CACHE OPTIMIZATION AND PERFORMANCE EVALUATION OF A STRUCTURED CFD CODE - GHOST

Anand B. Palki

University of Kentucky, apalki@gmail.com

Recommended Citation
https://uknowledge.uky.edu/gradschool_theses/363

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.
ABSTRACT OF THESIS

CACHE OPTIMIZATION AND PERFORMANCE EVALUATION OF A STRUCTURED CFD CODE - GHOST

This research focuses on evaluating and enhancing the performance of an in-house, structured, 2D CFD code - GHOST, on modern commodity clusters. The basic philosophy of this work is to optimize the cache performance of the code by splitting up the grid into smaller blocks and carrying out the required calculations on these smaller blocks. This in turn leads to enhanced code performance on commodity clusters. Accordingly, this work presents a discussion along with a detailed description of two techniques: external and internal blocking, for data access optimization. These techniques have been tested on 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 modified version of the code was also ported to the current state-of-art architectures with successful results.

KEYWORDS: Cache Optimization, External blocking, Internal blocking, Structured CFD Code Optimization, Commodity Clusters

Anand .B. Palki
12/15/2006
CACHE OPTIMIZATION AND PERFORMANCE EVALUATION OF A STRUCTURED CFD CODE - GHOST

By

Anand .B. Palki

__________________________
Dr. Raymond .P. LeBeau
Director of Thesis

__________________________
Dr. L. S. Stephens
Director of Graduate Studies

__________________________
12/15/2005
RULES FOR THE USE OF THESIS

Unpublished thesis submitted for the Master’s degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgements.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name

Date
CACHE OPTIMIZATION AND PERFORMANCE EVALUATION OF A STRUCTURED CFD CODE - GHOST

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
Anand B. Palki
Lexington, Kentucky

Director: Dr. Raymond P. LeBeau
Assistant Professor of Mechanical Engineering
University of Kentucky Lexington, Kentucky
2006
To My Parents
ACKNOWLEDGEMENTS

The satisfaction and euphoria that accompany the successful completion of any task would not be complete without the mention of the people who made it possible, whose constant encouragement and guidance crowned the effort with success.

I would like to express my sincere gratitude to my academic advisor Dr. Raymond LeBeau, for his continuous support and encouragement throughout my work. The progress of this work would not be achieved without his guidance and numerous ingenious suggestions. I would like to thank Dr. P.G. Huang for helping me understand how a CFD code works. I would also like to thank my defense committee members Dr. Jamey Jacob and Dr. M. Seigler for their valuable time for serving on my committee and evaluating my work.

I am grateful to my student colleagues who made my stay at the University of Kentucky a memorable one. I am especially grateful to Abhishek T, Ajay Babu, Aditya C, Chaitanya Penugonda, Chetan Babu, Daniel R, Jacky Rhinehart, Karthik M, Narendra BK, Phanindra C, Radhika K, Sandeep B, Snehal P and Vijay N for their emotional support, entertainment, caring and most importantly for being the surrogate family over the past two years. Finally, I would like to thank my parents for their continuous encouragement and love without which none of this would have been possible.
Table of Contents

Acknowledgements........................................................................................................................................vii

List of Tables ...............................................................................................................................................x

List of Figures .............................................................................................................................................xi

List of Files ................................................................................................................................................xiii

1. Introduction .............................................................................................................................................1
   1.1 Overview.............................................................................................................................................1
   1.2 Background - Memory Hierarchy .......................................................................................................2
   1.3 Introduction to Problem ...................................................................................................................6
   1.4 Goals of Optimizing the Code ..........................................................................................................7
   1.5 Previous Work ...................................................................................................................................8
      1.5.1 General Cache Optimization Techniques ................................................................................8
      1.5.1.1 Techniques for Reducing Capacity Misses .......................................................................9
      1.5.1.2 Techniques for Reducing Conflict Misses .......................................................................11
      1.5.1.3 Techniques to Hide Effects of Cache Misses ..................................................................12
      1.5.1.4 Techniques to Improve the Replacement Decisions by Cache .....................................13
      1.5.2 Optimizations to CFD Codes ....................................................................................................13
      1.5.2.1 Techniques to Improve Parallel Performance .................................................................13
      1.5.2.2 Techniques to Improve Single Node Performance ...........................................................14
   1.6 External & Internal Blocking ...........................................................................................................15

2. Computational tools .................................................................................................................................17
   2.1 General Description of GHOST ......................................................................................................17
      2.1.1 GHOST Flowchart ..................................................................................................................18
      2.1.2 Governing equations ...............................................................................................................19
      2.1.3 Calculation at artificial boundaries .........................................................................................21
   2.2 Grid File Data ...................................................................................................................................22
      2.2.1 Finite Volume Method .............................................................................................................22
      2.2.2 Generalized coordinates .........................................................................................................22
      2.2.3 Description of G.F90 Output ....................................................................................................25
      2.2.4 Description of Input File .........................................................................................................26
   2.3 Compilers & MPI Environment ........................................................................................................26
   2.4 Valgrind [79] ....................................................................................................................................28
   2.5 Kentucky Fluid Clusters ...................................................................................................................30
   2.6 Method used to measure performance ............................................................................................31
   2.7 Summary ...........................................................................................................................................32

3. External Blocking Results .....................................................................................................................33
   3.1 Terminology .....................................................................................................................................33
      3.1.1 Terms related to Code Versions ............................................................................................33
      3.1.2 Terms related to performance study test results description .............................................34
      3.1.3 Terms related to cache behavior study test results ...............................................................34
   3.2 Test Case .........................................................................................................................................35
   3.3 Types of Tests ...................................................................................................................................35
   3.4 External Blocking .............................................................................................................................37
   3.5 Performance Test Results ................................................................................................................37
      3.5.1 KFC4 Results ..........................................................................................................................37
      3.5.2 KFC5 Results ..........................................................................................................................41
      3.5.3 Rectangular Blocks ..................................................................................................................42
      3.5.4 Effects of Compiler Optimization Levels on Performance .................................................43
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.5 Effects of Different Compilers on Performance</td>
<td>46</td>
</tr>
<tr>
<td>3.5.5 Effect of different hardware on performance</td>
<td>50</td>
</tr>
<tr>
<td>3.5.6 Steady Turbulent Case Performance Results</td>
<td>53</td>
</tr>
<tr>
<td>3.6 Valgrind Results</td>
<td>57</td>
</tr>
<tr>
<td>3.6.1 KFC4 Results</td>
<td>57</td>
</tr>
<tr>
<td>3.6.2 Comparison between G95 &amp; IFC</td>
<td>59</td>
</tr>
<tr>
<td>3.6.3 Effect of Cache Thrashing</td>
<td>62</td>
</tr>
<tr>
<td>3.6.4 Valgrind Results for Turbulent Case</td>
<td>63</td>
</tr>
<tr>
<td>3.7 Accuracy Test Results</td>
<td>64</td>
</tr>
<tr>
<td>3.8 Summary</td>
<td>67</td>
</tr>
<tr>
<td>4. Internal Blocking Results</td>
<td>68</td>
</tr>
<tr>
<td>4.1 Basic Principle</td>
<td>68</td>
</tr>
<tr>
<td>4.2 Implementation of Internal Blocking in GHOST</td>
<td>68</td>
</tr>
<tr>
<td>4.3 Primary Tests</td>
<td>71</td>
</tr>
<tr>
<td>4.4 Performance Test Results</td>
<td>72</td>
</tr>
<tr>
<td>4.4.1 Comparison of Performance Test Results between External and Internal Blocking</td>
<td>74</td>
</tr>
<tr>
<td>4.5 Valgrind Results</td>
<td>76</td>
</tr>
<tr>
<td>4.5.1 Comparison of Valgrind results between Internal and External Blocking</td>
<td>79</td>
</tr>
<tr>
<td>4.6 Accuracy Test Results</td>
<td>81</td>
</tr>
<tr>
<td>4.7 Summary</td>
<td>84</td>
</tr>
<tr>
<td>5. Unsteady Test Case Results</td>
<td>85</td>
</tr>
<tr>
<td>5.1 Laminar Unsteady Test Case</td>
<td>85</td>
</tr>
<tr>
<td>5.2 Performance Test Results</td>
<td>87</td>
</tr>
<tr>
<td>5.3 Accuracy Test Results</td>
<td>90</td>
</tr>
<tr>
<td>5.4 Summary</td>
<td>93</td>
</tr>
<tr>
<td>6. Conclusions And Future Work</td>
<td>94</td>
</tr>
<tr>
<td>6.1 Summary and Conclusions</td>
<td>94</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>98</td>
</tr>
<tr>
<td>Appendix</td>
<td>100</td>
</tr>
<tr>
<td>A.1 Steps to implement internal blocking to GHOST</td>
<td>100</td>
</tr>
<tr>
<td>References</td>
<td>116</td>
</tr>
<tr>
<td>Vita</td>
<td>122</td>
</tr>
</tbody>
</table>
LIST OF TABLES

TABLE 1-1 TYPICAL CHARACTERISTICS OF MEMORY HIERARCHY CONSTITUENTS ON A PC......6
TABLE 1-2 ILLUSTRATION OF LOOP FUSION .................................................................10
TABLE 1-3 ARRAY PADDING [15] ..................................................................................12
TABLE 2-1 SUMMARY OF THE SUBROUTINES IN WHICH THE MAIN ITERATIVE CALCULATIONS
OF GHOST ARE CARRIED OUT....................................................................................21
TABLE 2-2 COMPARISON OF THE VARIOUS PROCESSORS BASED ON CERTAIN PARAMETERS ...31
TABLE 3-1 SUMMARY OF PERFORMANCE IMPROVEMENTS OBTAINED ON KFC4............39
TABLE 3-2 SUMMARY OF PERFORMANCE IMPROVEMENTS ACHIEVED ON KFC5 ............42
TABLE 3-3 COMPARISON OF PERFORMANCE IMPROVEMENTS ACHIEVED BY USING IFC VS G95
...........................................................................................................................................46
TABLE 3-4 PERFORMANCE IMPROVEMENTS OBTAINED ON VARIOUS HARDWARE FOR V0 ....52
TABLE 3-5 PERFORMANCE IMPROVEMENTS OBTAINED ON VARIOUS HARDWARE FOR V0 ....53
TABLE 3-6 SUMMARY OF PERFORMANCE IMPROVEMENTS FOR THE TURBULENT TEST CASE
ACHIEVED ON KFC4.................................................................................................56
TABLE 4-1 VALUES OF WALLTIME FOR THE GRID OF SIZE 600 X 600 SPLIT UP USING VARIOUS
BLOCK SIZES .........................................................................................................72
TABLE 5-1 BLOCK SIZES USED TO BREAK UP THE AIRFOIL GRID.................................87
TABLE 5-2 PERCENTAGE IMPROVEMENT OBTAINED BY APPLICATION OF INTERNAL
BLOCKING ON THE VARIOUS HARDWARE PLATFORMS .............................................89
TABLE 5-3 PERFORMANCE IMPROVEMENTS OBTAINED BY THE APPLICATION OF EXTERNAL
BLOCKING TO LID-DRIVEN CAVITY PROBLEM........................................................90
TABLE 6-1 SUMMARY OF THE PERFORMANCE IMPROVEMENTS OBTAINED.....................98
LIST OF FIGURES

FIGURE 1-1 MEMORY HIERARCHY IN MODERN DAY COMPUTER [8] .......................................................... 3
FIGURE 1-2 REGISTERS IN CPU [9] ........................................................................................................ 4
FIGURE 1-3 ILLUSTRATION OF LOOP TRANSFORMATION .................................................................. 10
FIGURE 1-4 ILLUSTRATION OF LOOP BLOCKING .............................................................................. 11
FIGURE 1-5 SCHEMATIC ILLUSTRATING EXTERNAL BLOCKING ......................................................... 16
FIGURE 2-1 FLOWCHART DEPICTING THE WORKING OF GHOST ....................................................... 20
FIGURE 2-2 ILLUSTRATION OF THE ARTIFICIAL BOUNDARIES ....................................................... 23
FIGURE 2-3 A GRID IN GENERALIZED COORDINATE SYSTEM [75] ....................................................... 24
FIGURE 2-4 ILLUSTRATION OF A SINGLE CONTROL VOLUME IN GENERALIZED COORDINATE .... 24
FIGURE 2-5 GRID COORDINATE TRANSFORMATION ......................................................................... 24
FIGURE 2-6 DESCRIPTION OF A INPUT FILE ......................................................................................... 27
FIGURE 2-7 KENTUCKY FLUID CLUSTERS (KFC) 4 AND 5 ..................................................................... 30
FIGURE 2-8 SCHEMATIC DIAGRAM SHOWING THE LID-DRIVEN CAVITY .......................................... 36
FIGURE 2-9 INPUT FILE USED TO GENERATE A SINGLE BLOCK GRID ............................................. 38
FIGURE 2-10 INPUT FILE GENERATED TO SPLIT THE ABOVE SHOWN GRID INTO TWO BLOCKS .. 38
FIGURE 3-1 WALLTIME AS A FUNCTION OF SUBGRID SIZE FOR GHOST COMPILED WITH G95 ON KFC4 FOR THE LID-DRIVEN TEST CASE. (A) V0 (B) V3 .............................................................. 40
FIGURE 3-2 WALLTIME AS A FUNCTION OF SUBGRID SIZE FOR GHOST COMPILED WITH IFORT (VERSION 9.1) ON KFC5 FOR THE LID-DRIVEN TEST CASE. (A) V0 (B) V3 ........................................... 44
FIGURE 3-3 WALLTIME AS A FUNCTION OF SUBGRID SIZE FOR GHOST COMPILED WITH G95 ON KFC4 FOR THE LID-DRIVEN TEST CASE (RECTANGULAR BLOCKS) (A) V0 (B) V3 ............. 45
FIGURE 3-4 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST COMPILED WITH G95 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) V0 B) V3 ......................................................... 47
FIGURE 3-5 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V0 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) G95 B) INTEL FORTRAN COMPILER VER. 7 .................................................................................................................. 48
FIGURE 3-6 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V3 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) G95 B) INTEL FORTRAN COMPILER VER. 7 .................................................................................................................. 48
FIGURE 3-7 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V0 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) G95 B) INTEL FORTRAN COMPILER VER. 7 .................................................................................................................. 49
FIGURE 3-8 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V3 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) G95 B) INTEL FORTRAN COMPILER VER. 7 .................................................................................................................. 49
FIGURE 3-9 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V0 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) G95 B) INTEL FORTRAN COMPILER VER. 7 .................................................................................................................. 49
FIGURE 3-10 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V3 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) G95 B) INTEL FORTRAN COMPILER VER. 7 .................................................................................................................. 50
FIGURE 3-11 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V0 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) G95 B) INTEL FORTRAN COMPILER VER. 7 .................................................................................................................. 51
FIGURE 3-12 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V3 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) G95 B) INTEL FORTRAN COMPILER VER. 7 .................................................................................................................. 52
FIGURE 3-13 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V0 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) G95 B) INTEL FORTRAN COMPILER VER. 7 .................................................................................................................. 53
FIGURE 3-14 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V3 ON KFC4 FOR THE LID-DRIVEN TEST CASE. A) G95 B) INTEL FORTRAN COMPILER VER. 7 .................................................................................................................. 54
FIGURE 3-15 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V0 ON KFC4 WITH G95 FOR THE LID-DRIVEN TEST CASE. A) V0 B) V3 .................. 55
FIGURE 3-16 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V3 ON KFC4 WITH G95 FOR THE LID-DRIVEN TEST CASE. A) V0 B) V3 .................. 56
FIGURE 3-17 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V0 ON KFC4 WITH IFC FOR THE LID-DRIVEN TEST CASE. A) V0 B) V3 .................. 57
FIGURE 3-18 WALLTIME AS A FUNCTION OF SUBGRID SIZE (OR GRID SIZE FOR A SINGLE NODE CASE) FOR GHOST V3 ON KFC4 WITH IFC FOR THE LID-DRIVEN TEST CASE. A) V0 B) V3 .................. 58
FIGURE 3-19 COMPARISON BETWEEN IFC AND G95 .............................................................................. 59
FIGURE 3-20 VALGRIND PLOT FOR V0 (A) 30 X 30 (B) NO SUBBLOCK .................................................. 58
FIGURE 3-21 VALGRIND PLOT FOR V3 (A) 30 X 30 (B) NO SUBBLOCK .................................................. 58
FIGURE 3-22 L2 CACHE MISS RATE V3 G95 .......................................................................................... 60
FIGURE 3-23 DATA CALLS V3 G95 ........................................................................................................ 60
FIGURE 3-24 VALGRIND PLOT FOR V0 30X30 SUBBLOCK (A) G95 (B) IFC ......................................... 61
FIGURE 3-25 DATA CALLS COMPARISON (A) G95 (B) IFC ................................................................. 61
FIGURE 3-26 L2D MISSES COMPARISON (A) G95 (B) IFC ................................................................. 62
FIGURE 3-27 VALGRIND PLOT FOR V0 (A) 30 X 30 (B) NO SUBBLOCK .................................................. 63
FIGURE 3-28 VALGRIND PLOT FOR V3 (A) 30 X 30 (B) NO SUBBLOCK .................................................. 64
LIST OF FILES

1. APalki.pdf - 1.15 MB (File Size)
CHAPTER - 1

1. INTRODUCTION

1.1 OVERVIEW

In today’s highly competitive industry, getting a good quality product out on the market in the minimum amount of time is the essence of success. This has led the industries to use various computational tools during the initial phase of the product design lifecycle. Computational fluid dynamics (CFD) has emerged as one of the major tools of engineering analysis and design in recent years. This has been possible due to the advent of the high speed computers and the development of novel and efficient algorithms for solving the equations of fluid dynamics. The development of numerical methods for solving the Euler and Navier Stokes equations is far from complete and the search for the best method is still ongoing. For example, the famous physicist Richard Feynman said that turbulence is the last unsolved problem in classical physics. CFD is critical in the quest to find answers to the yet unanswered questions of fluid dynamics since it has been recognized that CFD is the third dimension in fluid dynamics and complements the other two dimensions namely experiments and theory [1]. To get a better insight into various flow phenomena which are observed in nature and to be able to simulate those accurately, sophisticated computational platforms are necessary. Only the major government laboratories, certain big industries and some universities have access to these computing centers. But with improvements in microprocessor technology, these days we have desktop computers with the processing power close to that of an early 1990’s supercomputer. Also with the development of commodity clusters [2] built using inexpensive PCs, today a majority of simulations can be run successfully on clusters. Due to this, apart from the aerospace and automobile industry CFD has recently cut across all disciplines where the flow of fluids is important, such as civil engineering applications, industrial manufacturing, environmental engineering and circuit board design. But complex, turbulent and unsteady flow CFD simulations still can require months to be completed. There are several options available to decrease the simulation time. Better hardware (i.e., getting more powerful processors) that can carry out the calculations at a faster rate, or more processors can be added to a existing cluster. If these are not possible due to availability or budget constraints, another option is the optimization of the existing CFD code. Usually when codes are written for the first time, the programmer’s main
goal is to get an accurate solution to flow problems. The programmer is not as concerned with the efficiency of the code. Hence there is usually a large scope for improving the performance of the code. By conducting certain performance tests, the efficiency of the code can be quantified. Based on this, a decision can be taken to see if it is justifiable to spend time on optimizing the code. Code optimization can have a two pronged approach; rewriting the code so as to perform in an efficient manner by modifying the basic framework of the code or optimize the code by making modifications to the existing code without changing the basic framework. In this work the last approach has been chosen since it was felt that if the various techniques that would be studied were successful, they could be applied to other similar CFD codes.

