University of Kentucky
UKnowledge

University of Kentucky Master's Theses

Graduate School

2006

PERFORMANCE EVALUATION AND OPTIMIZATION OF THE UNSTRUCTURED CFD CODE UNCLE

Saurabh Gupta
University of Kentucky, saurabh737@yahoo.com

Recommended Citation

Gupta, Saurabh, "PERFORMANCE EVALUATION AND OPTIMIZATION OF THE UNSTRUCTURED CFD CODE UNCLE" (2006). *University of Kentucky Master's Theses*. 360.
https://uknowledge.uky.edu/gradschool_theses/360

ABSTRACT OF THESIS

PERFORMANCE EVALUATION AND OPTIMIZATION OF THE UNSTRUCTURED CFD CODE UNCLE

Numerous advancements made in the field of computational sciences have made CFD a viable solution to the modern day fluid dynamics problems. Progress in computer performance allows us to solve a complex flow field in practical CPU time. Commodity clusters are also gaining popularity as computational research platform for various CFD communities. This research focuses on evaluating and enhancing the performance of an in-house, unstructured, 3D CFD code on modern commodity clusters. The fundamental idea is to tune the codes to optimize the cache behavior of the node on commodity clusters to achieve enhanced code performance. Accordingly, this work presents discussion of various available techniques for data access optimization and detailed description of those which yielded improved code performance. These techniques were tested on various steady, unsteady, laminar, and turbulent test cases and the results are presented. The critical hardware parameters which influenced the code performance were identified. A detailed study investigating the effect of these parameters on the code performance was conducted and the results are presented. The successful single node improvements were also efficiently tested on parallel platform. The modified version of the code was also ported to different hardware architectures with successful results. Loop blocking is established as a predictor of code performance.

KEYWORDS: Unstructured CFD Code Optimization, Commodity Clusters, Cache Performance Enhancement, Data Access Optimization, Loop Blocking

Saurabh Gupta

04/14/06

PERFORMANCE EVALUATION AND OPTIMIZATION OF THE UNSTRUCTURED CFD CODE UNCLE

By

Saurabh Gupta

Dr. Raymond P. LeBeau, Jr.

Director of Thesis

Dr. P. G. Huang

Director of Graduate Studies

014/14/2006

THESIS

Saurabh Gupta

The Graduate School
University of Kentucky
2006

PERFORMANCE EVALUATION AND OPTIMIZATION OF THE UNSTRUCTURED CFD CODE UNCLE

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in Mechanical Engineering in the
College of Engineering at the University of Kentucky

By
Saurabh Gupta
Director: Dr. Raymond P. LeBeau, Assistant Professor of Mechanical Engineering
Lexington, KY
2006

Dedication

To my family, friends and my advisor

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge the help of many people who made this thesis possible. I cannot overstate my gratitude to my advisor Dr. Raymond P. LeBeau Jr. He has been a constant source of motivation and inspiration through out this work. I would like to thank him for his sound advice, good company and numerous insightful conversations during the developments of the ideas used in this thesis.

I would like to thank Dr. P.G. Huang for letting me use his code UNCLE for the purpose of this research and for his valuable time for being on my committee. I would also like to thank my defense committee member Dr. J.D. Jacob for taking time to evaluate my work.

I am grateful to my student colleagues for their stimulating discussions and providing a fun environment in which to learn and grow.

I am thankful to all the teachers who have taught me through out my academic career and imparted me knowledge which helped me to be the person I am.

Finally, I am forever indebted to my parents Suman and Lal Chandra Gupta and my sister Bhavana for their unconditional love and support throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

1.1 Overview

The advent of technology has brought us sophisticated computational tools. With the presence of state of the art supercomputers, computational solutions to fluid dynamics problems have become a more viable option. Computational Fluid Dynamics (CFD) has grown in popularity as a potent method for solutions of practical Fluid dynamics problems. Now that we have an established alternate method for solving fluid dynamics problems, improving these known methods seem like a logical step in progression. CFD simulations are typically done on very sophisticated computational platforms (e.g. NASA's supercomputer '*Columbia*'). As these computational facilities are usually in short supply, the time available for computation on these kinds of machines is not in abundance. Added to the problem of time constraint one is often faced with the fiscal problem associated with the costly computational times of these machines.

With these existing problems it is not desirable to run unoptimized codes on the supercomputers. Unoptimized codes will lead to inefficient use of the computational time and will prove costly to run. One of the ways to overcome this problem is to improve the hardware architecture of the computational platforms but this area is pretty much limited by the technological advances made in that field over which a CFD researcher has no control. So a logical solution to the problem would be to optimize the existing CFD code(s) to make them run faster so that the computational time could be used more effectively and economically.

For most problems CFD simulations often lag behind the real physical time which means that one minute of CFD simulation time takes more than one minute of the physical time. Optimization of the CFD codes may not be able to remove the disparity totally but would be an initial step in that process.

1.2 What is CFD?

CFD is sophisticated computer based design and analysis technique that enables one to model and study the dynamics of any thing that flows. It is based upon complex sets of non-linear mathematical equations that describe the fundamentals of fluid, heat and mass transport. The concerned physical domain is divided into a computational grid. These equations are then solved iteratively, on each small grid cell, with the use of very complex computer algorithms or a code. It allows the user to build virtual models of a fluid flow problem and assign physical and chemical properties to the fluid in the model and CFD then predicts the solution. CFD can provide solutions for problems concerned with flow of gases and liquids, moving bodies, heat and mass transfer, chemical reactions, multi-phase physics, fluid structure interaction, and acoustics with computer based modeling. It was primarily used in fields of aerospace, maritime, and meteorological sciences and is widely used in automobile and chemical industries and is rapidly gaining popularity in other fields. A sample CFD grid for space shuttle launch vehicle and a naval ship is shown in Fig 1.1[1]

1.3 Why CFD?

CFD has a couple of inherent advantages over the traditional methods used to solve the fluid dynamics problems. Some of the major ones are

Insight

Often when a user is faced with a design which is difficult to model or test through experimental techniques, CFD can provide insight by virtually moving into the user's design and test its performance. It allows the user to witness phenomena which is not otherwise possible by other means. It provides all desired information at every point of the computational domain which is often not possible by traditional methods.

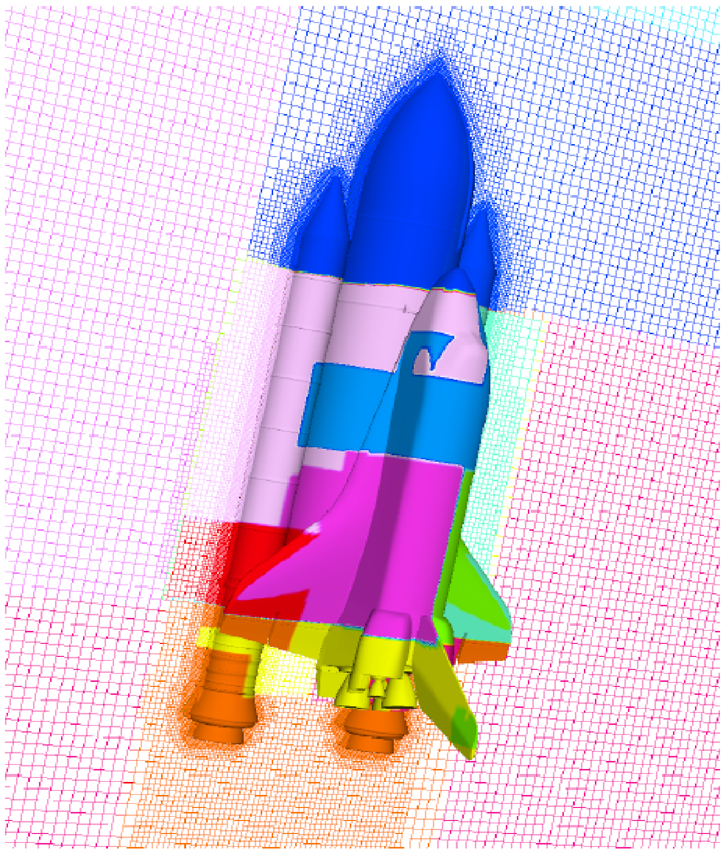


Figure 1.1(a) A CFD grid for a space shuttle launch vehicle (SSLV) [1]

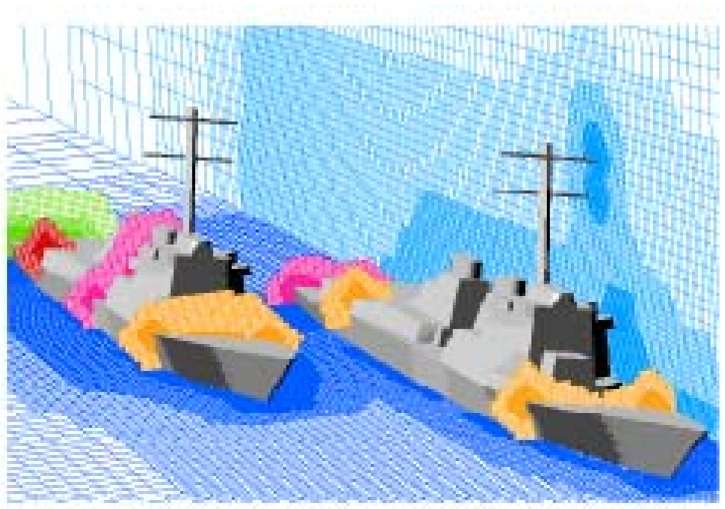


Figure 1.1(b) A CFD grid for a naval ship [1]

Foresight

When confronted with problem of choosing between various possible alternatives for a modeling problem, CFD can prove to be handy. It allows for application of various boundary conditions to the problem and gives the predictions in a short time. This facilitates testing of various alternatives until an optimal solution is reached. This saves valuable time which would be lost in physically prototyping and testing each alternative.

Efficiency

The insight and foresight that CFD provides helps in better and faster design, saving money, shortening the design cycle, and industry compliance. The equipment required can be built, installed, and upgraded with a very little downtime. Also, the physical space requirement is minimal. It allows for faster prototyping and design which leads to faster fabrication of the product.

1.4 A Brief History of CFD

Fluids have captivated the mankind since the dawn of civilization, whether it was waters in the rivers and oceans, winds in the skies, or the flow of blood in our body. Archimedes has been credited for starting the field of static mechanics and hydrostatics. He was the first person to determine ways to measure the volumes and densities of an object. Leonardo Da Vinci contributed to field of fluid mechanics by pictorially depicting various fluid phenomena he observed and he summed it up in his work *"Del moto e misura dell'acqua"* which covers waves and their interference, eddies, free jets, water surfaces, and many other related phenomena.

Isaac Newton, in late 17th century made an attempt to quantify and predict the fluid motion with his set of physical equations. His major contributions to this field includes, his (Newton's) second law of motion, the concept of viscosity, the reciprocity principle, and the relationship between the speed and wavelength of a wave at the surface. Daniel Bernoulli contributed by giving us the famous Bernoulli's equation which describes the behavior of fluid along a streamline. Leonard Euler derives the Euler equations which explain the principle of conservation of mass and momentum for an inviscid fluid. He also put forth the velocity potential theory.

In the 19th century Claude Louis Marie Henry Navier and George Gabriel Stokes added the viscosity to the Euler equations resulting in the Navier-Stokes (N-S) equations. This set of differential equations which describe the conservation of mass, momentum, species, pressure and turbulence are the basic fundamentals of the modern day computational fluid dynamics. One of the shortcomings of the full N-S equations was that they were so closely coupled that it required huge computational power for solution of practical flow problems, which only became available in the form of digital computers in the 1960s. In the 19th century noticeable contributions were made to the field of fluid dynamics by Osborne Reynolds, Jean Marie Louis Poiseuille, John William Rayleigh, John Le Rond D'alembert, Pierre Simon de Laplace, Joseph Louie Lagrange, and Simeon-Dennis Poisson.

The 20th century saw major development of new theories and improvement of existing theories in boundary layer and turbulence. Ludwig Prandtl proposed the famous boundary layer theory, the Prandtl number, compressible flows and the concept of mixing length. Theodore Von Karman studied the Von Karman vortex sheets. Geoffrey Ingram Taylor put forth a statistical theory of turbulence and the concept of Taylor microscales. Andrey Nikolaevich Kolmogorov proposed the universal energy spectrum and the idea of Kolmogorov scales for turbulence. George Batchelor contributed to the theory of homogeneous turbulence.

The first real attempt to use CFD for prediction of a physical fluid phenomenon was made by Lewis Fry Richardson, in late 19th century, when he tried to numerically predict the weather by dividing the physical space into grid cells and use the finite difference approximation of the Bjerknes's equations. It resulted in a failure because a prediction of an eight hour period took six weeks. In 1933, Thom did the numerical simulation of flow past the cylinder. Kawaguti, in 1953, achieved a similar result from his simulations of flow past a cylinder.

In 1960s, theoretical division of NASA at Los Alamos, introduced many numerical methods such as Particle-in-cell (PIC), Marker-and-cell(MAC), Vorticity -Stream function, Arbitrary-Lagrangian Eulerian (ALE) methods and the $k-\varepsilon$ turbulence model.

In 1970s Brian Spalding and his associates developed the SIMPLE algorithm, a modified form of $k-\varepsilon$ turbulence equations, the concept of upwind differencing, 'Eddy break-up' and '*presumed pdf*' combustion models. The 1980s saw development of commercial codes for use in the industry to solve the problems of heat, flow and mass transfer. Since then CFD has evolved from being an indispensable part of aerodynamics and hydrodynamics industry to being a vital part of any discipline which deals with craft of designing and manufacturing moving things.

1.5 Introduction to the Problem

Computational fluid dynamics is always in search of more computer power. The future of computational fluid dynamics requires major improvements in large scale numerical simulation of the Navier-Stokes equations. These however, require enormous and very expensive computing resources which are limited. This problem has been further compounded by the CFD expanding from traditional aerospace and meteorological applications to more diverse applications.

Large scale, time dependent problems which incur high computational cost are generally solved on sophisticated supercomputer facilities. The access to these supercomputers is limited primarily to government organizations and agencies doing research funded by the government. This leaves the non-government organizations and academic research community with almost no access to these facilities. An option for interested large organizations to overcome this hindrance is to buy a supercomputer facility and share it among them. However, these supercomputers are very expensive, difficult to upgrade, and have high maintenance costs. Thus, this option is not often economically viable. Apart from that smaller research facilities and organizations simply cannot afford this cost. So the high upfront cost of the CFD tool has put it beyond the reach of scientists and researchers whose studies and applications will benefit CFD the most.

Numerous advances are being made in the field of commodity computer hardware, which make computing better and cheaper. So commodity clusters have come up as an inexpensive alternate option to the high cost supercomputers. They are easy to build, maintain and upgrade. Also, as they cost considerably less than the supercomputers they can serve the interests of all research communities. The CFD group at the University of Kentucky has come up with commodity clusters which allow the high cost CFD to be performed on low cost clusters. Presently, there are five Kentucky Fluid Clusters (KFC's) each built with a different architecture, serving the computational needs of the CFD group.

UNCLE is an in-house, unstructured CFD code developed by Dr. George Huang at the University of Kentucky. It is capable of handling incompressible, two/three-dimensional calculations and varied geometry types. It uses a cell-centered pressure based approach based on the SIMPLE algorithm. It is second order accurate in both time and space. It is also adept at handling parallel computations. A detailed description of the code is presented in the later chapters.

UNCLE is a well developed code fully capable of handling computations of complex geometries which appear in practical computation. As the state of art computational facility is available at the CFD group in the form of KFC machines, it makes appropriate sense to choose them as a platform for the optimization work. One of the appealing features of computational fluid dynamics is the time and speed associated with it. The performance optimization would make it even faster thus saving costly computer time.

This exercise will reveal the performance shortcomings of this code and try to correct them and the insight gained during the process will help us design better and efficient codes in the future. A sample plot showing the walltime comparison results between the optimized and unoptimized versions of the code is shown in Fig. 1.2. The plot indicates that there is about 50% improvement in the performance of the optimized code. This thesis presents the various techniques that were tried and used to achieve these improvements in the code performance. The successful optimization techniques could be used to improve the performance of other codes. Optimization of CFD codes in general improves the overall performance so that limited and expensive computational time can be efficiently utilized.

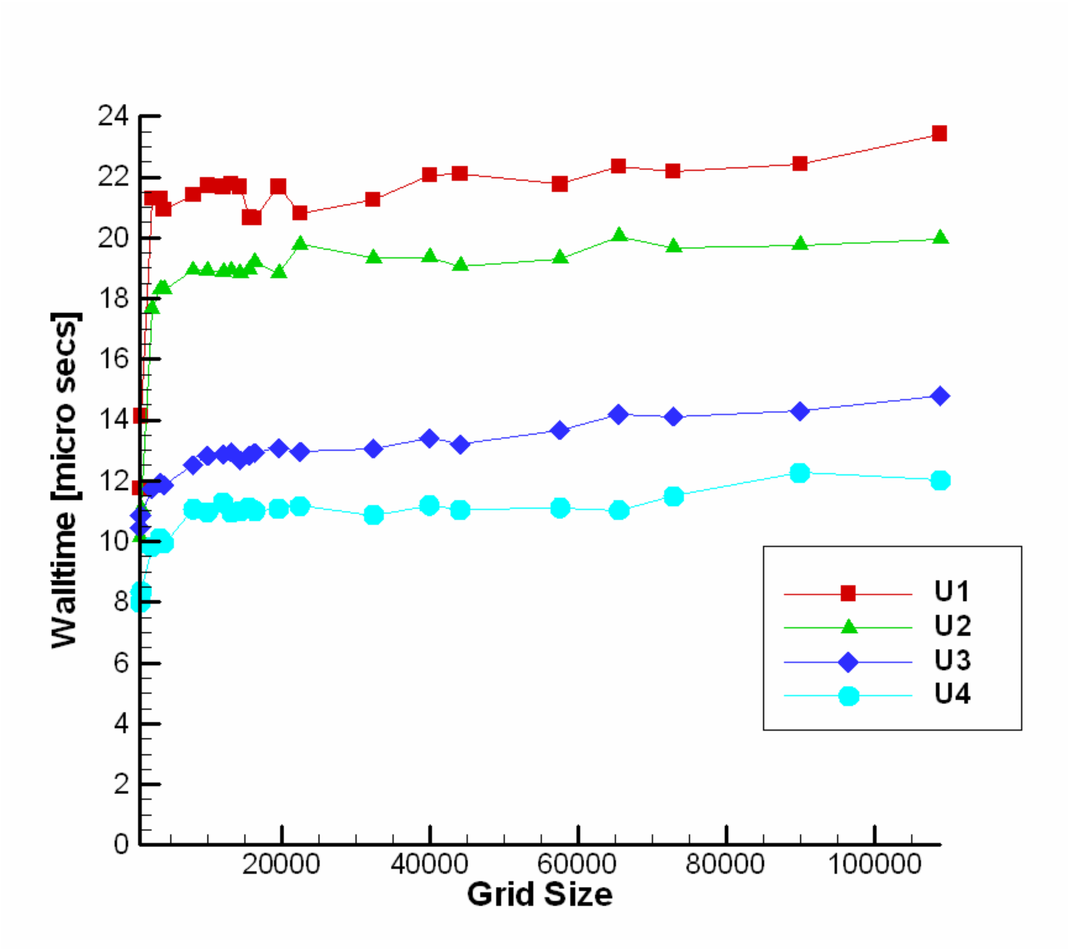


Figure 1.2 Walltime comparisons between the unoptimized and optimized versions of the code

1.6 Approach

With the foray of computers in everyday life, today's engineers are taking keen interest in the inner details of working of a computer. The memory architecture of the modern day computer is virtually evolving everyday. A wide array of memory types and fast microprocessors are available today giving the user the numerous options to design the machine conducive to his needs. With a computational facility custom made for CFD calculations available in form of KFC machines, this study will focus on enhancing the performance of UNCLE code by optimizing the cache behavior on these systems. Various techniques for data access optimization will be employed and their effect on the code performance will be studied in detail. This will also help us identify the key parameters of the memory architecture that influence the code performance. These parameters once identified, can be tuned, if necessary, to gain additional improvements. This cache based optimization can then potentially be used on various other codes and platforms.

1.7 Previous Work

Kadambi et al. [2] addressed the problem of moving the data between the memory and processors in their optimization of a program which solved three dimensional Euler equations. The first technique employed was to reduce the number of memory references by reallocation of the data space. The second scheme to improve the cache behavior and reducing the memory reference cost was restructuring the program structure. They achieved an improvement of 45% in their best case. The primary (L1) cache miss rate was reduced by more than a factor of four but the secondary cache miss rate did not show any significant changes.

Hauser et al. [3] used array of structures to group multiple fundamental variables together in a cache friendly manner instead of using separate arrays for each variable. This improves the spatial locality of the data. They reprogrammed their code DNSTool to make sure that the nested loops accesses multidimensional array in such a manner that the array with the smallest stride is in the innermost loop. In the later publication [4], the work was extended to other codes LESTool and Overflow. The effect of these changes on scalability was studied by conducting test on multiple numbers of nodes and different computer architectures. The codes showed near linear scalability of different degrees on different cluster architectures.

1.8 Background

To comprehend the methodology adopted in this research it is essential to understand the behavior of the cache. Caching is a technology based on the memory subsystem of a computer. It appears on a computer in various forms such as memory caches, hardware caches, software caches, and page caches. The current work aims at optimizing the usage of memory cache to enhance the code performance. To understand the technology behind the existence of cache in the computer we will have a look at the working of a computer and its memory architecture.

Numerous advances in the field of chip design have given us varied and faster microprocessors otherwise known as just processors or Central Processing Unit (CPU). They are basically the heart of any computer. The speed of a microprocessor is measured in clock speed. Clock cycle is the time taken by the processor to execute the simplest instruction. Clock speed is the number of these clock cycles executed per second by the processor. It is generally measured in megahertz (Mhz) or gigahertz (GHz). Each processor requires a certain fixed number of clock cycles to execute an instruction. So the faster is the clock cycles, the faster is the processor.

1.8.1 Memory Architecture

The data in the computer is stored in hard disk which a rigid magnetic disk which can used to both write and erase digital data. It is the primary storage unit of a computer. The next level in the computer architecture hierarchy is Random Access Memory (RAM). RAM is a volatile memory used to temporarily store the information for processing. Successful execution of a program includes sending data and instructions explaining what is to be done with the data, to the processor. The clock cycles of the RAM are around 60 nanoseconds and that of a hard disk is around 12 milliseconds. These are pretty fast times by themselves but when compared to the clock cycle of a 2GHz processor (which has a clock cycle of about 0.5 nanoseconds) these devices are fairly slow in speed of their operations.

When the processor is working on a data and requests further data, which is not available to the processor at that moment, to carry on the execution of a program, it goes back to the RAM and checks for its availability. A latency time is usually associated with all such operations. Latency time or latency is the time usually wasted by a device while it is waiting for another device to execute its function. If the data is found in the RAM, the processor goes ahead with the execution of the program and if it not found in the RAM it goes all the way to the hard disk to fetch the data. As the RAM and the hard disk have slower clock cycles it takes a long time to locate and fetch the data to the processor (hence increasing the latency time), while the processor is idle at the time when the data is being searched or fetched. This affects the efficiency of the processor and also the overall computational time.

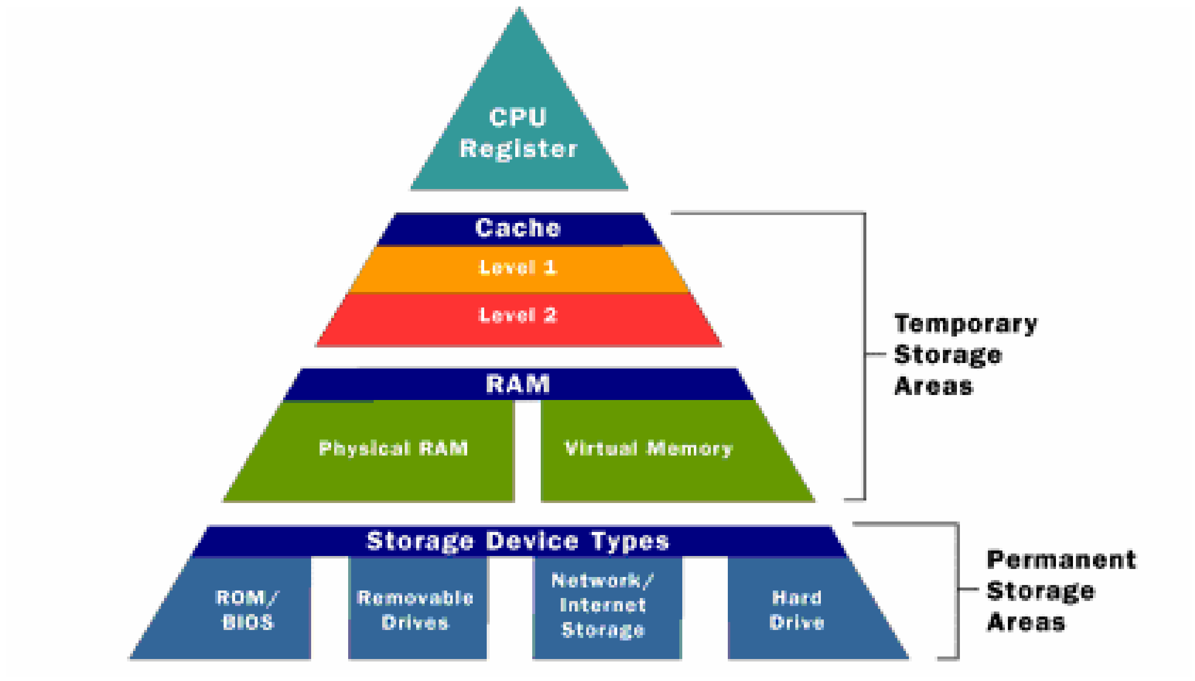


Figure 1.3 Memory Hierarchy in a typical computer [5]

This problem can be handled by the introduction of an intermediate smaller and faster memory type between the processor and the larger memory type (i.e. RAM). This is called the cache memory. Typical memory hierarchy architecture is shown in the Fig. 1.3. The design of the hierarchical memory systems in the modern computer is based on two fundamental concepts, time and space locality. Time locality is the guess by the memory system that it is likely that the memory objects referenced recently will be referenced again in the near future. The cache memory allows the storage of the recently referenced data in a faster memory type so that it can be accessed quickly. Spatial locality is the guess that the memory objects referenced recently will be adjacent or nearby to memory objects that will be referenced in the near future. Hierarchical memory systems use spatial locality by dividing the memory space into smaller chunks called ‘cache blocks’ or ‘cache lines’. The data is transferred from larger memory to cache in cache-lines. Cache-line is the smallest unit of memory (data) transferred between the main memory and the cache. It contains contents of contiguous block of the large memory.

So when a request for data arises from the processor, the cache is accessed first before moving to a larger memory type. If the required data is found in the cache it is called a cache hit. A cache hit saves us valuable computational time as it has a small latency. It is important to note that whenever a small piece of any data is requested by the processor it gets all the data present in the cache line. If the required data is not found in the cache it is called a cache miss. When a cache miss occurs the computer searches for the data in the larger memory types.

There is a theoretical concept called locality of reference in computer science that says that a program spends about 90% of the computational time in about 10% of the code. As the program uses 10% of the code most of the time, if a substantial or all of that 10% (depending on the size of the program) can be made available in the faster cache the program can run faster. This is one the basic reasons cache helps in faster execution of a program. It is possible to have multiple numbers of caches between the processor and the main memory. In most of the personal computers (PCs) we have a two layered cache, L1 cache being the cache closest to the processor in terms of access and L2 cache being the one between the L1 cache and the larger memory RAM.

A table showing the typical characteristics of these memory types is shown below.

Type	**Typical speed**	**Typical size**
L1 cache	10 nanoseconds	4 – 128 KB
L2 cache (e SRAM)	20 – 30 nanoseconds	128 – 512 KB
Main memory (e RAM)	60 nanoseconds	128 MB – 2 GB
Hard disk	10 milliseconds	20 – 100 GB

Table 1.1 Typical characteristics of memory types

During a computation if the processor requests data, it is first searched for in the L1 cache. If it is found in L1 cache, it is called a L1 cache hit; otherwise it is called a L1 cache miss. Then the data is searched in the immediate higher memory, in this case the L2 cache. If the data is found in the L2 cache it is a L2 cache hit or if the data is not in the L2 cache the next higher memory RAM is searched and it is called a L2 cache miss. So ideally, for the perfect execution of the program without any time delay we need to have data stored or data access pattern streamlined in such a fashion so that there are no cache misses. This is possible only for very small grids which fit into L1 or L2 cache, but this is highly impractical for CFD calculations. So the idea is to reduce the L1 and L2 cache misses and brings them as close as reasonable to zero.

The codes can be made to run at accelerated speed if we can use larger memories which are as fast as the cache but the faster memory tends to be very costly and not economically viable. So the memory architecture design is a trade off between its cost effectiveness and its efficiency. This also dictates that the subsequent memory types between the large memory and processor has to be smaller in size when compared to its predecessor. Memory architecture is said to have an inclusive design if the information in the smaller memory type is duplicated in the larger memory type. If the information is not duplicated in the larger memory, it is said to have an exclusive design.

1.8.2 Space Filling Curves (SFC)

Space filling curves have been fascinating to mathematicians and students of mathematics for over a century now. George Cantor, in 1878, proved that any two finite dimensional smooth manifolds have the same cardinality, irrespective of their dimensions. This implied that the interval [0, 1] can be mapped bijectively onto the square $[0, 1]^2$. In 1879, E. Netto proved that such a bijective mapping is necessarily discontinuous. So attention was focused to find a surjective mapping, if any existed, that could satisfy these properties. In 1890, Guiseppe Peano found the first space filling curve. Therefore the space-filling curves are also often referred to as Peano curves.

Definition: An *N*-dimensional space-filling curve is a continuous, surjective (onto) function from the unit interval [0, 1] to the *N*-dimensional unit hypercube $[0, 1]^N$. It is a simple way of mapping multi-dimensional space into one-dimensional space. A 2-dimensional space-filling curve is a continuous curve that passes through every point of the unit square $[0, 1]^2$. The concept of space-filling curves is not limited to 2-dimension and could be extended to *N*-dimensions. It may be thought of as the limit of a sequence of curves which are traced through the space.

There are numerous variants of pace filling curve available today. Some of the well-known spaces filling curves are Hilbert's SFC, Peano's SFC, Sierpinski's SFC, Lebesgue's SFC, Schoenberg's SFC, and Osgood's Jordan curves.

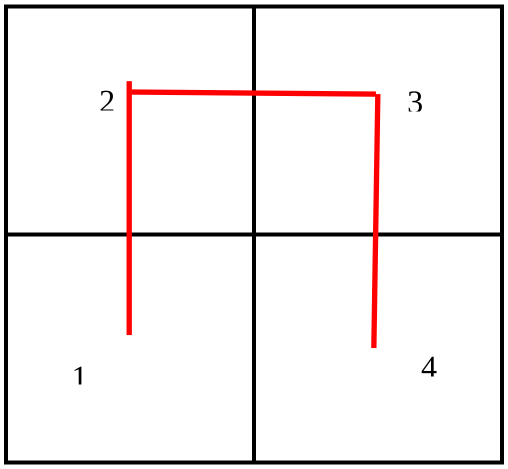


Figure 1.4 First order space filling curve

David Hilbert was the person to highlight the geometrical manifestation of the space filling curves. He also identified the general geometric generating procedures for construction of entire class of these curves. Hilbert's space filling curves consists of a set of perpendicular lines traversing through the unit square. The orthogonal lines render it easy to construct and formulate a repetitive procedure for coding. Therefore Hilbert's space filling curve was used to test the idea of grid blocking with the use of space filling curves.

1.8.2.1 Construction of Hilbert Space Filling Curve

As mentioned earlier the basic function of a space filling curve is to traverse through the space and pass through every single point in the space. The Hilbert's space filling curve necessitates the partition of the domain into '2^{2n}' sub domains. Here 'n' represents the order of the curve. The higher the is value of the 'n', the higher is the number of sub domains and higher is order of the space filling curves.

The construction of the first order curve requires the partition of the unit square in to 4 ($2^{2 \times 1}$) sub domains. This is shown in Fig. 1.4. The space filling curve (shown in red) passes through all the sub domains, starting from the bottom left hand corner and passing through the other sub domains in the order mentioned by the numbers (inside them) and reaching the end at the bottom right hand corner. The first order Hilbert curve is approximately similar to an inverted cup or an inverted 'U' in appearance. The second order curve is constructed by dividing each original sub domain of the previous order into 4 sub domains (as shown in the Fig. 1.5) which gives us a total of 16 (4x4) sub domains for the second order. This number of sub domains correlates to the previously mentioned partition requirement which also provides the same number of sub domain 16 ($2^{2 \times 2}$). The original (first order) sub domain is differentiated by different colors. The first order curve is rotated in clockwise direction by ninety degrees and positioned in the lower left hand corner of the domain (shown in purple). Similarly another one is rotated by ninety degrees in the anti-clockwise direction and placed in the bottom right hand corner (shown in violet). Two of them are placed unchanged in the top two quadrants and all the four of them are joined by lines (dotted). This gives us the second generation of the curve.

The next member of the family is obtained by treating each quarter as the whole and repeating the operation. The actual Hilbert Curve is not a member of this family; it is the limit that the sequence approaches. A fourth order space filling curve is shown in the Fig. 1.6. Hypothetically, at a certain limit the sub domain shrinks to the size of a single at which the space filling curve would be passing through every single point of the unit square, thus traversing the complete domain.

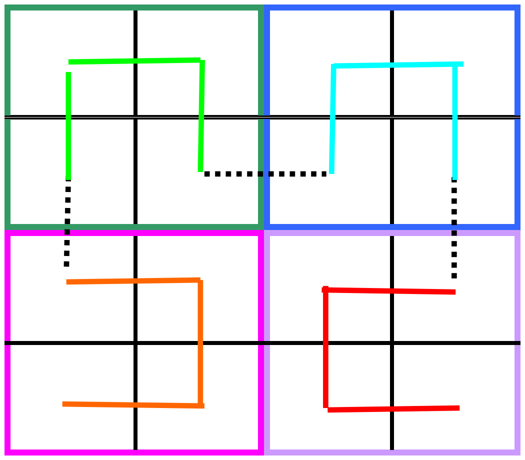


Figure 1.5 Second order space filling curve

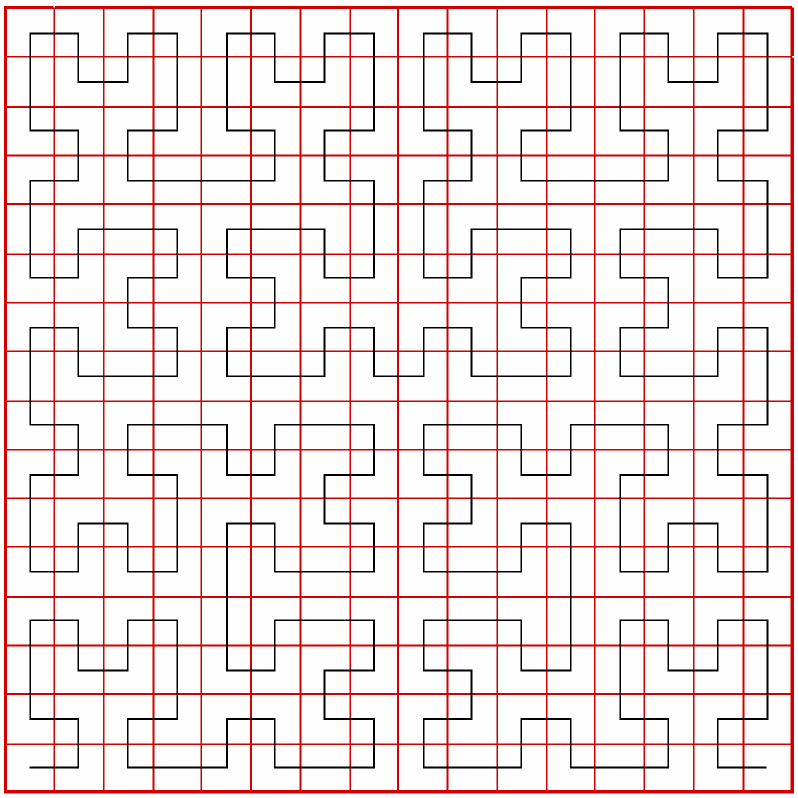


Figure 1.6 Fourth order space filling curve

1.8.2.2 Previous Applications of SFC

Space filling curves has been used in varied number of applications. Ogawa [7] used it for parallelization of an adaptive Cartesian mesh solver based on 2^N- tree data structure. The space filling curves were utilized to isotropically divide the domains in smaller subdomains and number them. The load balancing for computation on multiple processors is done by using the numbering done by space filling curve. Aftosmis et al. [1] utilized space filling curves in various applications for Cartesian methods for CFD. They employed the space filling curves for reordering, multigrid coarsening, inter-mesh interpolation, and to generate single pass algorithms for mesh partitioning.

Dennis [8] employed SFC for partitioning of domain in his study of atmospheric models. A gnomonic projection of a cube onto the surface of a sphere was done. The space filling curves were used to partition this domain for better parallel performance. Behren et al. [9] developed an efficient load distribution algorithm for unstructured parallel grid generation. Phuvan et al. [10] used the space filling curves for texture analysis. They showed that a one-dimesional image scan which follows a Peano curve to a desired resolution preserved two-dimensional proximity is efficient for wavelet transform and artificial neural network pattern recognition. Dafner et al. [11] developed a context based space filling curve approach for scanning images.

1.8.3 Techniques for Data Access Optimizations

Data access optimizations are modifications made to a code which changes the way in which the loop nests are executed. These changes maintain all the data dependencies and numerical accuracy of the original computations. There are numerous transformations that could be made to a particular code. Also various combinations of these individual modifications can be applied to a code with varying degrees of success. The combination of changes yielding the best performance is highly code specific. The optimized performance is dependent on the various intricate details of the code structure which changes from code to code; therefore no particular technique is suggested as universally effective. However, the modifications that were considered or rendered, in their original or modified form, to the code in this study are given below

1.8.3.1 Loop Interchange

Loop interchange involves reversing the order of two adjacent loops in a loop nest. This transformation can be extended to more than just two loops, which is referred as loop permutation. This can be generally applied to loops whose order of execution bears no effect on the final result. Loop interchange can reduce the stride of an array based computation in some cases. The stride is defined as the number of array elements accessed in the memory within consecutive loop iterations. The shorter strides lead to better data locality and encourage register reuse. In the Algorithm 1.1, simply the `'i'` and `'j'` loops interchanged.

```
         Original loop nest                      Interchanged loop nest
INT a[n][n],sum                         INT a[n][n],sum
DO i = 1,n                          DO j = 1,n
    DO j = 1,n                          DO i = 1,n
        sum = sum + a[i][j]                 sum = sum + a[i][j]
    ENDO                                ENDDO
ENDDO                               ENDDO
```

Algorithm 1.1 Loop Interchange

The scope of improvement by the use of this technique is elaborated by Fig. 1.7. We assume that all the elements in the array `'a'` are stored in a column major. The array elements are next to each other in the column if their second indices are consecutive numbers. It could very well be stored in row major order and the transformation will still hold. If the array is large enough it will not fit totally into the cache.

