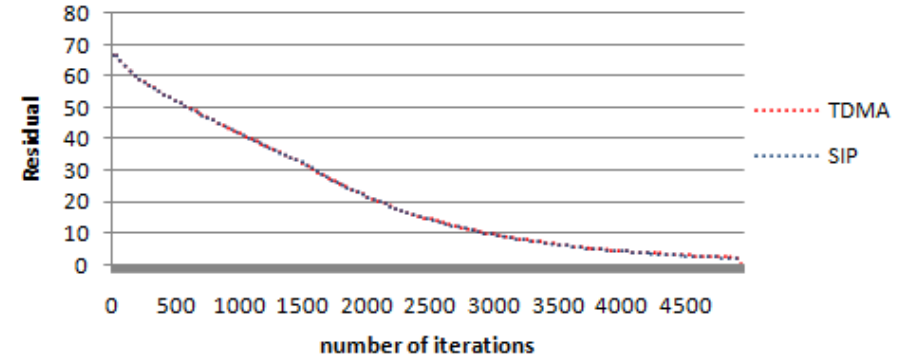
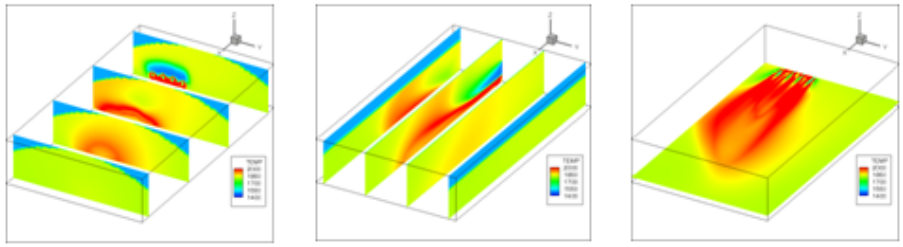


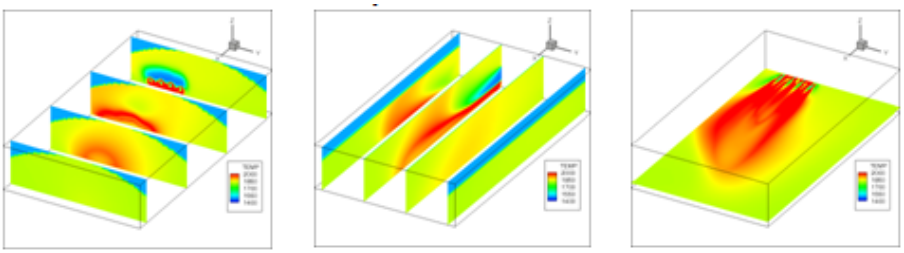
Figure 4.2. Parallel convergence curves for linear TDMA and SIP solvers (with no optimization flags - O0).

Finally, to verify that the solutions computed by the original sequential code and the parallelized code produce the same solution, we have plotted the temperature distributions at various cross-sections. Figures 4.3 and 4.4 shows side-view of temperature distribution among combustion space at several x, y and z cross-sections respectively.

The solutions obtained are indeed identical.

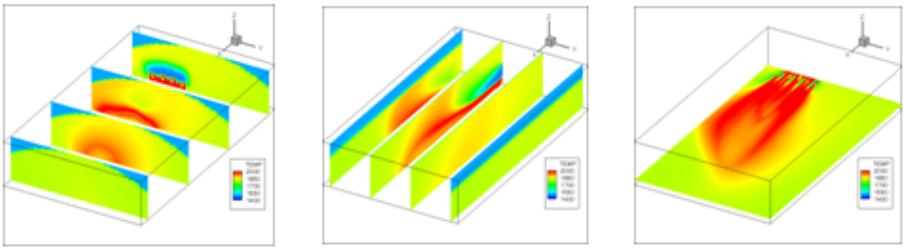


(a) Sequential Solution

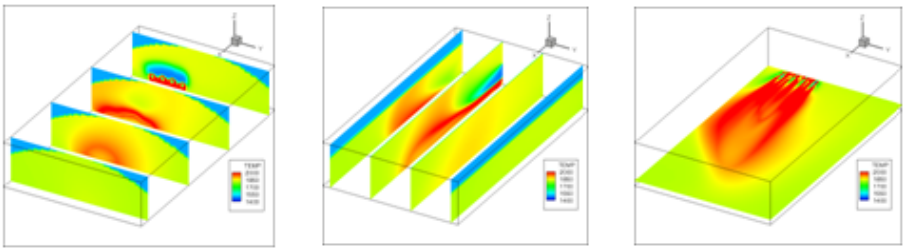


(b) Parallel Solution

Figure 4.3. Temperature distribution for computation using TDMA solver (with no optimization flags - O0).