1.2 BACKGROUND - MEMORY HIERARCHY

Most of the work that has been carried out in the area of CFD code optimization has been with respect to the improvement of the code performance in parallel. But it has been found that to improve the overall performance of a code, the single CPU performance needs to be improved [3,4]. In order to better understand how the application of the various optimization techniques to the code improves the single CPU performance, we need to have a basic understanding of how computer memory is organized.

All computer systems consist of at least these two parts: a processing unit and memory [5]. At the dawn of electronic computers in 1940, they had two levels of memory storage system: the main memory now known as RAM and secondary memory now known as disks. Processors at that time could only work on the data present on the main memory [6]. In 1961 a group at Manchester, England introduced the concept of virtual memory. This gave the programmer the illusion that he had access to an extremely large main memory even though the computer actually had relatively small main memory [7]. They came up with an algorithm that would move the information that was not currently being used, back into the secondary memory. All this was carried out by the operating system. This concept was widely used in most of the operating systems in the 1960s. In 1965 Maurice Wilkes proposed the “slave memory”, which was a small fast access storage device on the processor to hold a small amount of the instructions and data most recently used by the processor. This was later called “Cache Memory” in 1968 [6]. All this was done to make it easy for the programmers to write programs without worrying about the amount of main memory available. Also during those days the processing unit ran at clock cycles (viz., the time taken by the processor to execute a single simple instruction) comparable to the time required to access information from the memory. Hence the process was delayed while
waiting for the data to reach them from the memory. But as processor speed increased, it became starved for memory bandwidth because the memory access speed was not increasing at the same pace. Fast-access storage was extremely expensive; hence the computer memories of very large overall capacity were eventually organized hierarchically. In modern day computers the memory is organized as shown in the Figure 1-1. This was done to reduce the latencies, viz., the amount of time it takes for the requested data to get to the processor after being called. As we move down the hierarchy, memory size and latency increase.

![Figure 1-1 Memory Hierarchy in Modern Day Computer](image)

A register is a memory storage area which is located within the CPU as illustrated in
Figure 1-2. The processor can only work on the data available in the register. Hence the memory access from the register is extremely fast. It is usually on the order of 1 clock cycle. All the data and instructions required to perform the necessary operations on the CPU have to be stored on the registers. Registers are classified as Instruction Registers and Data Registers. Instructional Registers hold the instructions required for performing an operation, while Data Registers hold the data upon which the operations need to be performed [9]. If the processor requires data not present in the registers, it first looks into the cache.

A cache is a small subset of the main memory that contains a copy of the data that has been recently requested by the processor or the data which is anticipated to be requested by the processor in the near future. Hence when the processor requires any data, it first looks into the cache to see if the required data is available there. If the data is not available in the cache then it moves on to the main memory.

In a typical personal computer the cache is broken down into two levels [10]. The first level known as the L1 cache is located on the processor. It is the smallest memory holding device present in the memory hierarchy. Typical amounts of memory capacity are 4Kb to 32Kb. The processor can access the data present in the L1 cache in about 10 nanoseconds. The second level of cache, known as the L2 cache, is located close to the processor. It is larger compared to the L1
cache varying between 128Kb – 2Mb. The processor can access the data present in the L2 cache in about 20 nanoseconds. If the data present in the L1 cache is also present in the L2 cache, they are called inclusive (e.g., Intel Pentium 2, 3 and 4). If the data is present at most in either the L1 or the L2 cache, they are called exclusive (e.g., AMD Athlon) [11]. The exclusive caches can hold more data compared to inclusive caches. But the downside is the penalty due to time spent in transferring data from L2 to L1. This leads to an increase in latency. In the case of inclusive design, data from L2 is directly written on L1 by deleting some part of the data already present.

Data within the cache are stored in cache lines. A cache line holds the contents of a contiguous block of main memory. Cache line refers to the smallest amount of data transferred between the main memory and the cache by a cache-line fill or write back operation. Cache line size is 32 bytes. This means that even if the CPU requests just 1 byte of information contained in a particular cache line, it will get whatever other information is contained within that cache line. The reason it is designed this way is to take advantage of the principle called Spatial Locality [10], which states that the code which is together is more likely to be executed together.

The size of the cache is usually on the order of a few kilobytes, which means it cannot hold a lot of data compared to the hard disk which is usually on the order of gigabytes. Yet the cache helps in increasing the speed of the program execution due to the principle of Locality of Reference which states “Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends around 90% of its execution time in only about 10% of the code.” [12] Hence if the processor is only using 10% of the code most of the time, we could maintain that information close to the processor so that it can access the information the faster. This is especially true in numerically intensive codes.

If the processor finds the data it requires in the cache it is referred to as a cache hit. If the data is not present in the cache it looks in the main memory. This is referred to as a cache miss. Henessy & Patterson [13] have classified cache misses into three categories:

- **Compulsory cache misses**: These occur when the cache line has to be brought into the cache when it is accessed for the first time. They are unavoidable.

- **Capacity cache misses**: They are related to the limited size of the cache preventing all the necessary data to be in the cache simultaneously. New data brought into the cache may have to overwrite old entries.
• **Conflict cache misses:** The cache is designed in such a way that certain rules known as mapping rules have to be followed before placing data into the cache. Due to this the effective cache size is usually smaller than the physical cache size. The effective cache size would be equal to the physical cache size only if the data from the main memory could be put into any location in the cache. Since this is not the case, unoccupied cache line slots will frequently be found in the cache, thus reducing the effective cache size. This leads to conflict cache misses. The extreme case, called *cache thrashing*, is when most of the physical cache space is not available because of these mapping rules.

A cache miss leads to a reduction in the efficiency of the code significantly. When the processor is unable to find the necessary data in the cache, it has to go look for it in the main memory or random access memory [RAM]. This leads to a latency of around 60 nanoseconds.

The RAM is another kind of data storage used in the computer and is present between the cache and the hard drive. It can be thought of as a larger and slower cache which allows random access to the data that is stored on it. Similar to the cache, RAM loses its data when the computer is switched off. The typical characteristics of all the memory stores discussed up to this point have been summarized in Table 1.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Typical Access Speed</th>
<th>Latency</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>~2 nanoseconds</td>
<td>~ 0 - Cycles</td>
<td>~1b</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>~10 nanoseconds</td>
<td>~1 – Cycle</td>
<td>~ 4 KB – 256 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>~20 – 30 nanoseconds</td>
<td>~ 10 – Cycles</td>
<td>~ 128KB – 4 MB</td>
</tr>
<tr>
<td>RAM</td>
<td>~60 nanoseconds</td>
<td>~ 100 – Cycles</td>
<td>~ 128 MB – 4GB</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>~10 milliseconds</td>
<td>-</td>
<td>~ 20GB – 500 GB</td>
</tr>
</tbody>
</table>

### 1.3 INTRODUCTION TO PROBLEM

The field of CFD has been one of the key fields which have been driving the computer hardware industry to build faster processors. The processing speed of a CPU has historically increased at the rate of about 55% per year, whereas the main memory access speed has increased at a rate of only 7% per year [13]. This means that the gap in the speed to access the memory increased at about 45% per year [5]. If the processor had to get all the information directly from the main memory, it would take excessively long to perform a simple operation because it takes a lot of time to get the information from the main memory. Hence if a computer
code is to run efficiently, the data required by the processor must be available to it with the least amount of time lag. This problem was overcome by breaking up the memory into a hierarchy, as discussed in the previous section. Usually, there is a small and expensive high speed memory sitting at the top of the hierarchy. As we move away from the CPU, the layers of the memory keep getting bigger and slower. All the layers in the memory hierarchy are the subsets of the subsequent lower layer. This is done in order to improve the speed of accessing frequently used data. An exception to this is the exclusive cache design which was discussed earlier. For efficient program execution, the codes should be designed in such a way that they make efficient use of the memory hierarchy. Present-day compilers can perform certain simple code optimizations, but they are not sophisticated enough to change the codes so that they can make use of the memory hierarchy. It was noticed by Beyls et al [14] that the processor stalled on data memory access for almost 50% of the execution time for the SPEC2000 programs which were compiled with the highest level of optimization present in Intel’s state-of-the-art compiler. Hence most of this work is left to the programmer. This is particularly true with respect to numerically intensive codes [15]. These codes require a lot of data to be processed i.e. they repeat the same set of calculations over a very large data set. Hence if the code is designed in such a way that the data required by the CPU in the near future is already present in the top layers of the memory hierarchy, the performance of the code will be enhanced greatly. This is where code optimization to enhance cache performance comes into the picture. As discussed in the previous section, a cache miss is a relatively costly affair. In optimizing the code, we make modifications to the code such that the data required by the CPU for processing is available in the cache, thereby reducing the number of cache misses. In this work, we have made use of an optimization technique known as Grid Blocking for enhancing the cache performance.

1.4 GOALS OF OPTIMIZING THE CODE

The main purpose of this work is straightforward; make a CFD code run faster by optimizing the cache usage, so that the user of the code gets accurate results to the problem they are solving in the least amount of time. We intend to do this by using techniques which are code independent i.e. techniques which can be applied to different CFD codes.

Every CFD code spends time in performing the following two operations; numerically solving the conservative flow equations and input and output of data. The focus of this work is improving the performance of the “solver” portion of the CFD job. Usually a CFD job is not I/O intensive, unless the problem being solved is an unsteady problem and the user requires the
results at repeatedly short time intervals [17]. Generally the solver portion is rather computationally intensive. A focus on the reduction in solver time will lead to a greater decrease in the wall clock time (viz. the amount of time that passes if you are looking at a clock on the wall for the code to finish solving a problem) of the code.

Apart from decreasing the wall clock time, a few other aspects of code performance are considered. Code modification must maintain the accuracy of the code, since it does not make much sense to improve the performance of the code only to find out that the code no longer gives an accurate solution or, even worse, gives a completely wrong solution. Another aspect is to make sure that improved performances are repeatable, confirming that we get the same performance every time the code is run and for different kinds of problems. Getting good performance just once or for only one kind of a problem is often of limited use.

Finally, while gaining performance improvement is the main goal of this work, we want to study and see what price in terms of manpower and time is required to get this improvement. There may be things that could be done which could improve the performance of the code, but if the effort required in implementing them is a highly time consuming or expensive then it may not be worthwhile for others to try it.

1.5 PREVIOUS WORK

Since the optimum use of cache is critical to the efficient running of programs, thousands of research papers have discussed cache optimizations in the last forty to forty five years [18]. The proposed optimizations range from hardware modifications to compiler based optimizations and changes to operating systems to improvements to existing algorithms. We will first consider a few of these cache miss optimization techniques. Later we will examine the optimization work that has been carried out specifically on CFD Codes.

1.5.1 General Cache Optimization Techniques

Most of the program optimizations done to improve cache performance can largely be classified into one of four categories [5]: (1) reduce capacity misses by maximizing reuse of the data present in the cache; (2) reducing conflict misses by improving data layout or changing computation order; (3) hiding memory latency by performing parallel computations; (4) modifying the rules used to replace the data in the cache.
1.5.1.1 Techniques for Reducing Capacity Misses

The idea behind optimizations carried out to reduce capacity misses is as follows. Assume that N computations have to be carried out on all the data elements in the program. If one useful computation is carried out on the data element when it is present in the cache, each data element will have to be loaded onto the cache N times (assuming they are not on the same cache line). Which means possibly N cache misses due to the data element. Now if we can modify the program in such a way that it carries out two useful computations on the data element every time it is present in the cache, we cut down the cache miss frequency by half i.e., N/2. Another way to look at this is that we are trying to hold on to the data in the cache for a longer time so that all the necessary calculations in which the data is required is carried out before it leaves the cache.

Some of the techniques that use this principle are mentioned below:

- **Loop Interchange**: This transformation reverses the order of two adjacent loops in a loop nest [4,19]. Generally speaking, loop interchange can be applied if the order of the loop execution is unimportant [15]. This technique has been illustrated in Figure 1-3. According to Fortran convention, the physical ordering of data in the memory is row wise rather than column wise. Hence by changing the sequence of loop execution, the code traverses row wise. The data pertaining to adjacent cells in a single row will be stored in a cache line [Figure 1-3]; hence the use of the data available in the cache is maximized and the number of data calls is reduced.

```
!Original Code
Dimension a(8,6), b(8,6)
Loop A
Do i=1,8
  Do j=1,6
    a(i,j) = b(i,j)
  End Do
End Do

! After loop interchange
Dimension a(8,6), b(8,6)
Loop A
Do j=1,6
  Do i=1,8
    a(i,j) = b(i,j)
  End Do
End Do
```
• **Loop Fusion:** This transformation takes two adjacent loops that have the same iteration space and combines their bodies into a single loop as shown in Table 1-2 [15, 20]. We can carry out loop fusion as long as output dependencies in the fused loop exist for which instructions from the first loop depend on instructions from the second loop [21]. Loop fusions tend to reduce unnecessary memory references [4,19,22,23,24,25,26,27,28]. Loop fusion is carried out by most compilers at the highest level of optimization [36].

Table 1-2  Illustration of loop fusion

<table>
<thead>
<tr>
<th>Original Code</th>
<th>After loop fusion</th>
</tr>
</thead>
</table>
| \( \text{Do } I = 1, n \)  \
  \( b(i) = a(i) + 1.0 \)  \
  End do  \
| \( \text{Do } I = 1, n \)  \
  \( b(i) = a(i) + 1.0 \)  \
  \( c(i) = b(i) \times 4.0 \)  \
  End do |

• **Loop Blocking:** This technique, also referred to as loop tiling [29,30,31,32,33,34,35], tends to increase the depth of a loop nest with depth \( n \) by adding additional loops to the loop nest [15]. This is illustrated in Figure 1-4. As mentioned earlier that the cache works based on the Principle of Locality of reference, the data pertaining to points surrounding a cell will be saved in the cache. Since the code traverses through the blocks, most of the data necessary will be stored in the cache in advance. This helps to improve performance by reducing cache misses.
- **Data/Computational Reordering**: This involves two approaches; Data Reordering involves changing the location of the elements of the data but not the order in which these elements are referenced [37]. For example, first touch data reordering and space filling curve data reordering. Computational Reordering involves changing the order in which data elements are referenced, but not the locations in which these data elements are stored. For example, space filling curve computational reordering and computational reordering by blocking. These techniques help in the same manner as loop blocking.

1.5.1.2 Techniques for Reducing Conflict Misses

If a block of data is evicted from the cache to make way for new data even if there is still space, but it is inaccessible because of the mapping rules and if this evicted data is required in the very next access or in the near future it leads to a conflict miss. In order to reduce conflict misses the code has to be structured in such a way that the data it accesses are evenly distributed in the cache.

![Original Code](image)

- **After loop blocking**

![After loop blocking](image)

Figure 1-4 Illustration of Loop Blocking

- **Array Padding**: This transformation involves in improving the programs data layout [38,39,40]. This is done by increasing the array dimensions to make sure that array elements
in the same column or row are more evenly distributed over all cache sets [5]. A simple example of how this can be implemented is shown in Table 1.3.

Table 1-3 Array Padding [15]

<table>
<thead>
<tr>
<th>Original Code</th>
<th>After applying array padding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double a[1024] double b[1024] for i = 1 to 1023 sum+ = a[i] * b[i];</td>
<td>Double a[1024] double pad[x] double b[1024] for i = 1 to 1023 sum+ = a[i] * b[i];</td>
</tr>
</tbody>
</table>

- **Data Copying**: Research has shown that [34,41,42] in certain cases loop blocking leads to a high degree of conflict misses. One way to avoid this is by the using the technique of data copying, where non-contiguous data from a block are copied into a contiguous area of memory [34]. Another technique that has been suggested is the use of hierarchical array layouts [42].

1.5.1.3 Techniques to Hide Effects of Cache Misses

Accessing a data item for the first time leads to a compulsory cache miss. Such misses cannot be avoided, but the delay caused due to such misses can be hidden. Techniques to do this have to be applied by the compiler or by modifying the hardware.

- **Prefetching**: Usually due to the high processing power and lower bandwidth the data cannot consistently reach the processor fast enough. When it is known in advance that a data element is going to be used, it can be requested in advance from the main memory and stored in the cache, ready to be used. Usually this technique is implemented in compilers because prefetching can be implemented using machine language [43,44,45]. This can also be implemented directly by modifying the compiled code executable.