When an element is requested for computation in row major order, say `'a[i]  [j]'`in this case, a lot of data along with the requested data is grabbed in a stride. We assume that the stride size is six array elements. Thus the data grabbed might not contain the information pertaining to the required neighboring array elements. So another request for the data is made and it is obtained. So in every stride only information pertaining to one element is being received and rest of the data is not being used at all. This leads to large number of strides to do the task which translates into large computational time. An immediate instinctive approach to rectify this problem of ineffective use data will be to change the direction of the stride. So if we can orient the stride to the allocation of the data in the memory so that consecutive elements of the array `'a'` can be accessed in a single stride, as shown in transformed algorithm, we can reduce the extra data calls associated with each computation inside the loop. Substantial improvements in cache performance can be achieved by reducing the excess number of data calls and enhanced data reuse. This method will show substantial improvements only if the whole array is larger than the cache size.

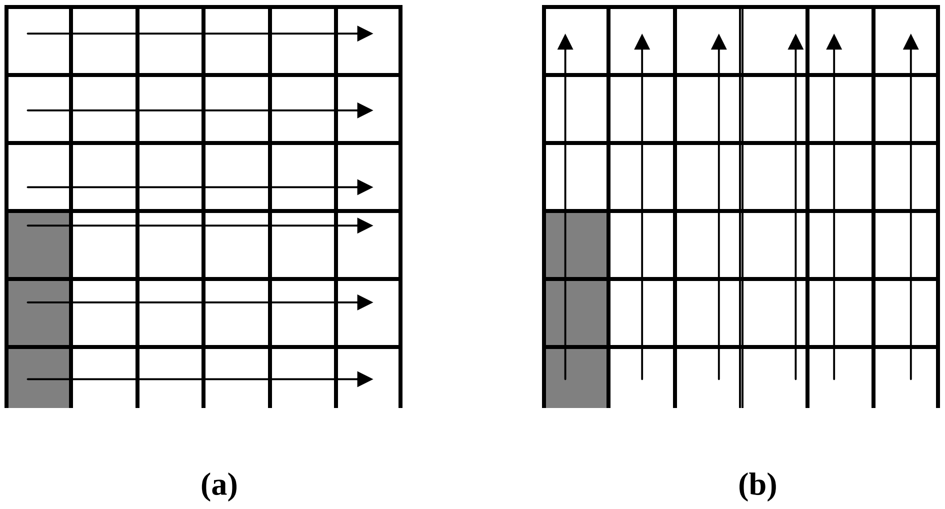


Figure 1.7 (a) Row wise stride access (b) Column wise stride access

1.8.3.2 Loop Fusion

Loop fusion involves taking two neighboring loops which have the same amount of iteration space to negotiate during the computation and combining their bodies under a single loop instead of two different loops as shown in Algorithm 1.2. This is also referred to as loop jamming. This is only plausible only when the bodies of two loops are independent of each other. There cannot be any flow, anti, or output dependencies in fused loop for which the instructions from the first loop are dependent upon instruction from the second loop. Loop fusion results reducing the loop overhead as there is only a single body inside the loop, instead of two separate bodies in two different loops. In this case the overhead will be reduced by a factor of two. This reduction of loop overhead largely depends on the complexity of the loop bodies.

Original loop nest

```
   DO i = 1,n
    b[i] = a[i]*10
      ENDDO
   DO i = 1,n
 C[i] = b[i]/5
ENDDO
```

Loop Fusion nest

```
DO i = 1,n
 b[i] = a[i]*10
      c[i] = b[i]/5
ENDDO
```

Algorithm 1.2 Loop Fusion

Loop fusion also helps improve the data locality. This could be seen in the algorithm shown. We can clearly see in the original algorithm that the loop sweeps over three different arrays in two loops. If the size of the arrays are large enough (larger than the cache), then the values of `‘b[i]’`calculated in the first loop can be swept out of the cache before they can used in the computation n the second loop. When the second loop begins the whole array `‘b[i]’` has to be loaded again from the higher memory. This leads to ineffective use of data. However, this problem can be overcome by the loop fusion. The values of `‘b[i]’`are calculated and reused for further calculation in the same body before it can be kicked out of the cache memory. This leads to fewer data calls which in turn leads to lower cache misses.

1.8.3.3 Loop Blocking

Loop blocking is dividing the whole computational domain into smaller blocks for the purpose of computation. This is generally useful when the body of the nested loop contains many large arrays and calculation is repeated over the loop more than once at a single call. In the Algorithm 1.3, shown below, we see that the body of the loop contains three arrays `‘a’, ‘b’, and ‘c’`, associated with each unit/cell. The whole grid is divided in to small blocks. There are `‘n’`such blocks in the whole grid. Also we notice that the calculation over the body of the loop is repeated `‘m’` number of times. We further assume that these arrays are larger than the cache size.

Original loop nest

```
     DO j = 1, m
     DO i = 1, n

  a[i][j] = a[i-1][j] + 1
  b[i][j] = b[i][j+1] * 10
  c[i][j] = b[i+1][j] / 5


 ENDDO
 ENDDO
```

Loop Blocked nest

```
DO i = 1, n/b
DO j = 1, m
DO ii = (i-1)*b +1, i*b
 a[i][j] = a[i-1][j] + 1
 b[i][j] = b[i][j+1] * 10
 c[i][j] = b[i+1][j] / 5

ENDDO
ENDDO
ENDDO
```

Algorithm 1.3 Loop Blocking

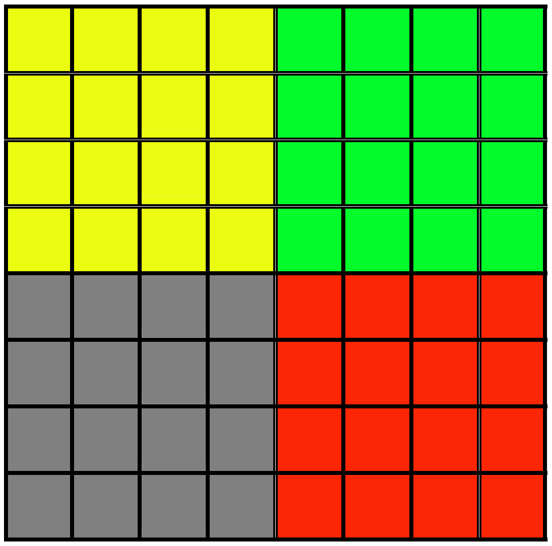


Figure 1.8 Computational domain (Blocked)

CFD calculations often require simultaneous calculations over contiguous block of cells. We see that in Algorithm 1.3 that the calculations of each array element require the data pertaining to the neighboring element. In the original loop, the whole computational domain shown in the Fig. 1.8 is solved at the same time. If time the loop sweeps over the entire domain in a row or column wise stride and calculations of each unit cell are done using the information stored in the neighboring cells, the information stored in this original cell be will be required when the calculations on associated neighboring cells are being performed. This information, however, could be moved out of the memory by the time it is recalled. This information had to be searched and reloaded for calculation which leads to high latency and cache misses. This situation is compounded since the calculation has to be repeated `‘m’` number of times and the information has to be reloaded `‘m’` number of times leading to high data misses.

Instead of solving the whole computational domain at the same time, the domain can be divided into smaller blocks (indicated by the color) as shown in Fig. 1.8 and computation can be carried over these smaller blocks. As shown in the blocked loop nest, the computational domain is divided into smaller blocks which fit into the cache and the calculation is repeated `‘m’` number of times over this small block. This ensures that the data in the cache is effectively reused before it is moved out of the cache resulting in enhanced cache performance.

1.8.3.4 Array Merging

This technique is a useful when different elements of an array or data structure are often accessed together but are not close together in the memory. This technique is also known as *group-and-transpose* and improves the spatial locality between the array or data structure elements. The simplest way to implement this is use of multidimensional array. If the array or data structure is already multidimensional this process could be facilitated by addition of an extra dimension as shown in Algorithm 1.4.

Original array

```
     INT a[n],b[n],c[n]
  DO i = 1,n
   c[i] = a[i]+b[i]
  ENDDO
```

Merged array

```
     INT abc[n,3]
  DO i = 1,n
   abc[i,3] = abc[i,1]+abc[i,2]
  ENDDO
```

Algorithm 1.4 Merged Array

Another simple technique for multidimensional arrays is to transpose the array dimensions by interchanging them as shown in the Algorithm 1.5. This is similar to the loop interchange.

Original array *Transposed array*

```
    INT abc [n, 3]                INT abc [3, n]
```

Algorithm 1.5 Transposed Array

1.8.3.5 Loop Unrolling

Loop unrolling is simply replicating the body of the loop more than one time to reduce the cost of the loop overhead. The loop index in advanced by a prescribed number instead of advancing by a single step each time and the loop body is replicated the same number of times inside the loop. Instead of checking the loop termination condition after every iteration, it is done once in the prescribed number of iterations. The loop overhead is reduced because the loop condition is checked less often. More computation is performed in every iteration when the loop is unrolled. An example of unrolled loop is shown in Algorithm 1.6.

Original loop *Unrolled loop*

```
  DO i = 1,n                    DO i = 1,n
       b[i] = a[i]*10                b[i] = a[i]*10
            c[i] = b[i]/5                 c[i] = b[i]/5
  ENDDO                              b[i+1] = a[i+1]*10
                                     c[i+1] = b[i+1]/5
                                     i = i + 2
                                ENDDO
```

Algorithm 1.6 Unrolled Loop

1.8.3.6 Prefetching

Modern processors are much faster than the memory in which the program and the data is stored. This means that the instructions cannot be read and the data cannot be fetched fast enough to keep the processor busy at all the time. Prefetching is the processor action of getting instruction and data from the memory before the processor needs it. If the prefetching is done efficiently, a continuous data flow to the processor can be achieved. Modern processors have prefetching instructions programmed in them. The prefecthing can be customized depending upon the requirements of a code and this would require writing machine language instructions at appropriate locations in the code, which is not a trivial task. This is generally a suggested method for very large codes.

CHAPTER 2

COMPUTATIONAL TOOLS

This chapter presents a comprehensive description of computational tools and platforms that were used in this study. To begin with a discussion of the CFD code UNCLE and the numerics involved is presented. The computational architecture employed in this study is also described in detail. A brief description of the profiling tool '*gprof*' and the cache simulator tool '*Valgrind*' follows. These tools will be used to assess the cache performance of the codes.

2.1 Description of UNCLE

UNCLE is an in-house code at the University of Kentucky, written by Dr. George Huang, designed to meet the challenges of physical problems with complex geometries, complicated boundary conditions on parallel computers while maintaining high computational efficiency. It was validated by Dr. Chen Hua at University of Kentucky using various challenging test cases. The detailed description of the code and validation process is presented in the Hua et al [13]. It is a two/three- dimensional, finite volume, unsteady, incompressible, Navier-Stokes solver with cell-centered pressure based SIMPLE algorithm. The code is second-order accurate in both time and space. It is very flexible in geometry as it can handle grids of various types such as triangular, quadrilateral, tetrahedral and hexahedral. METIS [14], a program based on multilevel graph partitioning schemes, is used for grid partitioning. The parallel construction of code is done using message passing interface (MPI) protocols and it has worked successfully on systems ranging from commodity PC clusters to supercomputers.

The cell-centered pressure-based method is based on the SIMPLE algorithm with second order accuracy in both time and space. The numerical flux on the interfaces is computed using a second order upwind scheme for all the advection terms and a second order central difference scheme for all the diffusion terms. A collocated grid system with the Rhie and Chow momentum interpolation method [15] is employed to avoid the checkerboard solution of the pressure based scheme. The code is written in FORTRAN 90 and has been successfully ported to several computational platforms. The details of the code formulation are given below and further details may be found in Hua et al [13].

2.2 Governing equations

The governing equations for unsteady incompressible viscous flow under the assumption of no body force and heat transfer are:

Conservation of Mass

$$\frac{\partial}{\partial t}\int_V \rho\, dV = -\oint_S \rho u_i n_i\, dS \tag{1}$$

Conservation of Momentum

$$\frac{\partial}{\partial t}\int_V \rho u_j dV = -\oint_S \rho u_i n_i\, u_j dS - \oint_S p n_j dS + \oint_S \tau_{ij}\, n_i dS \tag{2}$$

Conservation of Energy

$$\frac{\partial}{\partial t}\int_V \rho E dV = -\oint_S \rho u_i n_i\, E dS - \oint_S p u_j n_j dS + \oint_S u_j\, \tau_{ij} n_i dS \tag{3}$$

where ρ is density, p is pressure, u_i are the components of the velocity vector, n_i is unit normal vector of the interface, τ_{ij} is tensor of shear force, and specific internal energy is $E = e + \frac{1}{2}(u^2 + v^2 + w^2)$

2.2.1 Convective and diffusive fluxes

Figure 2.1(a) shows the schematic diagram for the integration areas for convective fluxes. The flow properties on the interface can be obtained by using Taylor series expansion, as shown in Eq. (4)

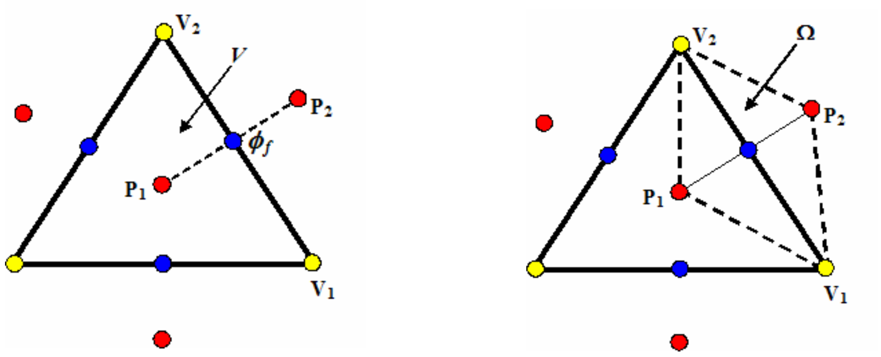


Figure 2.1 Schematic diagrams for integration areas. (a) Convective fluxes, and (b) Diffusive fluxes [13].

$$\phi^{LHS} = \phi_{P_1} + \left.\frac{\partial \phi}{\partial z}\right|_{P_1} (x_f - x_{P_1}) + \left.\frac{\partial \phi}{\partial y}\right|_{P_1} (y_f - y_{P_1}) + \left.\frac{\partial \phi}{\partial z}\right|_{P_1} (z_f - z_{P_1}) + HOT$$

$$\phi^{RHS} = \phi_{P_2} + \left.\frac{\partial \phi}{\partial z}\right|_{P_2} (x_f - x_{P_2}) + \left.\frac{\partial \phi}{\partial y}\right|_{P_2} (y_f - y_{P_2}) + \left.\frac{\partial \phi}{\partial z}\right|_{P_2} (z_f - z_{P_2}) + HOT$$

(4)

where ϕ stands for the velocity components and any scalar quantities, the superscript *RHS* and *LHS* denote the approximation from the right-hand side and left-hand side of the interface respectively, and *HOT* represents higher order terms. By substituting Eq. (4) into Eq. (5), interfacial flow properties ϕ_f can be obtained.

$$\phi_f = \frac{1}{2}(\phi^{RHS} + \phi^{LHS}) - \frac{1}{2} sign(1, \dot{m})(\phi^{RHS} - \phi^{LHS}) \qquad (5)$$

The gradients at the nodal points (cell centers) are evaluated by the Gauss's divergence theorem

$$\int_V \frac{\partial \phi}{\partial x_i} dV = \int_A \phi n_i dA$$

$$\frac{\partial \phi}{\partial x_i} \approx \frac{\sum_{k=1}^{N_{face}} \phi n_{i,k} \ A_k}{V} \qquad (6)$$

where N_{face} is the total number of interfaces of the cell and V denotes the volume of the control cell.

The schematic diagram for diffusive fluxes is shown in Fig 2.1(b). The gradients at the interface can be evaluated by using the chain rule as Eq. (7)

$$\begin{aligned}
\frac{\partial \phi}{\partial x} &= \frac{\partial \phi}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial \phi}{\partial \eta}\frac{\partial \eta}{\partial x} + \frac{\partial \phi}{\partial \zeta}\frac{\partial \zeta}{\partial x} \\
\frac{\partial \phi}{\partial y} &= \frac{\partial \phi}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial \phi}{\partial \eta}\frac{\partial \eta}{\partial y} + \frac{\partial \phi}{\partial \zeta}\frac{\partial \zeta}{\partial y} \qquad (7) \\
\frac{\partial \phi}{\partial z} &= \frac{\partial \phi}{\partial \xi}\frac{\partial \xi}{\partial z} + \frac{\partial \phi}{\partial \eta}\frac{\partial \eta}{\partial z} + \frac{\partial \phi}{\partial \zeta}\frac{\partial \zeta}{\partial z}
\end{aligned}$$

where the local coordinate system (ξ, η, ζ) is defined by the orientation of the face

For the triangular mesh in Fig. 2.1(b), ξ is the vector from nodal point P$_1$ to P$_2$, η is the vector from vertex V$_1$ to V$_2$, and Ω is the integration area for diffusive fluxes. The diffusive fluxes can be approximated by Eq. (8)

$$\begin{aligned}
\left(\frac{\partial \phi}{\partial x}\right)_f &\approx \frac{1}{2\Omega}[(\phi_{P_2} - \phi_{P_1})(y_{P_2} - y_{P_1}) - (\phi_{V_2} - \phi_{V_1})(y_{V_2} - y_{V_1})] \\
\left(\frac{\partial \phi}{\partial y}\right)_f &\approx \frac{-1}{2\Omega}[(\phi_{P_2} - \phi_{P_1})(x_{P_2} - x_{P_1}) - (\phi_{V_2} - \phi_{V_1})(x_{V_2} - x_{V_1})]
\end{aligned} \qquad (8)$$

where ϕ_{Pi} denotes the properties at nodal points and ϕ_{Vi} denotes the properties at vertices. The values at vertices are obtained by averaging surrounding nodal values using inverse distances from all surrounding nodal points as the weighting function.

2.2.2 Center pressure based SIMPLE algorithm

With the use of an initial pressure field, P^n, we can obtain u^n, v^n, and w^n by solving the momentum equations in a sequential manner. The solution method is based on the first order delta form of the left-hand side (*LHS*) and the momentum equations can be written in the form as Eq. (9)

$$\begin{aligned} a_c\Delta u &= \sum_{nb} a_{nb}\Delta u + RHS_u \\ a_c\Delta v &= \sum_{nb} a_{nb}\Delta v + RHS_v \\ a_c\Delta w &= \sum_{nb} a_{nb}\Delta w + RHS_w \end{aligned} \tag{9}$$

where the coefficients a_{nb} and a_c are

$$a_{nb} = \max(-\dot{m}_f, 0) + \mu_f(\xi_x n_1 + \xi_y n_2 + \xi_z n_3)A, \tag{10}$$

$$a_c = \sum_{nb} a_{nb},$$

and the *RHS* term can be written as

$$\begin{aligned} RHS_u &= -\sum_{i=1}^{N_{face}}[\dot{m}_i u_i^n - \left(\tau_{11}^n - p^n\right)_i n_{i,1}A_i - \left(\tau_{21}^n\right)_i n_{i,2}A_i - \left(\tau_{31}^n\right)_i n_{i,3}A_i] \\ &= -\sum_{i=1}^{N_{face}}[\dot{m}_i u_i^n - \left(\tau_{11}^n\right)_i n_{i,1}A_i - \left(\tau_{21}^n\right)_i n_{i,2}A_i - \left(\tau_{31}^n\right)_i n_{i,3}A_i] - \frac{\partial p^n}{\partial x}V_c \\ RHS_v &= -\sum_{i=1}^{N_{face}}[\dot{m}_i v_i^n - \left(\tau_{12}^n\right)_i n_{i,1}A_i - \left(\tau_{22}^n - p^n\right)_i n_{i,2}A_i - \left(\tau_{32}^n\right)_i n_{i,3}A_i] \\ &= -\sum_{i=1}^{N_{face}}[\dot{m}_i v_i^n - \left(\tau_{12}^n\right)_i n_{i,1}A_i - \left(\tau_{22}^n\right)_i n_{i,2}A_i - \left(\tau_{32}^n\right)_i n_{i,3}A_i] - \frac{\partial p^n}{\partial y}V_c \\ RHS_w &= -\sum_{i=1}^{N_{face}}[\dot{m}_i w_i^n - \left(\tau_{13}^n\right)_i n_{i,1}A_i - \left(\tau_{23}^n\right)_i n_{i,2}A_i - \left(\tau_{33}^n - p^n\right)_i n_{i,3}A_i] \\ &= -\sum_{i=1}^{N_{face}}[\dot{m}_i w_i^n - \left(\tau_{13}^n\right)_i n_{i,1}A_i - \left(\tau_{23}^n\right)_i n_{i,2}A_i - \left(\tau_{33}^n\right)_i n_{i,3}A_i] - \frac{\partial p^n}{\partial w}V_c \end{aligned} \tag{11}$$

where subscript *c* denotes the cell value to be solved, subscript *nb* denotes the neighbor cells, and *A* denotes the interfacial area. Equation (9) can be solved by using Gauss-Seidel point substitution. Then, we can obtain u^*, v^*, and w^* with Eq. (12)

$$
\begin{aligned}
u^* &= u^n + \Delta u \\
v^* &= v^n + \Delta v \\
w^* &= w^n + \Delta w
\end{aligned}
\tag{12}
$$

At this stage u^*, v^*, and w^* satisfy the momentum equations, but they do not necessarily satisfy the continuity equation. In order to satisfy mass conservation, the velocity has to be interpolated to the interface. In order to avoid checkerboard solutions, the interfacial velocity has to be driven solely by the pressure difference evaluated directly at the interfaces. To accomplish this without sacrificing any accuracy, the interpolated interfacial velocity is divided into two components: one is the velocity component without the pressure contribution and the other is solely the pressure contribution. The former is first evaluated at the cell center as

$$
\begin{aligned}
\tilde{u}^* &= u^* + \frac{\partial p^n}{\partial x}\frac{V_c}{a_c} \\
\tilde{v}^* &= v^* + \frac{\partial p^n}{\partial y}\frac{V_c}{a_c} \\
\tilde{w}^* &= w^* + \frac{\partial p^n}{\partial z}\frac{V_c}{a_c}
\end{aligned}
\tag{13}
$$

and then interpolated onto the cell faces. The latter is obtained directly from the pressure difference of the two adjacent nodal points, P_1 and P_2, such that the interfacial velocity can be expressed as:

$$
\begin{aligned}
u_f^* &= \tilde{u}_f^* - \left(\frac{\partial p^n}{\partial x}\right)_f \frac{V_f}{a_f} \\
v_f^* &= \tilde{v}_f^* - \left(\frac{\partial p^n}{\partial y}\right)_f \frac{V_f}{a_f} \\
w_f^* &= \tilde{w}_f^* - \left(\frac{\partial p^n}{\partial z}\right)_f \frac{V_f}{a_f}
\end{aligned}
\tag{14}
$$

where V_f/a_f is obtained by interpolation from the cell center to the interface as:

$$\left(\frac{V}{a}\right)_f = \frac{\left[\left(\frac{V_{P1}}{a_{P1}}\right)V_{P2} + \left(\frac{V_{P2}}{a_{P2}}\right)V_{P1}\right]}{V_{P1}+V_{P2}} \tag{15}$$

We further assume that there are corrections to u_f^*, v_f^*, and w_f^*, such that the continuity equation can be satisfied

$$\sum_{i=1}^{N_{face}} \rho_{f,i}[(u^*_{f,i} + \Delta u'_{f,i})n_{1,i} + (v^*_{f,i} + \Delta v'_{f,i})n_{2,i} + (w^*_{f,i} + \Delta w'_{f,i})n_{3,i}]A_{f,i} = 0 \tag{16}$$

We can rewrite Eq. (16) as

$$\sum_{i=1}^{N_{face}} \rho_{f,i}[\Delta u'_{f,i}n_{1,i} + \Delta v'_{f,i}n_{2,i} + w'_{f,i}n_{3,i}]A_{f,i} = -\sum_{i=1}^{N_{face}} \rho_{f,i}[u^*_{f,i}n_{1,i} + v^*_{f,i}n_{2,i} + w^*_{f,i}n_{3,i}]A_{f,i} \tag{17}$$

where the right-hand side in Eq. (17) represents the mass imbalance in the control volume cell. We assume that there is a corresponding pressure correction field, p', which drives the velocity corrections according to

$$\begin{aligned}
\Delta u'_f &\approx -\left(\frac{\partial p'}{\partial x}\right)_f \frac{V_f}{a_f} \approx -\frac{V_f}{a_f}\xi_x(p'_{P_2} - p'_{P_1}) \\
\Delta v'_f &\approx -\left(\frac{\partial p'}{\partial y}\right)_f \frac{V_f}{a_f} \approx -\frac{V_f}{a_f}\xi_y(p'_{P_2} - p'_{P_1}) \\
\Delta w'_f &\approx -\left(\frac{\partial p'}{\partial z}\right)_f \frac{V_f}{a_f} \approx -\frac{V_f}{a_f}\xi_z(p'_{P_2} - p'_{P_1})
\end{aligned} \tag{18}$$

By substituting the velocity correction equations into the equation for the mass imbalance, we can obtain the equations of the pressure correction

$$a_c p'_c = \sum_{nb} a_{nb} p'_{nb} + b \tag{19}$$

where a_{nb} and a_c in the continuity equation are

$$\begin{aligned}
a_{nb} &= \rho[\frac{V_f}{a_f}\xi_x n_1 + \frac{V_f}{a_f}\xi_y n_2 + \frac{V_f}{a_f}\xi_z n_3]A \\
a_c &= \sum_{nb} a_{nb}
\end{aligned} \tag{20}$$

and

$$b = -\sum_{nb} \rho [u_f^* n_1 + v_f^* n_2 + w_f^* n_3] A \tag{21}$$

Once the pressure correction is obtained, one can update the pressure field by

$$p^{n+1} = p^n + \alpha_p p' \tag{22}$$

where α_p is the under-relaxation factor for pressure and is generally a value in the range 0.5-0.8. Then the velocity correction on the interfaces as well as nodal points is updated according to Eq. (14).

2.3 Time discretization

A second-order fully implicit scheme is employed for the temporal discretization. Here, we take a one-dimensional equation example

$$\frac{3\phi^{n+1} - 4\phi^n + \phi^{n-1}}{2\Delta t} + \frac{\partial f(\phi^{n+1})}{\partial x} = 0 \tag{23}$$

where ϕ is primitive variable, *f* is interfacial flux, and the superscript *n* indicates the index in time. A deferred iterative algorithm is employed to obtain ϕ^{n+1} by substituting Eq. (24) into Eq. (23),

$$(\phi^{n+1})^{m+1} = (\phi^{n+1})^m + (\Delta\phi)^m \tag{24}$$

where the subscript *m* stands for the subiteration level. The final equation is

$$\begin{aligned} &\frac{3(\Delta\phi)^m}{2\Delta t} + \frac{\partial f(\Delta\phi)^m}{\partial x} = \\ &\frac{(\phi^n - \phi^{n-1})}{2\Delta t} - \frac{3((\phi^{n+1})^m - \phi^n)}{2\Delta t} - \frac{\partial f((\phi^{n+1})^m)}{\partial x} \end{aligned} \tag{25}$$

The right-hand side of Eq. (25) is explicit and can be implemented in a straightforward manner to discretize the spatial derivative term. The left-hand side terms are evaluated based on the first order upwind differencing scheme. The deferred iterative algorithm is strongly stable, and the solution ϕ^{n+1} is obtained by using inner iterations to reach the convergent solution of the right-hand side of Eq. (25), corresponding to $\Delta\phi$ approaching zero. At least one subiteration is performed at every time step so that this method is fully implicit.

2.4 Partitioning approach

Excellent load balancing between the subgrids on each node is achieved through using METIS for domain decomposition. METIS can partition an unstructured grid into any integer number of zones without losing load balance. It is compatible with many platforms and convenient for running CFD codes on a variety of supercomputer to cluster architectures. The present partitioning approach has been tested by a number of two/three-dimensional geometries. All results show good load balances. The details of partitioning are described in detail in Hua et al [13].

2.5 Description of the Input Files

UNCLE uses structures and arrays to store the flow and geometry data of the computational domain. Most of the data is stored in four key array of structures of the form $\Phi(x,y)$: $\phi_1..\phi_n$, where 'ϕ' is the structure which is associated with any given point (x,y,z), and 'ϕ_i' is the variable that contains data concerned with the point. This ensures that all the similar data associated with the variable 'ϕ_i' for all the cells is stored contiguously in the memory. These structures contain data related to node, face, vertex, and cell. The face structure is a subset of another structure which divides the faces into internal faces and boundary faces. UNCLE reads this required data from the input files which are given below.

cell.dat – This file contains the total number of cells and information whether the simulation conducted is two or three dimensional. Given below is an example of a typical file.

```
65536  2D
       1
       1       65536    fluid fluid*
```

The number '65536' denotes the total number of a grid points in the computational mesh. The number '1' denotes the grid is a one contiguous domain and 'fluid' denotes the model option.

face.dat – This file tells us which vertices and nodes are associated with each internal face. The faces are stored in order they occur, when one moves across the cells in a row or column wise *'i-j'* manner. This is followed by similar information about the boundary faces with specific boundary conditions. A sample of face.dat is shown below.

```
130560
   2      11      17      2      1
   2      9       17      4      1
   .
   .
    2      13      23      16      15

   4

   256 wall * wall_R
   2      10      14      17      16
   .
  .
   2      16      1       20      11

   256 wall * wall_Bottom
   2      6       11      21      1
   .
   .
  2      13      10      24      16

   256 wall * wall_L
   2      2       7       25      6
   .
  .
   2      9       6       28      1

   256 inl * inl
   2      1       3       29      11
   .
  .
   2      5       2       32      6
```

The first number '130560' indicates the total number of internal faces. It is followed by information about each of these internal faces. The faces occur in the increasing order of their numbering.

The first number in each line of the face information indicates the number of vertices associated with that face. It is followed by the vertex numbers and the two node numbers on either side of the face. The data is represented in this manner for all the internal faces. After the data pertaining to the internal faces is specified the boundary faces data is presented. The number ‘4’, following the internal faces, indicate the number of boundary condition for the domain. This is followed by the data pertaining to these boundary conditions. The first number ‘256’ indicates the number of faces on each domain boundary. It is followed by the name of the boundary condition describing if it is a wall, an inlet or some other boundary face. The ‘*’ indicates to the information pertaining the treatment of the boundary condition in relevant subroutines in the UNCLE code. This is again followed by the detailed data for each face similar to the internal faces in the manner described above. This pattern continues until all the boundary conditions are described.

vertex.dat – This file contains the information about the vertex numbers and their geometry coordinates. A sample is *vertex.dat* shown below. The first number in the file is the total number of vertices in the domain. It is followed by the vertex information. The first number in each line indicates the vertex number followed by its geometrical coordinates. The present file is for a two-dimensional case hence the *x* and *y*- coordinates are shown.

```
66049
 1      1.000000        1.000000
 2      0.000000        1.000000
 3      0.750000        1.000000
 4      0.500000        1.000000
```

dist.dat – This file is required for the turbulent calculations and stores data pertaining to distance of a node from the wall. A sample *dist.dat* is shown below. The first two lines containing data pertaining to the geometry of the boundary faces and also is the standard syntax which is used for plotting the data. The number of boundary points is 769 and the number of boundary elements is 768. It is followed by the geometrical coordinate data for the vertices. Since the case presented here is two-dimensional the *x* and *y*- coordinates are shown. The vertices occur in increasing order of their numbering. The following data is the node connectivity data which is useful in plotting.

```
 variables = "x", "y"
      zone n= 769  E= 768  F=FEpoint, ET=triangle
1.00000000000000        0.00000000000000D+000
   1.00000000000000        3.90600000000000D-003
.
.
.
   1.00000000000000         5.69400000000000D-003
   1.00000000000000        7.81200000000000D-003
       1           2           2
       2           3           3
       .
      .
       5           6           6
```

2.6 Description of Critical Subroutines

UNCLE contains various subroutines which perform a variety of operations required by the users. When executed the data is read from the input files and a few initial subroutines calculate the necessary information to set up the geometry, initial, and boundary conditions. The program then moves to the iterative core which contains iterative loops over numerous subroutines which compute the CFD calculations. As mentioned earlier, almost all numerically intensive codes spend 90% of the time in about 10% of the code while executing. This rule of thumb for large computer codes stands in the case of UNCLE. A profiling of the unmodified code revealed that a large percentage of the time was spent in seven critical subroutines inside the iterative core of the code. These core subroutines compute the various flow variables and solve them using the Gauss-Siedel solver. Given below is the discussion of these critical and other major subroutines.

1) i_sovler_gs_velocity

As mentioned earlier, UNCLE employs a delta form approach to solve the equations. An initial value for a physical property 'ϕ', if not provided, is assumed and a small change delta ($\Delta\phi$) is added to the initial value at every iteration until satisfactory solution is achieved. The calculation of each '$\Delta\phi$' involves the '$\Delta\phi$' of all the neighboring cells.

This subroutine solves for the delta change in velocity at the cell center using a Gauss-Seidel matrix solver. It contains a DO loop over cell centers (or cells). At the end of each complete sweep over all the cells the velocity at all the nodes is updated. To ensure a stable convergence and accuracy this iterative loop over cells is repeated more than once each time this subroutine is called, with the use of another iterative loop. This number of iterations will be from hereon referred to as inner iterations for velocity. The number of inner iterations for velocity depends upon the complexity of the case.

2) i_solver_gs_p

This subroutine solves for pressure-related terms at the cell center using a Gauss-Seidel matrix solver. It consists of a DO loop over cell centers. It also employs inner iterations for pressure for purpose of stability and accuracy similar to those in *i_sovler_gs_velocity*.

3) i_solver_gs_ke

This subroutine is similar to the above two subroutines applied to the turbulent calculations. It is also dominated by DO loops over cells. The variables of turbulence equations are updated at the end of each iteration. The inner iteration employed for stability will be referred as inner iteration for turbulence.

4) Continuity

This subroutine applies the Rhie and Chow momentum interpolation scheme to get information related to pressure terms. It is dominated largely by DO loops over the faces. All the coefficients of the pressure related terms are calculated in this subroutine. This subroutine calls *i_solver_gs_p*, *set_bc_vel* and *set_bc_p*.

5) cal_velocity

This subroutine determines fluxes on the faces and calculates the coefficient terms related to the velocity. Akin to *continuity* it largely contains DO loops over faces. This subroutine calls *i_solver_gs_velocity* and *set_bc_vel.*

6) cal_ke

This subroutine is similar to the *cal_velocity* and calculates the turbulent fluxes on faces. All the turbulence related coefficients are calculated here. This subroutine in turn calls *i_solver_gs_ke.*

7) gradients_2d

This subroutine calculates the velocity gradients required by other subroutines for the computations. It is also dominated by DO loops over faces.

8) set_bc_vel/_set_bc_p/set_bc_ke

These subroutines just set the boundary conditions for velocity, pressure, and the turbulence at the end of each time step.

To summarize the first seven critical subroutines, the first three are largely do-loops over the cells and involve repetitive inner iterations over these loops, making them the costliest subroutines in terms of computational time. The next four are dominated by loops over the faces, with smaller additional cell loops. These subroutines typically account for about 80-95% of the entire execution time depending upon the grid size.

2.7 Gprof

Profiling allows the user to investigate the working of his code. It tells what amount of time was spent in each subroutine and which other subroutines do a subroutine call while it is being executed. This allows the user to identify the slower subroutines and consider changes to make it execute faster. It tells which subroutines are being called for execution more or less often than expected and spot the bugs in the program that might otherwise go unnoticed. It can be helpful in case of complex and lengthy codes which are very hard to analyze by reading the source. If a certain feature (subroutine) of the code is disabled, no information regarding that feature will show up in the profiling. The 'gprof' tool was used to do the profiling of the codes in this work.

There are three steps to be followed to profile a code with *gprof* during the profiling:

1) Compile the program and link it with profiling enabled

The first step in generating a profile is to add the `'-pg'` option while compiling the code.If the general way of compiling the code using the `'mpi'` enabled FORTRAN 77 compiler is

```
mpif77  uncle.f90   -o uncle.exe
```

The profiling could be enabled by

```
mpif77  uncle.f90   -pg  -o uncle.exe
```

2) Executing the program

Once the program is compiled, it has to run to acquire the data need by the '*gprof* ' for profiling. The program is run as usual using the normal arguments.

```
mpirun –np 1 ./uncle.exe
```

The program will run normally giving the usual output but it will run a bit slowly because of the extra time spent in collecting and writing the required profiling data. The profile data is written to a file called 'gmon.out' just before exiting in the program's current working directory. Any existing file of that name and its content is overwritten.

3) Run '*gprof*' and analyze the profiling output

There are various options available for analyzing the output. They are given below.

Flat profile:

Flat profile gives the time spent in each function and the number of times each function was called. This is the kind of output desired in this study as it gives the time spent in each subroutine concisely and helps the key subroutines to be identified for the optimization. Hence it is described below.

The functions are sorted by the decreasing order of the time it took for execution, followed by decreasing number of calls made to them, which is followed by functions being listed in alphabetical order. The flat profile starts with information about the sampling which tells us how often the samples were taken. It gives a rough estimate of margin of errors associated with each time figure. In this example each sample count is 0.01 seconds which correlates to a 100 Hz sampling rate. It means that a sample was taken in every 0.01 seconds. If the runtime for the program is in the order of sample count, say 0.1 seconds, only 10 samples were taken during the run and the values reported cannot be considered statistically reliable, as the number of samples were very small. So for these values to be very reliable the runtime of the program should be considerably larger than the sample count. Some of the terminology that appears in the flat profile is given below.