(a) Sequential Solution



(b) Parallel Solution

Figure 4.4. Temperature distribution for computation using SIP solver (with no optimization flags - O0).

4.2. Efficiency and Scaling Analysis

In Tables 4.4 and 4.5, we give the speedup (equation 2.1) and efficiency values (equation 2.2) for tests on core numbers ranging from 1 to 8. It is observed that efficiency decreases as the number of cores increases up to 7. At 8 cores, efficiency increases slightly. Although the efficiency decreases as the number of cores is increased, in Figure 4.5 it is observed that execution time is still reduced. Additionaly, in Figure 4.6 a speed-up graph is plotted for both solvers.

4.3. Effects of Compiler Optimizations and OpenMP Loop Scheduling Policies

We also perform tests in order to measure the impact of different compiler optimization flags and OpenMP loop scheduling methods. Tests were performed with compiler optimization flags O0, O1, O2 and O3 and OpenMP loop scheduling policies static, guided and auto). Compiler optimizations O2 and O3 roughly makes the non-optimized code (i.e. compiled with O0) twice as fast. Even though O3 does more agressive optimization, O2 optimization leads to faster code than O3. As far as the loop scheduling methods are concerned, there is only slight differences in the timings obtained from each of them. The execution times and the speedups obtained for various combinations of compiler optimization flags and loop scheduling methods are shown in Table 4.6. The best performance is obtained when O2 compiler and static scheduling method is used. The reason why static scheduling produces the best performance is because static scheduling divides the loop iterations evenly among the threads. Since at each iteration more or less the same code is executed (i.e. each iteration has equal work), iterations are load balanced.

Since Static-Balanced method gives the best results, chunk size is not provided explicitly. However, to be sure, we have made several test runs to test the effect of chunk size. The results are shown in Table 4.6. For scheduling types guided and dynamic, a heuristic for the best iteration chunk size seems to be:

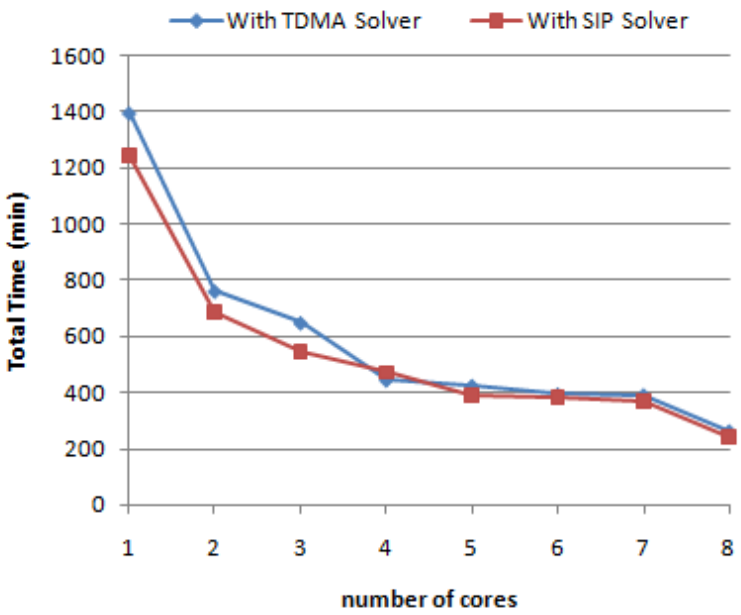
$$NII \,/\, \#ofthreads$$

Table 4.4. Efficiency and Speedup analysis for TDMA solver (with no optimization flags - O0).