- **Multithreading**: When a cache miss occurs the processor has to wait for the requested data to arrive from the main memory. While waiting it can go ahead with the data available in the cache and once the requested data reaches the cache, it can perform the previous task. In other words the processor can switch to another thread of execution. Many Multithreaded
processor can switch threads on every cycle [46]. To best hide the latency as many threads as possible should be created [5].

1.5.1.4 Techniques to Improve the Replacement Decisions by Cache

Usually processors use the *LRU* replacement policy (Least Recently Used) to decide which cache line needs to be evicted to make way for new data [47]. This is not always the best method. An adaptive replacement policy has been proposed wherein the compiler annotates hint bits on its instructions and these bits are used to adapt the replacement policy [48,49,50]. Another technique that has been suggested is the use of a “Victim Cache” which is a cache that holds data that has been evicted due to a cache miss to make way for the new data [51,52].

1.5.2 Optimizations to CFD Codes

The optimizations that have been carried out in this area can be classified into two broad classes: (1) Techniques to improve parallel performance and (2) Techniques to improve single node performance.

1.5.2.1 Techniques to Improve Parallel Performance:

With the current hardware technology available an economical way to increase processing power is by performing the calculations in parallel on computers which consist of several processors working concurrently. The cost of such hardware is decreasing compared to single processor systems of similar processing power. But in order for the codes to work efficiently in parallel the algorithm on which they are based should have a high degree of parallelism and low cost of communications [53]. Hence most of the work in this area comprises of designing new algorithms or modify existing algorithms to achieve these two goals.

- Often the equations solved in CFD for steady flow comprise of tri-diagonal systems. Intensive research has been carried out on the development of algorithms for efficient parallel tri-diagonal solvers [53,54,55].
- Another method used in parallelizing the code is to break up the grids into blocks, solving each block on different nodes by using MPI to communicate between nodes. Domain decomposition [56,57,58] and Chimera or Overset grid approach [59] are based on this principle.
1.5.2.2 Techniques to Improve Single Node Performance:

The basic principle behind improving the single node performance of a CFD code involves the improvement of its cache behavior. This has been mainly achieved by using the techniques mentioned in the previous section.

Kadambi et al. [57] studied an algorithm to solve compressible Euler equations with regard to temporal and spatial access of data. They optimized the code by using loop interchange, reallocation of data spaces and loop fusion. They achieved a performance 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.

Douglas et al. [61] have introduced a number of algorithms to solve elliptic boundary value problems using cache memories in a much more efficient manner than usual. They have applied these algorithms to solve a collection of problems on structured grids (2D and 3D) and unstructured grids (in 2D). They obtained speedup ranging from 100% to 300% over using standard, well coded implementations [62].

Hauser et al. [63] optimized several computationally expensive subroutines in their CFD code DNSTool by using array of structures to group multiple fundamental variables together in a cache friendly manner instead of using separate array for each variable. They also applied loop interchange. In a later publication [64] they partially extended this work to two other codes LESTool and OVERFLOW.

Gropp et al. [65,66] applied three simple techniques to the CFD code FUN3D, the first was: Interlacing, which leads to the high reuse of data brought into the cache, makes memory references closely spaced, and decreases the size of the working set of the data cache. The second was structural blocking, which lead to a significant reduction in the number of integer loads and enhanced the reuse of data in the registers. The last technique was edge and node reordering; which lead to a decrease in the TLB misses (viz. a kind of cache miss) by an order of two magnitude and decrease in the L2 miss by a factor of 3.5. The combination of the three techniques led to an overall improvement in the execution time by a factor of 5.7.

LeBeau et al. [4] tuned the 2-D structured CFD code GHOST by using loop fusion, loop interchange, and using array of structures. Due to the application of these techniques, the L2 cache miss rate dropped to less than 0.1% and also improved the codes walltime performance by close to a factor of 6 for larger grid sizes (> 250,000).

Gupta et al. [4,67] carried out a comprehensive study of the effects of application of various cache optimizing techniques to the 3-D unstructured CFD code UNCLE. They applied
space filling curve, loop blocking and optimized data access. An overall improvement of 50% in walltime was obtained from the application of these techniques.

1.6 EXTERNAL & INTERNAL BLOCKING

External and Internal blocking are two cache optimization techniques that help in reducing capacity misses in caches. The basic philosophy behind these two techniques is quite similar to that of loop blocking, but here the subroutine or the entire code is broken into blocks that readily fit into cache. External blocking is essentially the same as dividing the grid for parallel processing, but with the division requiring many more and much smaller subgrids than is typically demanded by the geometry, as shown in Figure 1-5. As such, this approach is as much a function of the grid generation process as it is of the CFD computation. In Internal Blocking instead of breaking up the grid externally during the grid generation process we do this internally once the grid is read by breaking up the arrays holding the grid data. This approach includes extra coding and memory overhead but is more opaque to the user. This technique is usually applied only to the costliest subroutines in terms of computational time.

In this work, the performance effects due to the application of these two optimization techniques to the structured 2D CFD code GHOST have been studied in detail. The optimization process has primarily been evaluated on steady laminar flows with relatively simple geometries, as these problems allow for easy manipulation and cover the primary central routines that form the bulk of most computations. As these simulations are by design not as complicated, we have later conducted an initial evaluation of how well the optimized codes translate to established flow problems that require increased modeling, complicated boundary conditions, and more routines.
Figure 1-5 Schematic illustrating External Blocking

Copyright © Anand B Palki, 2006
CHAPTER - 2

2. COMPUTATIONAL TOOLS

This chapter presents a comprehensive description of the computational tools and platforms that have been used in this study. It begins with a discussion of the numerics involved in the CFD code GHOST. It is followed by a discourse about contents of the grid file that is used by GHOST. A brief description of the cache simulator tool ‘Valgrind’ follows. This tool has been used extensively to assess the cache performance of the codes. The computational hardware architecture employed in this study has also been described in detail. The concluding part of this section contains a discussion about the method used to gauge or measure the performance improvements obtained by the application of the various techniques.

2.1 GENERAL DESCRIPTION OF GHOST

GHOST is a well established CFD solver. It has been used to carry out a number of published analyses of transitional turbo machinery flows and active flow control [68,69,70,71,72,73]. It is a two-dimensional incompressible finite-volume structured computational fluid dynamics code with chimera overset grids for parallel computing. The QUICK scheme is applied to discretize the advective terms in the momentum equations with second-order accuracy. A second-order central difference scheme is used for the diffusive terms. For the RANS turbulence equations, the Total Variation Diminishing (TVD) scheme is employed for the advective terms. Interfacial fluxes are determined through interpolation of cell-centered values. Second order upwind time discretization is employed for the temporal terms, using a delta form subiterative scheme. GHOST is written in FORTRAN90 and has been ported to a wide variety of platforms. GHOST was also originally designed to minimize memory usage, accomplished through extensive use of the allocation and de-allocation of variables in FORTRAN90.

GHOST uses a cell-centered partitioning approach, and the internode communication protocol is MPI. GHOST has mechanisms to do a form of automatic load balancing, but this is unnecessary for simple test geometries.

Flow and geometry data in GHOST for a given grid or subgrid are stored in individual arrays, as in $\phi_1(i,j)$, $\phi_2(i,j)$, ... $\phi_n(i,j)$. On a given grid, GHOST performs the majority of its calculations as a series of i,j bi-directional sweeps in nested double loops. In brief, the
momentum equations are solved implicitly in a delta form, shown here for the time discretization in one dimension:

\[
\frac{3(\Delta \phi)^n}{2\Delta t} + \frac{\partial f(\Delta \phi)^n}{\partial x} = \frac{(\phi^n - \phi^{n-1})}{2\Delta t} - \frac{3((\phi^{n+1})^m - \phi^n)}{2\Delta t} - \frac{\partial f((\phi^{n+1})^m)}{\partial x}
\] (2-1)

where \( \phi \) represents any variable, \( m \) is the subiteration level, and \( n \) is the time iteration level. The right-hand side of Eq. (2-1) 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 \( \phi^{n+1} \) is obtained by using inner iterations to reach the convergent solution of the right-hand side of Eq. (2-1), corresponding to \( \Delta \phi \) approaching zero. At least one subiteration is performed at every time step so that this method is fully implicit.

The resulting matrices generated at each subiteration based on the QUICK and TVD schemes as well as evaluation of source/sink terms are solved with an ADI-type decomposition into a pair of sweeps alternately in the \( i \)- and \( j \)-directions which are solved sequentially in tridiagonal matrices. This sequence may be repeated for improved accuracy. The techniques of Rhie and Chow [74] are then used to extract the pressure field from the continuity equation.

For clarity, most of the performance testing was conducted on the simplest form of GHOST in which there are no subiterations (the steady-state version of the code), no turbulence model (laminar flow), and only a single pair (one in \( i \), one in \( j \)) of ADI computations is completed per iteration. However, the iterative core of the code is retained even in this simplified version.

### 2.1.1 GHOST Flowchart

In the previous paragraph the working of the GHOST core was discussed in a fair amount of detail. But as mentioned in the introduction, the aim of this project is to optimize the code without modifying the underlying basic framework. In order to do that, a better understanding of the bigger picture of the working of GHOST is necessary. The flowchart of GHOST has been prepared [Figure 2-1] in order to understand this. A description of the flowchart is provided in the following paragraph. The subroutines which perform the operations described are mentioned in brackets.

The code starts by finding out how many grid files need to be read and how many processors have to be used in solving the problem (\textit{read_map}). This information is provided to the code by using a file titled “\textit{mpi.in}”. It then reads the grid files (\textit{read_data}). A detailed
description of the contents of these grid files is given in the successive paragraphs. The various
flow field variables and the boundary conditions are initialized next. In case the grids are located
on different processors, the boundary conditions are communicated between the grids by using
MPI broadcasts. Before the code starts the calculations, it checks to see if there are any restart
files present (read_restart). These files mainly contain the u, v and p values at each of the grid
points from the previous run. All the flow calculations that were mentioned in the previous
paragraph are carried out in the subroutine called calc_flowfield. This subroutine initially
calculates the variables which are necessary to solve the momentum and continuity equation
(cal_properties). It then goes ahead and calculates the u velocity by solving the x-momentum
equation (cal_u). This subroutine initially applies the QUICK scheme and calculates the required
parameters (quick). It then solves the tri-diagonal matrix that is formed using TDMA method
tdma). In a similar manner, the y-momentum equation is solved to get the v velocity field
(cal_v, quick, tdma). Once the x and y momentum equation have been solved and the velocity
field has been obtained, the pressure field is extracted from the continuity equation using the
Rhie and Chow technique [74] (cont, quick, tdma). If the energy equation is switched on then an
additional subroutine is used to calculate the temperature field (cal_t). If the turbulence model is
switched on, two additional subroutines are used to calculate the eddy viscosity (cal_tk, cal_ed).
Once all the required flow field variables have been calculated, the code broadcasts these values
to all the processors since they are required to calculate the values at the boundary points and to
update them. After the values at boundaries points are calculated, the code once again broadcasts
these newly calculated values. It then calculates the residuals and prints them. If the solution is
converged it goes ahead and writes the solution to a file, if not it starts off with another iteration.

2.1.2 Governing equations

The governing equations for unsteady incompressible viscous flow under the assumption
of no body force and heat transfer that are used to calculate the various flowfield parameters in
GHOST are as below

Conservation of Mass

\[ \frac{\partial}{\partial t} \int \rho \, dV = - \int_{S} \rho u_{i} n_{i} \, dS \]  \hspace{1cm} (2-2)

Conservation of Momentum

\[ \frac{\partial}{\partial t} \int \rho u_{j} \, dV = - \int_{S} \rho u_{i} u_{j} n_{i} \, dS - \int_{S} p n_{i} \, dS + \int_{S} \tau_{ij} n_{i} \, dS \]  \hspace{1cm} (2-3)
Read Grid Files and load arrays with grid data

Find out number of zones

Read Map

Read Data

Initialize initial values and boundary conditions

Init_flowfield
Update_real_bc
Update_ghost_bc

MPI Broadcasts

Read Restart Files if any

Read_restart

Iter = Iter + 1

Izone = Izone + 1

Calculate u,v,p

Calc_flowfield,
Cal_property,
Cal_u, Cal_v, Cont,
Tdma, Quick

Update Boundary Conditions

MPI Broadcasts

Is Izone > Nzone ?

YES

More MPI Broadcasts ...

Print residuals

Is Iter > Maxit ?

YES

Write solution & restart files

END

Figure 2-1 Flowchart depicting the working of GHOST
Table 2-1 Summary of the subroutines in which the main iterative calculations of GHOST are carried out.

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>cal_properties</td>
<td>Computes the values of variables necessary to solve governing equations.</td>
</tr>
<tr>
<td>Cal_u</td>
<td>Solves the x-momentum equation to calculate the $u$ - velocity field.</td>
</tr>
<tr>
<td>Cal_v</td>
<td>Solves the y-momentum equation to calculate the $v$ - velocity field.</td>
</tr>
<tr>
<td>cont</td>
<td>Extracts pressure field from continuity equation.</td>
</tr>
<tr>
<td>tdma</td>
<td>Tri diagonal matrix solver</td>
</tr>
<tr>
<td>quick</td>
<td>Implements the QUICK scheme to determine the advective fluxes.</td>
</tr>
<tr>
<td>cal_t</td>
<td>Solves the energy equation to calculate the temperature field.</td>
</tr>
<tr>
<td>Cal_tk</td>
<td>Computes the turbulent kinetic energy (k)</td>
</tr>
<tr>
<td>Cal_ed</td>
<td>Computes the energy dissipation rate ($\varepsilon$)</td>
</tr>
</tbody>
</table>

Conservation of Energy

\[
\frac{\partial}{\partial t} \int_V \rho EdV = -\oint_S \rho u_i n_i EdS - \oint_S p u_j n_j dS + \oint_S u_j \tau_{ij} n_i dS
\]

(2-4)

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

2.1.3 Calculation at artificial boundaries

GHOST uses chimera overset grid method to carry out parallel computations. Let us briefly look at the technique that is used to carry out the calculation at the artificial boundaries which are formed due to the breaking up of the grids into smaller blocks for parallel computations.

Figure 2-2 shows a grid that has been split into two halves for the sake of performing parallel computations. A magnified view of the region of overlap is shown below the actual grid. As can be noticed in the Figure 2-2, four grid points from each zone are overlapped. The last two overlapped grid points for each zone are refereed to as “Ghost Points”. No calculations are carried out at the Ghost Points. The number of Ghost Points required depends upon the order of accuracy of the code. GHOST is second order accurate; so it uses information from two grid...
points in each direction surrounding the grid point while calculating the diffusive and convective fluxes. Hence it requires two Ghost Points at the artificial boundaries.

Assume that each of these zones is located on two separate nodes of a parallel computer. For the first iteration, the code performs the calculation on each of the nodes simultaneously using the initial values and real boundary conditions. It is important to note here that there is no transfer of data between the nodes during this stage (This capability is used to implement internal blocking). At the end of the first iteration, using MPI communication, the boundary information is transferred between the nodes. For a laminar case with no heat generation, the values of velocities \( (u & v) \) and the pressure \( (p) \) are swapped between the nodes. The values at the boundary points of zone – 1, \( i.e. W_{n-1} \) and \( W_{n+1} \) are passed on to the ghost points of zone – 2, \( i.e. E_0 \) and \( E_1 \) and same is done between the ghost points of zone – 1 and boundary points of zone – 2. During the second iteration, the ghost point values are used to calculate the values at the boundary points. The same process is followed at the end of each iteration.

2.2 GRID FILE DATA

2.2.1 Finite Volume Method

GHOST uses the finite volume method to solve the governing flow equations. A basic understanding of this technique is necessary to understand the importance of the contents of the grid files. In the finite volume method, each computational cell makes up a control volume on which the governing integral equations shown above are enforced. A typical control volume (CV), along with the notations is shown in the Figure 2-3. The CV consists of four faces, denoted by lower-case letters (n, e, w, s) corresponding to their location with respect to the central node C. Adjacent nodes are denoted by upper case letters (\( i.e. N, E, W, S, \) etc). The values of the flow variables are calculated and stored at the cell centers \( i.e. \) nodes. The vertices around the central node C are denoted by lower-case letters (ne, nw, sw, se). The values of the flow variables at the vertices are calculated by taking the weighted average of the values at the nodes surrounding the vertex.

2.2.2 Generalized coordinates

The GHOST solver has been developed in generalized coordinates; hence it is capable of simulating the flow even when the grid is non uniform or curved as shown in Figure 2-4.
Figure 2-2 Illustration of the artificial boundaries
All the quantities are converted from the Cartesian coordinates \((x, y)\) to the generalized coordinates \((\eta, \xi)\) using transformations. The transformations used to convert the quantities for the east face \(i.e.\) the line (or face) joining the vertices \(se\) and \(ne\) [Figure 2-5] is given below. The subscripts denote the points in the illustration.