`% time`

This gives the percentage of the total time spent in a particular function. The sum of the %time taken by all the subroutines should add up to 100%.

`cumulative seconds`

This is the cumulative total number of seconds that were spent executing a particular function, in addition to the time spent in all the functions above this function in this table.

`self seconds`

This is the number of seconds spent on executing a particular function alone. This is the criterion by which the functions are sorted in the flat profile.

`calls`

This is the number of times a function was called. If a function was never called or was not compiled during the profiling enabled, this field would be left blank.

`self ms/call`

This gives the average number of milliseconds spent in a particular function every time it is called. If it is not compiled or never called, this field like the one above will be left blank.

`total ms/call`

This gives the average number of milliseconds used up by a function and its descendants per call, if the function is profiled. Otherwise, the field is left blank.

`name`

This simply gives the name of the function whose data is mention in the row.

A sample flat profile for UNCLE is shown below.

Table 2.1 Flat profile for UNCLE code

Each sample counts as `0.01` seconds.

```
  %   cumulative   self          self     total
 time   seconds   seconds  calls  ms/call  ms/call       name
 24.34  21.33     21.33   400   53.33    53.33   i_solver_gs_vel
 23.85  42.23     20.90   400   52.25    52.25   i_solver_gs_p_
 21.54  61.11     18.88   400   47.20    103.71  continuity_2d_
 15.89  75.04    13.93   400   34.83    88.29   cal_velocity_2d_
 8.73   82.69     7.65    401   19.08    19.08   gradients_2d_
 2.81   85.15     2.46    401    6.13    6.13    cal_vertex_v_
 1.84   86.76     1.61    400    4.03    4.03    cal_vertex_p_
```

The names of the subroutines are in the far right column. The left most column shows the percentage time taken by the subroutines. The subroutines are in the flat profile are always given in decreasing order of the percentage time consumed. The flat profile contains the data of all the subroutines in the program but for simplicity only the major subroutines are shown here. So a look at the percentage time column helps one identify the major time consuming subroutines. The next two columns contains data about the individual and cumulative time consumed while execution of each individual subroutines. It is followed by the column that shows the number of times these subroutines were called. As the computations were done for 400 iterations most of the subroutines have 400 against their name in this column. A couple of the subroutines which are used to set up the initial flow field are called one extra time, hence the number 401 in this column for the appropriate subroutines. The next column shows the times spent in the subroutine for each call without considering the time spent in the subroutines which this particular subroutine called. The following column gives the total time consumed by a subroutine with due consideration to the other subroutines called by this particular subroutine. As the number of calls for major subroutines is about the same the time taken per each call easily relates to the total time consumed by the subroutines which is presented in the third column.

Call Graph:

The call graph gives detailed analysis for each function. It tells how many times a function was called by other functions and how many other functions it called and estimated time associated with all of the above functions. This might help in eliminating time consuming function calls.

Annotated Source Listing:

The annotated source listing gives out a copy of the source program along with the number of times each line in the source code was executed.

2.8 Valgrind

Numerically intensive CFD codes are often plagued with memory and performance problems. One of many tools that provide help concerning this issue is Valgrind. Valgrind is a set of debugging and profiling tools for codes running on Linux. It helps in tracking the memory leaks and other performance issues. Valgrind is an open source tool and it does not require the user to recompile, relink, or modify the source code. On the other hand, it has the disadvantage of slower runtime.

Some of the benefits associated with Valgrind are:

- Uses dynamic binary translation so that modification, recompilation, or relinking of the source code is not necessary.
- Debugs and profiles large and complex codes.
- Can be used on any kind of code written in any language.
- Works with the entire code, including the libraries.
- Can be used with other tools, such as GDB.
- Serves as a platform for writing and testing new debugging tools.

Valgrind consists of five major tools `Memcheck, Addrcheck, Cachegrind, Massif,` and `Helgrind` which are tightly integrated into the Valgrind core.

Memcheck checks for the use uninitialized memory and all memory reads and writes. All the calls to malloc, free, and delete are instrumented when memcheck is run. It immediately reports the error as it happens, with the line number in the source code if possible. The function stack tracing tells us how the error line was reached. The tracks are addressed at byte level and initialization of values is addressed at bit level. This helps Valgrind detect the uninitialization of even a single unused bit and not report spurious errors on bitfield operations. The drawback of memcheck is that it makes the program run 10 to 30 times slower than normal.

Addrcheck is a toned down version of Memcheck. Unlike Memcheck it does not check for uninitialized data, which leads to Addrcheck detecting fewer errors than Memcheck. On the brighter side it runs approximately twice as fast (5 to 20 times longer than normal) and uses less memory. This allows the programs to run for longer time and cover more test scenarios. In summation, Addrcheck should be run lo locate major memory bugs while Memcheck should be used to do a thorough analysis.

Cachegrind is a cache profiler. It performs detailed simulation of the L1 and L2 caches in the CPU. It helps in accurately pinpointing the sources of cache misses in the source code. It provides the number of cache misses, memory references, and instructions executed for each line of source code. It also provides per-function, per-module and whole-program summaries. The programs run approximately 20 to 100 times slower than normal. With the help of the KCacheGrind visualization tool these profiling results can be seen in a graphical form which is easier to comprehend. This tool was exhaustively used in this study.

Massif is a heap profiler. The detailed heap profiling is done by taking snapshots of the program's heap. It produces a graph showing heap usage over time. It also provides information about the parts of the code that are responsible for the most memory allocations. The graph is complemented by a text or HTML file that includes information about determining where the most memory is being allocated. Massif makes the program run approximately 20 times slower than the normal.

Helgrind is a thread debugger. It finds data traces in multithreaded codes. It searches for the memory locations which are accessed by more than one thread but for which no consistently used lock can be found. These locations indicate of loss of synchronization between threads and could potentially cause timing-dependent problems.

2.9 Kentucky Fluid Clusters

In this section we will focus on two different clusters, Kentucky Fluid Clusters 4 and 5 (KFC4, KFC5). On both systems the Intel 8.0 and 9.0 FORTRAN90 compiler (ifort) with -O3 optimization and LAM MPI were used for the purpose of compiling UNCLE for this study. Since these clusters are controlled in-house, nodes can be readily restricted to a single job at a time; as such, the difference between the CPU time and the walltime has proven negligible, so walltime is used as the basis of the testing. Time values also exclude time associated with the input and output operation (I/O time).

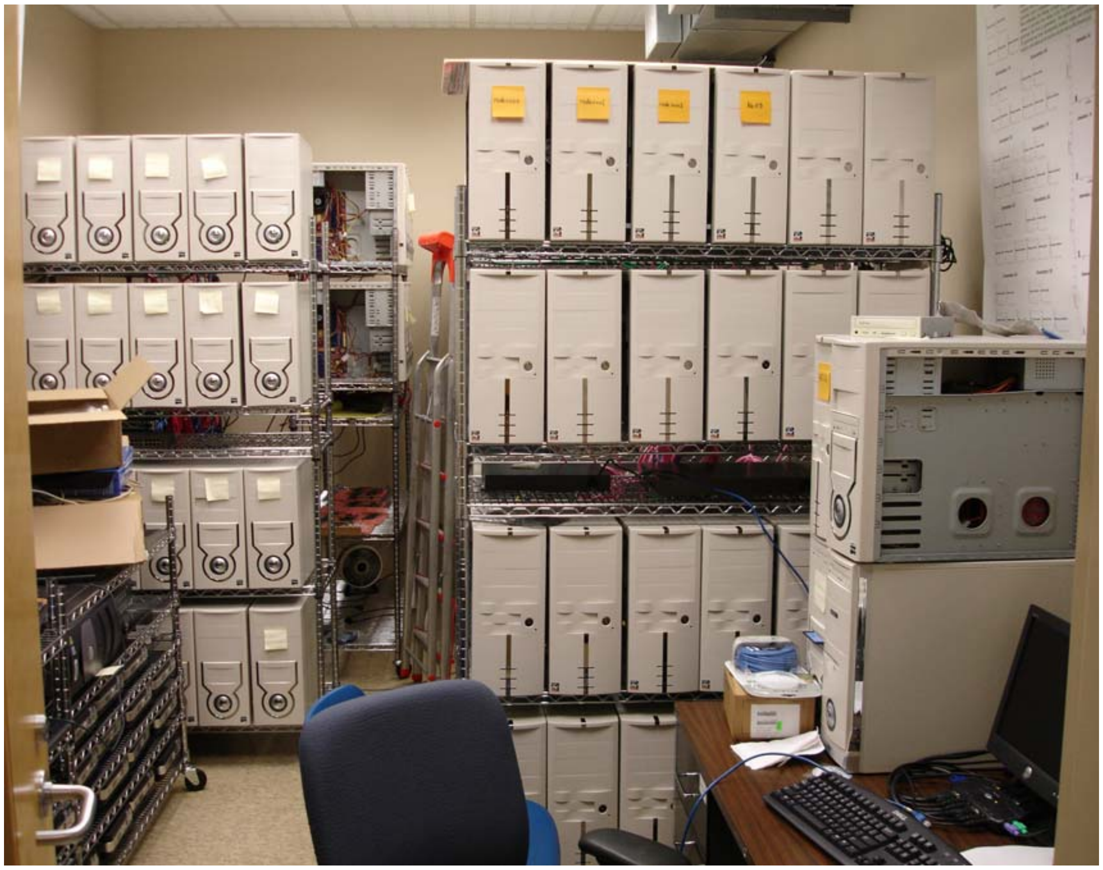


Figure 2.2 Kentucky Fluid Clusters (KFC) 4 and 5

Kentucky Fluid Cluster 4 is a 32 bit architecture constructed of AMD Athlon 2500+ Barton processors. The current configuration is a 47 node system linked by two networks: a single Fast Ethernet (100 Mb/s) switch and a single Gigabit (1Gb/s) switch. Each node has 512 MB of RAM and each processor has a L2 cache of 512 KB. The server is separate from the nodes and plays no direct role in the iterative computation. KFC4 is housed at the University of Kentucky.

Kentucky Fluid Cluster 5 is a 64-bit architecture, constructed of 47 AMD64 2.08 GHz processors linked by a single Gigabit (1Gb/s) switch. Each node has 512 MB of RAM and each processor has a L2 cache of 512 KB. The server is separate from the nodes and plays no direct role in the iterative computation. Like KFC4, KFC5 is housed at the University of Kentucky.

CHAPTER

3

PRELIMINARY RESULTS

This chapter presents a discussion of techniques which were employed with varying degree of success in this research. The original unmodified uncle code went through some modifications to arrive at a cleaned up version of the code. The code was checked for redundant *if-then* and *do* loops and they were either removed of modified accordingly to improve the performance of the code. The subroutines that essentially had the same iteration space were merged together in the fashion as described under the loop fusion section. Also the part of the code that was not used in the computation was commented out to avoid unnecessary data misses. An improvement of approximately 5% was observed in walltime for large grids. So this slightly tuned up version was used as the basic version for further modification.

3.1 Space Filling Curve Test Case

The lid driven cavity was chosen as the test case for steady state computations. The lid driven cavity, shown in the Fig. 3.1, is a fairly standard test case with simple boundary conditions. It was chosen because it has simply geometry yet a complex flow pattern. It is also easy to generate and divide into parts. It is treated as a unit square with a moving wall on the top. The top wall has the non-dimensional *u*-velocity value as unity and *v*-velocity value as zero. The other three walls have a no-slip boundary condition.

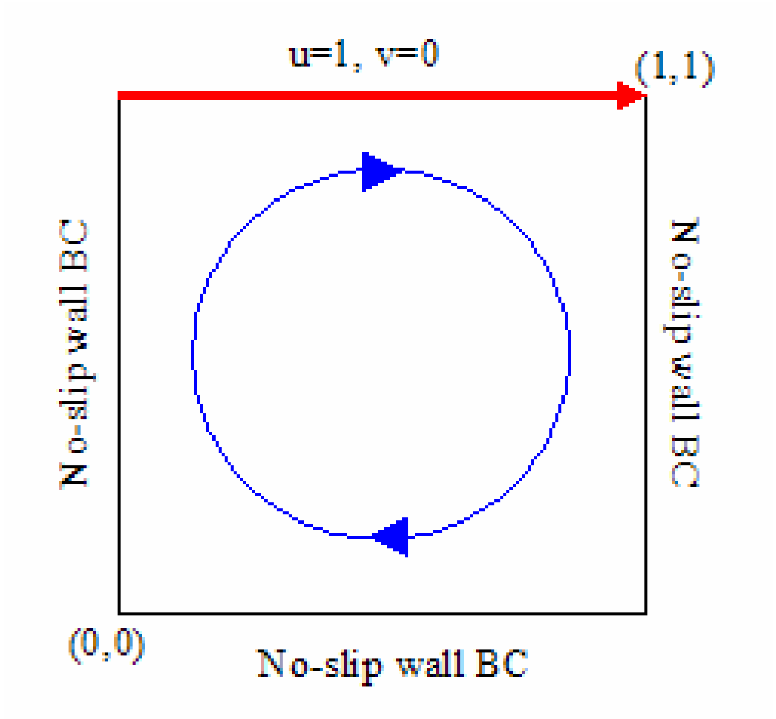


Figure 3.1 Schematic diagram showing the lid driven cavity

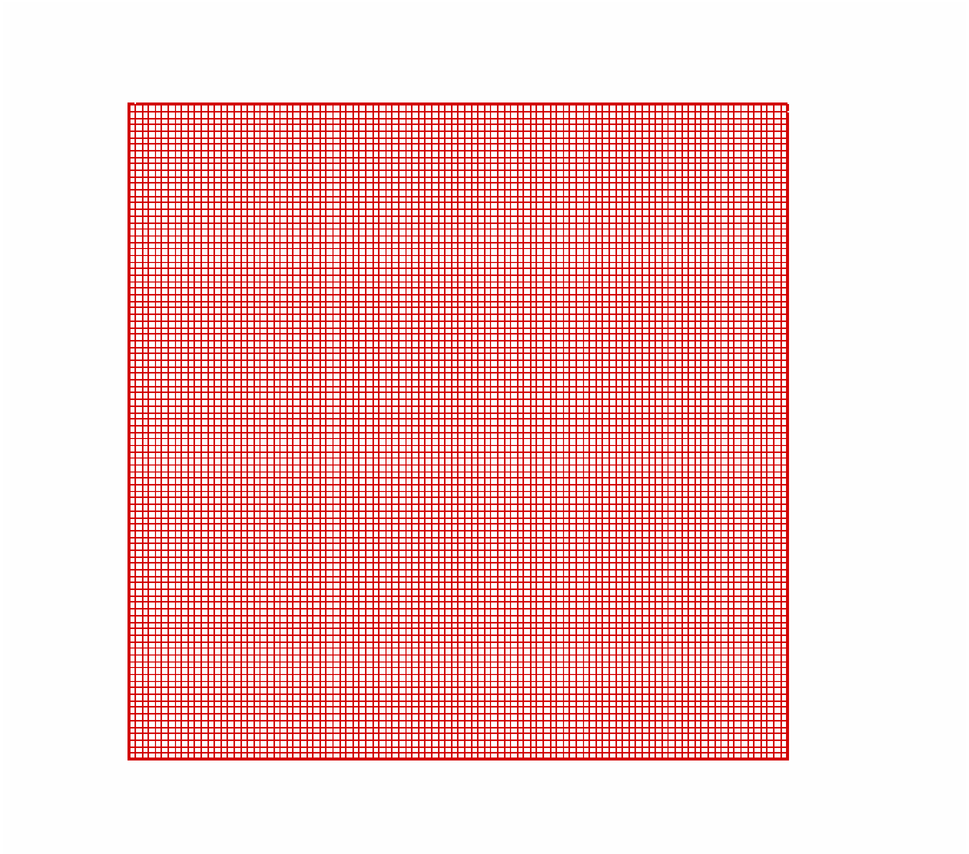


Figure 3.2 A 256x256 computational grid for lid driven cavity

The structured grid was used in this case as it is easy to construct and fit into the square cavity. A computational grid for a 256x256 grid is shown in Fig 3.2.

As mentioned earlier UNCLE requires certain input files namely *cell.dat*, *vertex.dat*, and *face.dat* to read in the input data. In the original grid generator, the numbering of the face, cell, and vertex was done by moving from cell to cell in a row wise (*i-j*) manner. The numbering started from the lower left hand corner. After the cell and all the internal faces and vertices were numbered, the next cell to be numbered was the adjacent one in the next column. After all the cells in a row were numbered, the numbering would continue with the cell at lower left hand corner of the next row, hence the name *i-j* path. The path traversed by this is shown with black arrows in the Fig. 3.3. In this fashion all the cells were numbered and the boundary faces were numbered only after all the internal

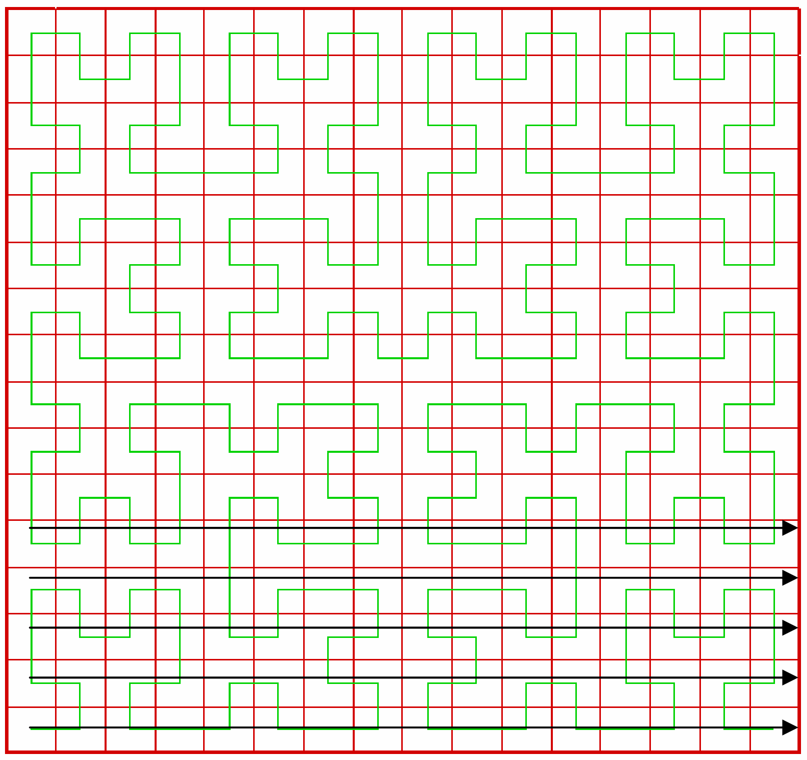


Figure 3.3 Contrasting i-j and SFC paths in a grid

faces were assigned numbers. The cell, face, and vertex numbers were written in these input files in the manner they were assigned.

To have flexibility in grid generation, a grid generation code *‘grid.c’* was written in C/C++, which generated files in the aforementioned manner. The time for code execution (walltime) was calculated using appropriate MPI commands. There were two walltime calculations, first including the time for reading input and writing output, the second being the time spent purely on calculations. As dedicated access to the clusters was ensured, the walltimes were pretty close to the CPU time and later was considered for timing tests.

As the approach adopted in this research is to optimize the cache behavior to speed up the CFD code, it is necessary to obtain a general pattern of cache behavior. It is not feasible to run the codes for various grid sizes till convergence to study the cache behavior because the time involved would be enormous. Fortunately the initial tests done with various grid sizes revealed that the cache behavior tends to stabilize after the initial developments by the time 400 iterations are completed [18]. The cache misses tend to flatten out after 400 iterations, so running a code for same number of iterations would give a very good estimate of the cache performance. So unless mentioned otherwise, all the cache simulation tests were done for 400 iterations.

A few preliminary speed tests were done using the *valgrind* cache profiling tool with grids of different size on KFC4 to get the basic cache performance data. The cache performances of these test cases are shown in the Fig. 3.4. Ideally, we would like to have zero miss rates in both the caches but it is not feasible in most practical cases. The smallest grid used was 16x16 and it showed the ideal zero L2 cache miss rate and a L1 miss rate of about 0.8%. As the grid size was increased, it was observed that the cache behavior started to depart from the optimum. When the grid size was increased to 25x25, the L1 cache miss jumped drastically from 0.8 % to 9.5% and L2 cache miss rate jumped from zero to 3.3%. The L1 and L2 cache misses were about 10% and 9% respectively for the grid size of 50x50. This high miss rate continued in the same range from hereon with the increase in the grid sizes. It is evident that the code had the capability to perform at near zero cache miss rate but the increase in the grid size clearly is causing departure from this behavior.

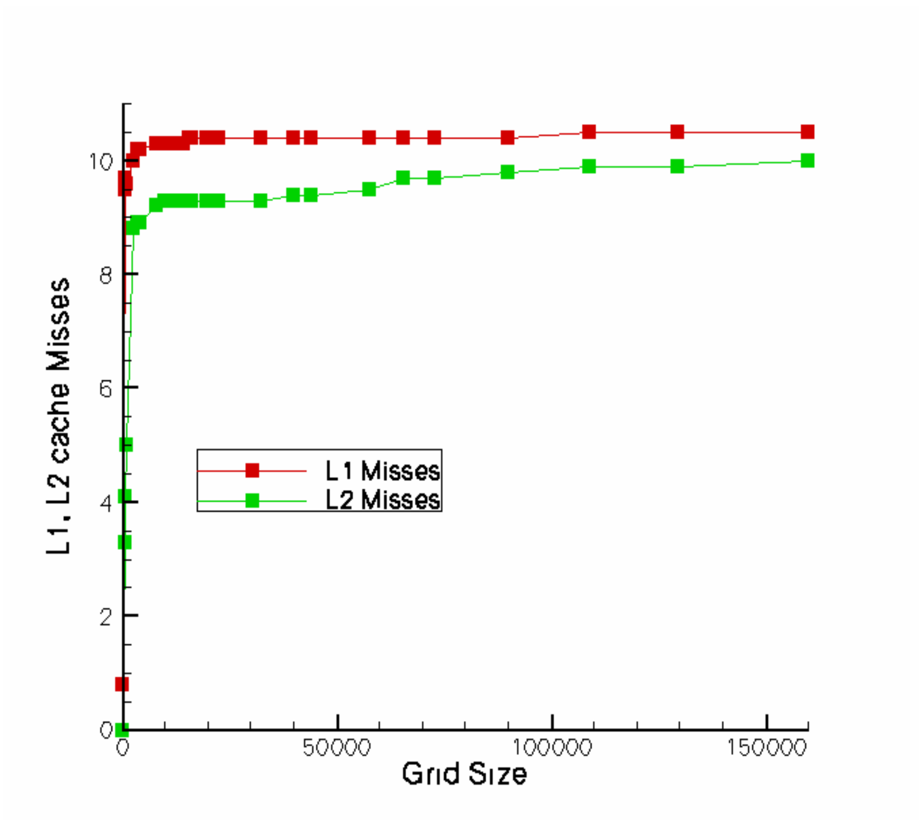


Figure 3.4 Cache performances for grids of various sizes

A profiling of the two-dimensional code version was done using the *gprof* tool and the walltimes taken by critical subroutines of two grid sizes 16x16 and 256x256 for 5000 iterations are shown in Fig. 3.5. As we move from a smaller grid to a larger grid we expect increase in time taken by these subroutines owing to the larger number of grid points. This behavior was extant in this case but the growth in these subroutines from a smaller grid to a larger grid size was found to be very disproportionate. The largest growth in time taken was observed in the Gauss-Siedel solver subroutines followed by the subroutines *gradients*, *cal_vel*, and *continuity*. The increased times taken by these subroutines were very consistent with the cache miss rate. So the attention was focused on a technique which could potentially reduce the time taken by all the subroutines at the same time without necessarily changing the way the computations were done.

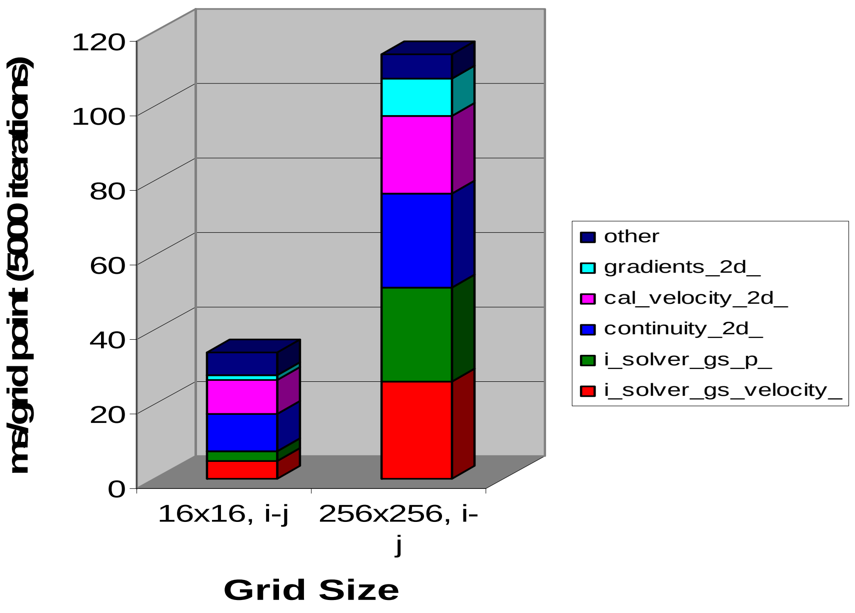


Figure 3.5 Comparison of times taken by critical subroutines

In the earlier cases the numbering was done in the traditional '*i-j*' manner. To calculate the physical parameters of a cell, UNCLE requires the values of the physical parameters of the neighboring cells. So it would be beneficial to have the parameters of the neighboring cells readily available when required. This is not always possible on large grids where the numbering was done in '*i-j*' manner because by the time calculations are being done at cells near the end of a row the data pertaining to cells at the start of the row is flushed out of the cache. This data is required again when the calculations on the cells at the start of the next row are being performed because the former are now a part of neighboring cell set. This process repeats itself whenever every new row is being calculated and ofcourse at every iteration and sub-iteration. This leads to a higher number of data calls and cache misses which makes the code run more slowly. So using the space filling curve for numbering the cells, faces, and vertices could be beneficial as the recursive nature of the space filling curve requires it to traverse neighboring cells before moving to the distant ones. As the space filling curves moves through every point in an N-dimensional space, every cell would be numbered.

Therefore the '*grid.c*' grid generator was accordingly modified to alter the numbering scheme using the Hilbert's space filling curve. As the space filling curves moves among the neighboring cells, covering each cell in the vicinity before moving to a farther one, it ensures better overall spatial data locality. This contrast to the '*i-j*' manner can be seen in Fig. 3.3. The data related to the adjoining cells almost always can be procured from the cache without reaching for the higher memory resulting in fewer data calls and lower cache misses. A larger computational grid and the space filling curve used to number the grid is shown in the Fig. 3.6.

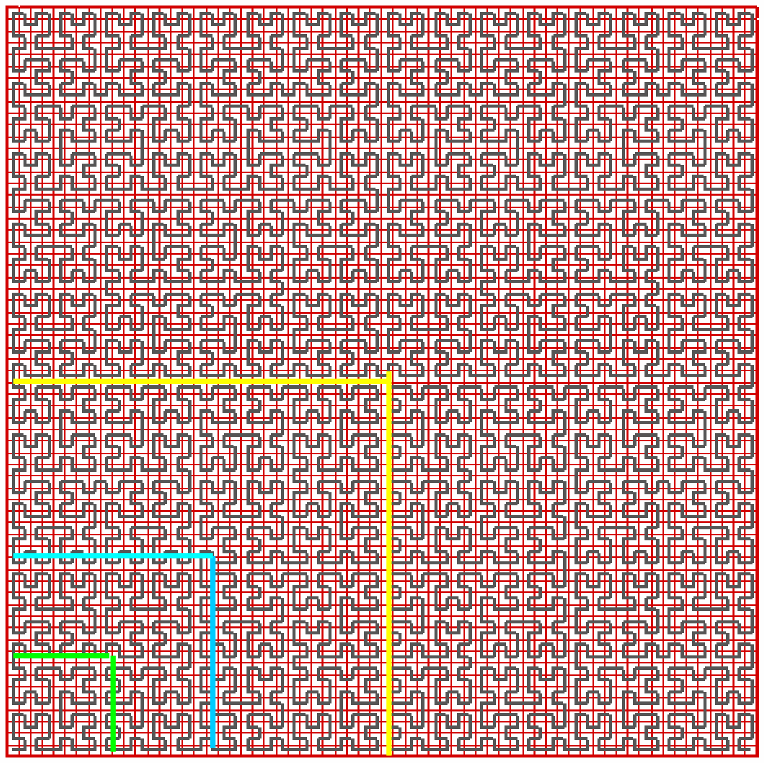


Figure 3.6 Computational grid blocked by space filling curves

The space filling curves inherently divides the whole grid into blocks of smaller sizes. The domain instead of being solved as a whole in some random manner is now being solved a 'block' at a time. This phenomenon can be clearly seen from Fig. 3.6. To start with all the cells inside the green square are solved first and when the calculations are done the next three blocks containing same number of cells in the green square are solved. It is to be noted that the next higher block designated by the blue square contains four smaller blocks of size designated by the green square. When all the four blocks inside the blue square are solved the whole process repeats itself with the blue square now being the basic block size, until whole of the grid is covered.

3.2 Space Filling Curve Results

The space filling curve scheme seemed promising but it did not yield encouraging results. The space filling curve can be easily generated only for a square grid with side of 2^n (where n can be any integer), which restricted our ability to test any random grid size. The results are presented for comparison in the Fig 3.7 and 3.8 and Table 3.1. The walltime presented is for 1000 iterations and is normalized by the grid size. The data calls are also divided by a factor of ten so that it can be easily compared to the cache misses.

Coincident with the initial findings we see that there are negligible L1 miss rates and almost zero miss rates for a grid size of 16x16 in both the cases. This indicates that the grid size nearly fits into the L1 cache. As we moved to the grid size of 32x32 we see an increase in the cache misses which increase a bit more for a grid of 64x64 and tends to asymptote for grids of higher sizes. There was a small reduction (~2%) in walltime for grids of smaller sizes and the effect was most noticeable on the 256x256 grid where the reduction in walltime was about 5%.

Though the space filling curve scheme did not alter the L2 cache miss rate considerably, a significant improvement in L1 cache misses was noted for all the larger grid sizes indicating the spatial data locality in the L1 cache. As the latency time associated with the L1 misses is not very large when compared to that of L2 and other misses, the L1 cache improvements could not manifest itself into large overall reduction in walltime.

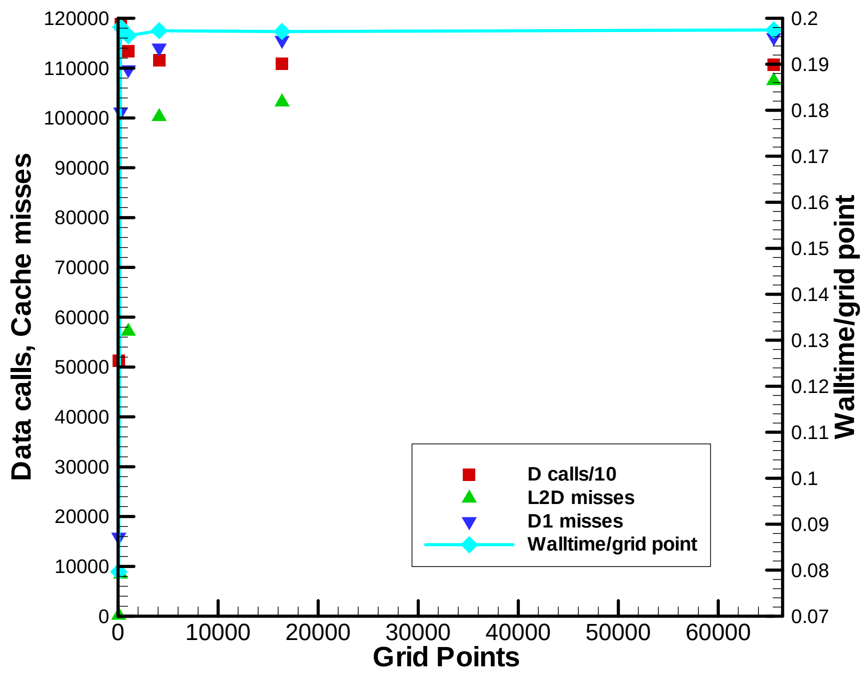


Figure 3.7 Walltime, data calls and cache misses as a function of grid size for i-j numbering scheme

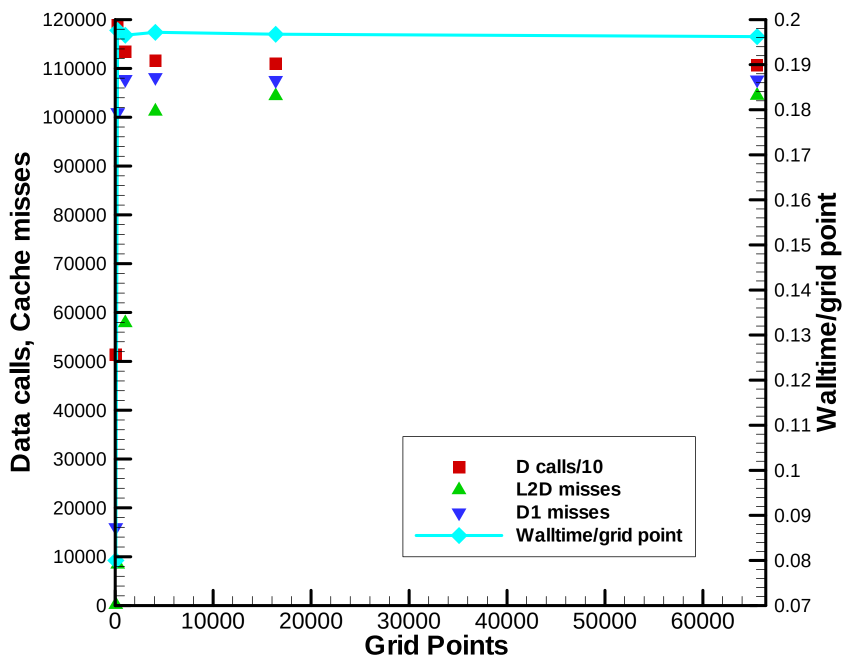


Figure 3.8 Walltime, data calls and cache misses as a function of grid size for SFC numbering scheme

Table 3.1 Comparative profiles of *i-j* path and SFC path for various grid sizes

	16x16		32x32		64x64		256x256	
Grid Type	IJ	SF	IJ	SF	IJ	SF	IJ	SF
Total Time (seconds)	0.702	0.688	6.723	6.542	33.78	33.70	597.9	567.8
i_solver_gs_velocity	0.12	0.20	1.89	1.82	7.62	7.49	136.8	135.8
i_solver_gs_p	0.18	0.18	1.33	1.35	5.39	5.33	132.9	131.2
cont_2d	0.11	0.05	1.16	1.08	8.14	8.17	132.0	124.0
cal_velocity_2d	0.14	0.09	1.23	1.15	8.32	8.33	110.2	90.0
gradients_2d	0.03	0.06	0.60	0.50	2.19	2.31	52.1	55.3
other	0.11	0.11	0.52	0.64	2.12	2.07	34.1	31.0

The ability of this technique to effect the L1 misses while causing no effect of L2 misses is intriguing but a possible explanation is discussed here. The grids constructed with the '*i-j*' manner were not exactly random; instead they were calculated in a deliberate pattern. The neighboring cells set for two contiguous cells were close together in most of the cases. Owing to the large L2 cache sizes the calculations on the neighboring cells were done near coincidently without any large L2 cache misses because the information was probably already present in the L2 cache. However the L1 cache is much smaller and could not retain the data pertaining to the neighboring cells all the time, hence we see more L1 misses in the '*i-j*' case when compared to the space filling curve path. If the grid was more random in manner we could probably have seen some significant improvements in the walltime but this hypothesis could not be tested because of lack of such random grids.

The results in the Table 3.1 for the large 256x256 grid clearly show small improvement in the *continuity* and *cal_vel* subroutines. On the other hand the Gaussian solver subroutines which are predominantly loop over cells showed negligible gains; hence they became primary focus in the next steps of the optimization process.

3.3 Loop blocking

The profiling of the unmodified code done using the *gprof* tool for various grids suggested that the two Gauss-Siedel solvers for the velocity (*i_gs_solver_vel*) and pressure (*i_gs_solver_P*), shown in Fig. 3.5, were the two critical and also the costliest subroutines in terms of computation for grids of substantial (or practical) sizes. Both these subroutines contained internal sub-iterations over the whole computational grid (or block in case of parallel processing) to ensure accuracy and a stable solution for each time step.

During these sub-iterations the change in velocity (Δv) and pressure (Δp) were calculated and added to the velocity and pressure respectively calculated at the previous timestep. To calculate the Δv and Δp of each cell the Δv and Δp of the neighboring cells are required which thereby translates into the data calls. This operation is repeated in each subroutine for a prescribed number of internal sub-iterations.

When the computation is performed over a large block or grid size the data called in for computation of a particular cell eventually squeezes out (data miss) of the L2 cache as there is data coming in for the computation performed over a distant cell and occupies the space in the L2 cache. As mentioned earlier, this operation is carried out for a prescribed number of sub-iterations, the data is squeezed out for the same prescribed number of times and this leads to subsequent recalls leading to huge data misses in the these subroutines.

The sub-iterations are only meant for the purpose of accuracy. So if the data can be reused before it can be moved out of the L2 cache, it will significantly reduce the data misses associated with these two subroutines. This leads to the idea of loop blocking. Loop blocking improves the temporal and spatial locality of the data. By dividing the domain into smaller cache friendly blocks we can ensure that the calculations are performed on a contiguous set of data and this data will be reused in the near future.