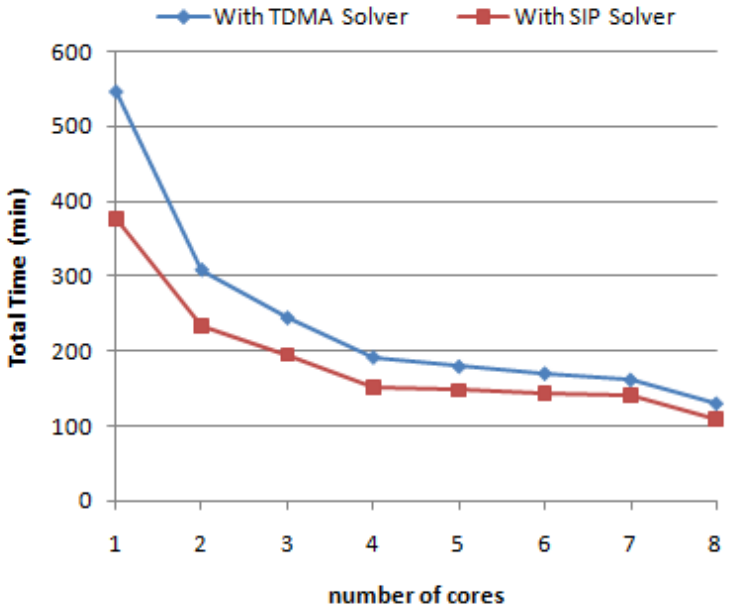
Core No.	**Iteration No.**	**Single iteraton time (sec/iter)**	**Total Time (min)**	**Speed-up**	**Efficiency**
Radiation On Every Iteration					
1	4984	6.66	1396.9	1	1
2	4998	9.12	759.3	1.84	0.92
3	4971	7.80	646.2	2.16	0.72
4	5020	5.27	441.3	3.17	0.79
5	4983	5.08	421.9	3.31	0.66
6	4984	4.71	391.3	3.57	0.59
7	4985	4.67	388.3	3.60	0.51
8	5002	3.11	259.3	5.39	0.67
Radiation On Every 5'th Iteration					
1	4925	6.66	546.5	1	1
2	4920	3.77	308.9	1.77	0.885
3	4920	2.98	244.7	2.23	0.744
4	4930	2.34	191.9	2.85	0.712
5	4925	2.19	179.6	3.09	0.619
6	4930	2.07	170.2	3.21	0.535
7	4920	1.97	161.9	3.38	0.482
8	4910	1.59	130.2	4.2	0.525

Table 4.5. Efficiency and Speedup analysis for SIP solver (with no optimization flags - O0).

Core No.	**Iteration No.**	**Single iteraton time (sec/iter)**	**Total Time (min)**	**Speed-up**	**Efficiency**
Radiation On Every Iteration					
1	5038	6.66	1242.9	1	1
2	5000	8.23	686.1	1.81	0.91
3	5000	6.51	542.3	2.29	0.76
4	5000	5.63	468.8	2.65	0.66
5	4973	4.71	390.1	3.19	0.64
6	4985	4.57	380.1	3.27	0.54
7	4971	4.42	366.5	3.39	0.48
8	4966	2.88	238.5	5.21	0.65
Radiation On Every 5'th Iteration					
1	4950	4.57	376.7	1	1
2	4950	2.83	233.2	1.62	0.808
3	4930	2.37	195	1.93	0.644
4	4925	1.85	151.7	2.48	0.621
5	4930	1.8	148.1	2.54	0.509
6	4930	1.75	144.1	2.61	0.436
7	4950	1.7	140.6	2.68	0.383
8	4925	1.33	109.1	3.45	0.432