First transform \(x\) & \(y\) into \(\eta\) & \(\xi\),

\[
\begin{align*}
\eta_x &= x_{ne} - x_{se}, \quad y_\eta = y_{ne} - y_{se} \\
\eta_\xi &= x_E - x_C, \quad y_\xi = y_E - y_C
\end{align*}
\]

The area of the face can be obtained by,

\[
A = ds = \sqrt{x_\eta^2 + y_\eta^2}
\]
Then using these values the normal vector across the face is calculated

\[
\vec{n} = \left( -\frac{y_\eta}{\sqrt{x_\eta^2 + y_\eta^2}}, \frac{x_\eta}{\sqrt{x_\eta^2 + y_\eta^2}} \right)
\]

In order to convert the various differentials e.g. \( \frac{\partial T}{\partial x} \) and \( \frac{\partial T}{\partial y} \) into \( \frac{\partial T}{\partial \eta} \) and \( \frac{\partial T}{\partial \xi} \) the following transformations are used:

\[
\begin{bmatrix}
\frac{\partial \phi}{\partial x} \\
\frac{\partial \phi}{\partial y}
\end{bmatrix} = \begin{bmatrix}
\xi_x & \eta_x \\
\xi_y & \eta_y
\end{bmatrix}
\begin{bmatrix}
\frac{\partial \phi}{\partial \eta} \\
\frac{\partial \phi}{\partial \xi}
\end{bmatrix} = \begin{bmatrix}
\xi_x & \eta_x \\
\xi_y & \eta_y
\end{bmatrix}
\begin{bmatrix}
\phi_2 - \phi_1 \\
\phi_2 - \phi_A
\end{bmatrix}
\]

where,

\[
\begin{bmatrix}
\xi_x & \eta_x \\
\xi_y & \eta_y
\end{bmatrix} = \begin{bmatrix}
x_{\xi} & y_{\xi} \\
x_{\eta} & y_{\eta}
\end{bmatrix}^{-1} = \frac{1}{D} \begin{bmatrix}
y_\eta & -y_{\xi} \\
x_{\eta} & x_{\xi}
\end{bmatrix}
\]

GHOST carries out all the calculations in the (\( \eta, \xi \)) coordinate system. The grid that is used will contain all the parameters in this coordinate system and the mapping functions.

### 2.2.3 Description of G.F90 Output

Since GHOST works off a generalized coordinate system, it requires a lot of grid data apart from just the \( x, y \) co-ordinates of the grid points. This data is generated by the code g.f90. In this section we briefly describe the grid data that is generated by the code. This information is critical in understanding the implementation of internal blocking.

The contents of the grid file for a non moving grid are as follows:

- Number of grid points in the \( x \) and \( y \) direction.
- Number of ghost points.
- Grid point weight in the \( x \) and \( y \) direction
- \( x \) & \( y \) co-ordinates of the grid points.
- Volume of the cell surrounding each grid point.
- Distance between the wall and the grid point.
- Values for the various transformation functions such as \( \eta_x, \eta_y, \xi_x \) and \( \xi_y \).
• A variable called “inx” which specifies if a particular grid point is a ghost point or not. If the value of inx for a grid point is 1, then that particular grid point is treated as a ghost point, whereas if its zero, it is treated as a normal point.

• Boundary conditions

  GHOST reads all these flow and geometry data of the computational domain from the grid files and stores them in a single structure consisting of various arrays for each of the above mentioned data.

2.2.4 Description of Input File

  As mentioned in the previous section, g.f90 is used to generate the grid data required by GHOST. In order for g.f90 to generate a grid it requires certain data regarding the size of the computational grid, boundary conditions and number of grid points. This data is provided using the file called “Input”. An Input file used to generate the grid is shown in Figure 2-6, with explanations inline. Under the column “Patch Zone Number” the value for the row labeled right is 2, which means that the grid specified as zone 2 is on its right. Similarly in the row labeled left in zone 2, we have specified the value as 1.

2.3 COMPILERS & MPI ENVIRONMENT

  The Intel Fortran Compiler and g95 Fortran compiler were used to compile the code for this work. Since GHOST is an MPI based code, an MPI environment has to be installed on the machines for it to be compiled and run. Hence we used LAM/MPI for this purpose.

  During the initial stages of testing, the code was compiled using the compiler developed by Intel called Intel Fortran Compiler Ver. 7.1 (Ifc). Later when the Intel Fortran Compiler Ver. 9 (Ifort) was released, we started to use that. Ifort had certain features such as [76] run-time array and string bounds checking, cross-file procedure interface checking and run-time uninitialized variable detection, which were not available in the earlier version. It also optimized the code better than Ifc. However there was a memory allocation problem in Ifort. It would lead to a segmentation fault due to a stack overflow when the code was run on grid sizes greater than 100,000. The workaround suggested for this problem was to set the stack size as unlimited by using the command:

  \texttt{ulimit -s unlimited}

This allowed us to go up to a grid size of around 150,000.
/*number_of_zone
2
/*zone_number 1
quadratic 20 20 4 2 (!Type of grid, Number of point in x direction, y direction,
No. of Boundary Conditions, Ghost Points)

(! The other options available for Type of grid are: grid, curv, circ and quad.)
0.0 0.0 (!Center co-ordinates)
1.5 0.5 0.5 1.5 (!X-co-ordinates of corners of the grid)
0.5 -0.5 0.5 0.5 (!Y-co-ordinates of corners of the grid)
0. 0. 0. 0. (!Wall co-ordinates)
0.99 1 (!Ratio to specify the grid density.
If < 1 (Eg. 0.89) then the grid density INCREASES from left to right,
If > 1 (Eg. 1.04) then the grid density DECREASES from left to right.)

(! Boundary Conditions
!Type of Boundary, Relative Position, Patch zone number)
m inlet left 1 1 1 99999 0
* patch right 99999 99999 1 99999 1
* patch bottom -99999 100000 1 1 0
* patch top -99999 100000 99999 99999 0

/*zone_number 2
quadratic 20 20 4 2
0.0 0.0
2.5 1.5 1.5 2.5
0.5 -0.5 0.5 0.5
0. 0. 0. 0.
0.99 1

m inlet left 1 1 1 99999 2
* patch right 99999 99999 1 99999 0
* patch bottom -99999 100000 1 1 0
* patch top -99999 100000 99999 99999 0

/*end

Figure 2-6 Description of a Input File
G95 [77] is a free, open source fortran 95 compiler. Work on this compiler began in the year 2000. We shifted to this compiler due to the above mentioned bug in the Ifort compiler. The bug was also present in the g95 compiler but the code would work fine when the stack size was set to unlimited. It allowed us to carry out test for grid sizes up to 360,000.

LAM/MPI [78] was originally developed at the Ohio Supercomputing Center. Is is a high quality implementation of the Message Passing Interface (MPI) Standard. LAM/MPI provides high performance on a variety of platforms, from small off-the-shelf single CPU clusters to large SMP machines with high speed networks. In addition to high performance LAM provides a number of usability features key to developing large scale MPI applications. In order to compile a MPI based code using a particular compiler, the LAM/MPI platform must be compiled and installed using the same compiler. Hence, every time you want to compile a code with a different compiler, LAM/MPI needs to be compiled. Then when the compiler is invoked, it will have access to the MPI libraries.

2.4 Valgrind [79]

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 track 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 of 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 non-initialization 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 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 to 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, D1, 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 KCachegrind visualization tool ([http://kcachegrind.sourceforge.net](http://kcachegrind.sourceforge.net)) 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 races 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.
2.5 Kentucky Fluid Clusters

In this section we will focus on two different clusters, Kentucky Fluid Clusters 4 and 5 (KFC4, KFC5) [Figure 2-7]. Apart from these we will also discuss the hardware that are being tested to be used for the construction of Kentucky Fluid Cluster 6 (KFC6). The Intel FORTRAN90 compiler (ifort) with -O3 optimization, the G95 compiler also with -O3 optimization and LAM MPI were used for the purpose of compiling GHOST 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 I/O.

Figure 2-7 Kentucky Fluid Clusters (KFC) 4 and 5

Kentucky Fluid Cluster 4 is constructed with AMD Athlon 2500+ 1.826 Ghz 32 bit 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.

Before a cluster is constructed, code performance tests are carried out on the various state of the art processors that are currently available in the market at that time and that are within the budget. Tests are conducted on these processors to study the performance of the in-houses codes GHOST and UNCLE and other codes such as EPIC which are extensively used by the lab members.

Such tests are carried out to pick out the processor to be used for the construction of KFC6. KFC6 is going to be a commodity cluster similar in design to KFC4 and 5. The processors tested are the dual core processors from AMD and Intel. The details of these processors which are relevant to our study are given in the Table 2-2.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock Speed</th>
<th>L1 Cache Size</th>
<th>L2 Cache Size</th>
<th>FSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel E-6400</td>
<td>2.13 GHz</td>
<td>2 X 32 Kb</td>
<td>1 X 2 Mb</td>
<td>1066 MHz</td>
</tr>
<tr>
<td>AMD 4200+</td>
<td>2.2 GHz</td>
<td>2 X 128 Kb</td>
<td>2 X 512 Kb</td>
<td>2000 MHz</td>
</tr>
<tr>
<td>AMD 4600+</td>
<td>2.4 Ghz</td>
<td>2 X 128 Kb</td>
<td>2 X 512 Kb</td>
<td>2000 MHz</td>
</tr>
</tbody>
</table>

2.6 METHOD USED TO MEASURE PERFORMANCE

In the computing world two ways of measuring the time taken by a code are referred to as “wall clock time” and “CPU time” [16]. Wall clock time is the time that passes if you are looking at a clock on the wall for the code to finish a problem. CPU time is the amount of time spent by the CPU in carrying out the calculations. The CPU time excludes time for events such as passing the data across the network, I/O time, CPU interrupt time and processing TCP packets. All these usually affect the total time taken to complete the job. Hence CPU time can miss critical time costs for someone doing CFD runs on parallel systems.

We use wall clock time to measure the code performance improvements in spite of running all our tests on a single node. All our tests have been carried out on clusters that are controlled in-house. Hence it was seen to it that only a single job is running on the node while
carrying out the timing tests. Based on the initial tests that were conducted, it was noticed that the difference between the wall time and CPU time was minimal under these conditions.

Walltime is the sum of the time taken by the code to read the grid files and to write the solution (I/O time) and time taken to complete the calculations (Solver time). Since this work is concerned with the optimization of the solver portion of the job only, the solver time alone was considered while calculating the performance improvements gained.

For a given problem and a code, walltime is a function of the grid size and the number of iterations. If the number of iterations is kept constant and the grid size increased, the wall time will increase too. In order to compare the performance improvements obtained with varying grid sizes, the wall time has been normalized by the grid size and number of iterations. Hence the walltime that has been used to measure the performance improvement is approximately the wall clock time of the code to perform a single iteration on a single grid point. This is further detailed in the next chapter.

2.7 SUMMARY

GHOST is a generalized 2D incompressible structured CFD solver with an ability to perform computations in parallel using MPI environment. The single node performance of the code was found to be poor due to the high cache miss percentage when it was tested on the in-house clusters. Work was started at reducing the cache misses by applying various techniques which were proven to reduce the cache miss on various non-CFD codes. The next chapter contains a discussion about the theory, applications and results obtained by the implementation of one of these techniques (External Blocking) on GHOST to improve cache performance.
CHAPTER – 3

3. EXTERNAL BLOCKING RESULTS

This chapter provides a detailed description of the methodology used to apply external blocking to the GHOST code. It further presents the results of the detailed study that was conducted to test the efficacy of this technique by solving the lid-driven cavity problem. The study comprises of tests to study the performance improvements obtained, changes in the cache behavior, and the accuracy of the solutions obtained by the application of this technique.

3.1 TERMINOLOGY

Any subject has its own set of jargon. A good example of this is the subject of thermodynamics. If one does not know the difference between energy and internal energy a person is bound to get frustrated. Hence before we look at the discussion of the results and further explanation of the concept and implementation of the various blocking techniques, it is necessary to understand some of the jargon that will be used frequently during the discussion of the results. What follows is a reference to the terminology used in the remaining part of this thesis.

3.1.1 Terms related to Code Versions

External blocking has been applied and tested on two versions of the GHOST code, Version 0 and Version 3; referred to as V0 and V3. V3 is the version of GHOST code which was previously tuned in LeBeau et al [4]; it is the best-tuned version of the laminar code at the start of this project. The modifications made to V0 to get to V3 are: (1) Correcting the orientation of the $i,j$ sweeps to the cache-conserving form (i.e. outer loop $j$, inner loop $i$) consistent with the storage in memory, (Loop Interchange) (2) Aggressive cleaning of redundant computations, unnecessary divisions, and other excessive mathematical activity, (3) Removal of unwanted if-then structures, particularly on sweeps that do not encompass the full $i,j$ grid, and (4) Restructuring the variables from the single array form, $\phi_1(i,j)$ and $\phi_2(i,j)$, to an array of structures $\Phi(i,j) : \phi_1, \phi_2$. Due to these modifications, this version has lower L2 cache miss rates and a more predictable behavior with changing grid size when compared to the original code, V0. Application of the same techniques used to create the laminar code V3 were applied to the
turbulence subroutines to create a turbulent V3 code. The untuned turbulent code is labeled V0 and is generally similar to the laminar V0 with unoptimized turbulence subroutines.

Tests were performed to study the effects on the code performance when compiled with different compilers. Three compilers were used in this study: Intel Fortran Compiler Ver 7.1 (ifc) and Ver 9.1 (ifort), and the g95 compiler (g95). The terminology used to present the results of these tests is the name of the compiler used to compile the code followed by the code version (V0 or V3). Hence the optimized version of the GHOST code compiled with the Intel Fortran Compiler Ver 7.1 is referred to as “ifc-V3”. The codes have been compiled using the -O3 level of optimization.

3.1.2 Terms related to performance study test results description

The one term that is frequently used during the discussion of the performance test results is walltime. As mentioned previously, walltime or wall clock time is the amount of time that passes if you are looking at a clock on the wall for the code to finish a problem or to run a fixed number of iterations. Performance tests were carried out on varying grid sizes. The time taken by the code to perform a fixed number of iterations (usually 5000 iterations) was considered the walltime. This time did not include the time taken to read the grid files and to write out the solutions. In order to relate the walltime obtained to the various grid sizes, the walltime value was normalized by the grid size. In order to be able to compare the walltime with other tests that were carried out for a different number of iterations it was normalized by the number of iterations. The resultant walltime value that has been used to compare the various results is the amount of time spent by the code to perform all the calculations on a single grid point for a single iteration. This is the value that has been used in the plots to illustrate the performance improvements obtained.

In the current chapter, external blocking has been referred to as blocking or subblocking. The results of the various tests that are obtained by the application of this technique are referred to as blocked grid results, blocked code results, or just blocked results. The results obtained using a full normal grid i.e. without the application of this technique are referred to as nonblocked or noblock case results.

3.1.3 Terms related to cache behavior study test results

A good understanding of the terms related to cache is necessary to better understand the results. These have been discussed in prolix in the first chapter. To reiterate, a cache miss can be
either an L1 cache miss or an L2 cache miss. If not specified, the words cache miss or L2D miss refers to a L2 cache miss. Data calls refer to the number of requests made by the processor to the memory hierarchy for data. This number was quite high when compared to the other parameters that were being studies. In order to be able to clearly represent the various cache parameters on a single plot, the number of data calls was normalized by 10. Hence it is referred to as Dcalls/10 in all the plots.

3.2 TEST CASE

A two-dimensional incompressible flow in a square cavity at a Reynolds number of 400 was used to check the effectiveness of the various methods. This is also known as lid-driven cavity flow. The lid-driven cavity, as shown in Figure 3.1, is a fairly standard test case with simple boundary conditions. It was chosen due to its simple geometry yet complex flow pattern. It is also easy to generate and divide the grid 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 of unity and a $v$-velocity value of zero. The other three walls have a no-slip boundary condition. The driven cavity flow lacks an exact solution; hence an existing accurate numerical solution for this problem is used as a benchmark for comparing the results. Ghia et al. [77] presented numerical studies using the vorticity-stream function formulation for solutions up to Re=10,000 with 256 X 256 grid points and these simulation results have been widely used as benchmark for the driven cavity problem.

3.3 TYPES OF TESTS

Three main tests were carried out to assess the effectiveness of the blocking techniques: (1) Performance Test (2) Cachegrind Test and (3) Accuracy Test

**Performance Test:** These tests were carried out in order to study the performance gains obtained due to external blocking. Another objective of these tests was to find out the optimum size of the grid block that would fit into the cache and at the same time give maximum performance improvement. The majority of the tests were carried out on the Kentucky Fluid Cluster 4 [KFC4], which was described in the previous chapter. Each node on this cluster has 512 MB of RAM and each processor has 64 Kb L1 cache and L2 cache of 512 KB. The GHOST code was compiled with –O3 optimization level on both the compilers *ifc* and *g95*. The test comprised of running the code for 5000 iterations on various grid sizes and block sizes and comparing the walltime. The codes were timed using inbuilt UNIX functions.
Since the optimizations that we carried out were not applicable to the input/output operations, the time taken by these was not considered. Hence the time values which are used in the plots are the amount of time which was used to carry out the calculations only.

- **CacheGrind Tests:** These tests were carried out to get a better insight of how and why grid blocking worked. CacheGrind data were helpful in studying the statistics of various cache parameters such as L1 and L2 cache miss rate and the number of data calls to the cache. From the plots obtained using these results, we get a better understanding of the relationship between the cache misses, data calls, and wall time per grid point. The main disadvantage is that when a code is run using cacheGrind it runs up to ten times slower than the normal code. However, it was found that the cache behavior tends to stabilize after the initial developments by the time 400 iterations are completed [4] over various grid sizes. The cache misses tended to flatten out after 400 iterations, so running a code for about the same number of iterations yields a good estimate of the cache performance. Hence all the cache simulations tests were done for 500 iterations.

- **Accuracy Tests:** These tests were carried out to study the effects of the blocking techniques to the accuracy of the solutions obtained. The solutions obtained by Ghia *et.al.* [80] were used as benchmarks for comparison.
3.4 EXTERNAL BLOCKING

External blocking involves the breaking up of the grid into smaller sized cache friendly blocks. This step is carried out during the grid generation process. As mentioned earlier, the code used to generate the grid data for GHOST is called g.f90. This code requires the various grid parameters such as the number of grid points, dimensions, boundary conditions and number of ghost points to be defined in a file called Input, which was described in the previous chapter. It is a tedious and lengthy process to create an input file manually for g.f90 to generate the blocked grid. In order to overcome this problem, a code which generates the input scripts for g.f90 called Gridsubblocker.f90 was created. This code reads the input file required to generate a single block grid and generates the input file to create the same grid, but consisting of multiple blocks. This has been illustrated with the help of Figure 3-2 and 3-3.