The domain available for computation was further divided into smaller cache-friendly subblocks during the computation in the two Gauss-Siedel solver subroutines as shown in the algorithms 3.1 and 3.2. The size of the cache-friendly subblocks was determined by conducting some experiments with subblocks of different sizes. The subblock size of 30x30 was found to fit snugly into the L2 cache. During the computation of the Δv and Δp of all the cells in the smaller subblock most of the data required for the computations remained in the cache and was readily available for calculation (and subsequent recall for internal iteration).

```
Do iter = 1, nstep
Do i= 1, block
    Call $\Delta$v or $\Delta$p of neighboring cells
    Calculate the $\Delta$v or $\Delta$p of cell
Enddo
Enddo
```

Algorithm 3.1 The Gauss-Siedel subroutines in the *unmodified* UNCLE code

```
Do i= 1,nblock
Do iter = 1, nstep
Do j=1,subblock
    Call $\Delta$v or $\Delta$p of neighboring cells
    Calculate the $\Delta$v or $\Delta$p of cell
Enddo
Enddo
Enddo

subblock = cache friendly block size
nblock = block/subblock
```

Algorithm 3.2 The Gauss-Siedel subroutines in the *modified* UNCLE code

In this way the data can be efficiently used before being moved out of the L2 cache which leads to reduction in the data misses associated with those two subroutines and reduction in the walltime. An important thing to note is that the computation done in the code with the loop blocking is fundamentally similar to the ones done in the non-blocked version but they are not exactly the same at every step. By applying the loop blocking in the Gaussian solver subroutines we are changing the way the computation of Δv and Δp is done in the internal subiteration, because some of the Δv's and Δp's at the boundary of the blocks cannot be updated until the end of the subiteration. This might lead to minor discontinuities at the boundaries but results presented later show that they negligible.

3.4 Efficient Data Calling/Optimized Data Access

After the data misses and the corresponding walltimes of the two Gauss-Siedel subroutines were reduced, the attention turned to next set of critical subroutines, c*al_vel* and c*ontiniuty,* and a minor subroutine, g*radients.*

UNCLE is an unstructured code, which means there is no definite and defined structure for all the cells in a domain. The cells can assume any shape among the triangle, quadrilateral, hexagon, and octagon and cells of different geometry can coexist inside the same computational domain. As the cells used in the computation are of fairly complex structure, it is almost impossible to align their faces to standard Cartesian axes. To overcome this difficulty local Cartesian axes are assumed parallel and perpendicular to a face and later transformed to standard Cartesian axes by multiplying with a transformation matrix. For a two-dimensional case this transformation of each face has four elements in it. Another four elements associated to a face are its two vertices and two cells on either side of it. Also we have the *x*-coordinate and *y*-coordinate of the center of each face associated with it. This leaves ten elements being associated with each face. Some additional elements are added to the existing ones in the three dimensional calculations.

The faces are further divided into two data structure types. The first being '*Internal*' which contains all the internal faces of a computational domain and the other being '*BC*' (Boundary Condition) which contains all the faces on the boundary of a computational domain. The above mentioned ten elements are required for computation in the aforementioned subroutine *cal_vel*, c*ontiniuty* and g*radients*, all of which contain the two data structure types '*Internal'* and *'BC'*.

All of the ten elements are called for each face in the two data structure types and assigned to a local variable and the computation is done using the local variable. The assignment is done in the following manner

For '*Internal'*

```
   xx = internal%face(iface)%x
```

and for '*BC'*

```
   xx = bc%face(iface)%x

   xx = local variable
   x = element associated to the face
```

As all ten elements are typically called upon at the same place in the code, it would be beneficial if all of them were stored at the same place using a cache-friendly two dimensional array instead of a pointer. This could make use of the spatial data locality principles. When a data is accessed, a bunch of data along with the requested data is grabbed from the memory. If all the data pertaining to the cell is stored in the same place, chances are that relevant data is grabbed along with the requested data and is already present in the cache when the request for it is made. This will act like data prefetching. The difference between the two methods can be seen in Fig. 3.9. Two new two-dimensional array variables '*facex'* (for integers) and '*facedata'* (for floating point numbers) were introduced and were used to store the data. The allocation to the local variable was made in the following manner as compared to above

For ‘*Internal’*

```
xx = internal%facex(iface,num)
xx = internal%facedata(iface,num)
```

For ‘*BC’*

```
xx = bc%facex(iface,num)
xx = bc%facedata(iface,num)
```

where each ‘*num*’ corresponds to a different element of the face.

This leads to more efficient data storage and memory allocation and also to reduction in the data misses and the walltimes. Further improvement can be achieved by integrating the two data structure types *‘Internal’ and ‘BC’* and thus removing the dependence of the variables ‘*facex*’ and *‘facedata’* on the two data structure types (removal of the remaining pointer %). This though being a simple concept can turn out to be complicated in application and also undermine the advantages of having two data structure types in various other parts of the code. It is to be noted that no fundamental change is done in the way the computation is being done unlike the case with loop blocking. Data that is being is called upon for computation is being called in a more cache friendly manner.

3.5 Code Versions

The tuned up version as described in the beginning of the chapter, after removal of redundant statements and rearrangement of statements with in a loop, was designated as the basic version U1. The code that incorporated the two dimensional arrays instead of the pointers was named U2. The version that employed the subblocking in the Gaussian solver subroutines was called U3. Finally, all the modifications of the above two codes U2 and U3 were combined in the same code to arrive at the final version U4. All the versions with turbulent calculations have been designated with ‘T’ instead of ‘U’.

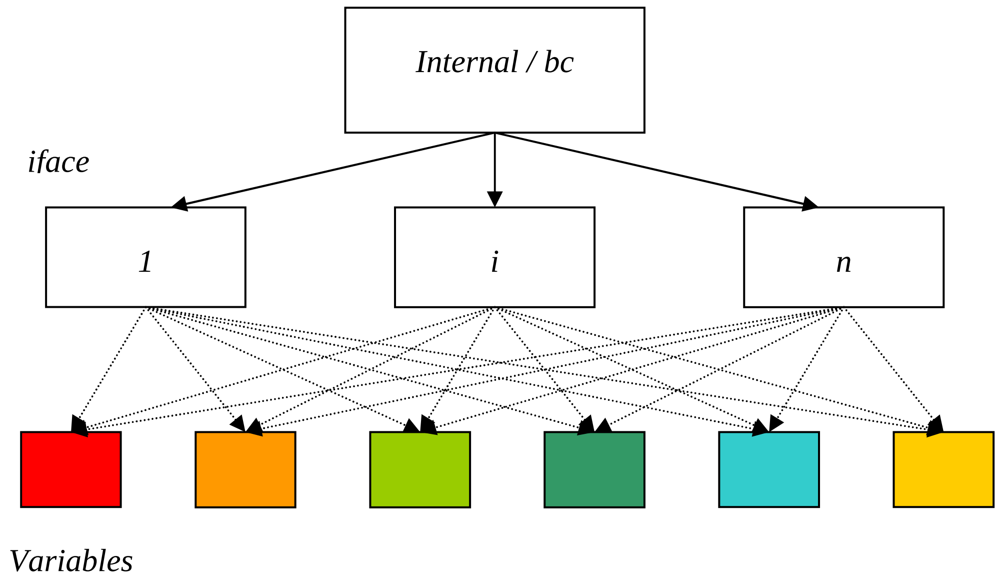


Figure 3.9(a) Original data structure

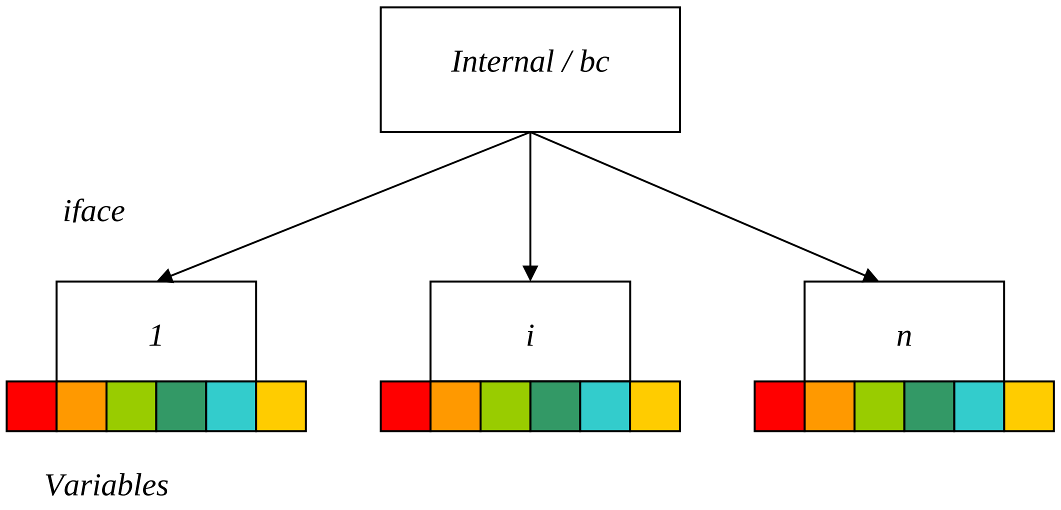


Figure 3.9(b) Modified data structure

3.6 Results

It is evident the smaller subblocks will fit better in the memory and have fewer data misses. But if we use very small subblocks then the extra communication cost may eclipse the advantage gained in walltime due to improved cache performance. Also we might have instabilities or discontinuities growing at the boundaries of the subblocks because they are not being updated at every subiteration within the Gauss-Siedel point solvers. So a series of tests with subblocks of different sizes were done.

The results of the preliminary tests done with subblock of various sizes are shown in Fig. 3.10. The walltime considered throughout this study is the time taken for computation excluding the time taken for file input and output. It is normalized by number of iterations and grid size for ease of comparison. In this case the walltime presented here is for 1000 iterations We can see that the code with subblocks of 70x70 shows an improvement (~10-12%) over the basic unmodified version but it is less than the improvements made by the code with 10x10 subblocks which is about 35%. The subblock size of 10x10 was the most efficient subblock size. A subblock of lower size showed diminished performance possibly due to increased loop overheads.

A comparison of the results of basic UNCLE version U1 and the subblock 10x10 for smaller grids revealed that the codes behave in a similar fashion up to a grid size of about 900 (30x30 subblock) grid points as seen in Fig. 3.11. At that point we see a sudden and jump in the walltime of the basic version which indicates the likelihood of deteriorating cache performance at that point. The walltimes increased initially after this point and then it started to asymptote displaying the same poor performance.

This suggests that the subblocks as big as 30x30 can be used instead of 10x10 without any significant loss of performance because the subblock of 30x30 could easily fit into the L2 cache. Though a sublock of 10x10 is a bit faster, a subblock of 30x30 would require dividing the domain in lesser parts and this would ensure that we have less instability building up at the boundaries of the subblocks. To confirm the cause of the deteriorating code performance a valgrind cache simulation was done and the results are

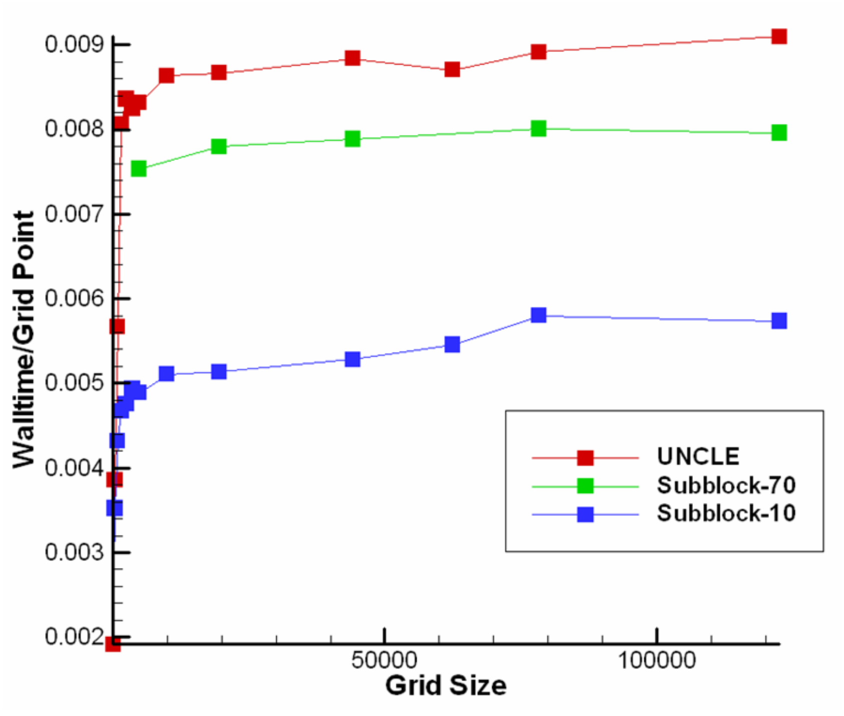


Figure 3.10 Walltime comparisons for various code versions

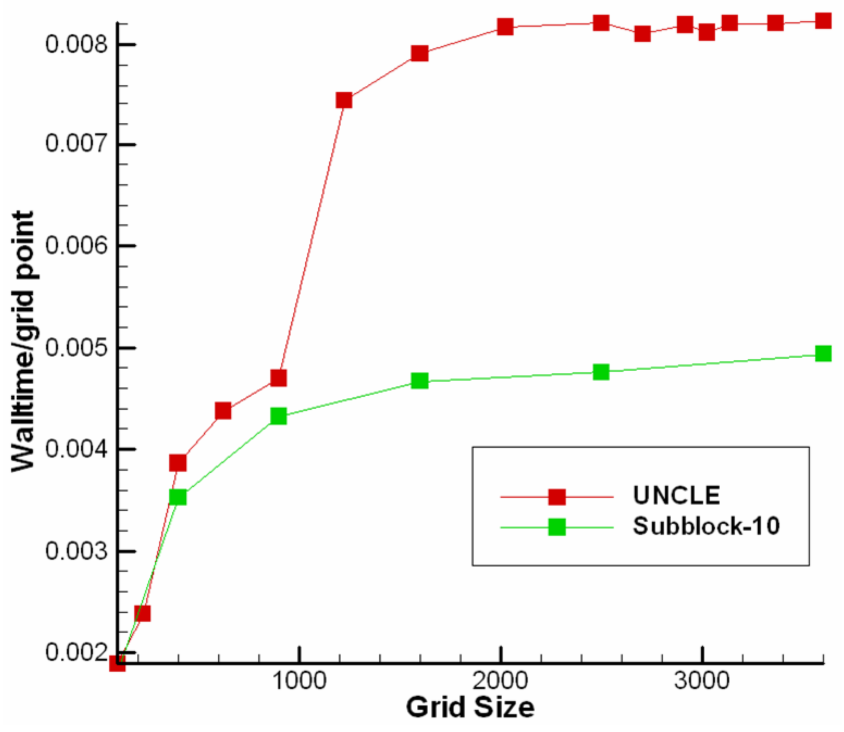


Figure 3.11 Determination of efficient subblock size

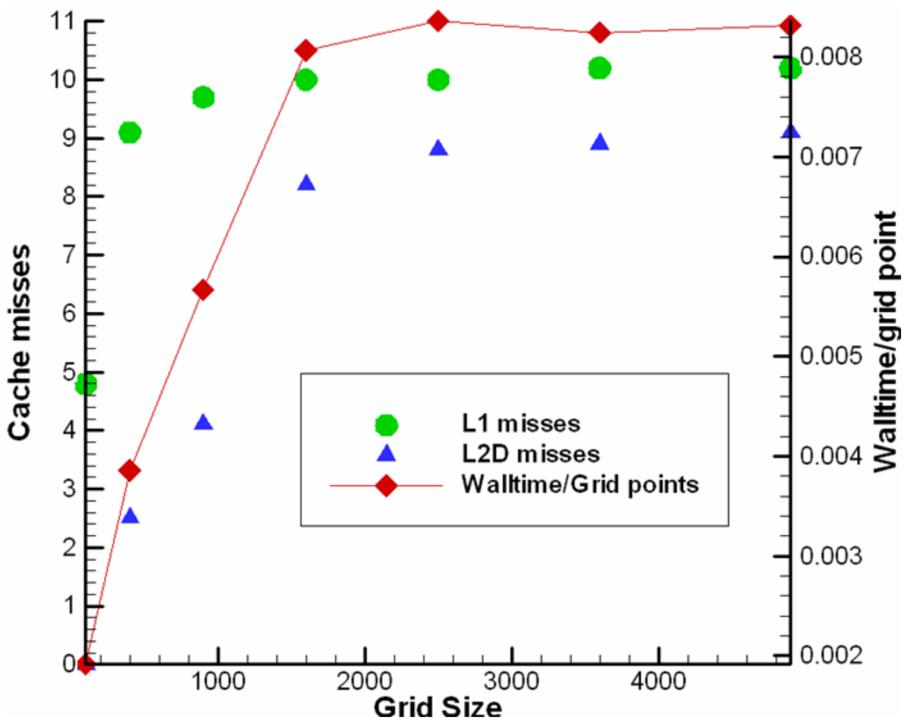


Figure 3.12 Walltime and cache misses for various grid sizes

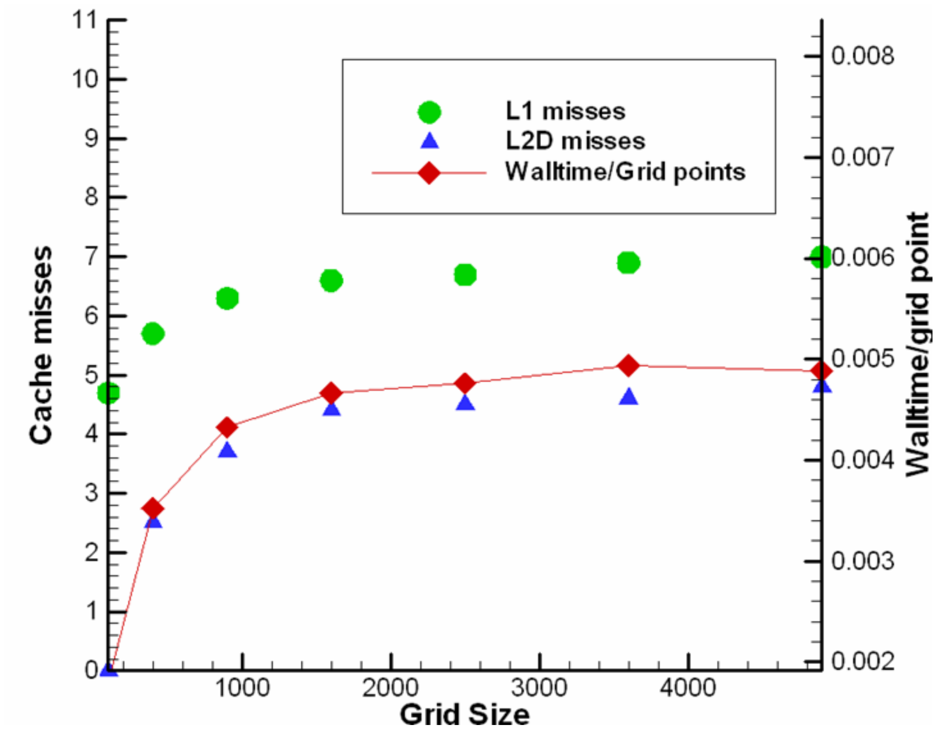


Figure 3.13 Walltime and cache misses for smaller grids

shown in the Fig. 3.12 and 3.13. It can be noted that there is a strong correlation between the cache misses and the walltime in both the cases. It appears that the performance enhancement was indeed cache driven.

The changes in the face data structure was made in the subroutines throughout the code. The loop blocking was applied to Gauss-Siedel solvers for velocity and pressure because the performance of these subroutines deteriorated the most at the larger grids when compared to the small grids as shown in Fig. 3.5. The walltime tests were done with various versions of the code on KFC4 for numerous grid sizes and are shown in Fig 3.14. The walltime was normalized with the grid size for sake of comparison among various grid sizes. The data calls were divided by a factor of ten so they could be placed on the same figure. The walltimes and the cache data for all the four versions are presented in Fig. 3.15.

Version U2 which incorporates just the data structure change showed improvement of about 15% for almost all the grid sizes. There was not a noticeable reduction in the data calls but a significant reduction in the cache misses was observed. This indicates that the data was now being called in a more efficient manner and that the improvement was cache driven. The use of a two-dimensional array instead of a pointer ensured the spatial locality of the data.

Version U3 with loop blocking applied to the Gauss-Siedel solvers showed approximately 35% improvement in the walltime compared to the version U1. There was a significant reduction in the data calls and cache misses. The version U4, which is the combination of changes made in the version U2 and U3 showed further improvements in the walltime along with the decrease in the data misses. Overall from version U1 to U4 the walltime was reduced by half and the L2 cache misses were reduced by more than 50%.

The normalized walltime remained roughly the same for all the practical grid sizes. This same behavior was noticed in the L1 and L2 cache miss for most of the grid sizes. Fig. 3.18 shows the overall improvement of the modified codes over the original version U1.

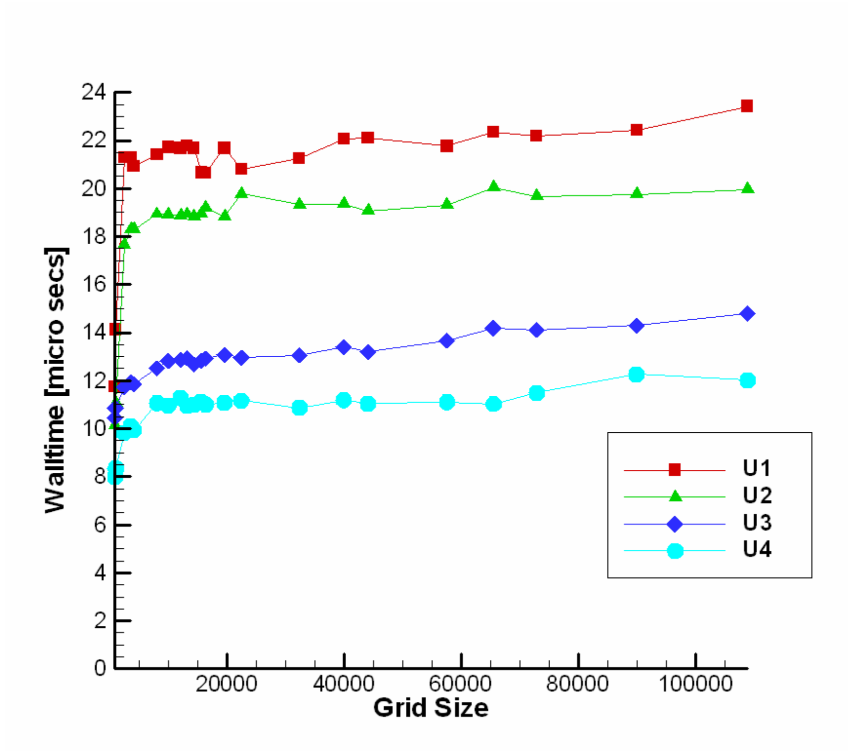


Figure 3.14 Walltime for different versions of the code

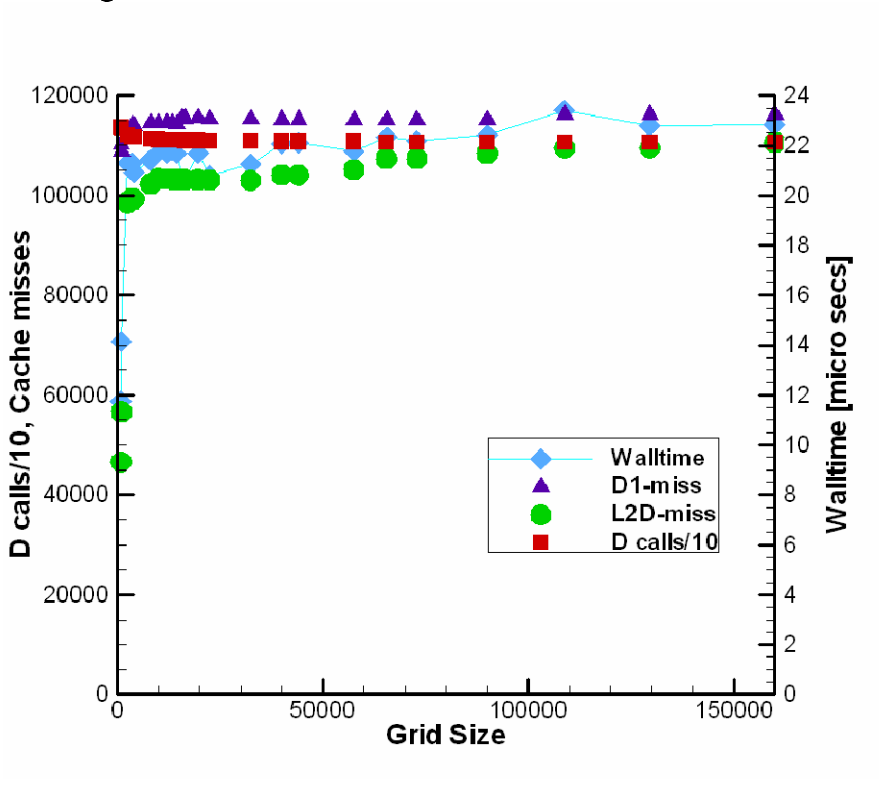


Figure 3.15(a) Walltime and cache data for U1

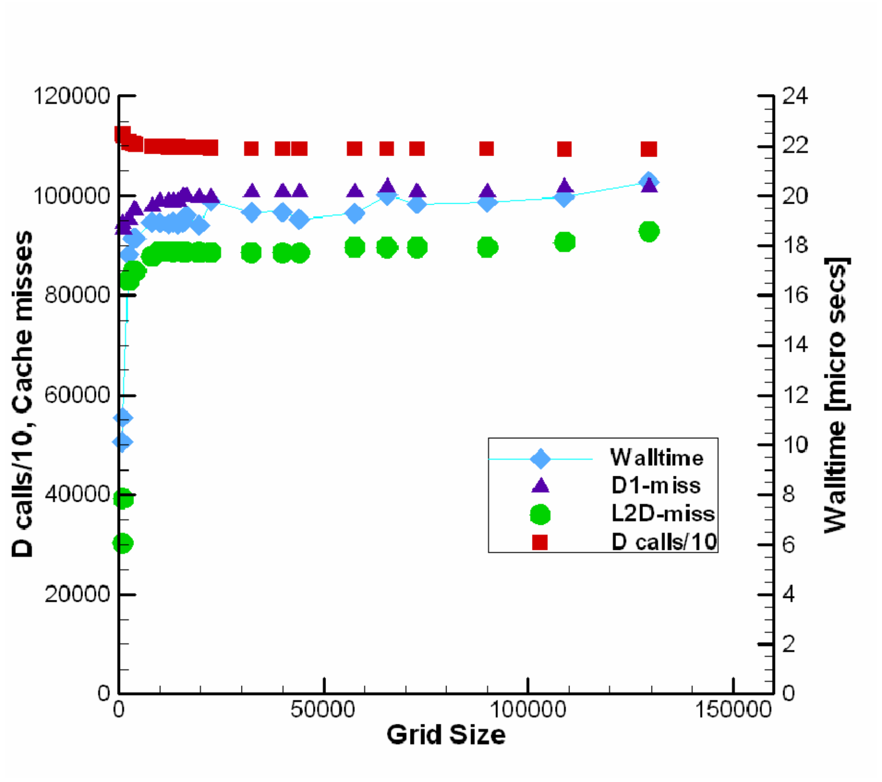


Figure 3.15(b) Walltime and cache data for U2

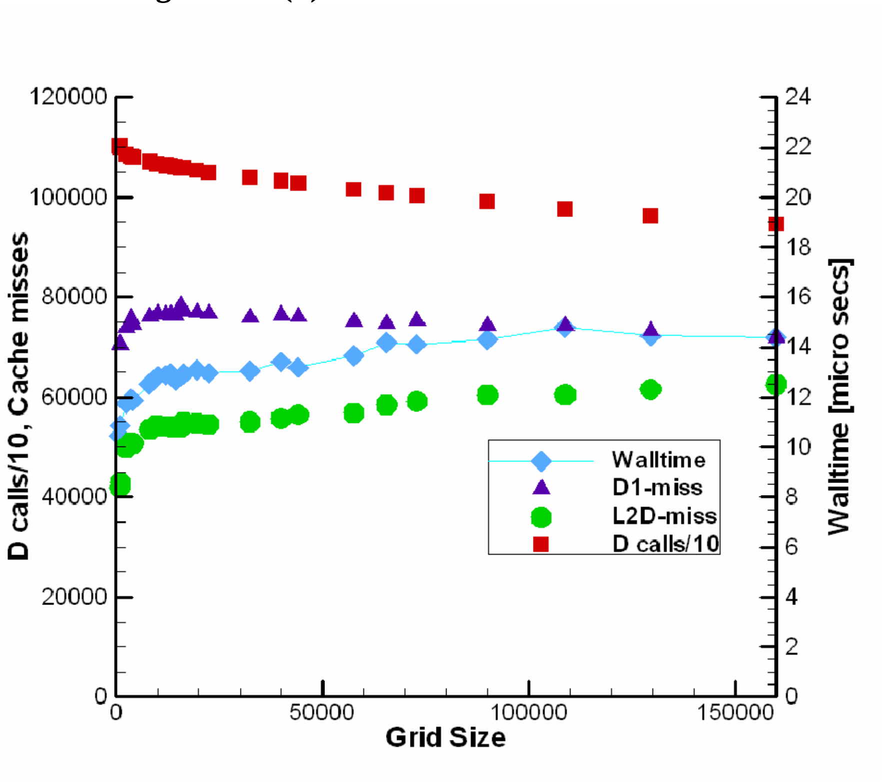


Figure 3.15(c) Walltime and cache data for U3

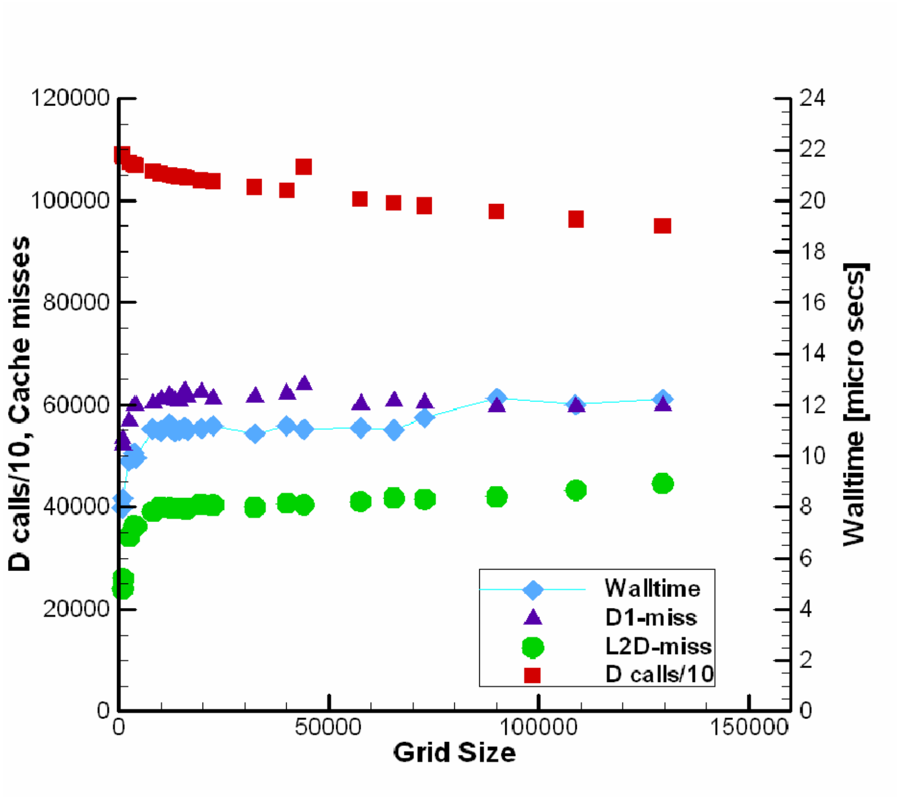


Figure 3.15(d) Walltime and cache data for U4

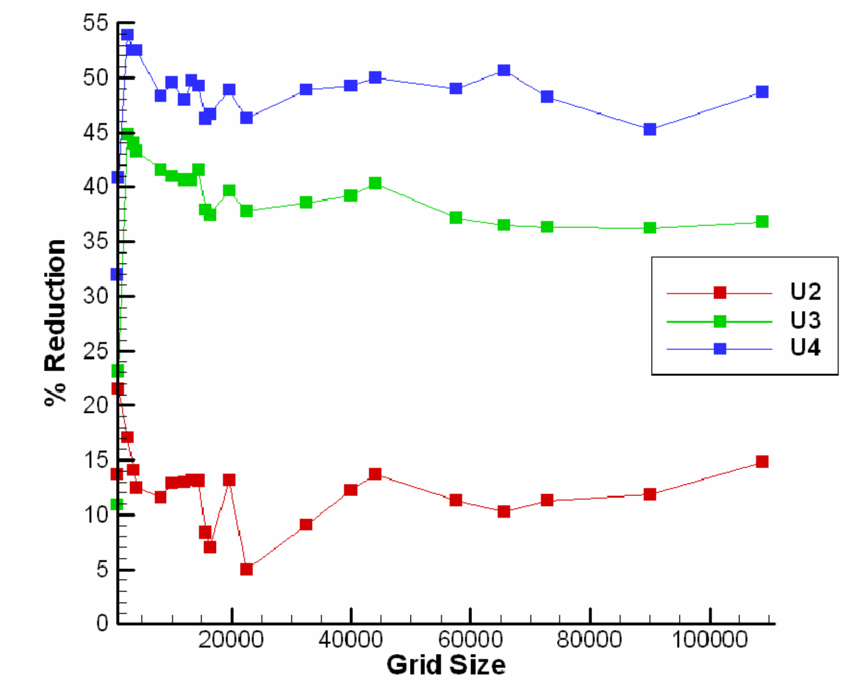


Figure 3.16 Performance improvement percentage over U1

A *gprof* was done on the versions U1 and U3 and the results are shown in Fig. 3.17. The Gauss-Siedel routines to which the loop blocking was applied shows an improvement of about 400%. It is demonstrated that the methods used lead to significant improvements in the code performance. The codes were also run to steady state convergence to study the effect of these changes on the results. The Reynolds number used was 400 and the grid size was 256x256. Fig. 3.18 shows the streamline plots obtained by running versions U1 and U4 of the code. The improved version in clearly in good agreement with the original version and it competently captures all the complex vortices.

A comparison of the *u*-velocity contour is shown in the Fig. 3.19. The contours obtained from both the codes are overlapped and they are in good agreement. Also shown in Fig. 3.20(a) are the *u*-velocity at the vertical centerline obtained using the original and the optimized version of the code and they are in good agreement with the results of Ghia's et al. [19].