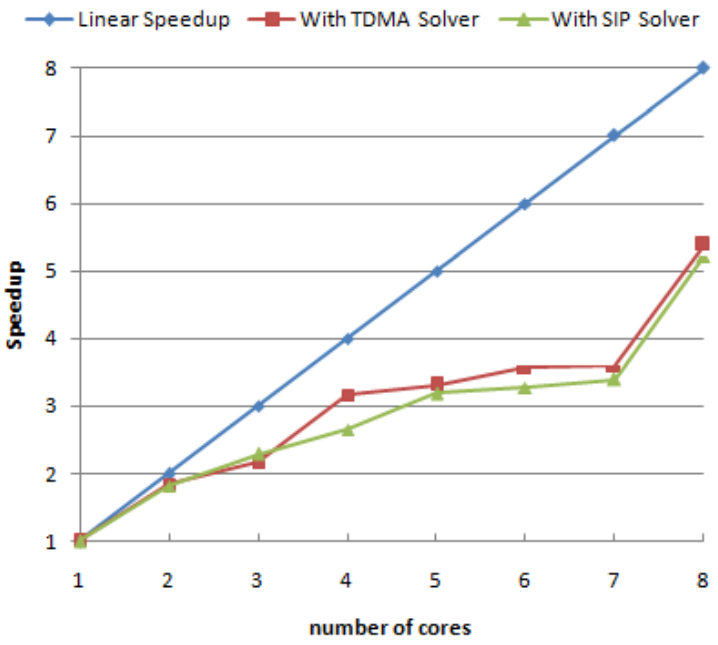


(a) Radiation on every iteration

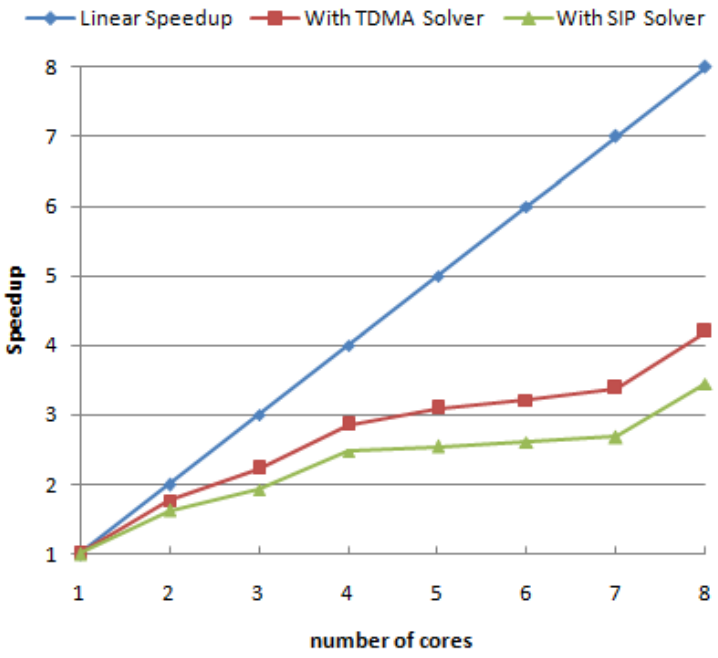


(b) Radiation on every 5'th iteration

Figure 4.5. Total computation times for TDMA and SIP solvers with no optimization flags - O0 (until error is reduced to 2%).



(a) Radiation on every iteration



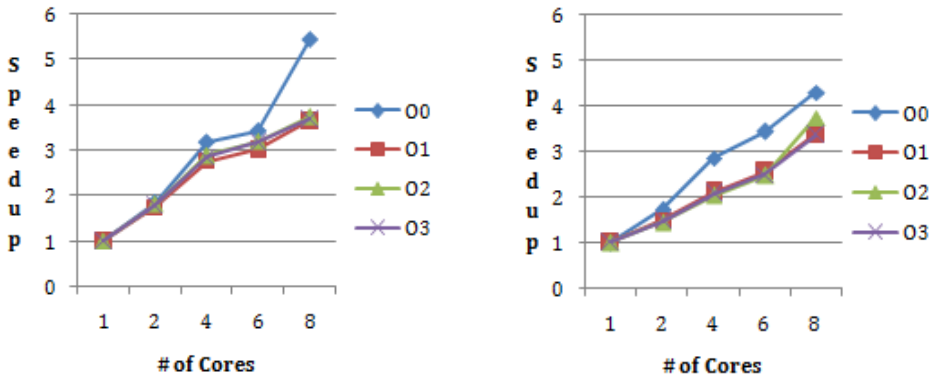
(b) Radiation on every 5'th iteration

Figure 4.6. Speed-up analysis for TDMA and SIP solvers with no optimization flags - O0 (until error is reduced to 2%).

In Table 4.6 the running time of program when O1, O2, O3 optimizations are used is shown. Especially, when the running time of program with O2 optimization is compared to the program executed without optimizations, it is seen that there is a speedup of 2. It is also observed that, O3 is not giving faster running times when compared to O2. Since numerical codes may be sensitive to code changes by the compiler, it is safer to use the O2 optimization level than on O3. Note that TDMA solver is used on the analysis that is done in Table 4.6. SIP is not used here since relative speedups are calculated.