3.5 PERFORMANCE TEST RESULTS

In LeBeau et.al. [4], it was found that for the laminar steady case, the grid sizes varying from 20x20 to 30x30 had effectively a 0% L2D cache miss rate, presumably because they fit into the L2 cache directly. In contrast, a grid of size 1000x1000 was found to fill over 90% of the 512 Mb RAM on a node in KFC4 and had high L2D miss rates. In all the performance tests that were carried out, we have used grid sizes varying from a minimum of 30x30 to maximum of 600x600. In order to test the effectiveness of external blocking, we split these grids into blocks varying from 30x30 to 70x70. As suggested by the cavity test problem, all the grids used had an equal number of points in $i$ and $j$, a convention that has been used in the majority of the analysis.

3.5.1 KFC4 Results

A plot of walltime as a function of block size for the untuned V0 code on KFC4 is shown in Figure 3-4(a). Once again, it is important to note that the walltime is effectively the average time for a single iteration over a 5000 iteration simulation and it is normalized by the number of grid points to eliminate the effect of increasing walltime with increasing grid sizes. The general trend that can be noticed from the plot is that blocking tends to maintain a constant walltime with the increase in overall grid size, while in the case of the unblocked grid the walltime tends to keep increasing with increasing grid size. Blocking of the grid leads to a reduction in the cache effects; hence, we notice a smooth and even walltime plot with the increase in the grid size. The walltime per grid point decreases with the decrease in the sub block size and it reaches a minimum when the grids are broken down into smaller blocks of size 30 x 30.
/*number_of_zone
1
/* zone_number 1

quadratic 10 10 4 2
0.0 0.0
-1.5 -0.5 -0.5 -1.5
-0.5 -0.5 0.5 0.5
0.0 0.0 0.0 0.0
1. 1.

m inlet          left    1       1       1        99999       0
* outflow        right   99999   99999   1        99999       0
* wall           bottom  -99999  100000  1      1           0
* wall           top    - 99999  100000  99999    99999       0
/end

Figure 3-2 Input file used to generate a single block grid

/*number_of_zone
2
/* zone_number 1

quadratic 5 10 4 2
0.00 0.00
0.00 0.50 0.50 0.00
0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00
1.0 1.0

* wall       left    1       1       1      99999  0
m inlet     top     -99999   100000  99999  99999  0
* patch      right  99999   99999   1      99999  2
* wall       bottom -99999  100000  1      1      0

/* zone_number 2

quadratic 5 10 4 2
0.00 0.00
0.50 1.00 1.00 0.50
0.00 0.00 1.00 1.00
0.00 0.00 0.00 0.00
1.0 1.0

* patch      left    1       1       1      99999  1
m inlet     top     -99999   100000  99999  99999  0
* wall       right  99999   99999   1      99999  0
* wall       bottom -99999  100000  1      1      0
/end

Figure 3-3 Input file generated to split the above shown grid into two blocks
The best performance improvement was obtained with the 30 x 30 block grid. In the case of the 600 x 600 grid the wall time for the 30 x 30 block grid is lower by a factor of 5.1 than that of the blocked grid.

The walltime plot for the tuned V3 code on KFC4 is shown in Figure 3-4(b). Due to the optimizations made to this code earlier, the walltime for the unblocked code does not increase with an increase in the grid size. The performance of V0 when subblocking is applied to it is similar to the performance of the V3 code without subblocking; fitting the computations into cache can apparently hide a variety of other program flaws. With the application of blocking, there is a decrease of 28.9% when compared to the unblocked code. The values of the walltime for the grid of size 600 x 600 for various subblock sizes are shown in Table 3-1. Unlike V0, the performance of the block sizes 30 x 30, 40 x 40 and 50 x 50 are almost the same for V3.

The externally blocked grid performance is also highly scalable and unchanging over a wide expanse of grid sizes i.e. the normalized walltime tends to stay almost constant irrespective of the grid size. Hence the actual normalized speed of a blocked computation is comparable to the single grid computation of the same size, even capturing some of the irregular behavior that arises when the computation only partially fits into the L2 cache such as the spike around 60 x 60 grids. In effect, a 30 x 30 external blocking extends the normalized computational speed of the 30 x 30 single grid to much larger grids. This suggests that by taking any CFD code and measuring its normalized walltime performance [Figure 3-5] on a series of smaller to larger grids, one could estimate the potential improvement achievable through sub-blocking of the whole grid. The nonblocked curve also reveals by its change in slope the region where cache effects become significant, which is about 15,000 grid points at which point the curve flattens.

Table 3-1 Summary of performance improvements obtained on KFC4

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Walltime/Grid Point/Iteration [µ secs]</th>
<th>V0</th>
<th>V3</th>
<th>% Decrease Compared to No Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Block</td>
<td></td>
<td>18.83</td>
<td>4.63</td>
<td>-</td>
</tr>
<tr>
<td>70 x 70</td>
<td></td>
<td>4.55</td>
<td>3.48</td>
<td>75.8%</td>
</tr>
<tr>
<td>60 x 60</td>
<td></td>
<td>4.75</td>
<td>3.40</td>
<td>74.7%</td>
</tr>
<tr>
<td>50 x 50</td>
<td></td>
<td>4.12</td>
<td>3.33</td>
<td>78.1%</td>
</tr>
<tr>
<td>40 x 40</td>
<td></td>
<td>3.97</td>
<td>3.31</td>
<td>78.9%</td>
</tr>
<tr>
<td>30 x 30</td>
<td></td>
<td>3.69</td>
<td>3.29</td>
<td>80.4%</td>
</tr>
</tbody>
</table>
Figure 3-4  Walltime as a function of grid size for GHOST compiled with g95 on KFC4 for the lid-driven test case. (a)V0 (b)V3
3.5.2 KFC5 Results
Performance tests were carried out on KFC5 in a similar manner to those carried out on KFC4. The Intel Fortran Compiler Ver. 9.1 (ifort) compiler was used to compile the code on this cluster. The results from the tests carried out on KFC5 and its 64-bit architecture revealed a similar pattern to KFC4, but with some noticeable differences [Figure 3-6]. As expected, due to better and faster processors the overall code performance is improved for the unblocked grid on KFC5 compared to KFC4. The potential gain obtained from improving cache performance is reduced due to the greater bandwidth and better memory hierarchy structure. However, blocking consistently increases the speed to where the blocked cases actually outperform the equivalent single grid, which can be noticed by the downward slope at the small grid end for all the sub-blocked results. After this drop, the performance is again highly scalable. With the application of 30 x 30 subblocking, for the largest grid of size 600 x 600, a decrease of 69% was observed for the V0 code and 27.8% for the V3 code when compared to the unsublocked code. The codes were compiled using the Intel Fortran Compiler Version 9.1.
Figure 3-5  Zoomed plot of walltime as a function of grid size for GHOST compiled with g95 on KFC4 for the lid-driven test case (a) V0 (b) V3

Table 3-2 Summary of performance improvements achieved on KFC5

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Walltime/Grid Point/Iteration [µ secs]</th>
<th>% Decrease Compared to No Block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V0</td>
<td>V3</td>
</tr>
<tr>
<td>No Block</td>
<td>6.51</td>
<td>2.41</td>
</tr>
<tr>
<td>70 x 70</td>
<td>2.32</td>
<td>1.89</td>
</tr>
<tr>
<td>60 x 60</td>
<td>2.31</td>
<td>1.86</td>
</tr>
<tr>
<td>50 x 50</td>
<td>2.18</td>
<td>1.78</td>
</tr>
<tr>
<td>40 x 40</td>
<td>2.03</td>
<td>1.76</td>
</tr>
<tr>
<td>30 x 30</td>
<td>1.98</td>
<td>1.74</td>
</tr>
</tbody>
</table>

3.5.3 Rectangular Blocks

Up to this point the results presented were for blocks with equal number of grid points in i, j direction i.e. square blocks. But it would not be practical to block all the grids using square grids only, especially at the boundaries of the grids. In order to study if the shape of the subblock affects the performance of the code, we carried out performance testing by blocking the code using rectangular blocks [Figure 3-8]. We tested four different blocks of sizes 25 x 35, 30 x 40, 35 x 50 and 40 x 45. The performance curves obtained had a similar pattern to that obtained
using square blocks. The average walltime for the rectangular subblock of size 25 x 35, which consists of 875 grid points was 3.72µs whereas for the square subblock of size 30 x 30, consisting of 900 grid points was 3.69 µs. This suggests that the effectiveness of subblocking depends primarily on the number of grid points in the blocks rather than the shape of the blocks.

3.5.4 Effects of Compiler Optimization Levels on Performance

Any compiler by default performs at least a few minor optimizations on the code during compilation. The user can specify the level of optimizations that the compiler should carry out using the option ‘-O’, with an optional digit specifying the optimization level. The higher the level of optimization selected, the faster the runtime, but the longer the compilation time. Some common characteristics of the various compiler optimization levels are [81):

- **-O0**, Default compiler state, performs minimal optimizations.
- **-O2**, Performs many powerful optimization techniques that generally result in improved code and do not result in excessive compile time.
- **-O3**, Performs the optimizations carried out at –O2 level and additional optimizations that might lead to the use of more machine resources during compilation and change the semantics of the code slightly.

Using a higher level of optimization level while compiling the code does not necessarily mean that the performance obtained is better [16]. Hence performance tests were carried out by compiling the code using the various levels of optimization to see if it had any effect on the performance of subblocking and to find out the best level of optimization. The G95 compiler was used to compile the code. The tests were carried out on KFC4. The plots comparing the walltimes are shown in Figure 3-9. There is a 50% decrease in the walltime for the code compiled with -O2 for both the V0 and V3 versions of the code when compared to the performance of the code compiled using -O0 optimization. The profiles of the walltime curves obtained for the various compiler levels is the same, only offset. The initial slope in the walltime curve for V3 compiled with -O0 option is smaller than -O2. The performance obtained using the code compiled by -O3 is the same as -O2.
Figure 3-6 Walltime as a function of subgrid size for GHOST compiled with IFort(Version 9.1) on KFC5 for the lid-driven test case. (a) V0 (b) V3
Figure 3-8 Walltime as a function of subgrid size for GHOST compiled with G95 on KFC4 for the lid-driven test case (Rectangular blocks) (a) V0 (b) V3
3.5.5 Effects of Different Compilers on Performance

The performance of the code is also dependent on the type of compiler used. Certain compilers might do a better job at optimizing the code during compilation. Hence performance tests were carried out by compiling the code with the g95 compiler and the Intel Fortran Compiler Version 7 [ifc]. Henceforth the codes compiled using G95 will be referred to as g95-V0 and g95-V3. The codes compiled using ifc as ifc-V0 and ifc-V3. ‘-O3’ level of optimization was used while compiling the code and the tests were carried out on KFC4. The walltime plots have a similar trend overall [Figure 3-10 & 3-11], but the performance of ifc-V0 for the unblocked case is better than g95-V0 for smaller grid sizes (up to ~40,000). Above this the performance is similar between the two [Figure 3-10 (a)(b)]. The performance of ifc-V3 is better than g95-V3 for the unblocked case. The codes compiled with g95 seem to work well for larger grid sizes [Figure 3-12], whereas when compiled using ifc they are more efficient for the smaller grid sizes. Hence the initial slope of the walltime plot is greater for the ifc compiled codes compared to the g95. Due to this, the percentage decrease with the application of subblocking is greater in this case (Table.3-3). The walltime curve for the ifc-V0 unblocked case has several spikes. The spike around the grid of size 3600 [Figure 3-13 (a)] also exists for the g95-V0 but it is smoother. Also from the plots we can notice that the walltimes for the unblocked version are much closer together for the code compiled with g95, when compared to ifc. All this suggests that as the steepness of the initial slope in the walltime curve of the unblocked grid increases, the effectiveness of subblocking increases proportionately.

Table 3-3 Comparison of performance improvements achieved by using ifc Vs g95

<table>
<thead>
<tr>
<th>Block Size</th>
<th>% Decrease for code compiled with Intel Fortran Compiler Ver. 7</th>
<th>% Decrease for code compiled with g95</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V0</td>
<td>V3</td>
</tr>
<tr>
<td>70 x 70</td>
<td>78.7%</td>
<td>24.4%</td>
</tr>
<tr>
<td>60 x 60</td>
<td>71.9%</td>
<td>22.7%</td>
</tr>
<tr>
<td>50 x 50</td>
<td>81.1%</td>
<td>36.1%</td>
</tr>
<tr>
<td>40 x 40</td>
<td>80.2%</td>
<td>36.8%</td>
</tr>
<tr>
<td>30 x 30</td>
<td>84.7%</td>
<td>40.2%</td>
</tr>
</tbody>
</table>
Figure 3-9 Walltime as a function of subgrid size (or grid size for a single node case) for GHOST compiled with G95 on KFC4 for the lid-driven test case. a) V0 b) V3
Tests were conducted to compare the performance of the Intel Fortran Compilers Ver. 7.1 (ifc) and Ver. 9.1 (ifort). These tests were carried on the KFC4 cluster. The plot comparing the performance of these two compilers along with g95 compiler [Figure 3-14] shows that the performance of the two Intel compilers is quite similar. Ifort results are slightly faster compared to ifc results for the V3 code, whereas for the V0 code it is the other way around. Hence the ifc compiler seems to be doing a better job at optimizing the unoptimized code. The spike observed around the 40 x 40 and 60 x 60 block in the ifc results is not present in the ifort results.

![Figure 3-10](image_url)  
**Figure 3-10** Walltime as a function of subgrid size (or grid size for a single node case) for GHOST V0 on KFC4 for the lid-driven test case. a) g95  b) Intel Fortran Compiler Ver. 7

![Figure 3-11](image_url)  
**Figure 3-11** Walltime as a function of subgrid size (or grid size for a single node case) for GHOST V3 on KFC4 for the lid-driven test case. a) g95  b) Intel Fortran Compiler Ver. 7
Figure 3-12 Walltime as a function of subgrid size (or grid size for a single node case) for GHOST on KFC4 for the lid-driven test case.

Figure 3-13 Walltime as a function of subgrid size (or grid size for a single node case) for GHOST on KFC4 for the lid-driven test case. a) V0 b) V3
3.5.5 Effect of different hardware on performance

The majority of the performance results that have been presented up to this point were carried out on KFC4. This cluster was constructed two years ago (mid 2004), which is a very long time in terms of semiconductor advances. The current state-of-art (mid 2006) dual core processors have 64 bit processing capability with 2 to 4 times bigger cache sizes, faster clock speeds and FSB. Since our work is focused on improving the cache performance, it is useful to study how the cache size and processor clock speed would affect the performance of the blocked code. These tests were carried out on the KFC6 test nodes. The processor details have been described in the previous chapter. The code was compiled using the Intel Fortran Compiler Ver. 9.1 (ifort) on these new processors.

The walltime plot comparing the performance of the three processors: Intel 6400, AMD 4200 and AMD4600 are shown in Figure 3-15 and 3-16. The performance of the unblocked V0 version on the new processors have a similar trend as that observed on the KFC4 cluster i.e. the
walltime increases with increases in grid size. It is interesting to observe here that the code performance on the Intel 6400 machine is better compared to the AMD processors in spite of its lower clock speed. The cache size on the Intel chip is twice that of the AMD chips, which should lead to a drop in cache misses. This point is confirmed by looking at the effect of blocking on the performance, where it can be seen that the performance on the AMD chips is better than the Intel chip in this case. Application of blocking is leading to better cache performance hence optimizing the utilization of the higher processor clock speed to perform calculations rather than wasting it waiting for the data to arrive. Looking at walltime plot for V3, the performance of the AMD processors is better for both the unblocked as well as the blocked code since the V3 code has already been cache optimized. These results clearly support the main objective of this project. They show that even if you get processors with higher clock speeds and larger caches their true potential cannot be realized until the code has been optimized.

Tables 3-4 and 3-5 present a summary of the average percentage performance improvements obtained by the application of blocking to the code on the KFC6 test nodes. For the Intel processor the performance improvement obtained using the 70x70 block is better than the 30x30 block. This suggests that as the cache size increases, larger sized blocks are more effective in improving the cache performance since they fit into the cache without

![Walltime comparison plot on KFC6 test nodes for V0](image-url)
the additional overhead of more blocks. The performance improvement obtained on the AMD processors with the application of blocking for the V0 code is significantly higher than the Intel processor. Hence as the cache size increases the performance improvements obtained from blocking diminish.

Table 3-4 Performance improvements obtained on various hardware for V0

<table>
<thead>
<tr>
<th>Block Size</th>
<th>AMD 4600</th>
<th>AMD 4200</th>
<th>Intel 6400</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 x 30</td>
<td>60.35%</td>
<td>61.58%</td>
<td>29.55%</td>
</tr>
<tr>
<td>40 x 40</td>
<td>59.11%</td>
<td>60.64%</td>
<td>30.85%</td>
</tr>
<tr>
<td>50 x 50</td>
<td>56.20%</td>
<td>57.25%</td>
<td>30.83%</td>
</tr>
<tr>
<td>60 x 60</td>
<td>52.73%</td>
<td>54.68%</td>
<td>30.51%</td>
</tr>
<tr>
<td>70 x 70</td>
<td>52.9%</td>
<td>54.23%</td>
<td>29.43%</td>
</tr>
</tbody>
</table>
Table 3-5 Performance improvements obtained on various hardware for V0

<table>
<thead>
<tr>
<th>Block Size</th>
<th>AMD 4600</th>
<th>AMD 4200</th>
<th>Intel 6400</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 x 30</td>
<td>21.55 %</td>
<td>22.78 %</td>
<td>11.18 %</td>
</tr>
<tr>
<td>40 x 40</td>
<td>21.05 %</td>
<td>22.18 %</td>
<td>13.6 %</td>
</tr>
<tr>
<td>50 x 50</td>
<td>20.56 %</td>
<td>21.94 %</td>
<td>14.66 %</td>
</tr>
<tr>
<td>60 x 60</td>
<td>17.44 %</td>
<td>18.91 %</td>
<td>15.28 %</td>
</tr>
<tr>
<td>70 x 70</td>
<td>17.02 %</td>
<td>17.92 %</td>
<td>15.11 %</td>
</tr>
</tbody>
</table>

3.5.6 Steady Turbulent Case Performance Results

With the inclusion of a turbulence model several more data arrays and computations are added to the calculations, leading to a decline in performance when compared to the laminar code performance. For a Reynolds-Averaged Navier-Stokes model such as the one used in GHOST (Menter’s SST two-equation model), two new flow variables are added and two new partial differential equations need to be solved. The subroutines which are activated when the turbulence model is turned on need to be optimized independently and the scale of grid that is cache-friendly may also change. As such, investigating the performance of a code on both laminar and turbulent problems is potentially worthwhile.