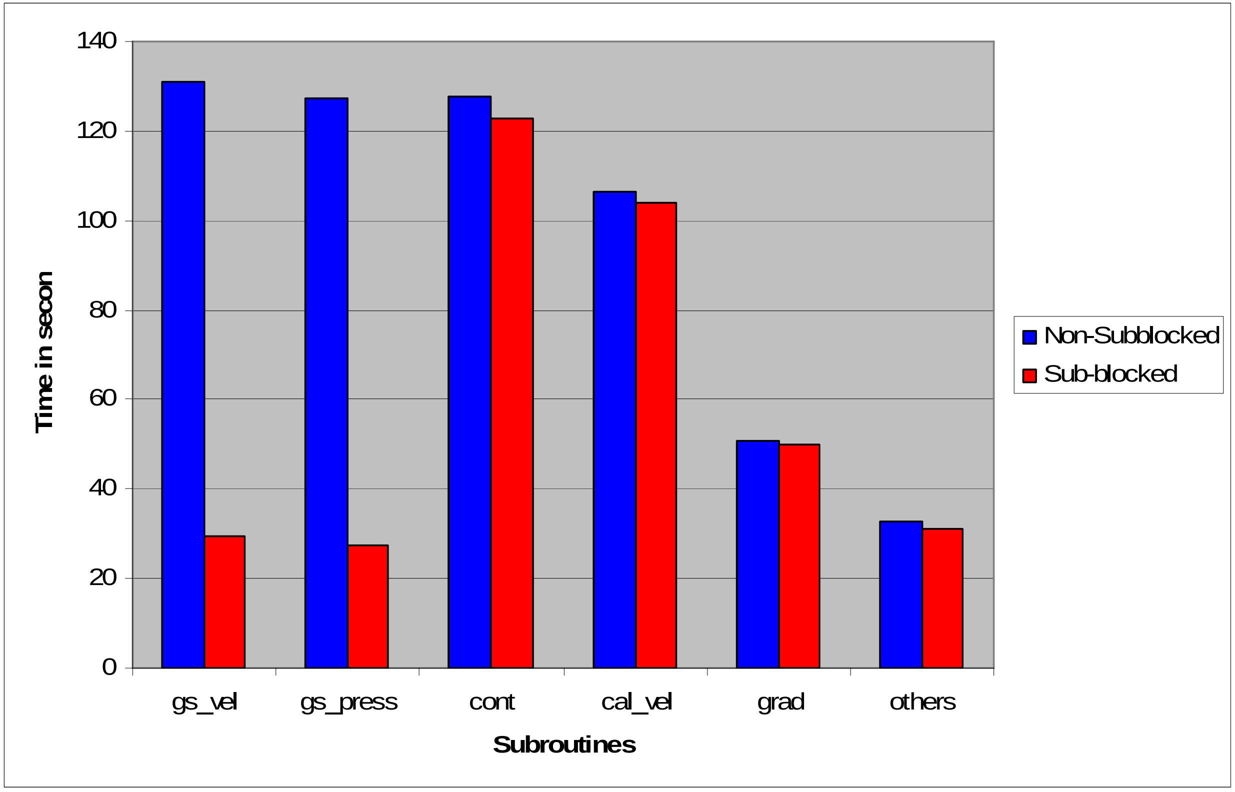


Figure 3.17 Comparison of time taken by critical subroutines for U1 and U3

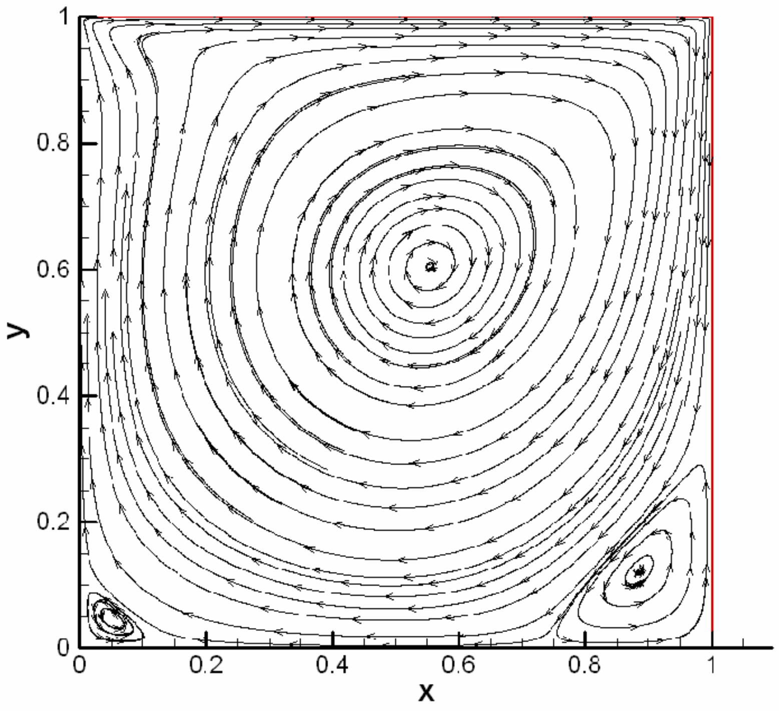


Figure 3.18(a) Streamline plot for U1

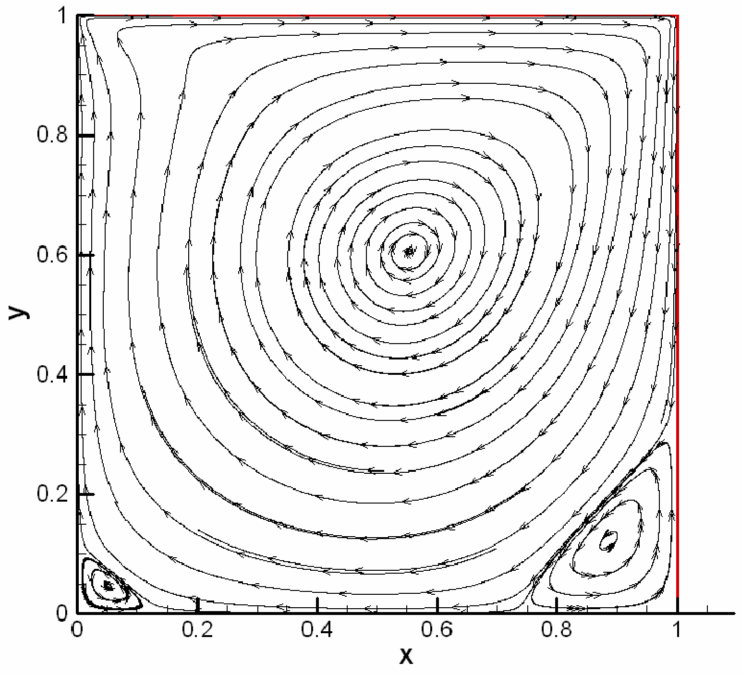


Figure 3.18(b) Streamline plot for U4

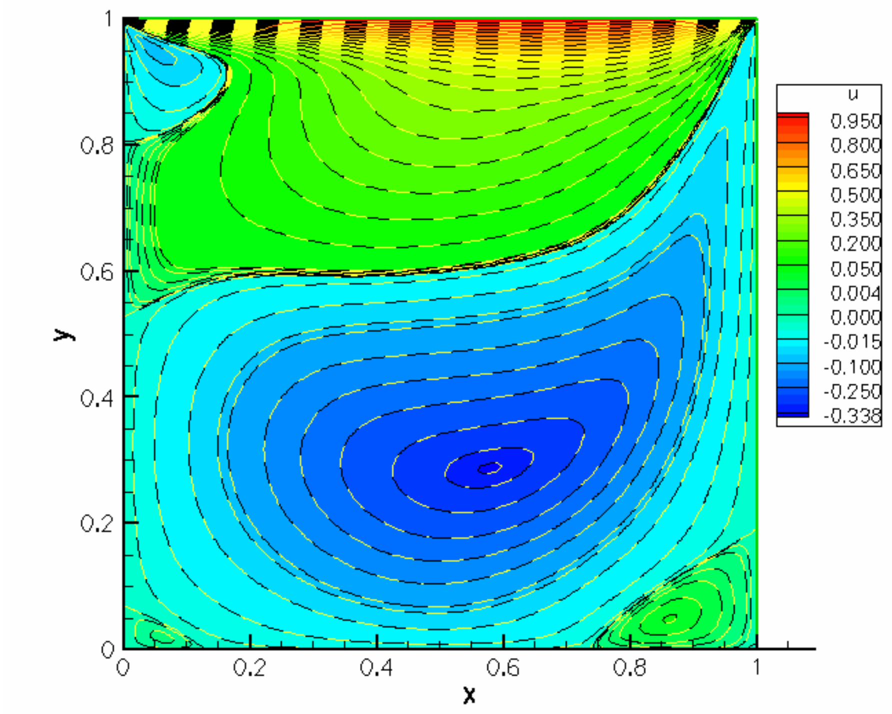


Figure 3.19 ***u*-velocity contour plots for U1(yellow) and U4(black)**

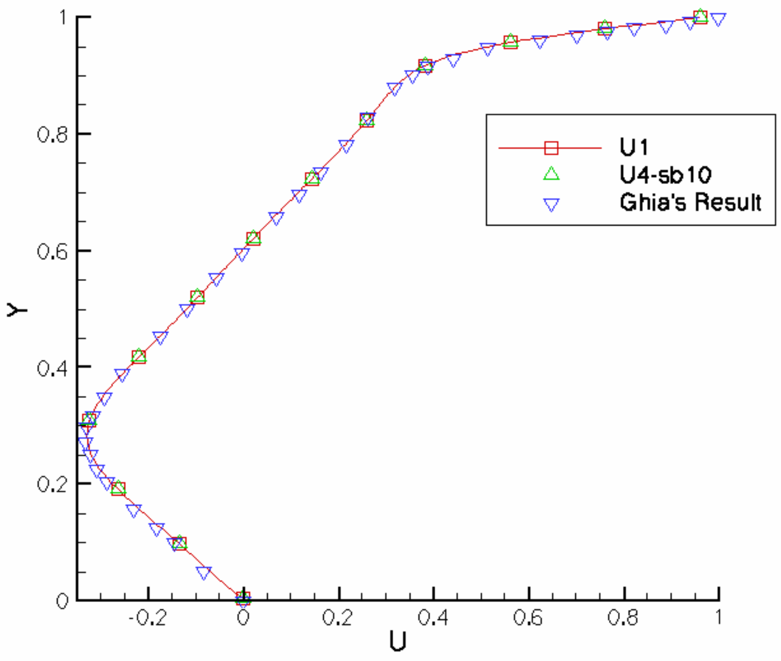


Figure 3.20(a) ***u*-velocity at the vertical centerline**

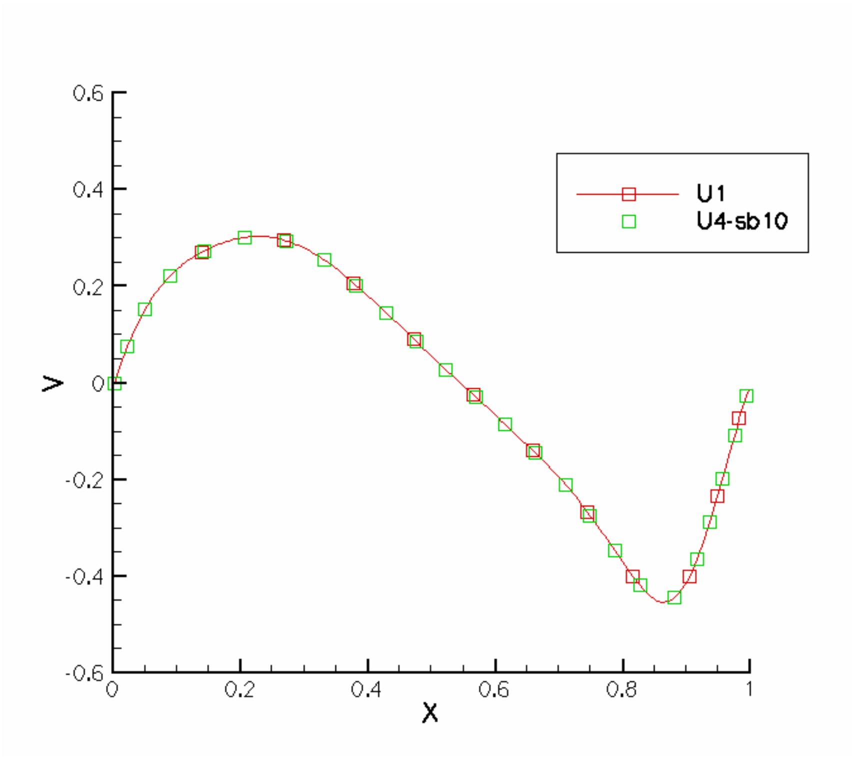


Figure 3.20(b) *v*-velocity at the horizontal centerline

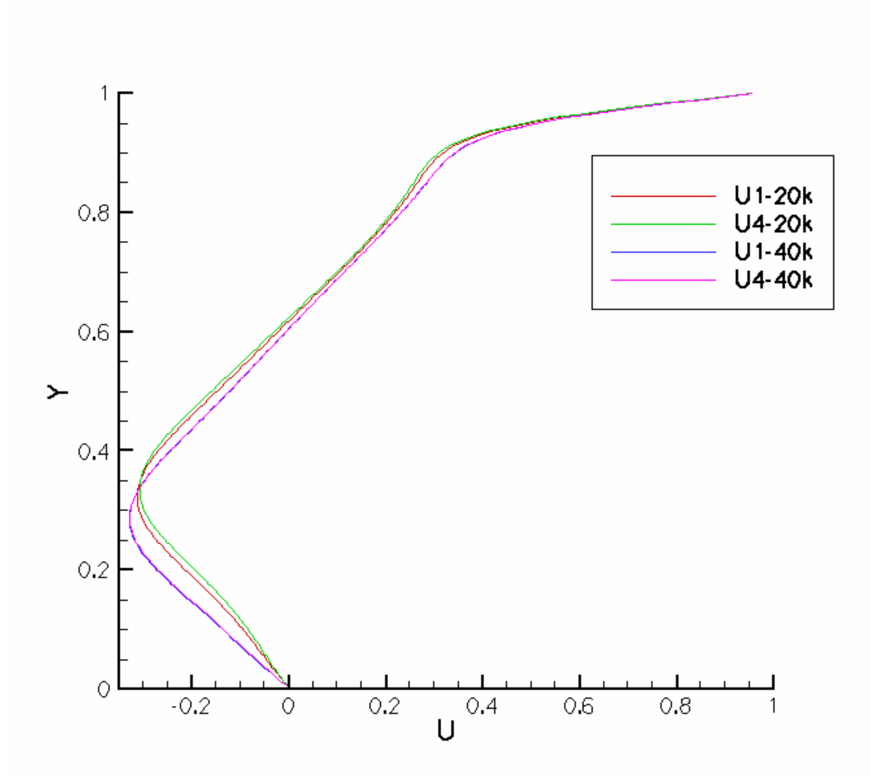


Figure 3.21 Evolution of *u*-velocity at the vertical centerline

Table 3.2 Accuracy comparison for versions U1 and U4

X	Y	*u*-vel up to 8 Decimal places		*u*-vel up to 10 Decimal places		*u*-vel up to 12 Decimal places	
		U1	U4	U1	U4	U1	U4
0.5	0.000000	0.00000000	0.00000000	0.0000000000	0.0000000000	0.000000000000	0.000000000000
0.5	0.062500	-0.09236981	-0.09236981	-0.0923698871	-0.0923698871	-0.092369887888	-0.092369887888
0.5	0.125000	-0.17838109	-0.17838109	-0.1783812287	-0.1783812287	-0.178381230067	-0.178381230067
0.5	0.187500	-0.26348443	-0.26348443	-0.2634845897	-0.2634845897	-0.263484591314	-0.263484591314
0.5	0.250000	-0.32085970	-0.32085970	-0.3208598292	-0.3208598292	-0.320859830583	-0.320859830583
0.5	0.312500	-0.32005546	-0.32005546	-0.3200555206	-0.3200555206	-0.320055521265	-0.320055521265
0.5	0.375000	-0.26626857	-0.26626857	-0.2662685769	-0.2662685769	-0.266268577152	-0.266268577152
0.5	0.437500	-0.19076444	-0.19076444	-0.1907644408	-0.1907644408	-0.190764440912	-0.190764440912
0.5	0.500000	-0.11510445	-0.11510445	-0.1151044484	-0.1151044484	-0.115104448463	-0.115104448463
0.5	0.562500	-0.04263024	-0.04263024	-0.0426302306	-0.0426302306	-0.042630230558	-0.042630230558
0.5	0.625000	0.03015695	0.03015695	0.0301569806	0.0301569806	0.030156980908	0.030156980908
0.5	0.687500	0.10532827	0.10532827	0.1053283267	0.1053283267	0.105328327362	0.105328327362
0.5	0.750000	0.18112395	0.18112395	0.1811240400	0.1811240400	0.181124040998	0.181124040998
0.5	0.812500	0.25196838	0.25196838	0.2519684972	0.2519684972	0.251968498546	0.251968498546
0.5	0.875000	0.31663297	0.31663297	0.3166331057	0.3166331057	0.316633107194	0.316633107194
0.5	0.937500	0.46945265	0.46945265	0.4694527578	0.4694527578	0.469452758959	0.469452758959
0.5	1.000000	1.00000000	1.00000000	1.0000000000	1.0000000000	1.000000000000	1.000000000000

The *v*-velocity profiles at the horizontal centerline are shown in Fig. 3.20(b) which are likewise in good agreement with each other. Fig 3.21 shows the evolution of centerline *u*-velocity with time. It can be seen that initially there is a slight disagreement between the *u*-velocity profiles but it disappears as the number of iteration increases.

The convergence criteria used for the above mentioned results was that the all residuals of velocities and pressure should be less that 10^{-6}.To check the numerical accuracy of result achieved to the last decimal place, the computations were done with increasingly stricter criterion for convergence. The simulations were done till the residuals were less than 10^{-8}, 10^{-10},10^{-12} successively and the *u*-velocity at the vertical center line obtained by using versions U1 and U4 was compared and is presented in Table 3.2. We can see that the solutions are in good agreement with each other. It was also observed that the rate of convergence of residuals from 10^{-6} to 10^{-12} was almost identical for both the versions. Version U4 took 10% more number of iterations than version U1, when reducing all the residuals to less than 10^{-6} was the convergence criteria. When a stricter criterion of 10^{-12} was used, the version U4 took about 3% more iterations.

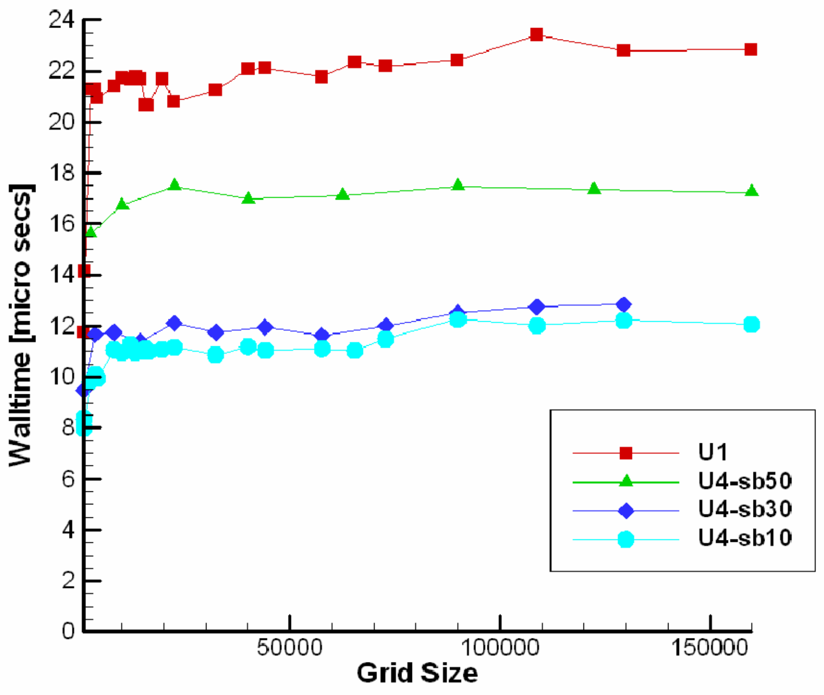


Figure 3.22 Prediction of performance based on the subblock size

The walltimes obtained using version U4 with subblocks of different sizes is shown in Fig. 3.22 There is a proportional reduction in the walltime with the decrease in the subblock size. There is not a large reduction between the walltimes of subblock 30x30 and 10x10 because a subblock of 30x30 fits into the L2 cache. So the improvements of 10x10 subblock over 30x30 subblock is minimal.

Another significant observation to be made is that the performance of the code with a particular subblock size is predictable. All the walltimes remain reasonably flat for grids of all sizes. The code performs approximately as if the smaller subblock is being solved instead of the whole grid. So running the tests on blocks of different sizes would help in predicting the potential gains that could be achieved by using the subblock of that particular size.

The convergence patterns of the residuals of two different versions for a grid size of 256x256 are shown in the Fig. 3.23 and 3.24. The residue shown is the square root of sum of square of the residuals of velocities and pressure. A few extra oscillations are seen in the U4 version and convergence is also a bit slow as seen in Fig. 3.23 for a subblock of 10x10. As the domain is solved in the blocks of 100 cells (10x10) small discontinuities build up at the boundaries and this probably causes the difference in the two patterns. The version U4 with subblock of 10x10 takes about 10% more iterations for convergence.

Unlike the 10x10 subblock the convergence of 30x30 was more similar to that of the original version U1 as seen in Fig. 3.24. This could be so because the 30x30 subblocks divide the domain into fewer internal blocks and fewer internal boundaries are created and hence the discontinuities are less. So it is beneficial to solve the domain in the larger subblocks and fewer subdomians. The code with the subblock of 10x10 is a bit faster than the subbblock 30x30 version but the convergence is faster in the latter version so the code performance of these two codes are comparable. A test conducted to compare the overall performance of these two subblock sizes showed that the 10x10 subblock is approximately 10% faster than the 30x30 version.

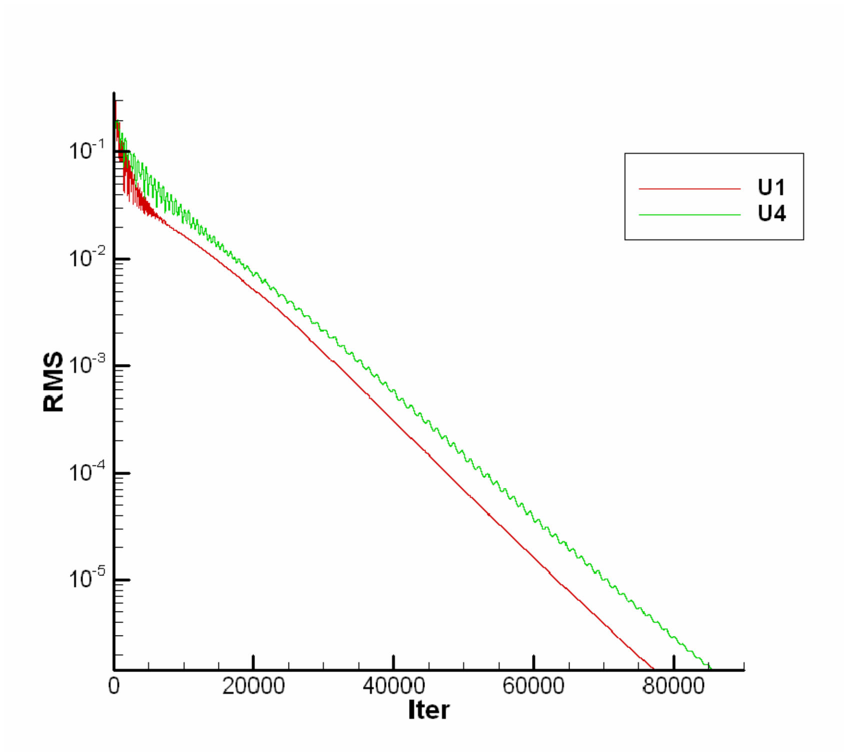


Figure 3.23 Residue convergence pattern for U1 and U4

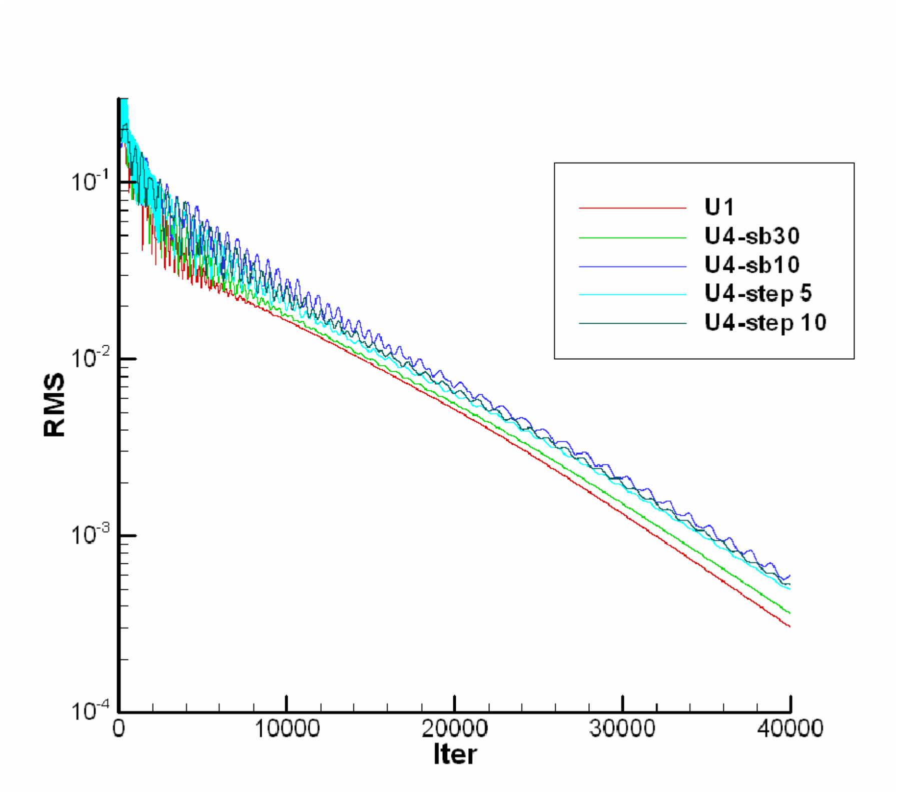


Figure 3.24 Residue convergence pattern for various versions of the code

A few tests were conducted to improve the convergence of the 10x10 subblock. The whole grid was solved every fifth or tenth sub-iteration instead of solving the subblocks in an effort to minimize the instabilities generated due to the subblocks. There is a slight improvement in the convergence when compared to that of the purely 10x10 subblocked version but it was not comparable to the convergence of the original or 30x30 subblock version.

The loop unrolling which involved replicating the body of the loop to reduce the loop overhead was also tested. The Gauss-Siedel solver subroutines for velocity and pressure were unrolled to investigate the potential improvements. The idea is to do more calculations per call so that less time is spent in checking the loop conditions. This technique though promising did not yield any significant improvements. This technique is usually successful in cases which require a lot of loop condition checking in a subroutine. In the case of UNCLE many redundant loops were already cleared to arrive at the original version. The minor improvements achieved due to the loop unrolling in these Gauss-Siedel solver subroutines could very well be masked by the diminished performances of L1 and L2 cache in these subroutines.

3.7 Summary

In this chapter a novel technique of numbering the cells in a domain by using the space filling curves was discussed. This promising technique was applied to the grid numbering system without any major improvements in the code performance. The loop blocking scheme was applied to the Gauss-Siedel solver subroutines in the code and tested on a steady laminar test case with successful results. Approximately 35% improvement in walltime was observed for grids of all sizes. It was shown that these improvements were achieved due to enhanced cache performance.

Another major change was made to the code by restructuring some of the data structures used to store data. The data structures with pointers were replaced by two dimensional arrays to provide better spatial locality of the data. This yielded approximately 15% improvement in the wall time for all the grid sizes. So overall, approximately 50%

improvements in walltime compared to the version U1 were observed. Using the domain numbered using space filling curve can add an additional 5% improvement to this existing 50%. Furthermore accuracy tests were performed to check the efficacy of the new techniques with successful results. A brief discussion on the effect of these techniques on various code performance parameters was presented. It was established that the performance of a loop blocked code could be predicted by doing a few simple tests.

In this chapter techniques were tested on a steady laminar case. These techniques now need to be tested on unsteady and turbulent test cases to establish the universality of the approach. Also they have to be tested on a different system to study their behavior on a different computational architecture. Parallel computations needs to be conducted to test the scalability of this modifications. Finally the work has to be extended to some three-dimensional and more challenging test cases. The aforementioned work is presented in the next chapters.

CHAPTER 4

UNSTEADY AND THREE-DIMENSIONAL RESULTS

In the previous chapter we have discussed the various optimization techniques, both successful and unsuccessful, and tested their efficacy on a steady laminar test case. The results obtained from the basic version of the code U1 were in very good agreement with the modified version U4 of the code. In this chapter we will present the results of tests done on time-dependent cases. The study will further extend to both steady and unsteady three-dimensional cases.

4.1 Unsteady Test Case

The flow over an unsteady cylinder was chosen as the test case for a time dependent problem. It is a fairly standard test case for unsteady simulations and has simple boundary conditions as shown in Fig. 4.1. The non-dimensionalized *u*-velocity is unity and *v*-velocity is zero at the inlet. A no-slip boundary condition is imposed on the bottom wall. The top and right walls are for the outflow. The Reynolds number for these simulations was 100.

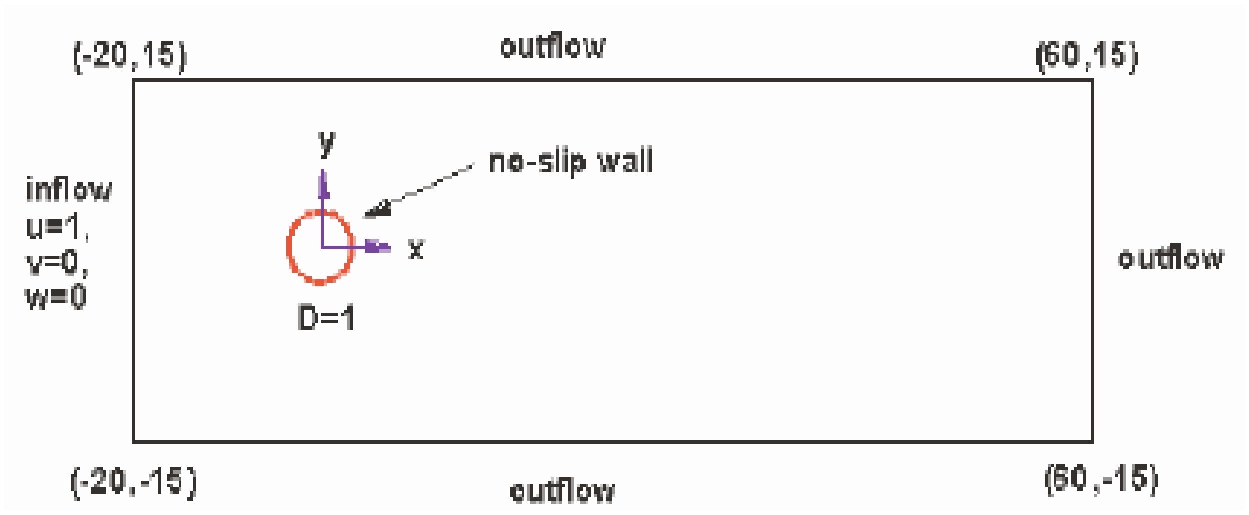


Figure 4.1 A schematic diagram of flow over a cylinder

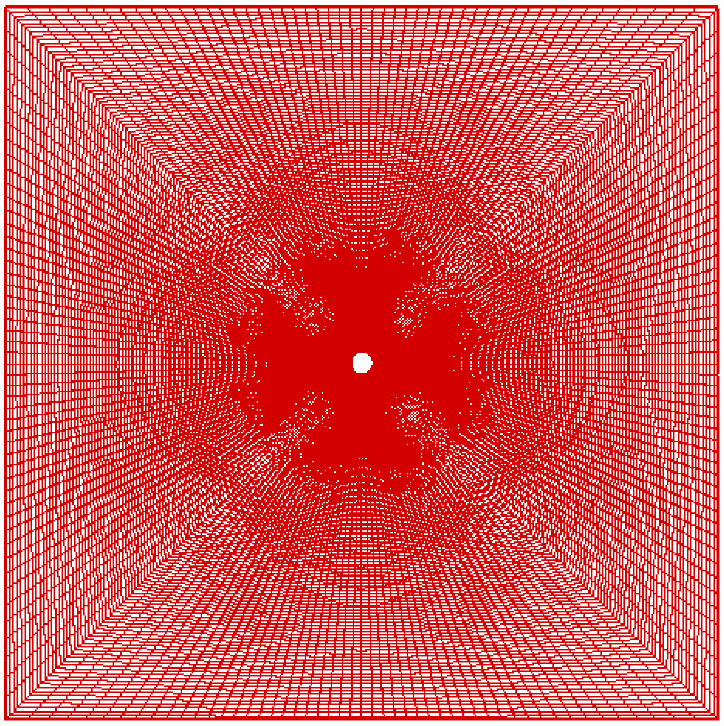


Figure 4.2 Grid used for the unsteady timing test

The grid used for the timing tests is shown in the Fig. 4.2. The cylinder has a unit diameter. The side of the square is 30 diameters. It is a square grid instead of a rectangle as shown in Fig. 4.4 which is 80 diameters in length and 40 diameters in height. The grid is dense and circular near the cylinder and as it moves out it becomes coarse and tends towards a structured square grid. The grid shown is 200x200 grid points with 200 points in circumferential '*i*' direction and 200 points in the radial '*j*' direction. The tests were conducted on the grids of multiple sizes of the same geometry as discussed above and the results are shown in Fig. 4.3.

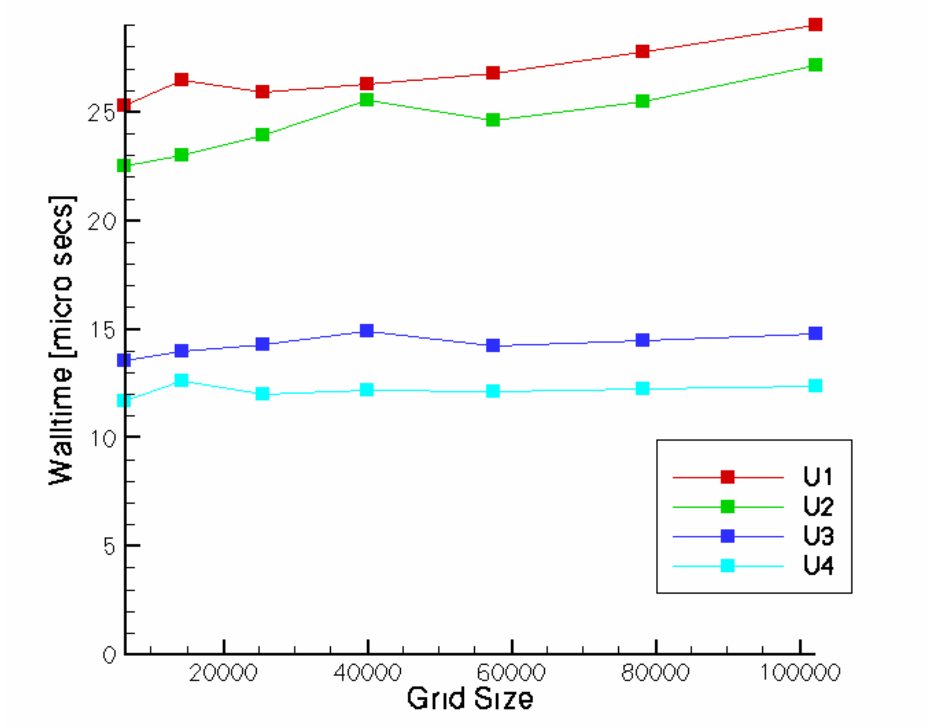


Figure 4.3 Walltime comparisons of various versions for a flow over a cylinder

4.1.1 Two-dimensional Unsteady Test Case Results

An improvement in the performance similar to the steady state case was observed. The performance improvement of smaller grids was slightly in excess of 50%. For grids of larger size the improvements achieved were close to 60%. There was an enhanced performance for large grids because the version U1 took larger time for calculation per

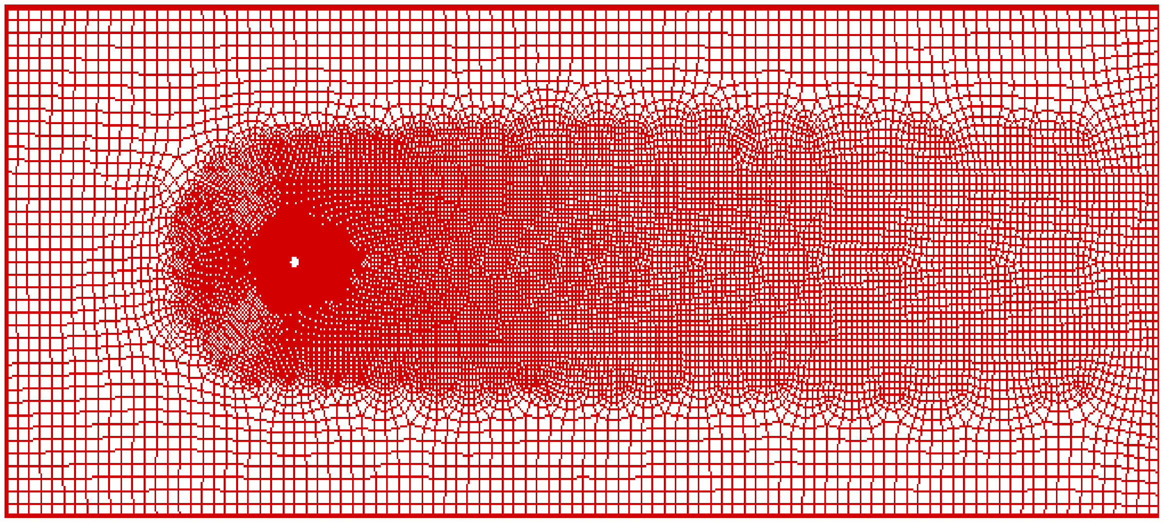


Figure 4.4 Grid used to do the accuracy tests for flow over a cylinder

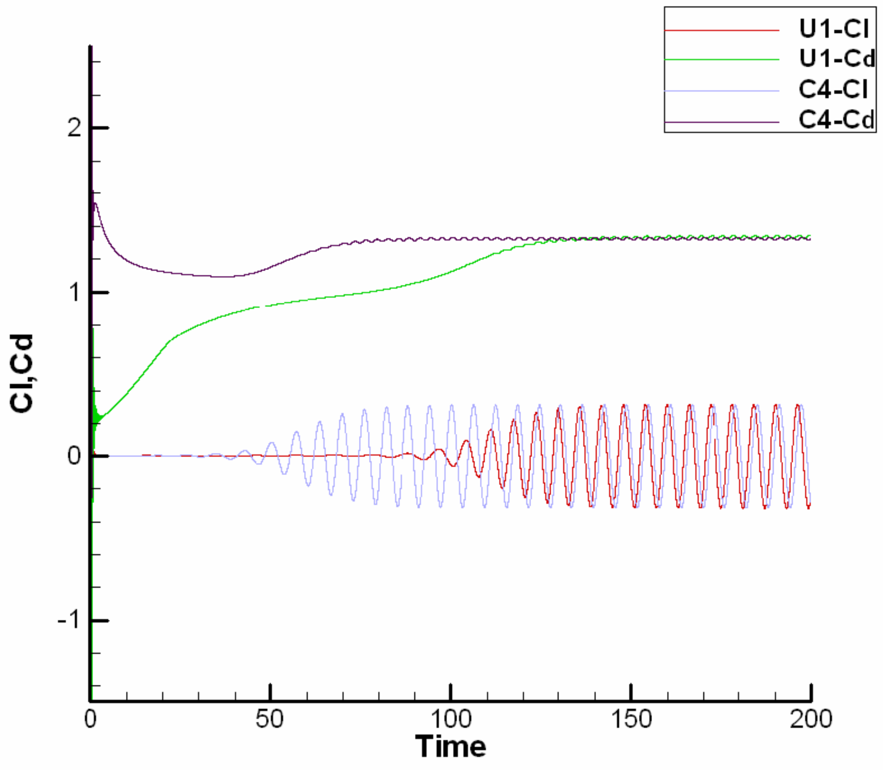


Figure 4.5 Lift and drag comparisons of versions U1 and C4

grid point whereas for the version U4 the time spent for calculation per grid point remained the same as the one for the smaller grids. The overall improvements were in excess of 50% for all the grid sizes for the cases which is slightly better than the steady state simulations because twice as many subiterations (10 for velocity and 20 for pressure) were used in the Gauss-Siedel solver for accuracy.

To check the accuracy of time dependent computations of the modified code the grid shown in Fig. 4.4 was used. It is slightly different when compared to the grid used for the walltime time tests. It is rectangular as opposed to the square grid used earlier and consists of 22705 cells. The cylinder in not in the center but it shifted towards the left with lot of grid points in the wake region. The grid is finer near the cylinder and it becomes coarser as we move away from the cylinder.

One of the basic reasons we chose to use a different grid for accuracy test was to compare it the results presented in Hua et al. [13] which were obtained during the validation process. The validation process used the exact same grid so it behooves us to use the same grid as it facilitates direct comparison. A direct comparison of lift and drag data for Re =100 between the code version C4 which was used in validation and version U1 is shown in Fig. 4.5 and Table 4.1. The version C4 had the same overall methodology of computation but there were minor differences in the way some calculations were done. This explains the initial disparity between the lift and drag patterns of the two codes. After the initial period of disagreement in the lift and drag they tend to concur as steady periodic oscillations are achieved.