Sequential program does its calculations on regular grids, so that the calculations inside the loops are regular. In short, iterations among a particular loop perform the same calculations. Thus, there is no load inbalance between iterations. Due to the fact that static-balanced loop scheduling of OpenMP shares the load between the processors in equal manner, and since it divides the loop in a static manner with less overhead, it can be concluded that it performs best when compared to other scheduling methods.

In Table 4.6, the speedups are calculated by dividing the O<level >optimized sequential program time by O<level >optimized parallel program time. For instance, the speedup of parallel O2 is calculated by dividing the running time of sequential O2 to the running time of parallel O2. It has a speedup of 4 when it is parallelized using OpenMP. This fact is also shown in Figure 4.7.

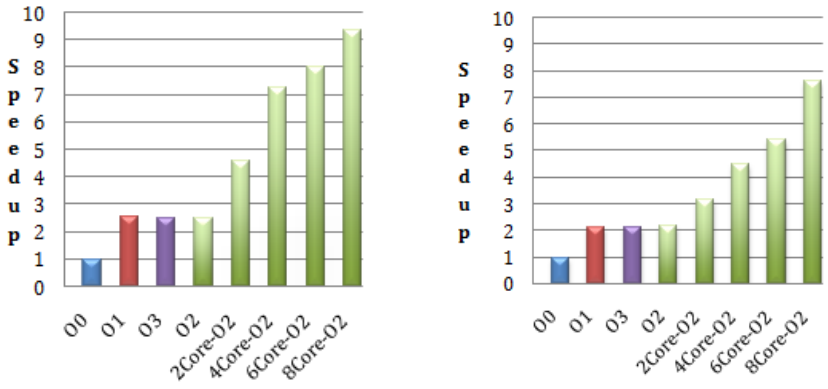


(a) Radiation on every iteration (b) Radiation on every 5'th iteration

Figure 4.7. Speedup Analysis for Intel compiler optimizations.

Table 4.6. Running Time and Speedups of the Program when Executed Using Compiler Optimizations.

Glass Furnace Simulation Program with TDMA SOLVER												
				STATIC Balanced			**GUIDED (chunk=10)**			**AUTO**		
Program(Optimization)	**# of Threads**	**Rad. on every X iter.**	**# of iterations**	**Overal Time (sec)**	**Time Per Iteraton (sec)**	**OpenMP Speedup**	**Overal Time (sec)**	**Time Per Iteraton (sec)**	**OpenMP Speedup**	**Overal Time (sec)**	**Time Per Iteraton (sec)**	**OpenMP Speedup**
Par. (O2)	8	1	100	226	2.26	3.73	232	2.32	3.6	230	2.30	3.64
		5	500	479	0.958	3.44	495	0.99	3.33	488	0.98	3.40
Par. (O3)	8	1	100	228	2.28	3.69	242	2.42	3.46	234	2.34	3.60
		5	500	507	1.014	3.35	495	0.99	3.43	503	1.01	3.37
Par. (O1)	8	1	100	227	2.27	3.66	250	2.5	3.36	245	2.45	3.44
		5	500	511	1.022	3.38	509	1.018	3.39	509	1.02	3.39
Par. (O0)	8	1	100	390	3.9	5.44	402	4.02	5.28	403	4.03	5.26
		5	500	848	1.696	4.3	866	1.732	4.21	861	1.72	4.24
Seq. (O2)	1	1	100	843	8.43	-	836	8.36	-	838	8.38	-
		5	500	1650	3.3	-	1650	3.3	-	1657	3.31	-
Seq. (O3)	1	1	100	842	8.42	-	837	8.37	-	843	8.43	-
		5	500	1700	3.4	-	1700	3.4	-	1697	3.39	-
Seq. (O1)	1	1	100	831	8.31	-	841	8.41	-	843	8.43	-
		5	500	1725	3.45	-	1725	3.45	-	1725	3.45	-
Seq. (O0)	1	1	100	2121	21.21	-	2121	21.21	-	2121	21.21	-
		5	500	3650	7.3	-	3650	7.3	-	3650	7.30	-



(a) Radiation on every iteration (b) Radiation on every 5'th iteration

Figure 4.8. Speedup analysis of Intel's O2 compiler optimization relative to O0.