For these investigations, two versions of GHOST (V0 and V3 turbulent) are considered - in V0, the turbulent subroutines like the laminar ones are untuned, while in V3 similar optimizations are applied to all subroutines. The test case considered is again the driven cavity flow but at a higher Reynolds number [500,000]. The codes were compiled using g95 and ifc [will be referred to as g95-V0, g95-V3, ifc-V0 and ifc-V3].

The overall characteristics of the walltime plots for the codes compiled with g95 [Figure 3-17] are same as the laminar results although the magnitude of the normalized walltime has more than doubled for the blocked results for both versions. However the relative performances of the blocked and unblocked cases are essentially the same as for the laminar case.

The results from the code compiled using ifc [Figure 3-18] are also similar to the laminar results. The spike in the normalized walltime around grid size 3600 is present and seems more pronounced. There is a dip initially for the 30 x 30 subblock case for the V0 code.

A notable difference is found in the unblocked V3 curve irrespective of which compiler is used, in that it continues to rise with grid size like V0 (although not as steeply) instead of flattening like the laminar case. As such, subblocking seems to be the good way to get improved
Figure 3-17 Walltime as a function of subgrid size (or grid size for a single node case) for GHOST on KFC4 with g95 for the lid-driven test case. a) V0 b) V3
Figure 3-18 Walltime as a function of subgrid size (or grid size for a single node case) for GHOST on KFC4 with ifc for the lid-driven test case. a) V0 b) V3
scaling characteristics with these versions of the code.

Comparing the performance between the codes compiled with the two different compilers [Figure 3-19], the performance for the unblocked case is similar for both V0 and V3 unlike the laminar results. The initial slope for the code compiled with g95 is almost the same as that compiled by ifc. But the results for the 30 x 30 block are very good in the case of the code compiled using ifc. In fact the performance of V0 compiled with ifc is better than the V3 compiled using g95. The effect of subblocking on the performance seems to be more significant when the code is compiled with the ifc compiler (Table 3-6). These comparisons suggest that the ifc compiler is optimizing the code better than the g95 compiler.

![Figure 3-19  Comparison between ifc and g95](image)

Table 3-6 Summary of performance improvements for the turbulent test case achieved on KFC4

<table>
<thead>
<tr>
<th>Block Size</th>
<th>% Decrease for code compiled with Intel Fortran Compiler Ver. 7</th>
<th>% Decrease for code compiled with g95</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V0</td>
<td>V3</td>
</tr>
<tr>
<td>70 x 70</td>
<td>76.6%</td>
<td>62.6%</td>
</tr>
<tr>
<td>60 x 60</td>
<td>67.4%</td>
<td>58.2%</td>
</tr>
<tr>
<td>50 x 50</td>
<td>77.2%</td>
<td>65.6%</td>
</tr>
<tr>
<td>40 x 40</td>
<td>76.1%</td>
<td>65.2%</td>
</tr>
<tr>
<td>30 x 30</td>
<td>80.3%</td>
<td>67.3%</td>
</tr>
</tbody>
</table>
3.6 **Valgrind Results**

Looking at the performance test results, significant improvements can be noticed for all the cases tested when external blocking was applied to the code. Theoretically it can be deduced that the reason for this improvement is due to better cache performance of the code. But in order to get a better insight into the connection between cache performance, overall walltime, and block size, and to quantify them, the various cache parameters were measured with the help of the “Cachegrind” cache profile simulator of the Valgrind Toolkit [79]. The tests were carried out in a manner similar to the performance tests. The only differences being that the code was run with the help of the cachegrind cache simulator instead of running it directly on the processor. When a code is simulated with the help of cachegrind, it runs up to 10 times slower than normal. Plus, it has been demonstrated earlier in LeBeau *et al.* [4] that the various cache parameters reach a steady state after 400 iterations. Hence, only 500 iterations were used while carrying out these tests. The code was compiled using the g95 compiler with -O3 level optimization and the -g option, which writes the debugging information. The -g option has to be used during the compilation of the code in order for it to be run using cachegrind.

3.6.1 **KFC4 Results**

Cache performance is presented in terms of overall data calls (divided by a factor of 10 for plotting convenience, Dcalls/10), data calls that miss the L1 cache (D1 misses), and data calls that miss the L2 cache (L2D misses), all normalized again by the number of grid points. The reason all the parameters have been normalized by the number of grid points is in order to be able to compare them across various grid sizes. In Figure 3-20, the effects of blocking on the cache miss rates for the V0 code have been illustrated. For the unblocked case, the L1 and L2 cache miss rate keep increasing as the grid size increases, leading to an increase in wall time. The introduction of blocking tends to smooth and reduce the L1 and L2 cache misses quite significantly. For the 30 x 30 blocks the L2D cache miss rate is less than 0.1%, the D1 miss rate is typically less than 2%, and these values are largely independent of the overall grid size. This compared to the unblocked case is quite low. Hence this leads to the significant improvements in the blocked case. The code modifications that were made to GHOST to get V3 in [4] have helped in reducing the L2 cache misses significantly and the data calls [Figure 3-21].
Blocking has reduced the L2 cache misses even further, but at the same time it has lead to an increase in data calls. The reason for this increase is the introduction of ghost points at the artificial boundaries. Looking at the L2 cache misses trend, there is a further drop when blocking is introduced. Plus it is highly scalable, remaining constant over a wide range of grid sizes, similar to walltime. All this suggests that the small blocks are effectively fitting into L2 cache, thereby achieving close to the best possible cache performance (whereas in theory the total cache miss rate would be near 0%, since there are unavoidable misses). The difference in performance
between the blocked V0 and V3 is primarily reflected in a reduction in D1 misses and in a lower number of data calls overall.

Figure 3-22 shows the L2 cache miss rate versus grid size on KFC4 for the V3 GHOST code over a series of block sizes. As the subblock size decreases the L2 cache misses decreases proportionately, which explains the decrease in walltime with the decrease in subblock size.

From the plot of data calls versus iteration over a series of block sizes [Figure 3-23], it is clear that the unblocked grid has the minimum number of data calls. As the subblock size decreases the number of blocks increases which leads to an increase in ghost points, leading in turn to an increase in the number of data calls. Let us look at this in further detail by looking at the increase in the total number of grid points due to the application of blocking to a grid. If a 600 x 600 block (i.e. 360,000 grid points) is split into blocks of 30 x 30, we would have to split up the grid into 484 blocks. Now each of these blocks will have 240 ghost points, which would mean a total of around 116,160 extra grid points over which the code must traverse during the calculations. Fortunately, these ghost points are not considered during the computationally intensive calculations such as the tri-diagonal matrix solves. Hence even though they do not affect the performance significantly, these points do lead to an increase in the data calls. This is the reason why the number of data calls increases when the grid is blocked versus when it is unblocked. Now if the above mentioned grid was split into blocks of 70 x 70, we would have 81 blocks, with each block having 560 ghost points, leading to an increase of 45,360 grid points. This is less than half the increase in points when the grid was split using the 30 x 30 blocks. Hence the number of data calls is significantly less in this case compared to the smaller blocks.

3.6.2 Comparison between G95 & IFC

From the plot of the various cache parameters for the V0 code compiled using g95 the number of data calls is around 20% higher when compared to the code compiled using ifc [Figure 3-24 (a)(b)], which means that the compiler does a better job in optimizing the code, leading to better performance compared to the code compiled with G95. The D1 misses [Figure 3-25] and L2D misses [Figure 3-26] on the other hand are almost the same in both the cases. This implies that subblocking has the same impact irrespective of how well the code is optimized by the compiler.
Figure 3-22  L2 cache miss rate V3 G95

Figure 3-23  Data calls V3 G95
Figure 3-24  Valgrind plot for V0 30x30 Subblock (a) G95 (b) IFC

Figure 3-25 Data calls comparison (a) G95 (b) IFC
3.6.3 Effect of Cache Thrashing

In the walltime plots for the code compiled using IFC [Figure 3-10(b), 3-11(b)] a significant spike in the walltime was noticed around the grid size 1600 and 3600. The normalized walltime for a 60 x 60 block was also higher than that for a 70 x 70 block. The reason for this discrepancy is the high D1 miss rate [Figure 3-27] for the 40 x 40 and 60 x 60 block. Cache thrashing which was discussed in the first chapter could be the cause for the high D1 miss rate. Although the exact source of the problem is not yet known.

Figure 3-26 L2D misses comparison (a) G95 (b) IFC

Figure 3-27 Comparison of D1 misses for various block sizes for code compiled using ifc. a)V0 and b)V3
3.6.4 Valgrind Results for Turbulent Case

The valgrind plot which illustrates the cache performance is presented in terms of overall data calls (divided by a factor of 10 for plotting convenience, Dcalls/10), data calls that miss the L1 cache (D1 misses), data calls that miss the L2 cache (L2D misses), normalized again by the number of grid points. In Figure 3-27, the effects of blocking on the cache miss rates for the V0 code have been illustrated. Similar to the laminar case, for the unblocked case, the L1 and L2 cache miss rates keep increasing as the grid size increases, leading to an increase in wall time. As seen previously the introduction of blocking tends to smooth and reduce the L1 and L2 cache misses significantly. This leads to the significant drop in walltime. The number of data calls is close to 2.5 times that of the laminar case, so the walltime is also higher. When the code modifications that were made to the laminar subroutines of GHOST to get V3 in [4] were applied to the turbulent routines, they did help in reducing the L2 cache misses and data calls considerably [Figure 3-28], but a slight increase in the L2 cache miss rate was noticed for larger grid sizes which was not present in the laminar V3 version of the code. This resulted in an increase in the walltime for larger grid sizes as noticed during performance tests. Blocking seems to have reduced the L2 cache misses even further, but as expected it has lead to an increase in the data calls.

Figure 3-27 Valgrind plot for V0 (a) 30 x 30 (b) No Subblock
3.7 ACCURACY TEST RESULTS

a.) Stream Line Plots:

The streamlines plots obtained using the subblocked grid results are in very good agreement with Ghia et al. [80].
b.) Contour Plots:

The overlapped u-velocity and v-velocity contour plots have been shown below. These have been plotted using the results obtained after 30,000 iterations. A 200 x 200 grid was used to carry out the test. The grid was split up using 30 x 30 blocks to test external blocking. There is a good match between both the results.

![Figure 3-30 Overlapped u-velocity contour plots at 30,000 iterations](image1)

Flood – 30 x 30 Blocked Grid
Dotted Line – Normal Grid

![Figure 3-31 Overlapped v-velocity contour plots at 30,000 iterations](image2)

Flood – 30 x 30 Blocked Grid
Dotted Line – Normal Grid
c.) Center line plot:

The midline u-velocity profile from simulations run to completion are in agreement with Ghia et al.[80].

![Figure 3-32 The midline u-velocity profile](image)

**d.) Residual plot:**

The convergence patterns of the residuals of the 30 x 30 subblocked and the unblocked version of the code is shown in the Figure 3-28. The residue shown is the square root of sum of square of the residuals of velocities and pressure. The rate of convergence for the blocked code is better than the unblocked code in spite of the large number of ghost points that are introduced.
3.8 SUMMARY

This chapter introduced the concept of external blocking along with a description of the methodology used in applying it to the GHOST code. Performance improvements of up to 80% for the V0 code and 28% for the V3 code in walltime were observed when compared to the unblocked code for the larger grid sizes. The best performance improvement was obtained by using 30 x 30 blocks. The results obtained by using cachegrind were also very encouraging. The L2 cache miss rate was reduced to almost 0% with the application of external blocking. By looking at the results from all the tests that were conducted, external blocking seems like a promising technique that can be used to improve the cache performance of a CFD code. A major disadvantage of this technique is the difficulty in implementing it, since it is not easy to break up a grid with a complicated geometry or boundary conditions into smaller blocks. Creating a viable automated system to split the grid led to the idea of internal blocking which is basically an automated version of external blocking. This technique is discussed in detail in the next chapter.
CHAPTER – 4

4. INTERNAL BLOCKING RESULTS

The previous chapter considered the primary results obtained by the application of external blocking techniques. With the application of this technique we were able to decrease the L2 cache miss rate to almost 0% and hence improve the performance by a factor of five times the original code on KFC4. Likewise there were significant performance improvements on the other hardware platforms. But one of the disadvantages of using this technique was that it is not easy to implement it on complicated grids with multiple zones and different boundary conditions. To overcome this problem we implemented an approach to automatically perform the blocking irrespective of the type of grid. This technique is called internal blocking. This chapter will present a detailed description of application of this technique to GHOST and the primary performance results that were obtained after its implementation.

4.1 BASIC PRINCIPLE

The underlying principle of internal blocking is quite similar to that of external blocking. It involves breaking up the grid into smaller cache fitting blocks, solving the governing equations on these smaller blocks, and then putting them back together before the start of the MPI communications to get the overall solution.

This concept has been graphically depicted in Figure 4-1. The grid is that of an airfoil at a certain angle of attack. In order to decrease the computation time through parallel processing, the grid has already been split into multiple blocks. The flow field across each of the blocks can be solved on a different processor. Internal blocking involves the splitting of each of these individual blocks into even smaller blocks such that they fit into the L2 cache of the processor on which the calculations are being carried out. This has been illustrated with the help of a magnified view of the sub blocks in Figure 4-1. Hence internal blocking can be thought off as a means of performing localized parallel processing on a single processor.

4.2 IMPLEMENTATION OF INTERNAL BLOCKING IN GHOST

The second chapter detailed the working of GHOST with the help of a flow chart (Figure 2-1) and also the methodology used by GHOST to carry out the calculations at the artificial boundaries that are created when the grid is split into multiple blocks to perform parallel processing. In order to implement internal blocking into GHOST, four additional subroutines
Figure 4-1 Illustration of Internal Blocking
have been added [Figure 4-2]. All the information pertaining to a grid such as the \( x \), \( y \)
coordinates of the grid points are stored in arrays. The subroutines split these arrays
systematically into smaller sized arrays such that each array will hold all the necessary
information pertaining to a smaller sized block. This is equivalent to breaking up the grid into
smaller sized cache fitting blocks. Since the grid parameters remain constant for a given grid,
this operation needs to be performed only once. The flow variables such as \( u \), \( v \), and \( p \) are also
stored in arrays. Unlike the grid parameters, the values of these variables are updated every
iteration. These updated values are required to calculate the values across the artificial
boundaries. Hence, the subroutines split up the arrays consisting of these flow variables at the
start of the calculations and then put them back together before the beginning of the MPI
communication. These steps have been detailed in the next paragraph.

The splitting up of the arrays consisting of the grid parameters takes place in the
subroutine “Internal_block”. The user has to insert the size of the block that he wants. Using
these sizes, the subroutine initially calculates the total number of blocks into which it will break
up the grid. At the boundaries, if there are not a sufficient number of grid points, then it either
creates a new block of smaller size or just adds the remaining grid points to the last block. Hence
the blocks that are created at the boundaries may either be a little smaller or larger than the block
size entered by the user. Once the number of blocks and the block sizes have been decided, the
subroutine creates two arrays which map the blocks to the grid \( i.e. \) they map the \((i,j)\) values of
the block to the equivalent \((i,j)\) values on the total grid. For example if a square grid of size 50 x
50 is split into four blocks, each of size 25 x 25, then for the block at the bottom left of the grid,
the point \((0,0)\) will coincide with the \((0,0)\) of the entire grid, where as for the block next to it, the
point \((0,0)\) will coincide with the point \((24,0)\) of the original grid. The reason for creating the
map is that it can be used to break up and put together the values of the various flow variables
easily. The subroutine then makes a copy of all the geometrical grid parameters into smaller
dimensioned arrays using these mapping arrays. These arrays will hold all the required data
pertaining to a block. The subroutine also initializes the value of the variable \( inx \) to 1 at the
boundaries of all the blocks. As mentioned in the second chapter, by default the value of this
variable is 0 at all the grid points but when the value of \( inx \) is 1 at a grid point, GHOST assumes
the point to be present on the ghost boundary and does not perform any calculations on it, in
other words it treats it as a ghost point.
Once the grid has been split, the code starts copying the boundary conditions. Hence if the boundary of a block lies along the real boundary of the grid, then it copies the boundary conditions along those boundary points and sets the \( \text{inx} \) value to 0 for those points. If the boundary of the block does not lie along the boundary of the grid or if it does lie along a pre-existing ghost boundary then the \( \text{inx} \) value is left as 1. The process described up to this point is performed only once.

Then the code enters the main iteration loop. At the beginning of the loop, if the internal blocking is switched on, it enters into a subroutine called “break_velocity”. As the name of the subroutine suggests, this subroutine breaks up the arrays containing the various flow variables into smaller arrays. This is done using the grid map that is constructed in the “internal_block” subroutine. This subroutine basically breaks up the arrays consisting of variables whose values change at the end of every iteration and are required at the next iteration.