Once it was established that the version U1 is comparable to C4, the accuracy tests were done for the version U4 with different subblock sizes. The lift and drag from these tests are shown in the Fig. 4.6. It can be seen that the lift and drag from different versions are in good agreement with each other. A close comparison of the lift data is shown in Fig. 4.7 (a). There is a slight shift in the phase but the magnitude and frequency of the lift remains the same. Fig. 4.7 (b) shows the blowup plot for the drag data. Again we observe

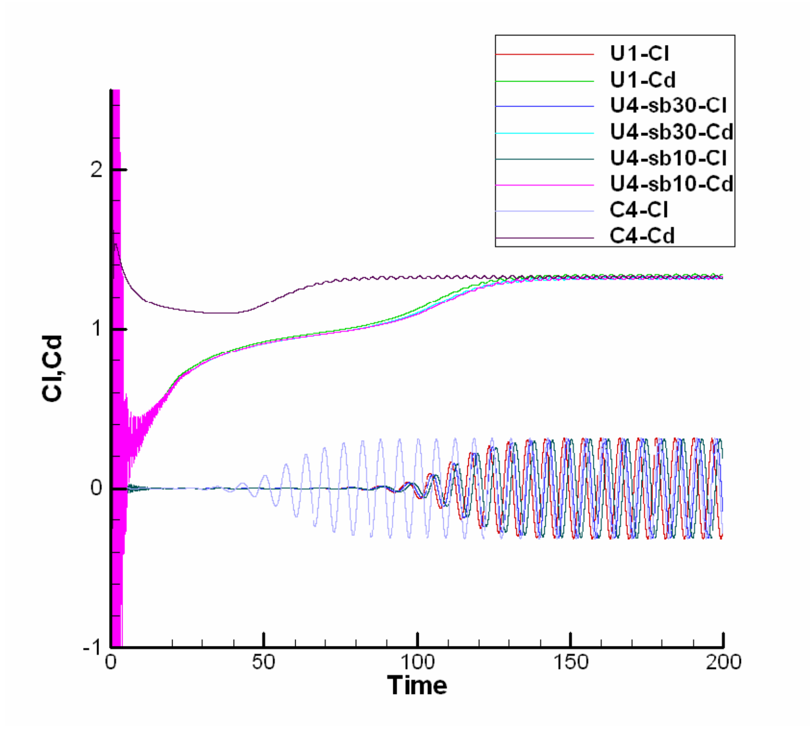


Figure 4.6 Lift and drag comparisons of various code versions

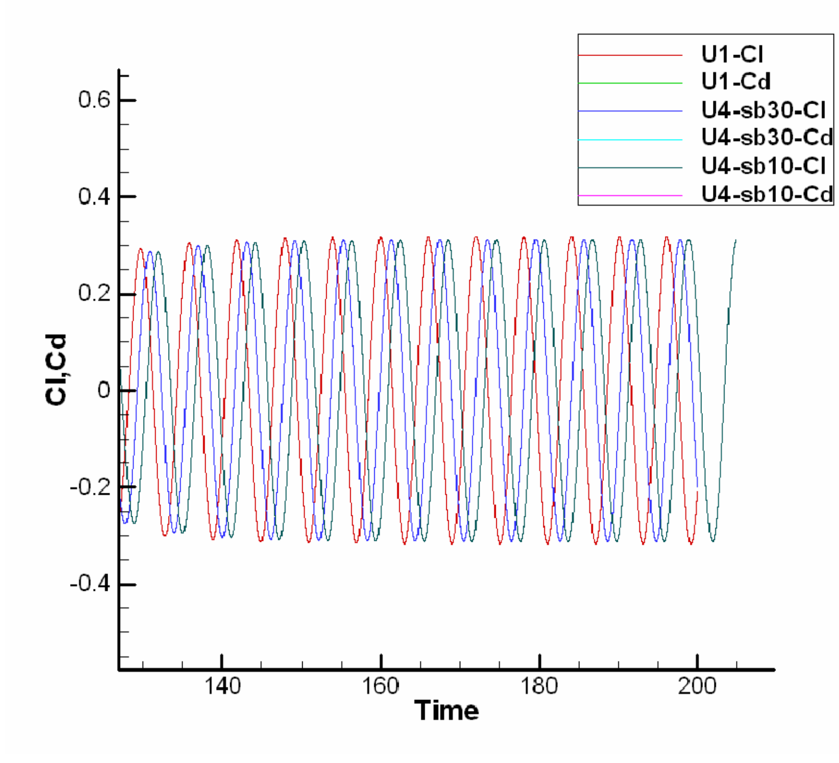


Figure 4.7 (a) Lift comparisons of various code versions

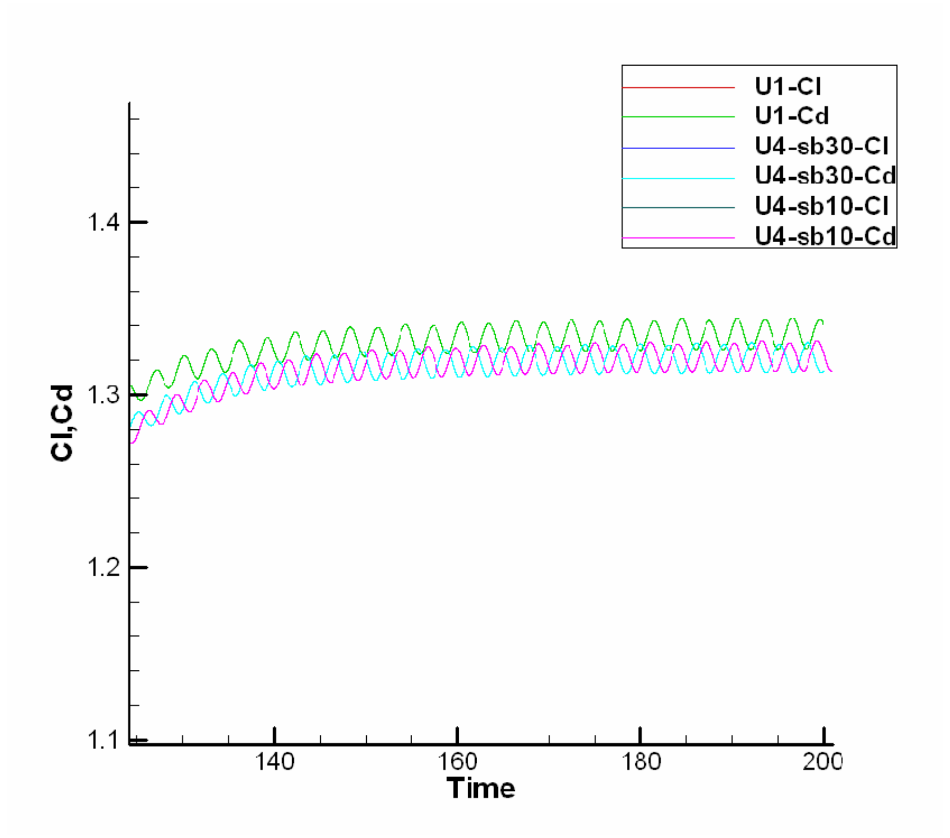


Figure 4.7 (b) Drag comparisons of various code versions

Table 4.1 Comparison of lift, drag and Strouhal number for various 2D codes

Source (Re = 100)	$\mathbf{C}_L$	$\mathbf{C}_D$	S_t	Speedup Comparison
Hua et al. [13]	± 0.314	1.325 ± 0.008	0.165	-5%
Ku et al. [20]	± 0.228	1.33~1.358	0.1675	-
U1	± 0.317	1.335 ± 0.008	0.1657	0
U4-sb30	± 0.3128	1.323 ± 0.008	0.1646	45%
U4-sb10	± 0.3134	1.3232 ± 0.010	0.1645	53%

a slight shift in the phase similar to the lift data but here we see a slight difference in the magnitude of the drag. A look at the Table 4.1 shows that the drag amplitude for the subblocked version is similar to that of version C4. The frequency of the oscillations which is denoted by the Strouhal number in Table 4.1 for various versions is in good agreement with each other. Also given in Table 4.1 are the speedup comparisons of different versions with U1.

4.2 Three-dimensional Simulations

4.2.1 Steady State Test Case

After successfully testing the two-dimensional steady and unsteady cases, the attention was focused on three dimensional test cases. The test case chosen for steady simulations was three-dimensional wall driven cavity as shown in Fig. 4.8.

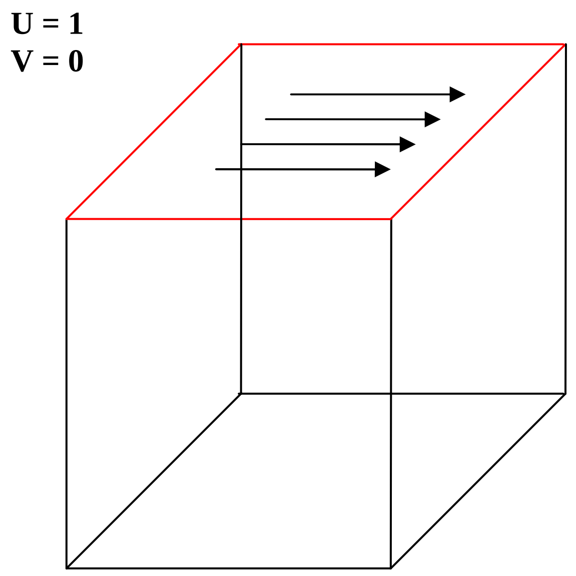


Figure 4.8 A schematic diagram showing the three-dimensional wall driven cavity

The computational grid consisted of 287496 cells so it was divided into four blocks for parallel computation as shown in Fig. 4.9. The simulations were done using version U1 and loop blocked version U3 with subblock size of 100 cells run till steady state convergence. The version U2 and U4 which incorporated the data structure changes encountered some problems during parallel computation so these versions were not used to conduct the simulations. The version U3 performed better than the version U1 and an improvement of 20% was observed in wall time. The results of these simulations are shown in the Fig. 4.10.

The contour plots of *u*, *v*, and *w*-velocity were taken at Z = 0.5 plane. The version U1 results are shown by the flood map and red dashed lines whereas the version U3 results are denoted by black dashed lines. It can be observed that the results are in good agreement with each other. The loop blocking yielded 20 % improvement in the walltime compared to the 35% in the two dimensional cases. This is because the Gauss-Siedel solver subroutines were the majority of calculations in the two-dimensional cases and their optimization lead to a large drop in walltime. In three-dimensional cases the Gauss-Siedel solver subroutines do not constitute the same proportion of calculation as compared to the two-dimensional case because of additional subroutines related to three-dimensional calculations, hence less overall improvement in the walltime. The residual convergence behavior, shown in Fig. 4.11, is similar to that of the two dimensional cases. The loop blocked version U3 took approximately 10% more iterations for equivalent convergence.

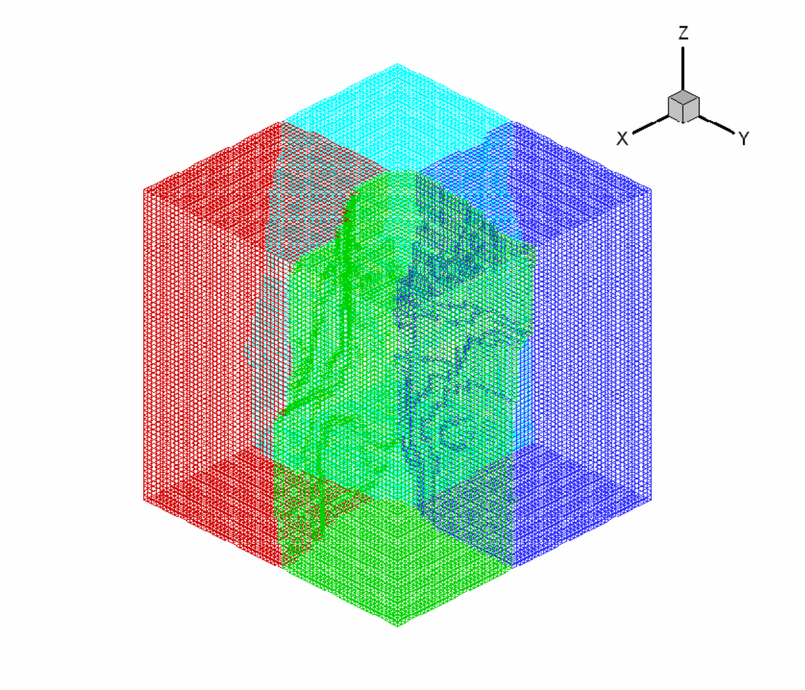


Figure 4.9 Computational grid for three-dimensional wall driven cavity (4 blocks)

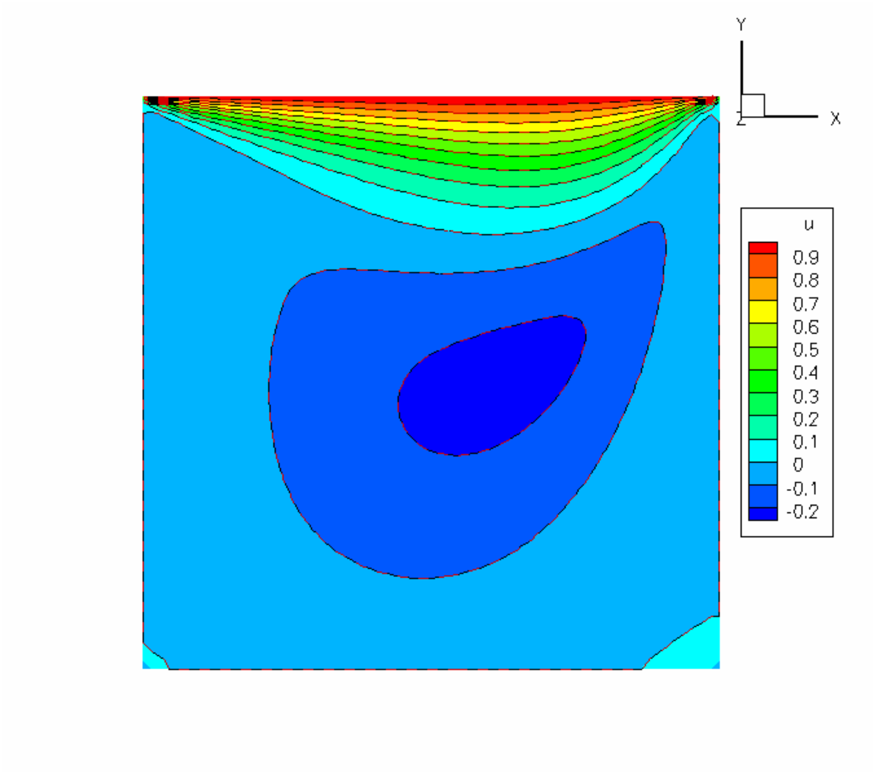


Figure 4.10 (a) *u*-velocity contour at Z = 0.5 plane

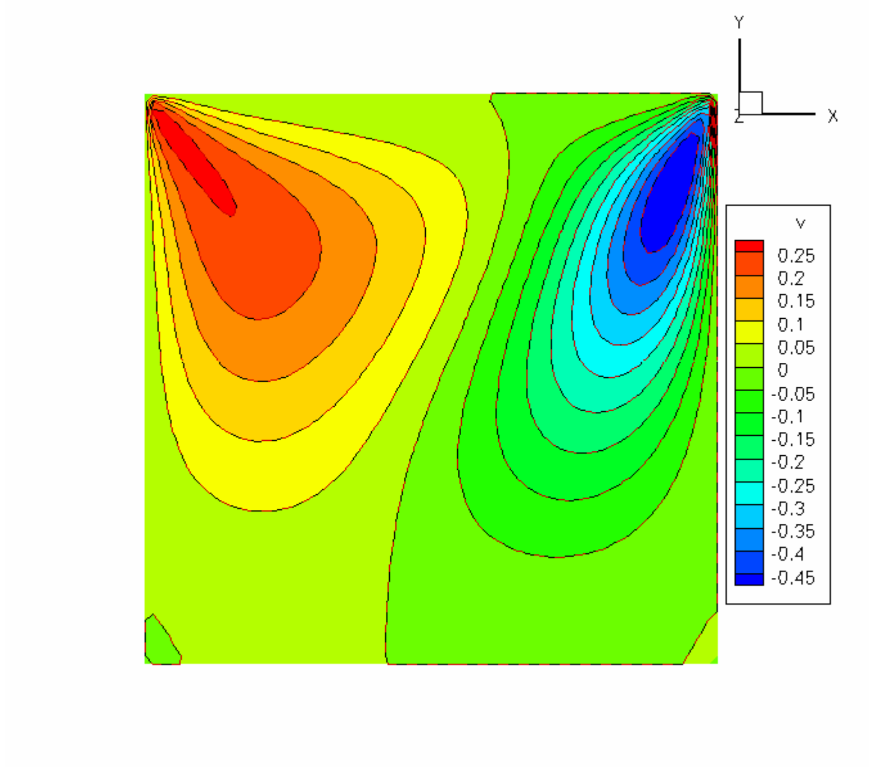


Figure 4.10 (b) *v*-velocity contour at Z = 0.5 plane

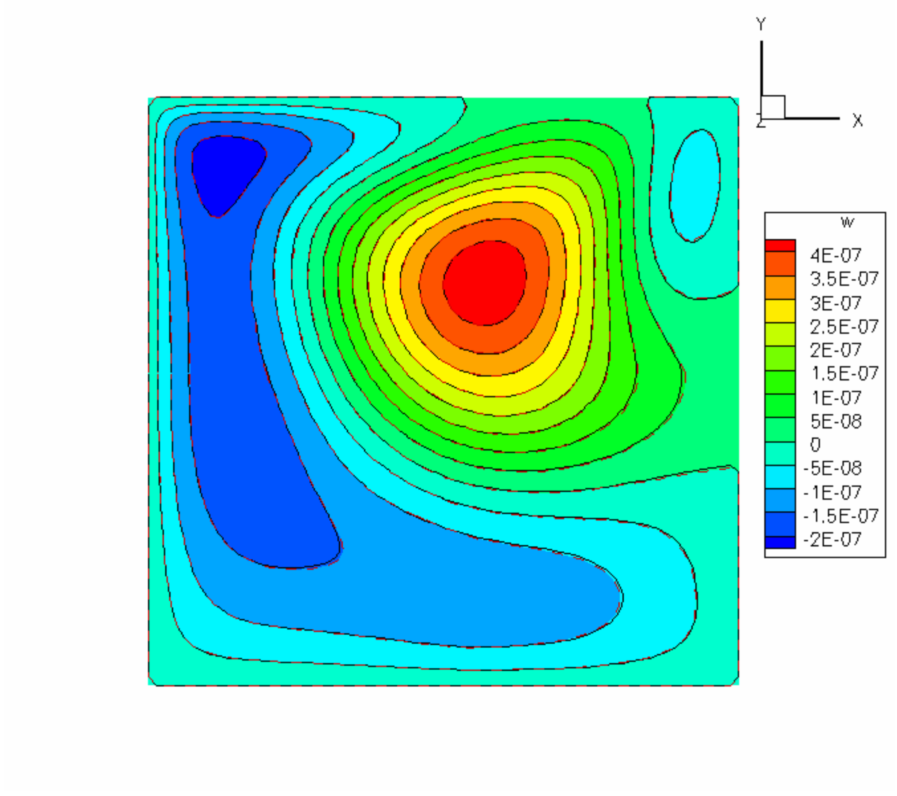


Figure 4.10 (c) *w*-velocity contour at Z = 0.5 plane

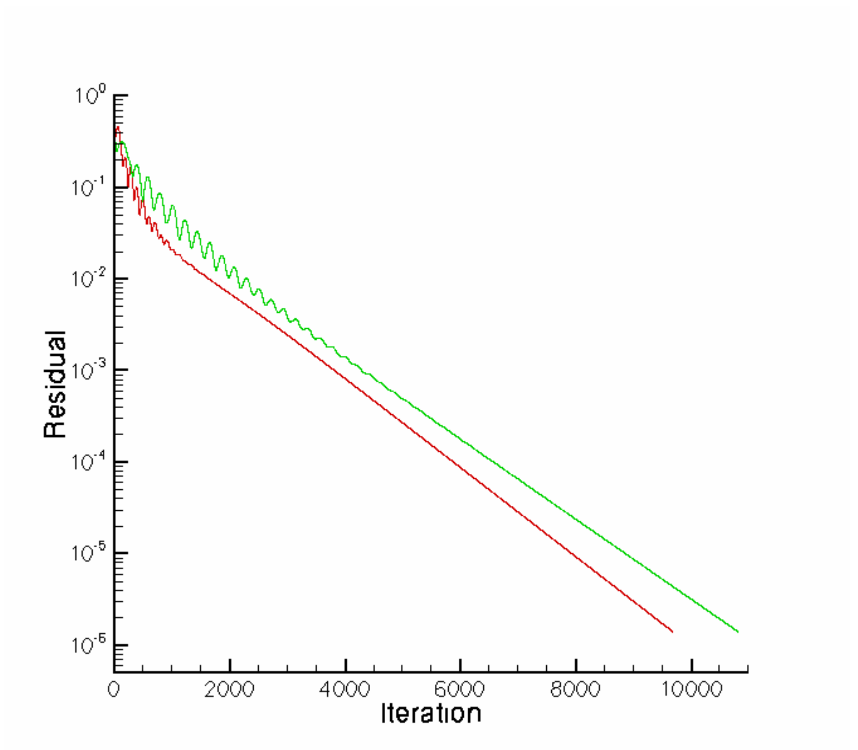


Figure 4.11 Residual convergence patterns for three-dimensional wall driven cavity

4.2.2 Unsteady Test Case

The flow over a cylinder was again chosen as the test case for unsteady calculations. A schematic diagram of the test case is shown in the Fig. 4.12. The grid used for simulation consisted of approximately 1.14 million points. It was divided into 32 parts for parallel computation. It is the same grid used by Hua et al. [13] in their computations. The computational grid is shown in Fig. 4.13. The simulations were conducted for Reynolds number of 200 and the lift and drag results are shown in Fig. 4.14. Table 4.2 shows the lift, drag, and Strouhal number comparison with other known experimental and numerical results.

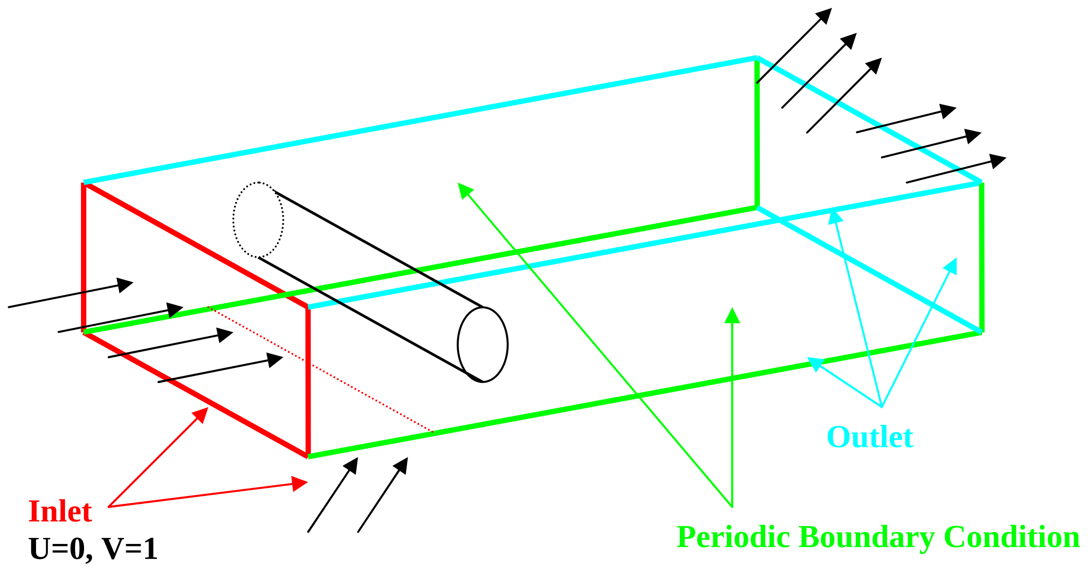


Figure 4.12 A schematic diagram for three-dimensional flow over a cylinder

The lift and drag are from both the versions are in good agreement with each other. Similar to the two-dimensional simulation results we see a slight shift in phase for both the lift and drag pattern but the magnitude is same for both the versions. A comparison shown in Table 4.2 reveals that the results from both unmodified U1 and loop blocked U3 version with subblock size of 10x10 are in good agreement with the results of Hua et al [13] and with those of other experimental results.

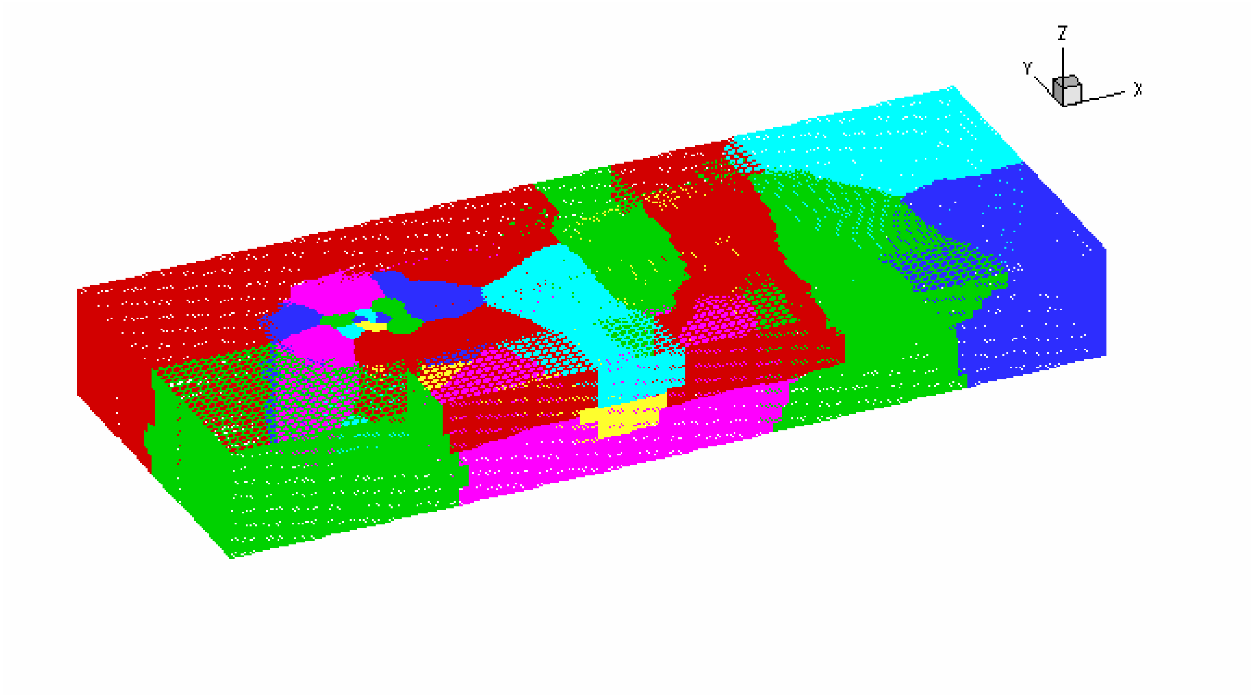


Figure 4.13 Three-dimensional grid for flow over a cylinder

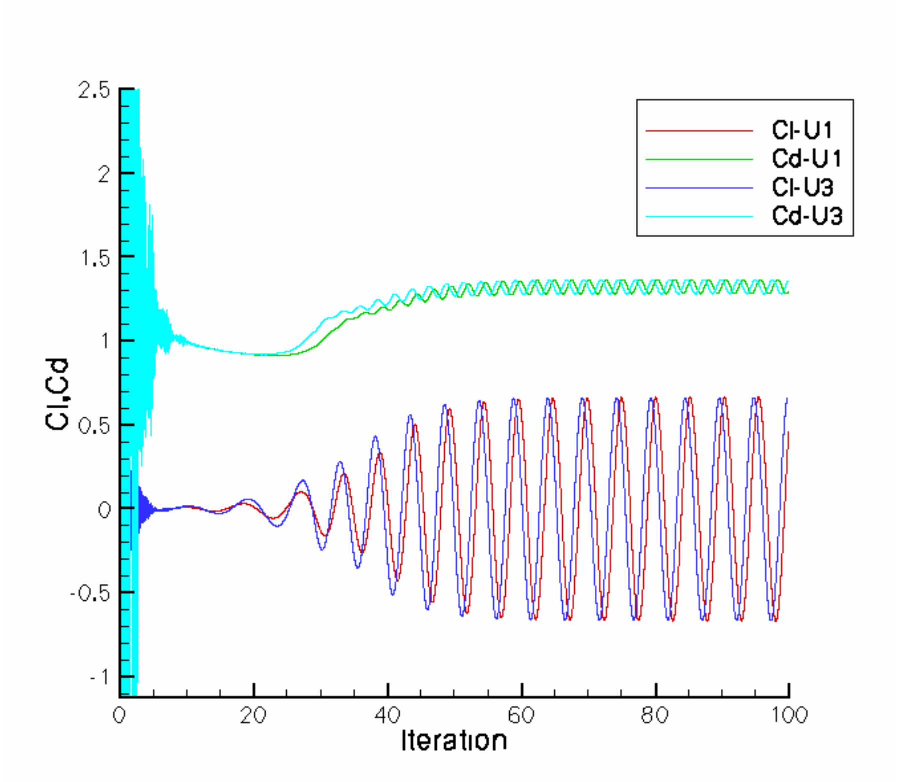


Figure 4.14 Lift and drag comparisons for three-dimensional simulations

Table 4.2 Comparison of lift, drag and Strouhal number for various 3D code simulations and experiments (* = Experimental)

Source **Re = 200**	**C** $_L$	**C** $_D$	S_t
U1	± 0.665	13255 ± 0.042	0.195
U4-sb10	± 0.664	13247 ± 0.042	0.195
Hua et al. [13]	± 0.664	1.324 ± 0.042	0.195
Henderson [21]			0.178
Roshko [22] *			0.19
Wille [23] *		1.3	
Williamson [24] *			0.197

4.3 Summary

This chapter presented the extension of the blocking techniques to time dependent simulations with successful results. Loop blocking was also extended to steady and unsteady three-dimensional simulations. The results obtained from the loop blocked version U3 of the code compared favorably to that of version U1. Loop blocking was established as a sound technique for enhancing code performance based on principle of improved cache behavior.

CHAPTER 5

TURBULENT RESULTS AND OTHER TECHNIQUES

This chapter presents the results of the turbulent simulations for steady and time dependent cases. All the tests conducted thus far were on KFC4. The results of tests performed to investigate the portability of the code modifications to different hardware architecture are discussed. The results of parallel tests performed to test the scalability of the code are presented

5.1 Steady Turbulent Test Case

The test case chosen was again the wall-driven cavity for the reasons mentioned earlier. However, the grid used, shown in Fig. 5.1, was slightly different from the one used for testing steady laminar cases. The grid is not uniform; instead it is finer at all the boundaries and coarser towards the center. This is done to ensure so all the turbulent features near the wall can be captured effectively. The Reynolds number used was 500000. The grid used is 256x256. The grid was divided into eight blocks and parallel computations were done. The tests were conducted on KFC5.

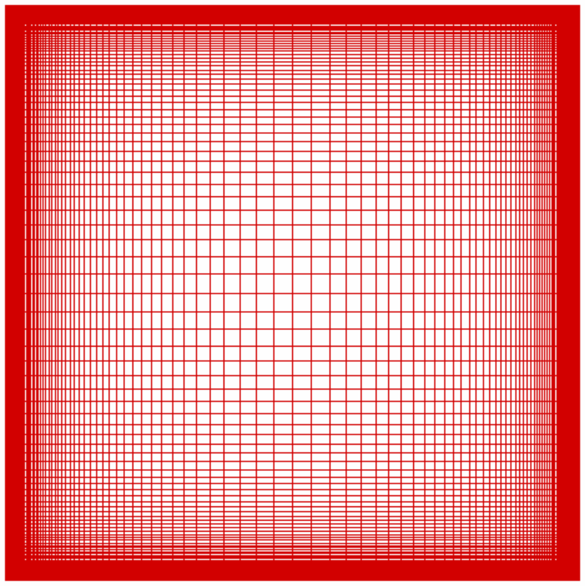


Figure 5.1 Grid used for testing steady turbulent case

5.1.1 Steady State Turbulent Simulation Results

The turbulent simulations are very sensitive to the instabilities or discontinuities that are generated in the system. It was observed that during the simulations minor instabilities had the potential to grow up abruptly and cause the calculations to diverge. The subblock size of 10x10 was used in Gauss-Siedel solver subroutines for velocity and pressure, whereas a subblock size of 30x30 was used in corresponding turbulent subroutine. It is highly recommended that higher subblock should be used in all the subroutines because they have fewer boundaries associated with them. The results of the turbulent simulations of wall driven cavity are shown in Fig. 5.2. An improvement of about 21% was observed in the walltime. The overlapped *u*-velocity contour plots for version T1 and T3 are shown in Fig. 5.2(a). The flood contour and the red lines correspond to the version T1. The black dashed lines correspond to version T3. It can be observed that the results are in good agreement with each other. Fig 5.2(b) shows the *v*-velocity contours which are also consistent with each other. It is to be noted that the loop blocked version T3 is capable of predicting the results of steady turbulent simulations with high degree accuracy.

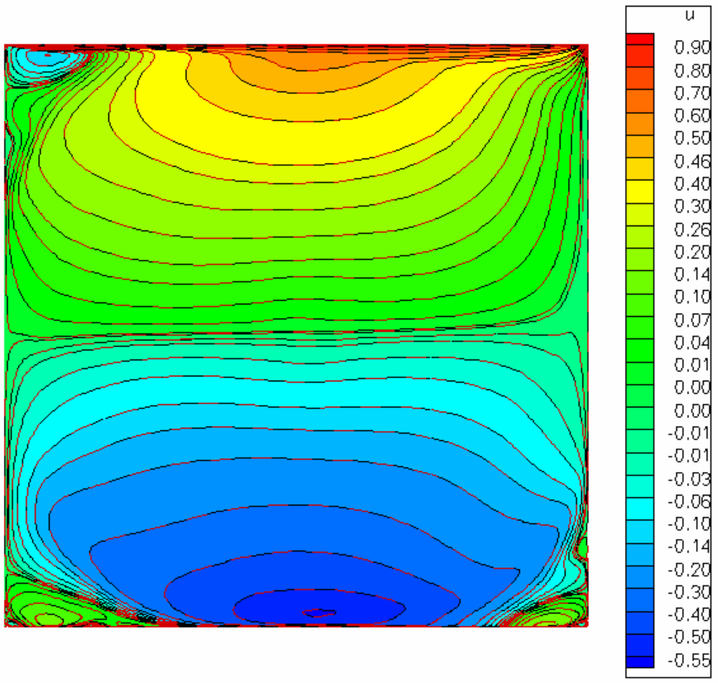


Figure 5.2(a) *u*-velocity contour plots for versions T1 (red) and T3 (black)

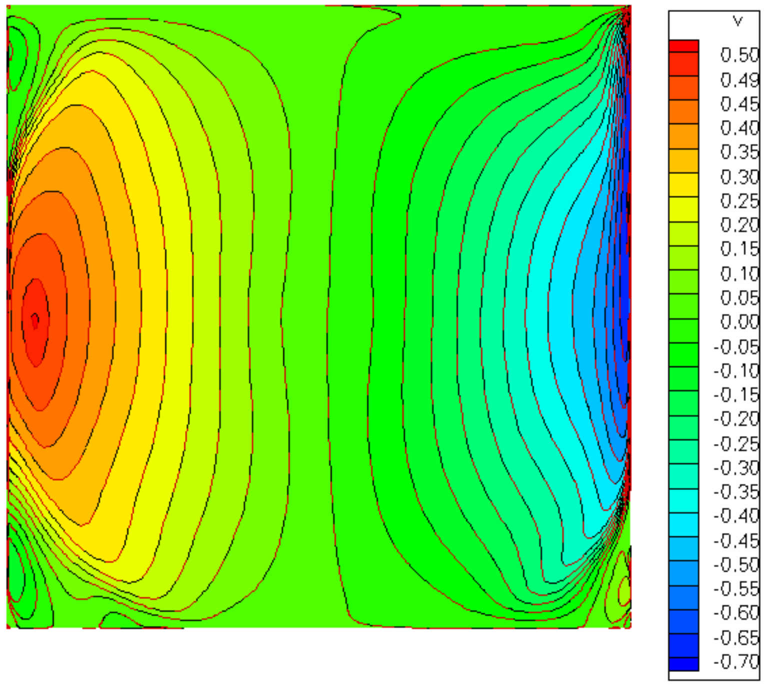


Figure 5.2(b) *v*-velocity contour plots for T1 (red) and T3 (black)

5.2 Time Dependent Turbulent Simulations

Active flow control methodology is one of the growing areas in the aerodynamics research. One of the means to do it is to superimpose forced oscillations with mean flow to obtain the desired results for e.g. suction or blowing using actuators. Synthetic jet demonstrates the capabilities to fulfill the requirements of these actuators. Hence it was chosen as a test case for unsteady turbulent simulations. This test case is the case 1 used in the CFDVAL 2004 workshop [24] which was organized to test the capabilities of various CFD codes. A numerical simulation of a synthetic jet into quiescent air was performed using the two different versions of UNCLE. A schematic diagram describing the geometry is shown in Fig. 5.3.