4.4. Effect of Problem (Grid) Size on Speedup

Since OpenMP runs loops by dividing them into chunks (equal chunks in case of static-balanced scheduling), it incurs an overhead. This overhead can be negligible if the input size is big enough, because as the input size increases, the chunks start to increase, however, the overhead remains the same. In this section, we investigate this issue by executing the program on different input sizes.

The TDMA part of the program is parallelized without changing its structure, since there is no dependency among blocks of planes. It only has overhead related to OpenMP thread management. However this is not the case in SIP. Almost all the loops inside this solver have a highly dependent structure. Thus, the paralellization involves modifications in the structure of loops which in turn means SIP will have extra overhead.

In [23], it is pointed out that the paralelized SIP solver will perform better and better as the input size increases, since the overhead related to the modified structure of the algorithm will be negligible. This can be observed in Figure 3.10. As the node size increases more blocks can be processed in parallel. Thus as the input size increases, it is expected that is SIP's running time will be relatively lower than TDMA.

In Table 4.8 and 4.9, the programs with TDMA and SIP solvers are run using the three different data sets given in Table 4.7. Additionally, radiation is executed on both implementations at every and at every 5th iteration.

The effect of increasing the input size on both of the solvers can be seen clearly on Figure 4.9 and 4.10. Program with the SIP solver has a better runnning time than the program with the TDMA with an increasing performance gap.

Table 4.7. Input specifications for three data sets that have roughly 200K, 1M and 2M nodes.

Parameter	**Data #1** (~**200K Nodes**)	**Data #2** (~**1M Nodes**)	**Data #3** (~**2M Nodes**)
NII	82	323	641
NJJ	86	86	86
NKK	28	36	36
NPRT	9	9	9
NEXT	9	9	9
NBRN	30	30	30
NPP	12	12	12
NMAX	86	86	86
NCL	40	40	40
NFF	10	10	10
NTT	6	6	6

Table 4.8. Running times for varying data parameters, when using TDMA solver for 100 iteration. Radiation is calculated on every and every fifth iteration.

Data	**Radiation on every X iter.**	**Overal Time (min)**
Data #1	1	14.1
	5	6.5
Data #2	1	77.6
	5	33.8
Data #3	1	160.8
	5	69.1

Table 4.9. Running times for varying data parameters, when using SIP solver for 100 iteration. Radiation is calculated on every and every fifth iteration.

Data	**Radiation on every X iter.**	**Overal Time (min)**
Data #1	1	11.9
	5	5.86
Data #2	1	65.4
	5	31.3
Data #3	1	130
	5	61.4