The subroutine “calc_flowfield” is called next. This subroutine is the same as the subroutine “calc_flowfield” which was described in the second chapter. The only difference being that the broken down arrays are used as the input to this subroutine. The values are sent to the subroutine one block at a time. Once all the blocks have passed through this subroutine, the “combine_velocity” subroutine is invoked. This subroutine does the exact opposite of “break_velocity”, \( i.e. \) it puts back the flow variables together and sends them ahead. After this subroutine, the code updates the boundary values and broadcasts the values to all the nodes through MPI subroutines and ends the iteration. The next iteration again starts off with the breaking up of the flow variables and the above mentioned processes go on until the solution converges or the maximum number of iterations is completed.

### 4.3 PRIMARY TESTS

The test case used to check the effectiveness of internal blocking was the same as that used to test external blocking, \( i.e. \) two-dimensional incompressible flow in a square cavity at a Reynolds number of 400. The same three tests as before were carried out with the help of this test case to assess the effectiveness of the technique: (1) Performance Test (2) Cachegrind Test and (3) Accuracy Test. These tests were performed in a similar manner as the ones performed to
assess external blocking. All the tests were carried out on the Kentucky Fluid Cluster 4 and the code was compiled with -O3 optimization level using the g95 compiler.

4.4 PERFORMANCE TEST RESULTS

A plot of walltime as a function of subgrid size for the untuned V0 code on KFC4 is shown in Figure 4-3(a). As mentioned earlier, it is important to note that the walltime is effectively the average time for a single iteration over a 5000 iteration simulation and it is normalized by the number of grid points to eliminate the effect of increasing walltime with increasing grid sizes. All the trends that were observed in the walltime plot obtained using external blocking can be noticed here. Internal blocking tends to maintain a constant walltime with the increase in overall grid size, whereas the walltime tends to keep increasing with increased grid size in the case of the unsubblocke d grid. The average walltime per grid point for the subblocked grid seems to be directly proportional to the subblock size. As the subblock size increases, we can notice an increase in the average walltime. The best performance improvement was obtained with the 30 x 30 subblock grid. In the case of the 600 x 600 grid the wall time for the 30 x 30 subblock grid is lower by a factor of 4.75 compared to the unsubblocked grid. The walltime plot for the tuned V3 code on KFC4 is shown in Figure 4-3(b). There is a decrease of 23.5% when compared to the unsubblocked code.

The values of the walltime for the grid of size 600 x 600 for various subblock sizes are shown in Table. 4-1. For the V0 code the best performance is obtained with the use of 30 x 30 blocks, while in the case of the V3 code 50 x 50 blocks showed the best performance, although for the V3 code the difference in the value of the walltime is quite small for the various block sizes. Hence the performance improvement obtained by splitting up the grid using only 30 x 30 blocks will be almost the same as a grid broken up using blocks of sizes varying from 30 x 30 to 70 x 70. This is usually the case since it is very tough to break up a grid using a single block size.

Table 4-1 Values of walltime for the grid of size 600 x 600 split up using various block sizes

<table>
<thead>
<tr>
<th>Block Size</th>
<th>V0</th>
<th>V3</th>
<th>V0</th>
<th>V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Block</td>
<td>18.83</td>
<td>4.63</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>70 x 70</td>
<td>4.74</td>
<td>3.57</td>
<td>74.82</td>
<td>22.89</td>
</tr>
<tr>
<td>60 x 60</td>
<td>4.43</td>
<td>3.50</td>
<td>76.47</td>
<td>24.406</td>
</tr>
<tr>
<td>50 x 50</td>
<td>4.24</td>
<td>3.40</td>
<td>77.48</td>
<td>26.56</td>
</tr>
</tbody>
</table>
Scalability, which was one of the most important features of external blocking, is also applicable to internal blocking. As noticed from the walltime plots above, the actual normalized speed of a subblock computation is comparable to the single grid computation of the same size. Hence we can obtain considerable performance improvements with the application of internal blocking.

4.4.1 Comparison of Performance Test Results between External and Internal Blocking

A combined walltime plot comparing the performance of external versus internal blocking is shown in Figure 4-4 (a) (b). The performance is almost the same for the 30 x 30 blocks up to a grid size of around 180,000 grid points. After this there is a slight decrease in the performance of the internally blocked code. This trend can be noticed in both V0 and V3 versions of the code. There are two possible explanations for this. One is that, as the grid size increases, it leads to an increase in the number of blocks which in turn leads to an increase in the number of ghost points and these excessive ghost points might be causing an increase in the walltime. The second is that, due to increase in number of blocks, it takes more time to break and put the various flow variables back together during each iteration. Looking at the performance for 70 x 70 blocks from Figure 4-4 (a) we can notice that the performance of the internal blocked code remains mostly constant. Hence looking as the performance curves for these two block sizes we can notice that, when larger sized blocks are used to split the grid, the number of blocks is lower and so are the number of ghost points, hence leading to a slightly better performance in breaking up and putting back together of the grid during each iteration.

From the bar charts (Figure 4-5 (a),(b)) comparing the average normalized walltime per grid point for the various block sizes for internal versus external blocks we notice that walltime for the internal blocks is slightly higher for most of the block sizes for both versions of the code. This is presumably due to the cost of breaking up the flow variables and putting them back together at each iteration. The cache effect that was causing a spike in the walltime for the 60 x 60 and 40 x 40 subblock when external subblocking was applied does not appear to happen with internal blocking, leading to better performance for these two block sizes in the V0 version of the code.
Grid Size

Walltime/Grid Point [Micro Seconds]

0 100000 200000 300000
0 5 10 15 20

V0 - No Subblock
V0 Int - 30x30
V0 Ext - 30x30
V0 Int - 70x70
V0 Ext - 70x70

V3 - No Subblock
V3 Int - 30x30
V3 Ext - 30x30

Walltime/Grid Point [Micro Seconds]

0 100000 200000 300000
0 5 10 15 20

Grid Size

a)

b)
Figure 4-4 Internal Vs External Walltime Comparison a)V0 b)V3

Figure 4-5 Average walltime comparison between internal and external blocking

4.5 Valgrind Results

The cache performance data obtained using the “Cachegrind” cache profile simulator of the Valgrind toolkit [79] on the KFC4 cluster is presented in this section. The format of the plots used to illustrate the various cache performance parameters are same as the ones used in the previous chapter during the illustration of the external blocking results.

In Figure 4-6 and 4-7, we can see the effect of subblocking on the cache miss rates for the V0 and V3 code version. Similar to external blocking, internal blocking also tends to smooth and reduce the L1 and L2 cache misses when compared to the unblocked grid. This leads to the significant improvements in the subblocked cases which were noticed during performance tests. All the other trends in the plots are very similar to those seen in external blocking.

Figure 4-8 shows the L2 cache miss rate versus grid size on KFC4 for the V0 version of the code over a series of subblock sizes. Unlike the unblocked case, the L2 cache miss rate for the internally blocked cases remains steady with increases in grid size. This is exactly the same trend as noticed with external blocking.

Looking at the trend of the number of data calls with the increase in grid size for the 30 x 30 block in Figure 4-9, we see that the number of data calls seems to increase slightly for grid sizes greater than 180,000. In contrast, for block sizes greater than 50 x 50 a constant slope is maintained.
Figure 4-6 Valgrind plot for V0 (a) 30 x 30 (b) No Subblock

Figure 4-7 Valgrind plot for V3 (a) 30 x 30 (b) No Subblock
Figure 4-8 L2 cache miss rate V0 G95
These results provide us with the proof to the explanation that was provided in the previous section to explain the increase in the walltime for the internally blocked code when compared to externally blocked code for the smaller sized blocks. As explained in the previous chapter, if a 600 x 600 block is split into blocks of 30 x 30, we would have to split up the grid into 484 blocks and add 116,160 extra ghost points over which the code must traverse during the calculations. Fortunately, these ghost points are not considered during the computationally intensive calculations such as the tri-diagonal matrix solves. Hence even though they do not affect the performance significantly, these points do lead to an increase in the data calls. This is the reason why the number of data calls increases when the grid is blocked versus when it is unblocked. Now if the above mentioned grid was split into blocks of 70 x 70, we would have 81 blocks and an additional 45,360 grid points by means of ghost points. This is less than half the number of points when the grid was split using the 30 x 30 blocks. Hence the number of data call is significantly less in this case compared to the smaller blocks. The number of blocks is six times lower in this case. Even though the total number of iterations will effectively remain the same during the breaking up and rejoining of the grid irrespective of the block size used, it seems to be more efficient when the number of blocks is lower. This might be leading to a slight decrease in the performance of internal blocking for the smaller sized blocks as the grid size increases.

4.5.1 Comparison of Valgrind results between Internal and External Blocking

The valgrind comparison is presented in terms of overall data calls, data calls that miss the L1 cache (D1 misses), data calls that miss the L2 cache (L2D misses), and walltime. All three terms have been normalized by the number of grid points. In Figure 4-10, from the plot of the various cache parameters for 30 x 30 subblocked grids we can see that the number of data calls starts off at a lower number and increases gradually with an increase in the grid size for the internally blocked code. While it remains close to constant for the externally blocked code. This feature is common to both versions of the code i.e. V0 and V3. One possible explanation for the lower number of data calls for the internally blocked grid at smaller grid sizes when compared to the externally blocked grid could be that the number of MPI communications would be much lower when compared to the externally blocked grid since the grid is put back together before the start of the various MPI operations. As the grid size increases the number of data calls for the
internal blocking gradually increases due to the overhead caused by breaking up and putting together the grid at each iteration and an increase in the number of ghost points.

Figure 4-10 Valgrind plot for V3 (a) 30 x 30 (b) No Subblock

In Figure 4-11 (a) & (b) comparing the number of data calls for the 30 x 30 and 70 x 70 blocks, we see that the number of data calls for the 70 x 70 internal blocked code remains the same as that of external block code, unlike the 30 x 30 block, consistent with the walltime trends as shown in Figure 4-4.
4.6 ACCURACY TEST RESULTS

The accuracy test results that were obtained using internal blocking are very similar to those obtained using external blocking.

a.) Stream Line Plot:

The streamline plots obtained are in very good agreement with Ghia et al. [80].

b.) Contour Plots:

The overlapped $u$-velocity and $v$-velocity contour plots have been shown below. These have been plotted using the results obtained after 30,000 iterations. There is a good match between the results.
Figure 4-13 Overlapped u-velocity contour plot at 30,000 iterations
Flood - Sub Blocked Grid
Dotted Line - Normal Grid

Figure 4-14 Overlapped v-velocity contour plot at 30,000 iterations
Flood - Sub Blocked Grid
Dotted Line - Normal Grid
c.) Center Line Plot:

The midline u-velocity profile from simulations run to completion is in agreement with Ghia et al. [80].

![Midline u-velocity profile](image)

Figure 4-15 The midline u-velocity profile

d.) Residual Plot:

The convergence patterns of the residuals for the 30 x 30 subblocked grid and the unblocked version of the code is shown in Figure 4-16. The residue shown is the square root of the sum of square of the residuals of velocity and pressure. The trend of the residue plot is similar to that obtained for external blocking. The rate of convergence of the internally blocked grid seems to be slightly slower than the unblocked grid.
4.7 SUMMARY

This chapter introduced the concept of internal blocking along with a detailed description of the methodology used in applying it to the GHOST code. It was noticed that the performance enhancements obtained by the application of this technique is comparable to that of external blocking. Performance improvements of up to 79% in walltime were observed when compared to the unsubblocked code. A minor decrease in the walltime performance of the 30 x 30 block was noticed for the larger grid sizes when compared to external blocking. But the overall performance of internal blocking was seen to be the same as external blocking. Also since internal blocking completely automates the process of breaking up the grid into smaller blocks, it is more convenient for the user.

Up to now we have tested the efficacy of the two blocking techniques namely External and Internal blocking with successful results when applied to the lid-driven cavity flow. These techniques now need to be tested on unsteady and turbulent test cases with more complicated grids and boundary conditions to establish the universality of the approach. Initial work in this direction is discussed in the next chapter.
CHAPTER – 5

5. UNSTEADY TEST CASE RESULTS

In the previous two chapters we discussed the concept of external and internal blocking. From the performance test results we noticed that both of these were successful in improving the performance of the code. The test case that we considered to test the performance improvements was a steady flow problem, with a simple grid and straight-forward boundary conditions. In this chapter we look at the initial results obtained with the application of internal blocking to a time dependent test case with more complicated grid and boundary conditions. The reason for carrying out the following tests using internal blocking only is because in the previous chapter we noticed that the technique of internal blocking performs almost the same as external blocking. Also, since internal blocking is an automated process, it is easier to apply it to complicated grids.

5.1 LAMINAR UNSTEADY TEST CASE

The flow over an airfoil inside a 24x24 inch wind tunnel section was chosen as the test case for a laminar time dependent problem. The experimental setup for this case is as shown in Figure 5-1. The wind tunnel experiments for this case have been carried out by Munday and Jacob [82,83]. Katam [84] has performed an extensive study of this case computationally using the GHOST code. The wing used for the numerical runs in our study is shown in Figure 5-2. It simulates a modified NACA 4415 airfoil. It comprises of 273 x 120 grid points. The overall computational grid consisting of two-dimensional multi-zonal blocks is shown in Figure 5-3. The airfoil grid overlaps the central background grid. This background grid is surrounded by eight other rectangular grids.

On the outer boundary, the left (inlet) boundary is fixed with a uniform dimensionless inlet velocity $u_\infty = 1.0$ and the upper and lower boundary condition are no-slip wall boundaries representing the top and bottom of the wind tunnel test section. For the airfoil blocks, the inner boundary condition is a no-slip wall boundary condition, and the outside boundary is set to “overlap” which allows the background grid points to be overlapped by the airfoil block grid points to interpolate values from the foreground airfoil grid points.
Figure 5-1 24 x 24 inch experimental test section

Figure 5-2 Airfoil grid used for computation
Computation information between adjacent blocks is exchanged by two ghost points. All the parameters chosen in the computation are dimensionless. Care is taken to see that the initial near-wall dimensionless $y^+$ values of the airfoil grid are less than 0.5.

![Figure 5-3 Grid used for 24 x 24 inch wind tunnel section](image)

Table 5-1 Block sizes used to break up the airfoil grid

<table>
<thead>
<tr>
<th>Zone</th>
<th>Grid Size</th>
<th>Subblock Size</th>
<th>Zone</th>
<th>Grid Size</th>
<th>Subblock Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$i \times j$</td>
<td>$sni \times snj$</td>
<td></td>
<td>$i \times j$</td>
<td>$sni \times snj$</td>
</tr>
<tr>
<td>1</td>
<td>273 x 120</td>
<td>30 x 30</td>
<td>6</td>
<td>300 x 40</td>
<td>30 x 40</td>
</tr>
<tr>
<td>2</td>
<td>50 x 100</td>
<td>50 x 50</td>
<td>7</td>
<td>50 x 40</td>
<td>50 x 40</td>
</tr>
<tr>
<td>3</td>
<td>300 x 100</td>
<td>30 x 50</td>
<td>8</td>
<td>50 x 40</td>
<td>50 x 40</td>
</tr>
<tr>
<td>4</td>
<td>50 x 100</td>
<td>50 x 50</td>
<td>9</td>
<td>300 x 40</td>
<td>30 x 40</td>
</tr>
<tr>
<td>5</td>
<td>50 x 40</td>
<td>50 x 40</td>
<td>10</td>
<td>50 x 40</td>
<td>50 x 40</td>
</tr>
</tbody>
</table>

5.2 PERFORMANCE TEST RESULTS

Table 5-1 contains the information regarding the grid sizes of the individual blocks in the grid. It also contains the block sizes that were used to break up the grid. We can see that the airfoil grid block contains the maximum number of grid points (32,760) compared to the other
blocks in the grid. Hence we split up the grid into blocks of size 30 x 30. The grid block behind the airfoil grid also consists of a high number of grid points (30,000), but a major part of this grid is overlapped by the airfoil and is not considered during calculations i.e. no calculations are performed on that part of the grid. Hence computationally the grid size for that particular block is lower than the actual grid size. So this grid was broken up using larger sized blocks. The four blocks at the corners of the grid are quite small (50 x 40). They are already quite cache efficient. Hence even if we did block them into smaller blocks we would not get any performance improvements. In fact it might lead to deterioration in the performance since we will unnecessarily be breaking up the grid and putting it back together after every iteration. The blocks above and below the background grid are long in the $i$ direction and small in the $j$ direction (300 x 40). Hence they were only split in the $i$ direction only. The other two grids were split into blocks of size 50 x 50.

The code was run for the duration of 5000 iterations. This was done by running the code for 500 time steps with 10 inner iterations at every time step. A Reynolds number of 100,000 was used. The performance tests were carried out on the various hardware platforms mentioned in chapter 2. A comparison of the actual walltime (i.e. walltime which has not been normalized by the grid size and number of iterations) of the internally blocked grid versus the unblocked grid for 5000 iterations and over the entire grid is shown in Figure 5-4. On KFC4 the code was compiled using the g95 compiler with the -O3 optimization level, whereas on the other three hardware platforms it was compiled using the Ifort compiler with the -O3 optimization level. The percentage improvement in walltime for the various platforms is mentioned in Table 5-2. The code used for this performance test was essentially version V0. The maximum percentage improvement of 27% was obtained on KFC4.