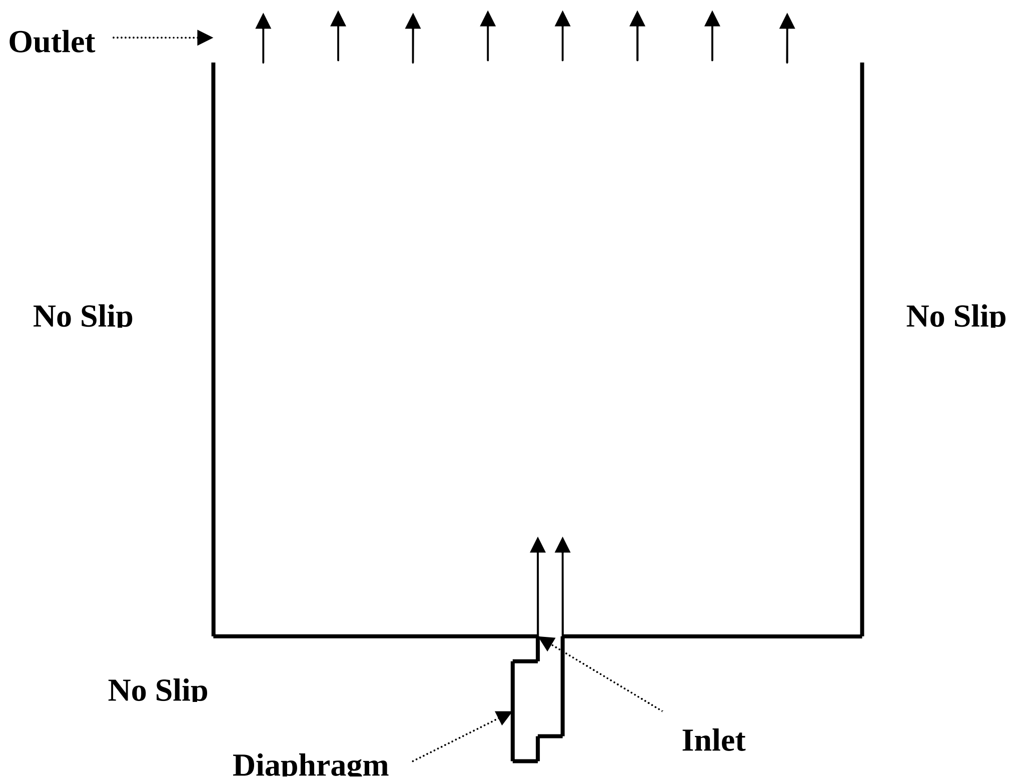


Figure 5.3 Schematic diagram showing the synthetic jet

The inlet velocity was defined by the displacement history of the center of the diaphragm. This in turn was provided by a periodic boundary condition defined by:

$$\frac{v}{V_\infty} = a + b.\cos(c.t + d)$$

where

$a = -0.0044205499$

$b = 0.026689782$

$c = 0.119707081$

$d == 4.0776928$

V_∞ is the maximum inlet velocity and v is the inlet velocity at dimensionless time t. The parameters a, b, c and d are dimensionless. The outflow boundary condition at the top satisfies the continuity equations. For all the other walls no-slip condition is imposed as the wall boundary condition. The computational grid is shown in Fig. 5.4

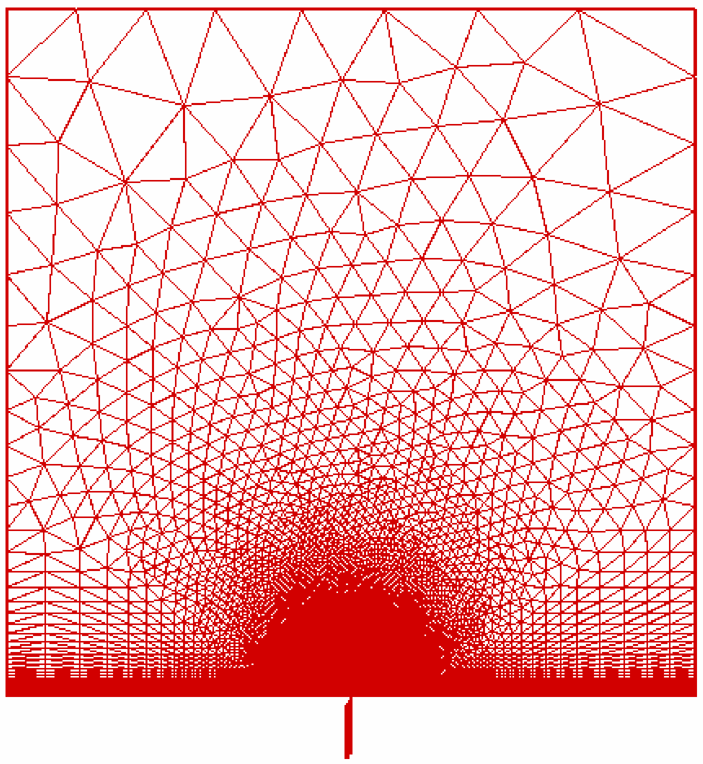


Figure 5.4 Grid used for synthetic jet test case

5.2.1 Results for Simulation of Synthetic Jet

The Reynolds number used was 2453.72 and the value chosen to non-dimesionalize the velocity was 30 m/sec. The non-dimesionalized time period for one cycle and the time step were 52.488 and 0.009 respectively. The unmodified version T1 and the version with modified data structure T2 were used to conduct these simulations. The tests were for two complete cycle and the final results are shown in Fig. 5.5. The contour flood and the red dashed lines represent the results from version T1 and the green dashed lines represent the version T2. Fig 5.5(a) shows the *u*-velocity contour plot and Fig. 5.5(b) shows the *v*-velocity contour plots for the same. We can see that the results are in good agreement with each other.

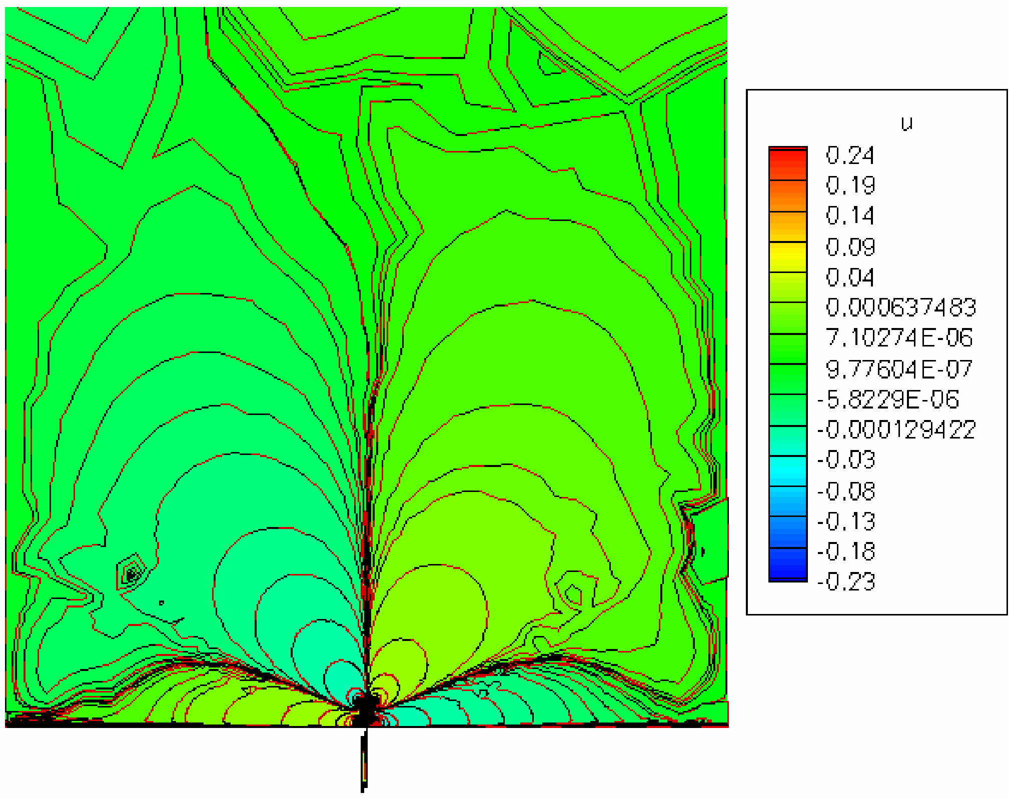


Figure 5.5(a) *u*-velocity contour of synthetic jet for versions T1 (red) and T2 (green)

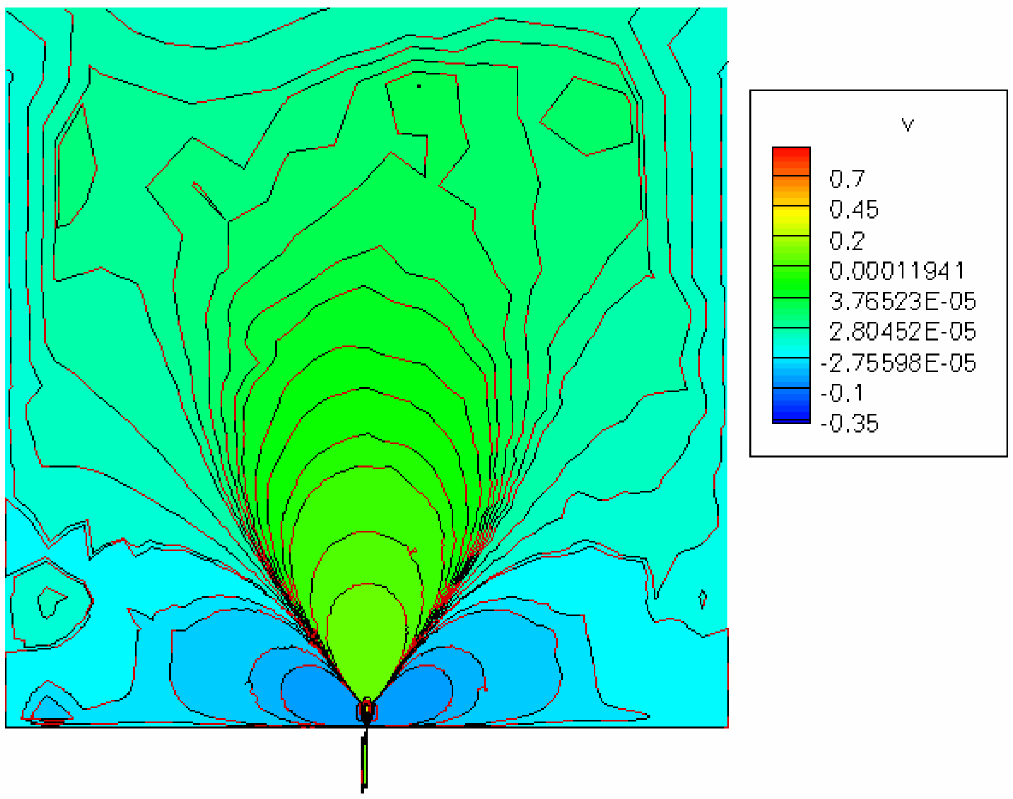


Figure 5.5(b) *v*-velocity contour of synthetic jet for versions T1 (red) and T2 (green)

5.3 External Blocking

The improvements achieved by the loop blocking of the Gauss-Siedel solver subroutines encouraged us to try out the new technique of external blocking. In the loop locking the blocking was applied to only a few critical subroutines, which ensured that the computation was done on small cache friendly blocks at a time instead of the full domain. The other subroutines of code were still solved on the whole domain at a time. To investigate the potential benefit of solving the whole code in small blocks instead of just a few subroutines the external blocking techniques was tested. A schematic diagram showing the external blocking mechanism is shown in Fig. 5.6.

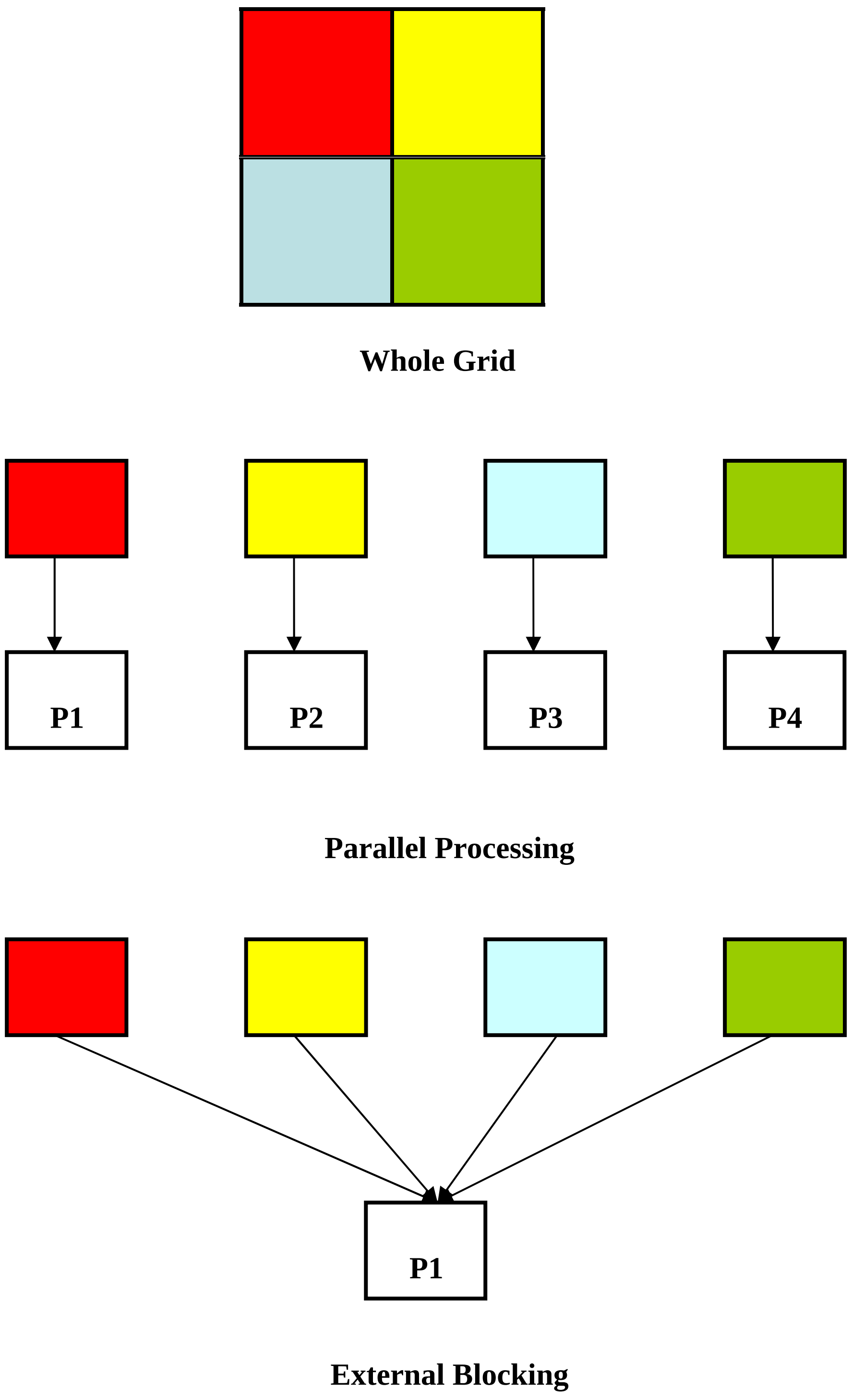


Figure 5.6 Schematic of external blocking

To perform parallel processing the domain is divided in to small sub-domains or blocks and then each of these small sub-domains is sent to a different processor for computations. So instead of solving the small sub-domains on different processors, if we could send them all to the same processor they would be solved a small block at a time. Thus the whole code not just a few subroutines could potentially benefit from the blocking.

5.3.1 External Blocking Results

The cache friendly block size which was determined earlier (and presented in an earlier chapter) was 900 cells. So the whole domain was divided into blocks which were approximately 900 cells for all the grid sizes. All the blocks were then solved on the same processor. These tests were first done for unmodified version U1. The walltime results for these simulations are shown in Fig. 5.7. We can observe that for smaller overall grids external blocking did not show any improvements and showed an increase in walltime instead. As we moved towards the grids of larger sizes we see considerable improvements in the wall time but this improvement was not comparable to the improvements achieved by the internal loop blocking.

The external blocking was also applied to the loop blocked version U3 of the code to investigate if the unfavorable performance of the external blocking for the smaller grids could be rectified and to study the potential improvement by combining both the blocking techniques. The results of these simulations are also shown in Fig. 5.7. We observe the same deteriorating performance for the external blocking for smaller grid sizes. For all the larger grids there was a notable improvement in the walltime. The results of the simulations from both of the codes show that the external blocking yield favorable improvements just for the larger grids compared to the improvements of the loop blocking where the improvements were universal i.e. for grids of all sizes.

A *gprof* profile of the code was done to investigate the reason for the deteriorating performance on the smaller grids. The walltime performances for the grid of 70x70 are identical in both externally blocked and unblocked versions of U1 and U3. This same behavior is reflected in the profiling data shown in the Fig. 5.8(a). The performance tends to deteriorate for the grids lying between 100x100 and 200x200. The profiling data for 150x150 grid is shown in the Fig. 5.8(b). For version U1 we observe a slight increase in the time taken by the Gauss-Siedel solver subroutines when the external subblocking was applied. Also a marginal increase in the '*other*' subroutines was observed. This explains the cause of detrimental performance at this grid size. On the contrary in version U3 we observe a high increase in the time taken by the subroutines '*cal_vel*', '*continuity*' and '*gradients*'. Only the Gauss-Siedel solver routines seemed to be benefited by the external blocking. The minimal time consumed by them earlier was further reduced by the application of external blocking. The grid size of 350x350 saw marginal improvements in most of the subroutines translating into overall better performance of the code, as seen in Fig. 5.8(c), when external blocking was applied. This behavior was observed for both versions U1 and U3 hence we had enhanced performance for this grid size in both the cases.

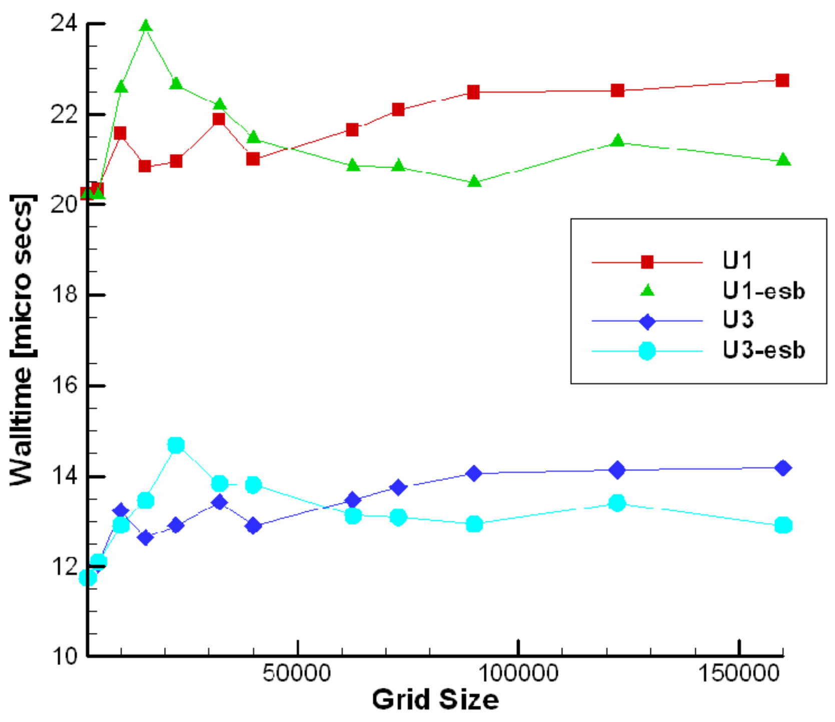
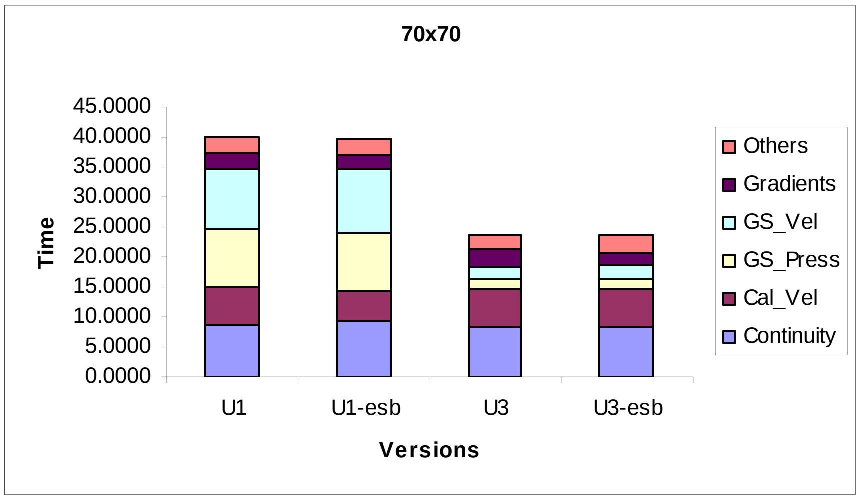
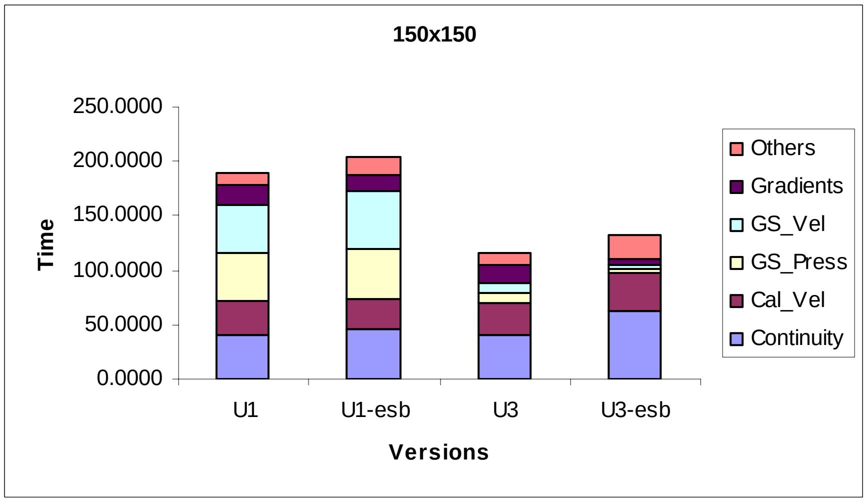


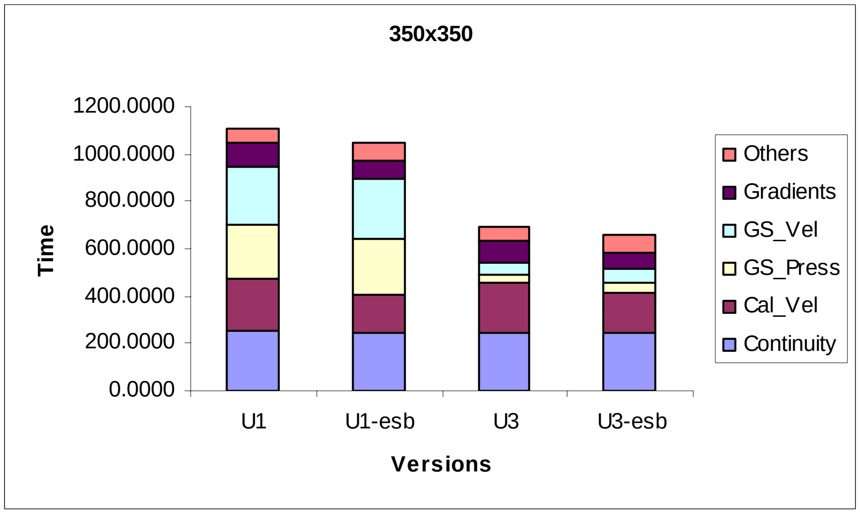
Figure 5.7 Walltime results for external blocking



(a)



(b)



(c)

Figure 5.8 Gprof results for grid sizes of (a) 70x70 (b) 150x150 (c) 350x350

5.4 Tests on Different Hardware Architectures

Once the successful simulations were performed for the steady and unsteady test cases on KFC4, the attention was now focused on a different machine KFC5. The idea was to investigate the effect of hardware architecture on the code performance. KFC5 has a higher data bandwidth and a faster processor with 64 bit architecture as opposed to the 32 bit architecture in KFC4. The cache and RAM sizes of 512 KB and 512 MB per node respectively are same for both the clusters.

Various versions of UNCLE were run on both KFC4 and KFC5 and the walltimes from the runs are shown in Fig. 5.9 and 5.10. A plot showing percentage gains on KFC5 over KFC4 for various grid sizes is shown in the Fig. 5.11. A quick look at the results show that the KFC5 runs are completed in about 45-50 % of the time taken by the KFC4 runs. Also the walltime profiles for all the versions are much smoother on KFC5. This enhanced cache performance is due to better hardware architecture. The data was transferred faster along with the faster calculations.

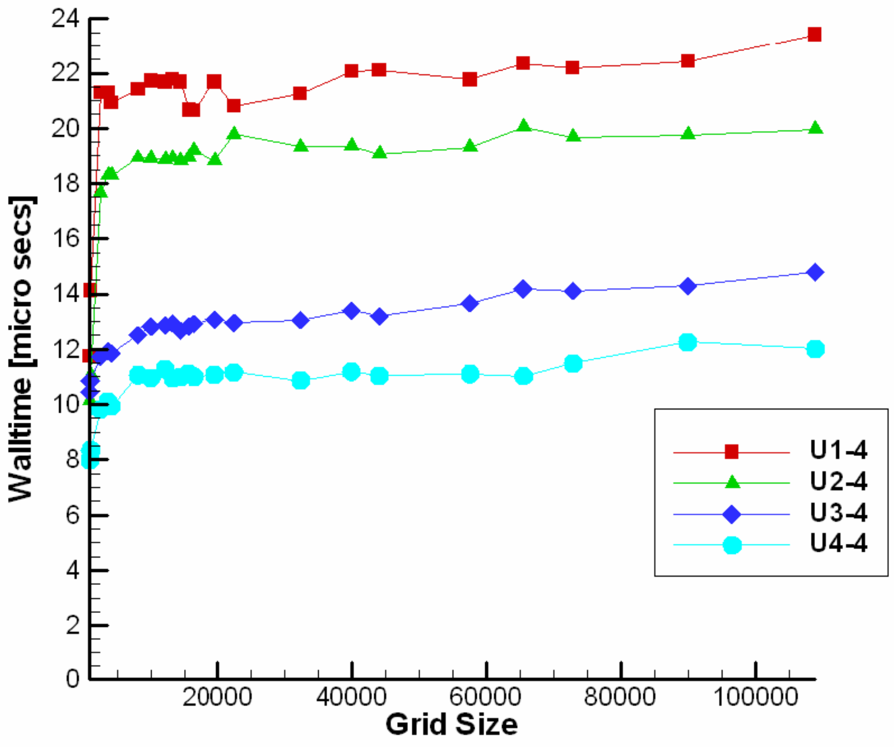


Figure 5.9 Walltime comparisons for various code versions on KFC4

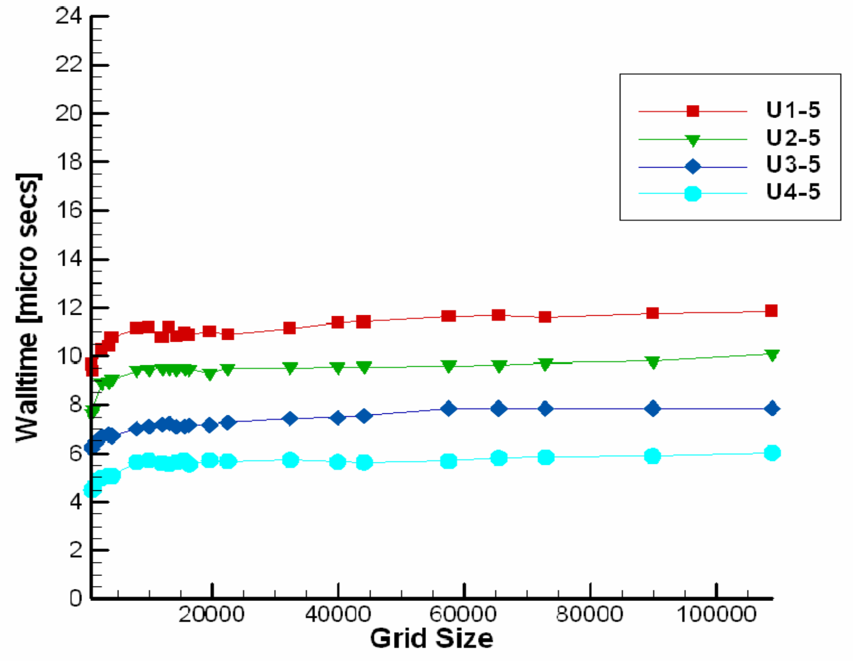


Figure 5.10 Walltime comparisons for various code versions on KFC5

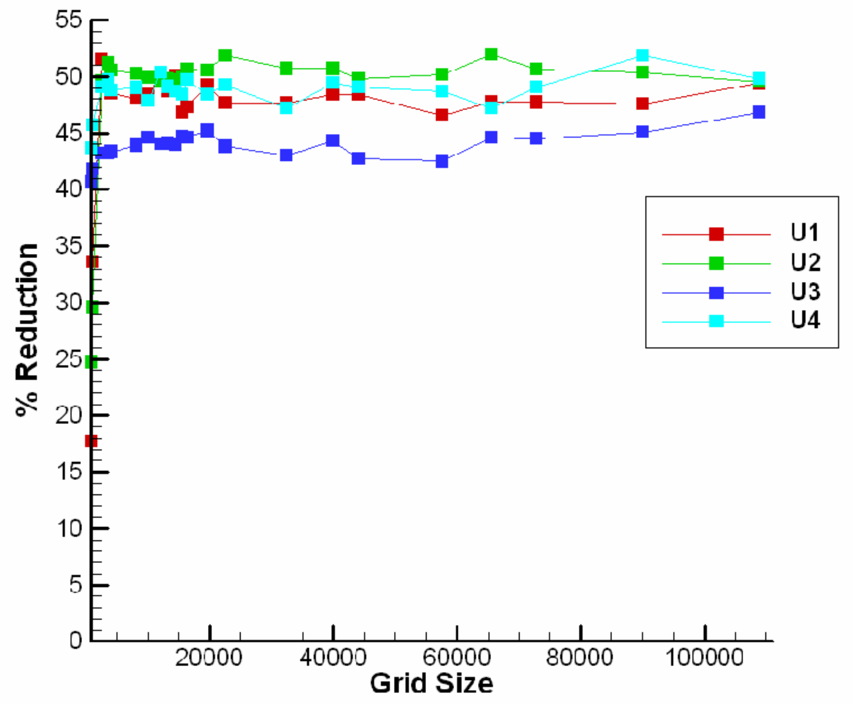


Figure 5.11 Percentage gains on KFC5 over KFC4 for various grid sizes

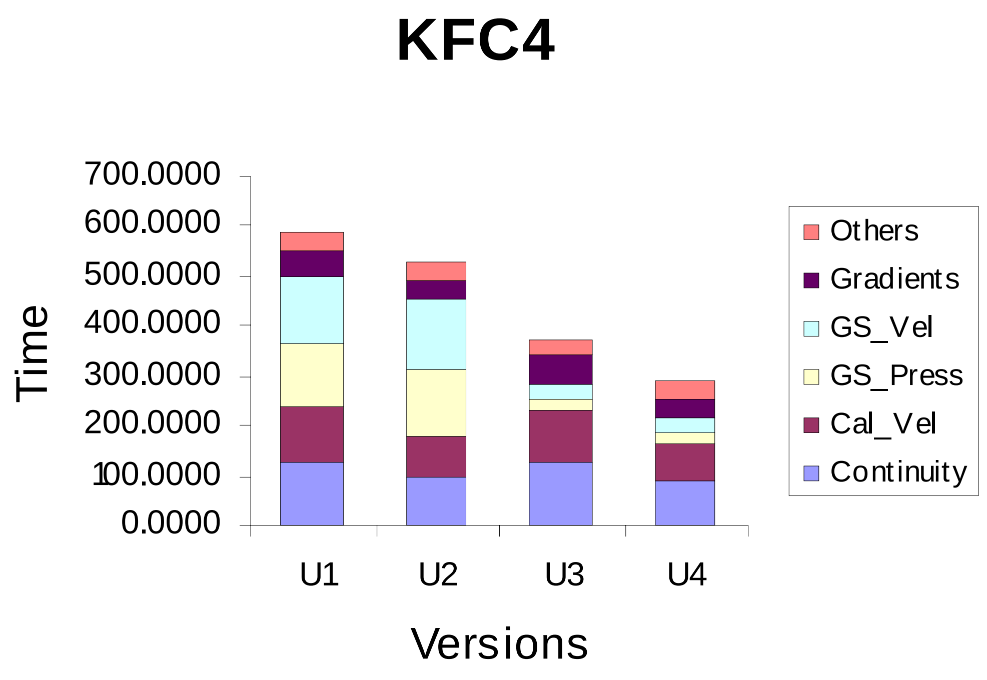


Figure 5.12(a) Gprof data for 256x256 grid on KFC4

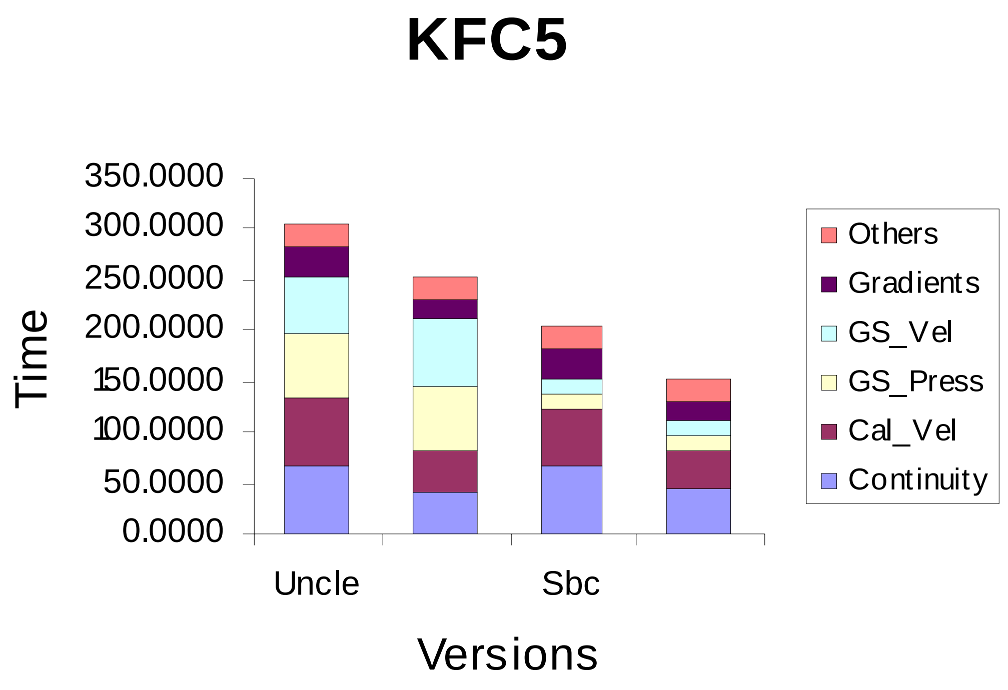


Figure 5.12(b) Gprof data for 256x256 grid on KFC5

It can be seen from Fig 5.11 that versions U2 and U4 which incorporate the data structure changes were slightly more benefited by the new architecture. Also the gap between the times taken by U2 and U3 versions has been reduced on the KFC5 platform. Fig. 5.12 and Table 5.1 show the *grof* profiles for a 256x256 grid on KFC4 and KFC5. The time taken in computation is shown in the Fig. 5.12 and the percentage time taken in computation is shown in the Table 5.1.It can be seen that there is a proportionate reduction in the times taken by various subroutines when the code was ported from KFC4 to KFC5. The subroutines that were optimized performed slightly better than the non-optimized subroutines. This can be observed in the slight increase in time taken by the *'others'* subroutines.

Table 5.1(a) Gprof profile data for 256x256 grid on KFC4

	Percentage time taken by subroutines			
KFC4	**U1**	**U2**	**U3**	**U4**
Continuity	22.15	18.4	34.08	30.46
Cal_Vel	18.49	15.66	28.96	26.14
GS_Press	21.85	25.35	6.06	8.8
GS_Vel	22.6	26.47	7.62	10.05
Gradients	9.32	7.58	14.59	13.42
Others	5.59	6.54	8.69	11.13

Table 5.1(b) Gprof profile data for 256x256 grid on KFC5

	Percentage time taken by subroutines			
KFC5	**U1**	**U2**	**U3**	**U4**
Continuity	22.1	16.29	32.77	28.34
Cal_Vel	21.61	15.66	27.21	25.75
GS_Press	21.23	25.62	6.5	8.45
GS_Vel	17.78	25.97	7.73	10.12
Gradients	9.75	7.46	14.26	12.5
Others	7.53	9	11.53	14.84

5.5 Parallel Tests

The primary goal of this research was the single node optimization of the code. A parallel test was also done to investigate the scalability of these code modifications. The parallel tests were done using the lid driven cavity case. The grid size used was 256x256.

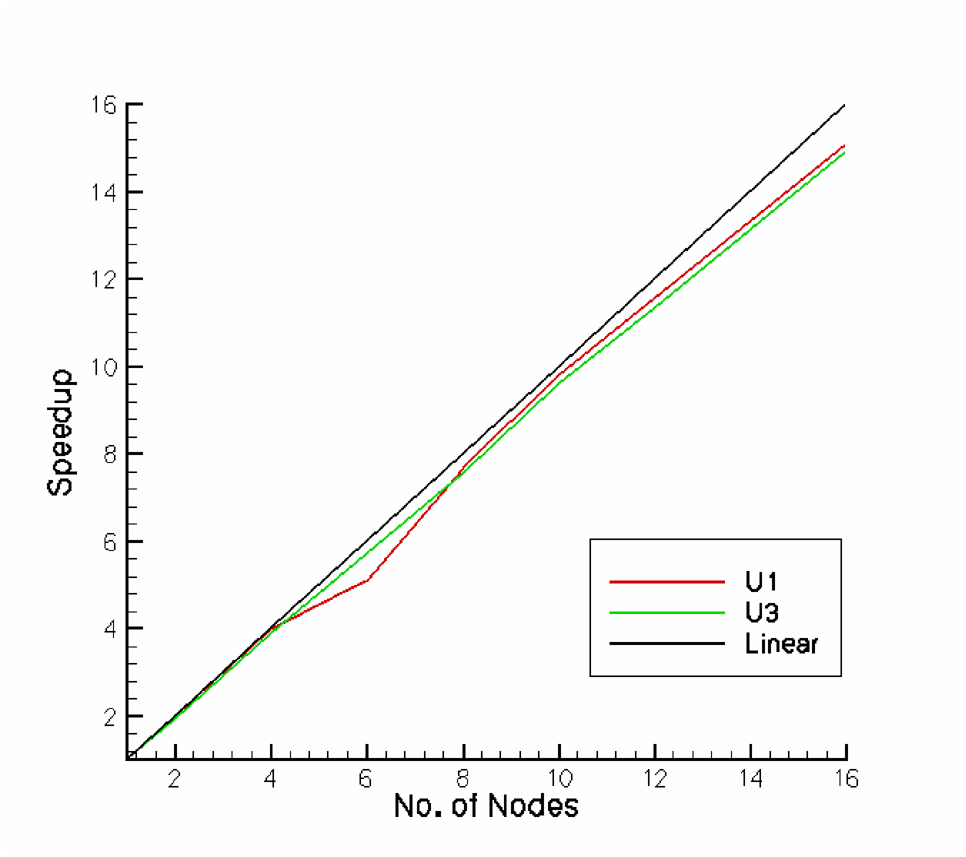


Figure 5.13 Parallel speedup comparison for versions U1 and U3

The tests were done on up to 16 processors and the speedup results are shown in Fig. 5.13. The ideal linear speedup line is shown in black. It can be observed that both the original and loop blocked versions of the code are slightly sublinear which is what we expect due to some time losses associated with the internode communication.The original version had a diminished performance when six nodes were used for calculations whereas this effect was not noticed in the loop blocked version which seemed to be smooth in performance. However, the overall performance of the original version U1 was slightly better than the loop blocked version U3 for all other cases. The versions U2 and U4 could not be tested due to the problems encountered by the new data structure in parallel computation.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

An initial optimization of the unstructured, 3-D code UNCLE was completed successfully. The technique to partition and solve the domain in a cache friendly manner was tested with positive results. Various techniques such as array merging, loop blocking, and external blocking were tested which reduced the cache misses and enhanced the code performance. A few of the data structures were modified to optimize the data access pattern with successful results. External subblocking was also applied and compared to the loop blocking technique.