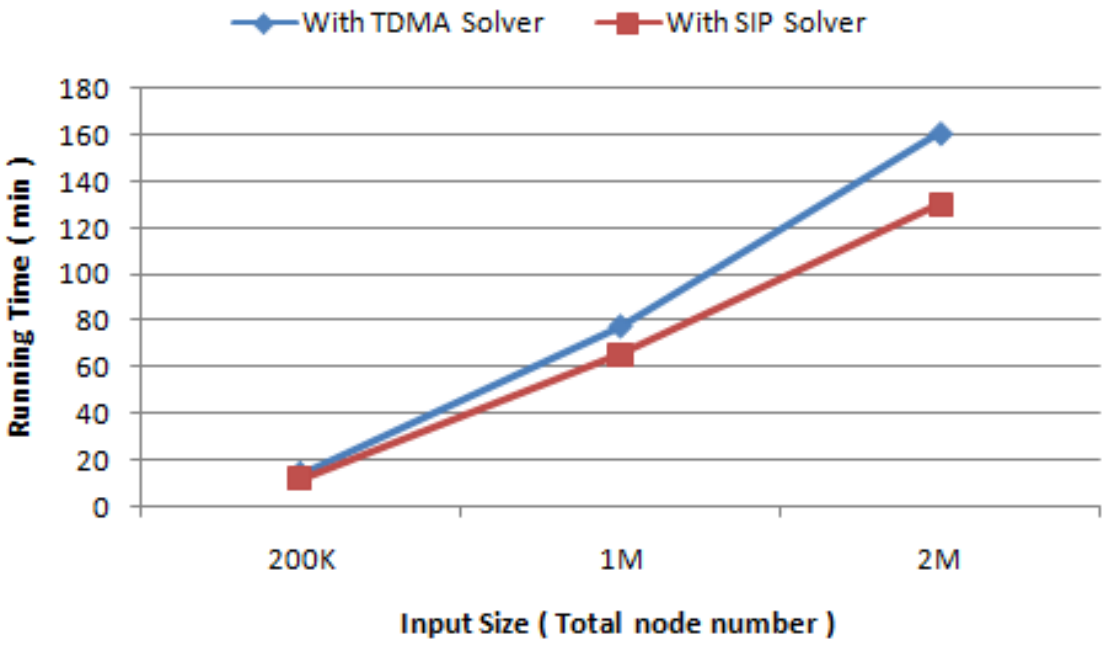


Figure 4.9. Running times of program with different solvers as the input size increases, where Radiation is calculated on every iteration.

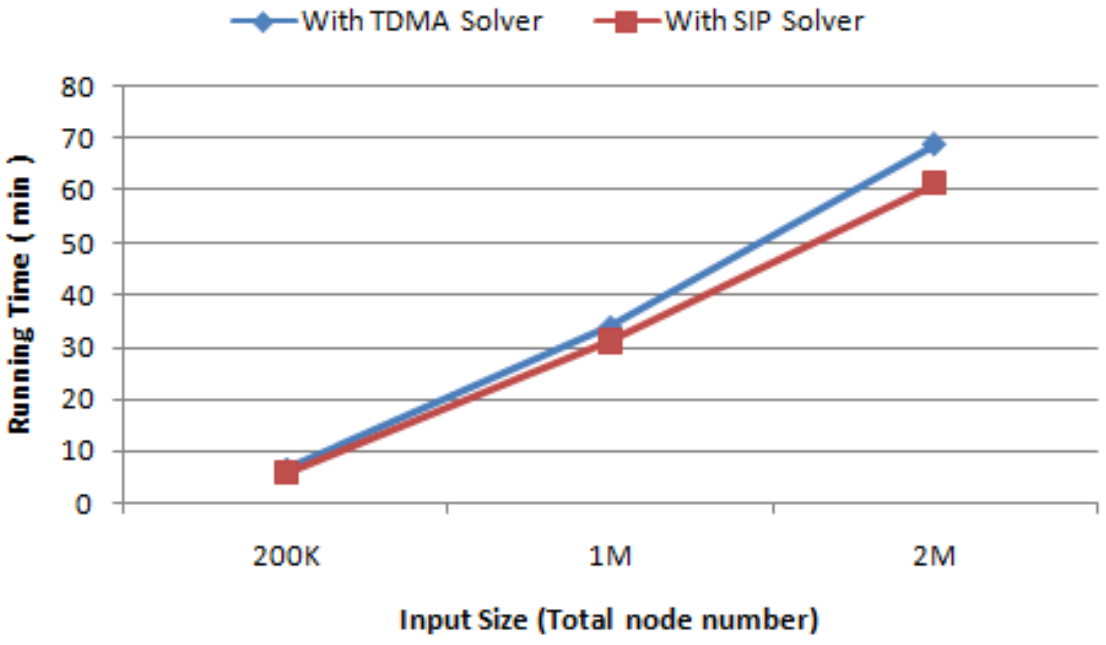


Figure 4.10. Running times of program with different solvers as the input size increases, where Radiation is calculated on every fifth iteration.

5. CONCLUSIONS

This thesis contributes OpenMP shared memory parallelization work for Sisecam's sequential Glass Furnace Simulation programs. The solution method that is used in the Glass Furnace Simulation is based on the SIMPLER method. The program consists of 11 phases all of which are parallelized. Most of the computations take place in the radiation and linear system solvers (TDMA and SIP). Hence, our parallelization efforts concentrated on these two parts. As a result of parallelization, speed-ups of roughly 4 was attained on 8 core Xeon systems. In the past, the program was used without any compiler optimizations. The use of O2 compiler optimization also doubles the speed of the program. As linear system solvers, TDMA or SIP can be used. The tests show that the code that uses SIP solver runs faster than that of TDMA. The results show that as the problem (grid) size is increased the speed-ups obtained indeed increase showing that there is no growth of overhead due to increased problem size.

As future work, we plan to carry out more tests on systems with higher number of cores. We believe that the parallelized code will scale when larger number of cores are used. In particular, it will be interesting to port the code to Xeon-Phi systems.

REFERENCES

1. Oruç, O., L. Önsel, Z. Eltutar and C. Özturan, "Parallel Computing of Combustion Space Model for Glass Furnaces", *AcerS Glass & Optical Materials Division*, 2002.

2. OpenMP, "OpenMP Application Program Interface Version 3.0", 2008, `http://www.openmp.org/mp-documents/spec30.pdf`, [Accessed January 2014].

3. Ferziger, J. H. and M. Perić, *Computational Methods For Fluid Dynamics*, Vol. 3, Springer Berlin, 2002.

4. McDonough, J. M., "Lectures on Computational Numerical Analysis of Partial Differential Equations", 2002, Departments of Mechanical Engineering and Mathematics University of Kentucky, `http://www.engr.uky.edu/~acfd/me690-lctr-nts.pdf`, [Accessed January 2014].

5. Patankar, S., *Numerical Heat Transfer and Fluid Flow*, CRC Press, New York, USA, 1980.

6. Thomas, L. H., "Elliptic Problems in Linear Differential Equations Over a Network", Technical Report, Columbia University, New York, 1949.

7. Murthy, J. Y., "Numerical Methods in Heat, Mass, and Momentum Transfer", Draft Notes for ME 608, School of Mechanical Engineering Purdue University, 2002.

8. Raithby, G. and E. Chui, "A Finite-Volume Method for Predicting a Radiant Heat Transfer in Enclosures with Participating Media", *Journal of Heat Transfer*, Vol. 112, No. 2, pp. 415–423, 1990.

9. Prasad, J. K. and P. V. Patil, "Finite Volume Numerical Grid Technique for Multidimensional Problems", *International Journal of Science and Research*, 2014.

10. Versteeg, H. K. and W. Malalasekera, *An Introduction to Computational Fluid Dynamics. The Finite Volume Method*, Longman Group Ltd., London, 1995.

11. Stone, H. L., "Iterative Solution of Implicit Approximations of Multidimensional Partial Differential Equations", *SIAM Journal on Numerical Analysis*, Vol. 5, No. 3, pp. 530–558, 1968.

12. Barney, B., "The Message Passing Interface (MPI)", 2012, `https://computing.llnl.gov/tutorials/mpi/`, [Accessed December 2013].

13. Geist, A., *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*, MIT press, 1994.

14. Barry, W., *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2/E*, Pearson Education India, 2006.

15. Intel®, "Threading Building Blocks(Intel® TBB)", 2014, `https://www.threadingbuildingblocks.org/`, [Accessed September 2014].

16. Barney, B., "The POSIX Threads Programming", 2014, `https://computing.llnl.gov/tutorials/pthreads/`, [Accessed September 2014].

17. "Quick-Reference Guide to Optimization with Intel Compilers Version 12", Technical Report, Intel Corporation, 2010.

18. Schäfer, M., E. Schreck and K. Wechsler, "An Efficient Parallel Solution Technique for the Incompressible Navier-Stokes Equations", *Numerical Methods for the Navier-Stokes Equations*, pp. 228–238, Springer, 1994.

19. Manshoor, B., M. N. Wan Hassan and N. Alias, "Incompressible Flow Simulation Using SIMPLE Method on Parallel Computer", *Journal of Science and Technology*, Vol. 1, No. 1, 2011.

20. Bernert, K., T. Frank, H. Schneider and K. Pachler, "Multi-Grid Acceleration of

a SIMPLE-Based CFD-Code and Aspects of Parallelization", *Cluster Computing, 2000. Proceedings. IEEE International Conference on*, pp. 217–223, IEEE, 2000.

21. Saldana, J. B., V. Sarin and N. Anand, "Parallelization of a Simple-Based Algorithm to Simulate Mixed Convective Flow Over a Backward-Facing Step", *Numerical Heat Transfer, Part B: Fundamentals*, Vol. 56, No. 2, pp. 105–118, 2009.

22. Iaccarino, G., "Sequential SIP Solver Code", 1995, `http://web.stanford.edu/class/me469a/codes/solvers/lapl3d.f`, [Accessed May 2014].

23. Deserno, F., "Basic Optimization Strategies for CFD-Codes", Technical Report, Regionales Rechenzentrum Erlangen, Friedrich-Alexander-Universität Erlangen-Nürnberg Martenstr, `https://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/Projekte/OptGuide.pdf`, [Accessed July 2013].