The preliminary work involved testing the space filling curves to inherently subblock the whole grid in a cache friendly manner. This technique showed improvements of about 5% for large grid sizes but remained ineffective for smaller grids. Since an approach aimed at improving the whole code performance at the same time did not yield overwhelming results, the attention was then focused on the critical subroutines. The initial tests included simulation of a steady, laminar lid driven cavity flow. The Gauss-Siedel solver subroutines involved repetitive calculations over the same data set and consumed about half of the computation time in these cases. Hence, they became a primary target for application of loop blocking. The *face* data structure was also modified in some of the major subroutines to investigate potential improvements through data structure modification.

In the two-dimensional steady laminar test case the loop blocking with subblocks of various sizes were tested. The subblock of 900 cells was found to fit into the L2 cache. The subblock size of 100 cells gave the best walltime results which was a 35% reduction in walltime for all grid sizes. The walltimes for this case were slightly less than the subblock of 900 cells. As the size of subblock was increased further from 100 cells, the walltimes also increased respectively. The data structure modifications showed 15% improvements over the original version. These changes compounded together yielded 50% improvement in runtime. The improvements in both the cases were largely cache driven. Both the methods significantly reduced the L1 and L2 cache misses which translated into faster code execution. Another technique called loop unrolling was tested but it did not show significant improvements in the walltimes.

While the implementation of a new data structure did not change the calculations, the loop blocking slightly altered the way the calculations were done. The implementation of loop blocking introduced new subblock boundaries which gave rise to small discontinuities whose effect was observed in the residual convergence patterns. The loop blocked versions tend to converge 10% more slowly than the original versions. But the improvements gained by the aforementioned techniques negate this small set back and yield large overall improvements. The accuracy tests conducted showed that the results were in good agreement with each other, thus establishing the modifications as a viable performance enhancing technique for CFD codes. Also, the steady laminar simulations were subblock insensitive. All the subblock sizes, large or small, gave the same final solution with few noticeable instabilities during the process. The only effect was seen in the residual pattern.

The walltime results of the loop blocked versions showed some interesting results. The walltime per node for loop blocked version while solving the whole domain was approximately similar to walltime per node while a domain of the size of subblock was being solved. The code and cache behaved as if the domain of size of the subblock was being solved instead of the whole domain. This test was conducted for various subblock sizes and this behavior was extant in all of them. So these results showed that the performance of a loop blocked code could be approximately predicted by conducting simple tests on small subblocks instead of running the code with the whole grid. The effective subblock could be chosen pertaining to ones need, if the subblocked subroutines took up a large percentage of time initially before subblocking, as in this case.

These results for time dependent simulations of flow over a cylinder showed performance improvements ranging from 55-60%. This slight improvement over the steady state versions is due to the increased number of inner iterations in the unsteady simulations. The three-dimensional simulations were also conducted successfully using the cavity and the cylinder test cases for steady and unsteady calculations respectively and they showed less overall improvement compared to the two-dimensional version. The results are summarized in Table 6.1. This happened because the targeted subroutines did not form a major portion of the code due to the introduction of extra three-dimensional and related calculations. The accuracy tests done for three-dimensional versions also showed that the results were in good agreement with each other.

After thorough analyses of laminar results were done, the tests were conducted for the turbulent cases. The steady state tests were done using the lid-driven cavity and the results of the loop blocked version were consistent with the ones achieved with the original version. Unlike the laminar simulations, the turbulent simulations were very sensitive to the size of the subblock. The smaller subblocks gave rise to numerous discontinuities and instabilities which caused a problem in convergence. So it is recommended that the subblock size of at least 900 cells be used for loop blocking in the case of UNCLE for turbulent simulations. Simulation of a synthetic jet in quiescent air was done with the code which incorporated modified data structures and the results compared successfully with the original version.

The technique of external blocking was also tested for the laminar cases. The external blocking effectively subblocks the whole grid instead of few specific subroutines. The results contrary to the expectations did not compare well with those of loop blocking. The performance of externally blocked codes deteriorated for smaller and moderate grid sizes and improvement of 5-10% was observed for the larger grid sizes. The cause for the deteriorating performances in small and moderate grid sizes was the excessive time spent in the Gauss-Siedel solvers. The external blocking in conjunction with loop blocking was applied to various grid sizes and performances similar to the external subblocking were observed in this case. One would reckon that the combination of these two schemes would provide large improvements but it was not observed in this case.

Tests were conducted in parallel to check the scalability of the code. The results showed that the loop blocked scaled successfully on multiple nodes and achieved slightly sub-linear speedup. The codes were also ported to faster and more modern hardware architecture. KFC5 which had 64 bit architecture compared to 32 bit of KFC4 and a high data transfer bandwidth showed large improvements in the performance of the code. The walltime performances of the various versions were reduced by 45-50%.

Although the data modification seemed to be fruitful, it encountered problems when executed on a parallel platform. The parallel computations require introduction of virtual boundaries when the grid is divided into several parts. The data structure allocation did not comprehensively address this issue and more investigation is required in this area to rectify this problem. The time dependent cases with the turbulent calculations were very sensitive to loop blocking. The small discontinuities generated in the system made the calculations very unstable. The instabilities grew randomly at a certain point in the calculation and caused the computations to diverge. The idea of loop blocking in turbulent simulations requires further investigation to find the cause of such behavior.

Despite of the few problems encountered, the optimization overall was largely successful. The techniques used were thoroughly investigated and established as performance enhancing methods for two/three-dimensional laminar simulations using unstructured

CFD codes. These techniques were proven successful for turbulent simulation albeit the time dependent ones. The critical hardware parameters were identified and tuned to obtain aforementioned performance enhancement. The summary of the improvements is shown in the Table 6.1.

Table 6.1 Summary of improvements achieved

Case	**Best Version/Method**	**% Improvement**
Large Grids	Space-Filling Curve	~5%
2D Steady Laminar	U4	~50%
2D Unsteady Laminar	U4	50-60%
3D Steady Laminar	U3	10%
3D Unsteady Laminar	U3	~23%
2D Steady Turbulent	T5	~22%
2D Unsteady Turbulent	T2	~6%
Large Grids	External Blocking	~10%

6.2 Future Work

The improvements in the performance were largely due to gains in the Gauss-Siedel solver subroutines which were targeted by the loop blocking. The loop blocking technique drastically reduced the time taken by these Gauss-Siedel solvers to the minimum possible, which leave little scope of improvement in these subroutines. A lot of other subroutines such as *cal_vel*, *continuity*, *gradients*, and *set_bc* need further attention as they would be the next target of the optimization work. The issue of sensitivity of the turbulent simulations with regard to subblocking requires thorough investigation.

A major potential lies in further modification of the remaining data structure. In this study only the *face* data structure was modified in few subroutines and it gave an encouraging 15% improvement. It was comprehensively shown that the multidimensional arrays are more cache friendly way of storing the data when compared to the pointers. The other dominant *cell* and *vertex* data structures are prime candidates for restructuring. The restructuring of the *cell* data structure can further reduce the time consumed by Gauss-Siedel solver subroutines. These modifications of *cell* and *vertex* data structures were beyond the scope of this research because it involved major restructuring of the code. This definitely should be a target for next level of optimization.

The cause of deteriorating performance of the external blocking for small and medium size grids needs to be investigated. Also the space filling curve based grid numbering though mildly successful in the two-dimensional holds a larger scope of improvement for the three-dimensional version and should be one of the focus of next stage of optimization.

APPENDIX

A1. Loop Blocking

The original and the modified *i_solver_gs_vel* and the loop blocked version are presented here. Similar changes were applied to Gauss-Siedel solver for pressure *i_solver_gs_p* and Gauss-Siedel solver for turbulence *i_solver_gs_ke*. The original code has three different loops with the same iteration space for solving the velocities *u*,*v*,*w*. In the loop blocked version they have been merged using some extra local variables, so that they can be solved simultaneously. This facilitates better data and spatial locality in these subroutines resulting in fewer cache misses and enhanced performances.

A1.1 ORIGINAL SUBROUTINE

```
SUBROUTINE i_solver_gs_velocity (cell, node, coef, ncell, nnode, BLOCK,
nblock, num_iter)
      INTEGER :: icell, ncell, iter, num_iter, nblock, nnode, iblock
      TYPE (cells), DIMENSION (:) :: cell
      TYPE (points), DIMENSION (:) :: node
      TYPE (coeffs), DIMENSION (:) :: coef
      TYPE (block_t), DIMENSION (:) :: block
      real (high), dimension(nnode) :: dt
      REAL (high) :: res, summ, dt1, dt2, dt3, dt4, dt5, dt6, summ1,
      errors=1.e-4_high
      dt = 0._high
      DO iter = 1, num_iter
        summ = 0._high
        do iblock=1,nblock
          IF(INDEX(BLOCK(iblock)%title,'soli')==0)then
          do icell=block(iblock)%n_begin,block(iblock)%n_end
            IF (cell(icell)%num_faces == 3) THEN
              dt1 = dt(cell(icell)%surround_p%i(1))
              dt2 = dt(cell(icell)%surround_p%i(2))
              dt3 = dt(cell(icell)%surround_p%i(3))
              res = coef(icell)%rhs_u + coef(icell)%an_u(1) * dt1 + &
                  & coef(icell)%an_u(2)* dt2 + coef(icell)%an_u(3)* dt3
            ELSE IF (cell(icell)%num_faces == 4) THEN
              dt1 = dt(cell(icell)%surround_p%i(1))
              dt2 = dt(cell(icell)%surround_p%i(2))
              dt3 = dt(cell(icell)%surround_p%i(3))
          dt4 = dt(cell(icell)%surround_p%i(4))
          res = coef(icell)%rhs_u + coef(icell)%an_u(1) * dt1 + &
              coef(icell)%an_u(2) * dt2 + coef(icell)%an_u(3)*dt3 &
              &+ coef(icell)%an_u(4) * dt4
        ELSE IF (cell(icell)%num_faces == 6) THEN
          dt1 = dt(cell(icell)%surround_p%i(1))
          dt2 = dt(cell(icell)%surround_p%i(2))
          dt3 = dt(cell(icell)%surround_p%i(3))
          dt4 = dt(cell(icell)%surround_p%i(4))
          dt5 = dt(cell(icell)%surround_p%i(5))
          dt6 = dt(cell(icell)%surround_p%i(6))
          res = coef(icell)%rhs_u + coef(icell)%an_u(1) * dt1 + &
              & coef(icell)%an_u(2)* dt2 + coef(icell)%an_u(3)* &
              & dt3 + coef(icell)%an_u(4) * dt4 + &
              & coef(icell)%an_u(5) * dt5 + coef(icell)%an_u(6) * &
              & dt6
        ELSE
          PRINT *, 'error 1', icell, cell(icell)%num_faces
        END IF
        summ = summ + abs (res-coef(icell)%ap_u/urfu*dt(icell))
        dt(icell) = res / coef(icell)%ap_u*urfu
      end do
      endif
    END DO
    IF(iter==1)then
      if(summ<tiny)exit
      summ1=summ
    else
      IF (summ/summ1 < errors) EXIT
    endif
  END DO
  node(1:ncell)%u  = node(1:ncell)%u + dt(1:ncell)

  dt = 0._high
  DO iter = 1, num_iter
    summ = 0._high
    do iblock=1,nblock
      IF(INDEX(BLOCK(iblock)%title,'soli')==0)then
      do icell=block(iblock)%n_begin,block(iblock)%n_end
        IF (cell(icell)%num_faces == 3) THEN
            dt1 = dt(cell(icell)%surround_p%i(1))
            dt2 = dt(cell(icell)%surround_p%i(2))
            dt3 = dt(cell(icell)%surround_p%i(3))
         res = coef(icell)%rhs_v  + coef(icell)%an_v(1) * dt1 + &
           & coef(icell)%an_v(2) * dt2 + coef(icell)%an_v(3) * dt3
        ELSE IF (cell(icell)%num_faces == 4) THEN
            dt1 = dt(cell(icell)%surround_p%i(1))
            dt2 = dt(cell(icell)%surround_p%i(2))
            dt3 = dt(cell(icell)%surround_p%i(3))
            dt4 = dt(cell(icell)%surround_p%i(4))
          res = coef(icell)%rhs_v + coef(icell)%an_v(1) * dt1 + &
              & coef(icell)%an_v(2) * dt2 &
              & + coef(icell)%an_v(3) * dt3 + coef(icell)%an_v(4)&
              & * dt4
        ELSE IF (cell(icell)%num_faces == 6) THEN
            dt1 = dt(cell(icell)%surround_p%i(1))
            dt2 = dt(cell(icell)%surround_p%i(2))
            dt3 = dt(cell(icell)%surround_p%i(3))
            dt4 = dt(cell(icell)%surround_p%i(4))
            dt5 = dt(cell(icell)%surround_p%i(5))
            dt6 = dt(cell(icell)%surround_p%i(6))
          res = coef(icell)%rhs_v + coef(icell)%an_v(1) * dt1 + &
              & coef(icell)%an_v(2)*dt2 + coef(icell)%an_v(3)*dt3 &
              & + coef(icell)%an_v(4)* dt4 + coef(icell)%an_v(5) &
              &* dt5 + coef(icell)%an_v(6) * dt6
        ELSE
          stop
        END IF
        summ = summ + abs (res-coef(icell)%ap_v/urfv*dt(icell))
        dt(icell) = res / coef(icell)%ap_v*urfv
      enddo
      endif
    END DO
    IF(iter==1)then
      if(summ<tiny)exit
      summ1=summ
    else
      IF (summ/summ1 < errors) EXIT
    endif
  END DO
  node(1:ncell)%v  = node(1:ncell)%v + dt(1:ncell)

  if(dim==2)return

  dt = 0._high
  DO iter = 1, num_iter
    summ = 0._high
    do iblock=1,nblock
      IF(INDEX(BLOCK(iblock)%title,'soli')==0)then
      do icell=block(iblock)%n_begin,block(iblock)%n_end
        IF (cell(icell)%num_faces == 3) THEN
            dt1 = dt(cell(icell)%surround_p%i(1))
            dt2 = dt(cell(icell)%surround_p%i(2))
            dt3 = dt(cell(icell)%surround_p%i(3))
            res = coef(icell)%rhs_w  + coef(icell)%an_w(1)* dt1 + &
              & coef(icell)%an_w(2) * dt2 + coef(icell)%an_w(3) * &
              & dt3
        ELSE IF (cell(icell)%num_faces == 4) THEN
            dt1 = dt(cell(icell)%surround_p%i(1))
            dt2 = dt(cell(icell)%surround_p%i(2))
            dt3 = dt(cell(icell)%surround_p%i(3))
            dt4 = dt(cell(icell)%surround_p%i(4))
            res = coef(icell)%rhs_w + coef(icell)%an_w(1) * dt1 + &
              & coef(icell)%an_w(2)* dt2 + coef(icell)%an_w(3)* &
              & dt3 + coef(icell)%an_w(4) * dt4
        ELSE IF (cell(icell)%num_faces == 6) THEN
            dt1 = dt(cell(icell)%surround_p%i(1))
            dt2 = dt(cell(icell)%surround_p%i(2))
            dt3 = dt(cell(icell)%surround_p%i(3))
            dt4 = dt(cell(icell)%surround_p%i(4))
            dt5 = dt(cell(icell)%surround_p%i(5))
            dt6 = dt(cell(icell)%surround_p%i(6))
            res = coef(icell)%rhs_w + coef(icell)%an_w(1)* dt1 + &
```

```
                  & coef(icell)%an_w(2)* dt2 + coef(icell)%an_w(3)* dt3
                  & + coef(icell)%an_w(4)* dt4+ coef(icell)%an_w(5) * &
                  & dt5 + coef(icell)%an_w(6) * dt6
            ELSE
              stop
            END IF
            summ = summ + abs (res-coef(icell)%ap_w/urfw*dt(icell))
            dt(icell) = res / coef(icell)%ap_w*urfw
          enddo
          endif
        END DO
        IF(iter==1)then
          if(summ<tiny)exit
          summ1=summ
        else
          IF (summ/summ1 < errors) EXIT
        endif
      END DO
      node(1:ncell)%w  = node(1:ncell)%w + dt(1:ncell)
      return
    END SUBROUTINE i_solver_gs_velocity
```

A1.2 LOOP-BLOCKED VERSION

```
SUBROUTINE i_solver_gs_velocity_sb (cell, node, coef, ncell, nnode,
BLOCK, nblock, num_iter)
      INTEGER :: icell, ncell, iter, num_iter, nblock, nnode,
iblock,counter,subblock,nsubblock,remainder
      TYPE (cells), DIMENSION (:) :: cell
      TYPE (points), DIMENSION (:) :: node
      TYPE (coeffs), DIMENSION (:) :: coef
      TYPE (block_t), DIMENSION (:) :: block
      real (high), dimension(nnode) :: dt,dt_u,dt_v,dt_w
      REAL (high) :: res_w,res_u,res_v,summ, dt1, dt2, dt3, dt4, dt5,
                  dt6, summ1, errors=1.e-4_high
      REAL (high) :: dt11,dt22,dt33,dt44,dt55,dt66,dt01,dt02,
                  dt03,dt04,dt05,dt06
      dt_u = 0._high
      dt_v = 0._high
      dt_w = 0._high
      subblock=100
!
      IF  ( subblock > ncell) then
    DO iter = 1, num_iter
         do icell= 1, ncell
              dt1 = dt_u(cell(icell)%surround_p%i(1))
              dt2 = dt_u(cell(icell)%surround_p%i(2))
              dt3 = dt_u(cell(icell)%surround_p%i(3))
              dt4 = dt_u(cell(icell)%surround_p%i(4))
              dt5 = dt_u(cell(icell)%surround_p%i(5))
              dt6 = dt_u(cell(icell)%surround_p%i(6))
              dt11 = dt_v(cell(icell)%surround_p%i(1))
              dt22 = dt_v(cell(icell)%surround_p%i(2))
              dt33 = dt_v(cell(icell)%surround_p%i(3))
              dt44 = dt_v(cell(icell)%surround_p%i(4))
              dt55 = dt_v(cell(icell)%surround_p%i(5))
              dt66 = dt_v(cell(icell)%surround_p%i(6))
              dt01 = dt_w(cell(icell)%surround_p%i(1))
              dt02 = dt_w(cell(icell)%surround_p%i(2))
              dt03 = dt_w(cell(icell)%surround_p%i(3))
              dt04 = dt_w(cell(icell)%surround_p%i(4))
              dt05 = dt_w(cell(icell)%surround_p%i(5))
              dt06 = dt_w(cell(icell)%surround_p%i(6))
                res_u = coef(icell)%rhs_u + coef(icell)%an_u(1)*dt1 + &
                  & coef(icell)%an_u(2) * dt2 + coef(icell)%an_u(3) * &
                  & dt3 + coef(icell)%an_u(4) * dt4 + &
                  & coef(icell)%an_u(5) * dt5 + coef(icell)%an_u(6) * &
                  & dt6
                res_v = coef(icell)%rhs_v + coef(icell)%an_w(1)* dt11 &
                  & + coef(icell)%an_v(2)* dt22 + coef(icell)%an_v(3) &
                  & * dt33 + coef(icell)%an_v(4) * dt44 &
                  & + coef(icell)%an_v(5)* dt55 + coef(icell)%an_v(6) &
                  & * dt66
                res_w = coef(icell)%rhs_w + coef(icell)%an_w(1)* dt01 &
                  & + coef(icell)%an_w(2)*dt02 + coef(icell)%an_w(3) *&
                  & dt03 + coef(icell)%an_w(4) * dt04 &
                  & + coef(icell)%an_w(5)* dt05 + coef(icell)%an_w(6) &
                  & * dt06
            summ = summ + abs (res_u- &
                  & coef(icell)%ap_u/urfu*dt_u(icell))+abs (res_v- &
                  & coef(icell)%ap_v/urfv*dt_v(icell)) &
                  & + abs (res_w-coef(icell)%ap_w/urfw*dt_w(icell))
            dt_u(icell) = res_u / coef(icell)%ap_u*urfu
            dt_v(icell) = res_v / coef(icell)%ap_v*urfv
            dt_w(icell) = res_w / coef(icell)%ap_w*urfw
         enddo
!
        IF(iter==1)then
          if(summ<tiny)exit
          summ1=summ
        else
          IF (summ/summ1 < errors) EXIT
        endif
   ENDDO
 ELSE

      remainder = mod(ncell,subblock)
      nsubblock = ncell/subblock
!
      IF(remainder > 0)then
      nsubblock = nsubblock + 1
      ELSE
      nsubblock = nsubblock
      ENDIF
!
        do counter = 1, nsubblock
         DO iter = 1, num_iter
         summ = 0._high
     IF (counter == nsubblock)then
     IF (remainder > 0)then
       do icell= (counter-1)*subblock+1,(counter-1)*subblock+remainder
```

```
         dt1 = dt_u(cell(icell)%surround_p%i(1))
         dt2 = dt_u(cell(icell)%surround_p%i(2))
         dt3 = dt_u(cell(icell)%surround_p%i(3))
         dt4 = dt_u(cell(icell)%surround_p%i(4))
         dt5 = dt_u(cell(icell)%surround_p%i(5))
         dt6 = dt_u(cell(icell)%surround_p%i(6))
         dt11 = dt_v(cell(icell)%surround_p%i(1))
         dt22 = dt_v(cell(icell)%surround_p%i(2))
         dt33 = dt_v(cell(icell)%surround_p%i(3))
         dt44 = dt_v(cell(icell)%surround_p%i(4))
         dt55 = dt_v(cell(icell)%surround_p%i(5))
         dt66 = dt_v(cell(icell)%surround_p%i(6))
         dt01 = dt_w(cell(icell)%surround_p%i(1))
         dt02 = dt_w(cell(icell)%surround_p%i(2))
         dt03 = dt_w(cell(icell)%surround_p%i(3))
         dt04 = dt_w(cell(icell)%surround_p%i(4))
         dt05 = dt_w(cell(icell)%surround_p%i(5))
         dt06 = dt_w(cell(icell)%surround_p%i(6))

           res_u = coef(icell)%rhs_u + coef(icell)%an_u(1)*dt1 + &
             & coef(icell)%an_u(2) * dt2 + coef(icell)%an_u(3) * &
             & dt3 + coef(icell)%an_u(4) * dt4 + &
             & coef(icell)%an_u(5) * dt5 + coef(icell)%an_u(6) * &
             & dt6
           res_v = coef(icell)%rhs_v + coef(icell)%an_w(1)* dt11 &
             & + coef(icell)%an_v(2)* dt22 + coef(icell)%an_v(3) &
             & * dt33 + coef(icell)%an_v(4) * dt44 &
             & + coef(icell)%an_v(5)* dt55 + coef(icell)%an_v(6) &
             & * dt66
           res_w = coef(icell)%rhs_w + coef(icell)%an_w(1)* dt01 &
             & + coef(icell)%an_w(2)*dt02 + coef(icell)%an_w(3) *&
             & dt03 + coef(icell)%an_w(4) * dt04 &
             & + coef(icell)%an_w(5)* dt05 + coef(icell)%an_w(6) &
             & * dt06
       summ = summ + abs (res_u- &
             & coef(icell)%ap_u/urfu*dt_u(icell))+abs (res_v- &
             & coef(icell)%ap_v/urfv*dt_v(icell)) &
             & + abs (res_w-coef(icell)%ap_w/urfw*dt_w(icell))

       dt_u(icell) = res_u / coef(icell)%ap_u*urfu
       dt_v(icell) = res_v / coef(icell)%ap_v*urfv
       dt_w(icell) = res_w / coef(icell)%ap_w*urfw

  enddo
ELSE
      do icell= (counter-1)*subblock+1,(counter)*subblock
         dt1 = dt_u(cell(icell)%surround_p%i(1))
         dt2 = dt_u(cell(icell)%surround_p%i(2))
         dt3 = dt_u(cell(icell)%surround_p%i(3))
         dt4 = dt_u(cell(icell)%surround_p%i(4))
         dt5 = dt_u(cell(icell)%surround_p%i(5))
         dt6 = dt_u(cell(icell)%surround_p%i(6))
         dt11 = dt_v(cell(icell)%surround_p%i(1))
         dt22 = dt_v(cell(icell)%surround_p%i(2))
         dt33 = dt_v(cell(icell)%surround_p%i(3))
         dt44 = dt_v(cell(icell)%surround_p%i(4))
         dt55 = dt_v(cell(icell)%surround_p%i(5))
```

```
          dt66 = dt_v(cell(icell)%surround_p%i(6))
          dt01 = dt_w(cell(icell)%surround_p%i(1))
          dt02 = dt_w(cell(icell)%surround_p%i(2))
          dt03 = dt_w(cell(icell)%surround_p%i(3))
          dt04 = dt_w(cell(icell)%surround_p%i(4))
          dt05 = dt_w(cell(icell)%surround_p%i(5))
          dt06 = dt_w(cell(icell)%surround_p%i(6))

            res_u = coef(icell)%rhs_u + coef(icell)%an_u(1)*dt1 + &
              & coef(icell)%an_u(2) * dt2 + coef(icell)%an_u(3) * &
              & dt3 + coef(icell)%an_u(4) * dt4 + &
              & coef(icell)%an_u(5) * dt5 + coef(icell)%an_u(6) * &
              & dt6
            res_v = coef(icell)%rhs_v + coef(icell)%an_w(1)* dt11 &
              & + coef(icell)%an_v(2)* dt22 + coef(icell)%an_v(3) &
              & * dt33 + coef(icell)%an_v(4) * dt44 &
              & + coef(icell)%an_v(5)* dt55 + coef(icell)%an_v(6) &
              & * dt66
            res_w = coef(icell)%rhs_w + coef(icell)%an_w(1)* dt01 &
              & + coef(icell)%an_w(2)*dt02 + coef(icell)%an_w(3) *&
              & dt03 + coef(icell)%an_w(4) * dt04 &
              & + coef(icell)%an_w(5)* dt05 + coef(icell)%an_w(6) &
              & * dt06
        summ = summ + abs (res_u- &
              & coef(icell)%ap_u/urfu*dt_u(icell))+abs (res_v- &
              & coef(icell)%ap_v/urfv*dt_v(icell)) &
              & + abs (res_w-coef(icell)%ap_w/urfw*dt_w(icell))

        dt_u(icell) = res_u / coef(icell)%ap_u*urfu
        dt_v(icell) = res_v / coef(icell)%ap_v*urfv
        dt_w(icell) = res_w / coef(icell)%ap_w*urfw
   enddo
  ENDIF
ELSE
   do icell= (counter-1)*subblock+1,counter*subblock
          dt1 = dt_u(cell(icell)%surround_p%i(1))
          dt2 = dt_u(cell(icell)%surround_p%i(2))
          dt3 = dt_u(cell(icell)%surround_p%i(3))
          dt4 = dt_u(cell(icell)%surround_p%i(4))
          dt5 = dt_u(cell(icell)%surround_p%i(5))
          dt6 = dt_u(cell(icell)%surround_p%i(6))
          dt11 = dt_v(cell(icell)%surround_p%i(1))
          dt22 = dt_v(cell(icell)%surround_p%i(2))
          dt33 = dt_v(cell(icell)%surround_p%i(3))
          dt44 = dt_v(cell(icell)%surround_p%i(4))
          dt55 = dt_v(cell(icell)%surround_p%i(5))
          dt66 = dt_v(cell(icell)%surround_p%i(6))
          dt01 = dt_w(cell(icell)%surround_p%i(1))
          dt02 = dt_w(cell(icell)%surround_p%i(2))
          dt03 = dt_w(cell(icell)%surround_p%i(3))
          dt04 = dt_w(cell(icell)%surround_p%i(4))
          dt05 = dt_w(cell(icell)%surround_p%i(5))
          dt06 = dt_w(cell(icell)%surround_p%i(6))

            res_u = coef(icell)%rhs_u + coef(icell)%an_u(1)*dt1 + &
              & coef(icell)%an_u(2) * dt2 + coef(icell)%an_u(3) * &
              & dt3 + coef(icell)%an_u(4) * dt4 + &
```

```
                  & coef(icell)%an_u(5) * dt5 + coef(icell)%an_u(6) * &
                  & dt6
                res_v = coef(icell)%rhs_v + coef(icell)%an_w(1)* dt11 &
                  & + coef(icell)%an_v(2)* dt22 + coef(icell)%an_v(3) &
                  & * dt33 + coef(icell)%an_v(4) * dt44 &
                  & + coef(icell)%an_v(5)* dt55 + coef(icell)%an_v(6) &
                  & * dt66
                res_w = coef(icell)%rhs_w + coef(icell)%an_w(1)* dt01 &
                  & + coef(icell)%an_w(2)*dt02 + coef(icell)%an_w(3) *&
                  & dt03 + coef(icell)%an_w(4) * dt04 &
                  & + coef(icell)%an_w(5)* dt05 + coef(icell)%an_w(6) &
                  & * dt06
            summ = summ + abs (res_u- &
                  & coef(icell)%ap_u/urfu*dt_u(icell))+abs (res_v- &
                  & coef(icell)%ap_v/urfv*dt_v(icell)) &
                  & + abs (res_w-coef(icell)%ap_w/urfw*dt_w(icell))

            dt_u(icell) = res_u / coef(icell)%ap_u*urfu
            dt_v(icell) = res_v / coef(icell)%ap_v*urfv
            dt_w(icell) = res_w / coef(icell)%ap_w*urfw

       enddo
   ENDIF
!
!
     IF(iter==1)then
          if(summ<tiny)exit
          summ1=summm
        else
          IF (summ/summ1 < errors) EXIT
        endif
      END DO
   enddo
!
  ENDIF
      node(1:ncell)%u  = node(1:ncell)%u + dt_u(1:ncell)
      node(1:ncell)%v  = node(1:ncell)%v + dt_v(1:ncell)
      node(1:ncell)%w  = node(1:ncell)%w + dt_w(1:ncell)

!
      return
    END SUBROUTINE i_solver_gs_velocity_sb
```

A2. Data Structure Change

Given below are the changes made to the data structure in the code. The new data structure with two-dimensional array replaced the pointer for the face data structure. The `v(1), v(2), p1, p2` were replaced by 1,2,3,4 respectively in the two-dimensionsal array. Similar changes were made to other variables also. The changes were made only to the face data structure to test the idea and also because it showed a major scope of improvement.

For ‘internal’

```
       internal%facex(iface,1) = internal%face(iface)%v(1)
       internal%facex(iface,2) = internal%face(iface)%v(2)
       internal%facex(iface,3) = internal%face(iface)%p1
       internal%facex(iface,4) = internal%face(iface)%p2
```

For ‘Boundary Conditions’

```
    boundary%bc(ibc)%facex(iface,1) = boundary%bc(ibc)%face(iface)%v(1)
    boundary%bc(ibc)%facex(iface,2) = boundary%bc(ibc)%face(iface)%v(2)
    boundary%bc(ibc)%facex(iface,3) = boundary%bc(ibc)%face(iface)%p1
    boundary%bc(ibc)%facex(iface,4) = boundary%bc(ibc)%face(iface)%p2
```

In the relevant subroutines the values associated with the variables were called using the new data structure as shown below. The new data structure proved to e more cache friendly and led to performance improvements.

A2.1 ORIGINAL DATA STRUCTURE

```
        v1 = internal%face(iface)%v(1)
        v2 = internal%face(iface)%v(2)
        p1 = internal%face(iface)%p1
        p2 = internal%face(iface)%p2
        a1 = internal%face(iface)%a(1)
        a2 = internal%face(iface)%a(2)
        ex = internal%face(iface)%e(1)
        ey = internal%face(iface)%e(2)
        nx = internal%face(iface)%n(1)
        ny = internal%face(iface)%n(2)
```

A2.1 MODIFIED DATA STRUCTURE

```
v1 = internal%facex(iface,1)
v2 = internal%facex(iface,2)
p1 = internal%facex(iface,3)
p2 = internal%facex(iface,4)
a1 = internal%facedata(iface,1)
a2 = internal%facedata(iface,2)
ex = internal%facedata(iface,3)
ey = internal%facedata(iface,4)
nx = internal%facedata(iface,5)
ny = internal%facedata(iface,6)
```

REFERENCES

1. M.J. Aftosmis, M.J. Berger, S.M Murman, “Applications of space filing curves to Cartesian methods for CFD,” 42nd Aerospace Sciences Meeting and Exhibit, Reno, NV, 2004.
2. S. Kadambi, “Accelerating CFD application by improving cached data reuse,” Thesis, Mississippi State University, May 1995.
3. TH. Huaser, T.I. Mattox, R.P. LeBeau, H.G. Dietz, P.G. Huang, “Code optimization for complex microprocessors applied to CFD software,” SIAM Journal on Scientific Computing, vol. 25, 2004, pp 1461-1477.
4. TH. Huaser, R.P. LeBeau, P.G. Huang, T.I. Mattox, H.G. Dietz, “ Improving the performance of Computational Fluid Dynamics codes on Linux cluster architectures,” 16th AIAA Computational Fluid Dynamics Conference, Orlando, FL, 2003.
5. http://www.howstuffworks.com as on 06/07/2005
6. H. Sagan, Space-Filling Curves, Springer-Verlag, New York, 1994
7. T. Ogawa, “Parallelization of an adaptive Cartesian mesh flow solver based on the 2^N-tree data structure.”
8. J.M.Dennis, “Partitioning with space filling curves on the cubed sphere”
9. J. Behrens, J. Zimmerman, “Parallelizing an unstructured grid generator with a space filling curve approach.”
10. S. Phuvan, T. K. Oh, N. Caviris, Y. Li, H. H. Szu, “Texture analysis by space-filling curves and one-dimensional Haar wavelets.”
11. R. Dafner, D.Cohen-Or, Y. Matias, “Context based space filling curves,” EUROGRAPHICS 2000, vol. 19.
12. M. Kowarschik, C. Weis, “An overview of cache optimization techniques and cahe aware numerical algorithms”.
13. H. Chen, P.G. Huang, R.P. LeBeau, “ A cell-centered pressure based method for two/three-dimensional unstructured incompressible Navier-Stokes solver,” Proceedings of the International Conference on Computational Fluid Dynamics 3, Toronto, Canada, 2004.

14. G. Karypis, V. Kumar, A software package for partitioning unstructured graphs, partitioning meshes, and computing fill reducing orderings of sparse matrices version 4.0. 1998.
15. C.M. Rhie, W.L. Chow, "Numerical study of the turbukent flow past an airfoil with tailing edge separation," AIAA Journal, vol. 21, 1983,pp 1525-1532.
16. http://gnu.org
17. http://www.linuxjournal.com/node/7930/print
18. R.P. LeBeau, P. Kristipati, S.Gupta, H. Chen, P.G. Huang, "Joint performance evaluation and optimization of two CFD codes on commodity clusters," 43rd Aerospace Sciences Meeting and Exhibit, Reno, NV, 2005.
19. U.Ghia, K.Ghia, C.T. Shin, " High-resolution for incompressible flow using the Navier-Stokes equations and a multigrid method," Journal of Computational Physics, vol. 48, 1982, pp. 387-411
20. H.C. Ku, "Solutions of flow in complex geometries by pseudo spectral element method," Journal of Computational Physics, vol. 117,1995, pp. 215-227
21. R.D. Henderson, "Unstructured spectral element methods: parallel algorithms and simulations," Ph.D Thesis, Princeton University, 1994.
22. A. Roshko, "On the development of turbulent wakes from vortex sheets," NACA Report 1191, 1954.
23. R. Willie, "Karman vortex sheets," Advances in Applied Mechanics, vol. 6, Academic, New York, 1960, pp. 273-287.
24. C. H. K. Williamson, "Three-dimensional vortex dynamics in bluff body wakes," Experimental Thermal and Fluid Sciences," vol. 12, pp. 150-168,1996.
25. Workshop conducted on "CFD validation of synthetic jets and turbulent separation control", Williamsburg, VA, 2004.

VITA

Saurabh Gupta was born in Jhansi, India on 7th of June 1980. He graduated from Fatima Convent High School in May 1996. He completed his Intermediate college from S.K.V.S Junior college, Vijayawada in 1998. He obtained his undergraduate degree in Mechanical Engineering from C.B.I.T, Osmania University, India in May 2002. He received certificate of merit for being among the top five students in Mechanical Engineering of all the affiliated colleges in the University. He entered the University of Kentucky in August 2003 to pursue his Masters degree in Mechanical Engineering and joined the University of Kentucky Computational Fluid Dynamics in January 2004. He conducted research on optimization of unstructured two/three-dimensional CFD codes which composed his dissertation. He presented his work at various conferences.

Conferences

Gupta S., Palki A., LeBeau Jr. R.P., (2006) "*Use of Cache-Friendly Blocking to Accelerate CFD codes on Commodity Hardware*," presented at the 44th AIAA Aerospace Science Meeting & Exhibit, Reno, January, 2006

Gupta S., Palki A., LeBeau Jr. R.P., (2006) *"Data Access Optimization by Cache Friendly Blocking for CFD Codes"* presented at 32nd AIAA Dayton-Cincinnati Aerospace Sciences Symposium, Dayton, Ohio. March, 2006

Gupta S., Huang Shih-Che, Hauser T, Palki A., LeBeau Jr. R.P., (2006) *"Parallel Performance Evaluation of Different Cluster Architectures for CFD Applications"* presented at 32nd AIAA Dayton-Cincinnati Aerospace Sciences Symposium, Dayton, Ohio. March, 2006

Gupta S., LeBeau Jr. R.P., Chen H., Kristipati P., Huang P.G., (2005) "*Joint Performance Evaluation and Optimization of Two Navier-Stokes Codes on Commodity Cluster Architectures,*" presented at 43rd AIAA Aerospace Science Meeting & Exhibit, Reno, January, 2005

Gupta S., LeBeau Jr. R.P., Kristipati P., Huang P.G., (2005) *"Performance Optimization of a Structured and Unstructured Code on Commodity Clusters,"* presented at 31st AIAA Dayton-Cincinnati Aerospace Sciences Symposium, Dayton, Ohio. March, 2005