An interesting observation can be made regarding the magnitude of the performance improvement obtained on the KFC5 versus KFC6 (AMD 4200+). The percentage improvement obtained on KFC5 is less than KFC6 in spite of KFC6 being a faster processor. This anomaly might have been caused due to the fact that even though the KFC6 - AMD 4200 has a greater clock speed (2.2 Ghz) its L2 cache size is 2 x 512 Kb. This processor has dual core, hence it is processing at a high speed, but the cache does not seem to be keeping up with it.
Table 5-2 Percentage improvement obtained by application of internal blocking on the various hardware platforms

<table>
<thead>
<tr>
<th>Platform</th>
<th>No Subblock Walltime (secs)</th>
<th>Internal Block Walltime (secs)</th>
<th>Percentage Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>KFC4</td>
<td>3323</td>
<td>2400</td>
<td>27%</td>
</tr>
<tr>
<td>KFC5</td>
<td>1559</td>
<td>1317</td>
<td>15.5%</td>
</tr>
<tr>
<td>KFC6 (AMD 4200+)</td>
<td>1434</td>
<td>1126.6</td>
<td>21%</td>
</tr>
<tr>
<td>KFC6 (Intel E6400)</td>
<td>1263</td>
<td>1150</td>
<td>9%</td>
</tr>
</tbody>
</table>

This could be leading to a higher rate of cache misses, in turn leading to bad performance for the unblocked grid. But when internal blocking is applied, it enhances the cache performance, in turn leading to better performance. Whereas KFC5 has a slightly slower processor (2.08 Ghz) with a 512 Kb L2 cache. This being a single core, the cache seems to be keeping up with the processor, due to which the performance improvement obtained by applying internal blocking is slightly less than that obtained on KFC6.
The least amount of performance improvement due to the application of blocking was observed on the KFC6 - Intel 6400 processor. Due to the larger L2 cache (2 Mb), the cache performance for the unblocked code is quite good. Hence with the application of internal blocking a significant improvement is not noticed.

A 27% performance improvement was noticed on the KFC4 node. Since the code used to carry out this test was an unoptimized GHOST version (V0), this value seems to be lower than the performance improvement reported for the lid-driven cavity flow test case. The reason for this decrease is that the grid of the airfoil test case comprises of smaller grids. The performance improvements obtained by the application of blocking on these grids is not as high as that obtained by blocking grids of sizes greater than 250,000. In Table 5-3, from the data that was available for the lid-driven flow problem, we have tried to extrapolate the performance improvements. The weighted average of the extrapolated values of the performance improvements is 31%. This shows that we can predict to a certain extent the amount of performance improvement that can be achieved by the application of internal blocking.

Table 5-3 Performance improvements obtained by the application of external blocking to lid-driven cavity problem

<table>
<thead>
<tr>
<th>Zone</th>
<th>Grid Size $i \times j$</th>
<th>Performance Improvement</th>
<th>Zone</th>
<th>Grid Size $i \times j$</th>
<th>Performance Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>273 x 120</td>
<td>39%</td>
<td>6</td>
<td>300 x 40</td>
<td>29%</td>
</tr>
<tr>
<td>2</td>
<td>50 x 100</td>
<td>10%</td>
<td>7</td>
<td>50 x 40</td>
<td>0%</td>
</tr>
<tr>
<td>3</td>
<td>300 x 100</td>
<td>32%</td>
<td>8</td>
<td>50 x 40</td>
<td>0%</td>
</tr>
<tr>
<td>4</td>
<td>50 x 100</td>
<td>10%</td>
<td>9</td>
<td>300 x 40</td>
<td>29%</td>
</tr>
<tr>
<td>5</td>
<td>50 x 40</td>
<td>0%</td>
<td>10</td>
<td>50 x 40</td>
<td>0%</td>
</tr>
</tbody>
</table>

5.3 Accuracy Test Results

The accuracy tests for this case were run on the KFC4 server alone. A Reynolds number of 100,000 along with a time step $dt = 0.00025$ was used. Twelve inner iterations were performed at each time step. The code was run for 15 non dimensional time steps. We were unable to get a good level of accuracy when the grid was subblocked in the same manner as it was done while conducting the performance tests. This might be due to a coding error in the way the boundary conditions are being implemented. Still, a good level of accuracy was obtained by performing internal blocking on the airfoil grid alone. This resulted in a performance.
improvement of 9% on KFC4. The airfoil was split into various sized blocks and the co-efficient of drag and lift results obtained were compared with the unblocked grid as shown in Figures 5-5, 5-6 and 5-7. The airfoil grid was initially split up into two halves of sizes 137 x 120, in the vertical direction. A good agreement of the results between the blocked and unblocked grid [Figure 5-5] was noticed. Later the grid was further split up into four blocks of size 69 X 120. The grid was also split horizontally into four blocks of size 137 X 60. The grid with 4 vertical blocks gave similar accuracy as its predecessor Figure 5-6.

But the horizontally split blocks coefficient [Figure 5-7] of lift results did not match the unblocked case up to the first seven non dimensional time steps. But after that, once the steady state oscillations were reached we noticed a good agreement between the two.

Figure 5-5 - Comparison of lift and drag curves for the blocked (block size - 137x120) and unblocked grid
Figure 5-6 - Comparison of lift and drag curves for the blocked (block size - 69x120) and unblocked grid

Figure 5-7 - Comparison of lift and drag curves for the blocked (block size - 137x60) and unblocked grid
5.4 SUMMARY

The technique of internal blocking was applied with a certain degree of success to a complex problem with complicated grid and flow conditions. This technique can be fully automated making it readily accessible. It was shown that the potential improvement that can be achieved by the application of internal blocking can be predicted from existing data. The accuracy of the solution obtained by blocking of the grid into smaller blocks to achieve optimum performance improvements was not good. Hence we were unable to achieve the maximum potential improvements possible by the application of this technique while maintaining accuracy. Also the accuracy of the solution was affected when the problem was solved on multiple parallel processors using MPI. These problems are not insurmountable. They are not being caused due to a weakness in the idea of internal blocking but seem to be caused due to an error in the code which can be fixed.
CHAPTER - 6

6. CONCLUSIONS AND FUTURE WORK

6.1 SUMMARY AND CONCLUSIONS

GHOST has been successfully optimized earlier by LeBeau et. al. [4]. They were able to obtain a performance improvement of 20% relative to the original code for small grids and up to 6 times faster for the largest grid. Still, at the end of that study it was noticed that there was still scope for improvement in the cache performance of the code.

In this work the main objective has been to optimize the code even further and to maximize the cache usage. Two techniques: External blocking and Internal blocking were implemented with the intention of achieving this goal and various tests were conducted to study their effectiveness and the performance enhancements that could be obtained. Both of these techniques did help significantly in reducing the cache misses and enhancing the code performance.

External blocking was tested initially. The philosophy behind this technique was to transfer the data required to carry out the calculations into the cache just before it was invoked by the processor, thereby reducing the idle time spent by the processor in waiting for the data. This was mainly achieved by splitting up the grid during the grid generation process into small blocks whose size was comparable to the L2 cache size. Blocks of sizes varying from 900 to 4900 grid points were tested. The initial tests were conducted using the laminar lid-driven cavity flow test case. With the implementation of this technique in the previously optimized laminar code (V3), a further improvement of up to 28% in performance was observed for all grid sizes. For the original unoptimized code (V0) the performance improvements were in the range of 75% - 80% for the larger grid sizes. The best performance improvement was obtained by using blocks of 900 grid points. These improvements were obtained on KFC4 which has a comparatively older processor design. In order to study the effectiveness of this technique on the newer hardware, tests were carried out on the currently state-of-the-art processors with dual core architecture from both Intel and AMD. A performance improvement for the unoptimized code of around 20% was observed on the Intel processor containing a larger L2 cache size (2 Mb) compared to a 48% improvement for the AMD processor with a smaller L2 cache size (1 Mb). From these tests it was noticed that the potential performance improvements that could be gained
by the application of external blocking were more related to the cache size on the processor rather than the processing speed. Tests were also carried out to study the effects on performance for different compilers and different compiler optimization levels. A noticeable performance improvement was observed in the code compiled using the Intel Fortran compiler (IFC) when compared to the code compiled with G95. The performance improvement gained with the application of external blocking to the optimized code was greater for the IFC compiled code of up to 40%, whereas an improvement of up to 28% was noticed when the code was compiled with G95. With respect to the various compiler optimization levels, as anticipated, the overall performance improved with the increase of the optimization level used to compile the code. The percentage improvement obtained with the application of external blocking remained the same with change in optimization levels. Another interesting test was a study on the effect of block shape on performance. For this test, blocks having the same number of total grid points as the square blocks but different number of grid points in the $i$ and $j$ direction i.e. rectangular blocks were used. The performance remained the same irrespective of the shape of the block. With this test it was verified that the performance improvements were dependent on the total block size irrespective of the shape. A couple of performance tests were carried out to evaluate the technique on turbulent test cases. Because when a turbulence model is switched on in the code, several more data arrays and computations are added to the calculations, leading to longer runtimes. With the application of external blocking to the unoptimized code, an improvement of around 70% - 75% for the larger grid sizes was observed. In the case of the optimized code, around 53% - 56% improvements were achieved which is almost twice the improvement that was noticed for the laminar code. Overall with the application of this technique a significant performance improvement was achieved.

The performance test results from external blocking suggested that the improvement potentially gained from blocking could well be approximated by looking at the single grid normalized walltime on small grids. Hence, if someone wants to decide if it is justifiable to spend time and effort into applying this technique to their code, they just need to conduct a couple of performance tests with various grid sizes without blocking it. Depending upon the initial slope obtained in the walltime plot, they can gauge the potential improvement that can be obtained by the application of this technique and make a decision based on that. Another advantage of this technique is that in order to use it, the user does not need to have an in-depth understanding of the CFD code. The user must just be able to break up the grid into smaller sized
blocks. In theory this technique can potentially be applied to any parallel structured grid code to improve its cache performance.

Looking at the performance curves from the tests, it was theoretically deduced that the performance improvements that were achieved with the application of external blocking were due to the reduction in cache misses. In order to test this theory, several cache simulation tests were carried out using “Cachegrind” cache profile simulator of the Valgrind Toolkit. From the results of these tests it was noticed that with the application of external blocking the L2 cache miss rate was reduced to almost 0% and this was the leading cause for the improvement in the performance. The penalty of external blocking was a significant rise in the number of data calls, caused due to the introduction of the ghost points at the boundaries of the blocks. Overall, compared to the amount of improvement in performance that was gained from reducing the cache misses, the decrease in performance due to the increase in data calls was not significant. These tests gave us a much better insight into how and why external blocking works. It proved that the philosophy behind the application of external blocking, which was to reduce the cache misses, was actually happening in practice and provided a method to quantify it. Multiple tests were carried out to verify the accuracy of the externally blocked simulations. An extremely good match was obtained between the blocked and unblocked grid results. The results were also compared with other sources and there was a good match even with them as well.

The biggest drawback of this technique was its difficulty in implementation. All the tests that were carried out using external blocking were done on the lid-driven cavity flow test case. This test case had a fairly simple grid with straightforward boundary conditions. In order to test this technique on this case, a separate code had to be written which would read the input script required by the grid generator for GHOST to construct the grid. It would then generate another input script with the help of which the grid generator could construct the lid-driven cavity flow grid which was split into multiple blocks. Hence a whole new code would have to be written for different grids/problems and in some cases if the grid consisted of multiple blocks or complicated boundary conditions, it would be challenging to write a code to externally block it. In spite of this technique being quite successful in terms of performance improvements and reducing cache misses; it was not the most practical approach to implement for GHOST. We had to come up with a viable automated system which would split the grid up into smaller blocks irrespective of the kinds of boundary conditions that the grid had or the number of zones. This
led to the idea of internal blocking which is basically an automated version of external blocking within the Ghost code.

The basic philosophy behind internal blocking is exactly the same as that of external blocking. Only the means used to achieve it is slightly different. With internal blocking, the process of breaking up of the grid is done inside the CFD code rather than doing it externally during the grid generations process. This technique was equally successful as external blocking at enhancing the performance of the code. Since the philosophy behind both the techniques was the same, fewer performance and cachegrind tests were performed to check the efficacy of this technique.

With the implementation of this technique to the previously optimized laminar code, a further improvement of up to 26% in performance was observed for all grid sizes compared to 28% observed with external blocking. For the unoptimized code the performance improvements were in the range of 74% - 79% for the larger grid sizes. The main reason for the slight decrease in performance is due to the overhead caused by the splitting and rejoining of the grid at each iteration.

The cachegrind results obtained showed similar trends as external blocking. For example, the reduction in L2 cache miss to almost 0% and an increase in the number of data calls caused due to the introduction of the ghost points. When compared to external blocking, there was a slight increase in the number of data calls due to the additional subroutines that were added.

Accuracy tests were conducted in a similar manner as the ones conducted to test the effectiveness of external blocking. The results obtained were a good match to the results obtained using the unblocked grid and those obtained from other references.

Looking at the results of all the tests that were carried out it was verified that the internal blocking technique was equally good as external blocking. All the tests had been carried out using the lid-driven cavity flow test case. In spite of the complicated flow in this problem, it is a laminar steady flow problem with a fairly simple grid and boundary conditions. In order to test the practicality of this technique it had to be tested on a problem consisting of a more complicated grid, boundary conditions, and flow. Hence, the flow over an airfoil problem was chosen. This problem is a classic unsteady flow problem and has been extensively studied both experimentally and computationally. This problem also has a complicated grid and various boundary conditions. The performance test results that were conducted using this case were quite encouraging. An improvement of 27% was observed with the application of internal blocking to
the unoptimized code. This reduction in the performance improvement when compared to what
was obtained with the lid-driven cavity (~78%) is due to the fact that the grid used in this
problem was already effectively partially externally blocked. This was originally done in order to
be able to perform parallel processing on the grid. But when the problem is solved on a single
processor, it leads to a certain amount of improvement due to external blocking. Also the larger
performance improvements that were mentioned earlier were for larger grid sizes (~ 250,000 to
360,000 grid points), whereas the airfoil grid consisted of grids of comparatively smaller sizes (~
32,760 grid points).

The accuracy test results from this case have not been encouraging so far. The results are
not accurate when the grid is blocked extensively. With the partial blocking of just the airfoil
grid a performance improvement of 9% has been achieved. This does provide accurate results. It
seems like the problem is being caused due to incorrect boundary conditions being created for
the block boundaries when the grid is blocked into smaller blocks. Work is on going to fix this
issue.

Despite the problems that have been mentioned above, the optimization process was quite
successful. Both the techniques have been thoroughly investigated and established as
performance enhancing methods for two dimensional simulations using structured CFD codes. It
has been shown that any technique that helps in reducing cache misses leads to an improvement
in the performance of the code. A summary of the maximum improvements that have been
achieved with both the techniques for the lid-driven cavity case are shown in the Table. 6.1

### Table 6-1 Summary of the performance improvements obtained

<table>
<thead>
<tr>
<th>Case</th>
<th>External Blocking</th>
<th>Internal Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V0</td>
<td>V3</td>
</tr>
<tr>
<td>2D Steady Laminar</td>
<td>80%</td>
<td>29%</td>
</tr>
<tr>
<td>2D Steady Turbulent</td>
<td>75%</td>
<td>56%</td>
</tr>
<tr>
<td></td>
<td>79%</td>
<td>26%</td>
</tr>
</tbody>
</table>

#### 6.2 Future Work

This work is the result of the future work from a previous effort [4]. It has been quite
successful to a great extent in attaining its main objective viz. to reduce the L2 cache miss to the
lowest possible extent. It does not seem possible to make changes to the code to get a justifiable
performance improvement by reducing the L2 cache miss rate further.

Some of the problems related to the accuracy of the solution that have been noticed
during the testing of the airfoil test case need to be resolved. The solution obtained is not
accurate when internal blocking is switched on and the problem is solved on multiple processors in parallel. This error might be caused due to a problem with the MPI communications. When the entire grid is split into smaller sized blocks, the solution is not accurate. This might be cause due to incorrect boundary condition allocation at the artificial boundaries.

When the turbulence model is switched on in the code, several more data arrays and subroutines are introduced. Internal blocking has to be implemented on these data arrays.

External blocking was seen to be more robust when compared to internal blocking plus there was no need to touch the main CFD code in order to implement it. Also the performance improvements obtained was slightly better. Hence, a possible option is to develop a code that can perform external blocking on complex grids by creating the appropriate input file to generate the externally blocked grid.
APPENDIX

A.1 Steps to Implement Internal Blocking to GHOST

1. Define the variables “internal”, “sni” and “snj”.

   LOGICAL::internal
   INTEGER::sni,snj

2. Define the type “sgrid” and “block”

   TYPE, public :: sgrid
   INTEGER :: sni,snj,xint,yint
   REAL (high), POINTER, DIMENSION (:, :) :: swfx, swbx
   REAL (high), POINTER, DIMENSION (:, :) :: swfy, swby
   REAL (high), POINTER, DIMENSION (:, :) :: sx, sy, svol, sdwall,i,j
   REAL (high), POINTER, DIMENSION (:, :) :: sae_x, sae_y, san_xt, san_yt
   INTEGER, POINTER, DIMENSION (:, :) :: sinx
   REAL (high), POINTER, DIMENSION (:, :) :: sx, sy, svol, sdwall,i,j
   REAL (high), POINTER, DIMENSION (:, :) :: sae_x, sae_y, san_xt, san_yt
   INTEGER, POINTER, DIMENSION (:, :) :: sinx
   REAL (high), POINTER, DIMENSION (:, :) :: su, sv, sp, stk, sed, sgamma, st
   REAL (high), POINTER, DIMENSION (:, :) :: suo, svo, stk0, sed0, sgamma0, sto
   REAL (high), POINTER, DIMENSION (:, :) :: suoo, svoo, stkoo, sedoo, sgammaoo, stoo
   REAL (high), POINTER, DIMENSION (:, :) :: stvis
   INTEGER :: n_real_bc_point
   TYPE (real_bc_desc_t), POINTER, DIMENSION (:) :: real_bc
   INTEGER :: n_ghost_bc_point
   TYPE (ghost_bc_desc_t), POINTER, DIMENSION (:) :: ghost_bc
   END TYPE sgrid

   TYPE, public :: block
   INTEGER :: xblock,yblock,xextra,yextra,nblock
   TYPE (sgrid), POINTER, DIMENSION (:) :: szone
   END TYPE block

3. Create the subroutines break_velocity, combine_velocity ,internal_block and gen_real_bc.

   SUBROUTINE break_velocity (u,uo,uoo,v,vo,voo,tk,tko,tkoo,ed,edo,edoo,gamma,
   gammao,gammaoo,t,to,too,p,tvis,szone,nblock,itr)
   IMPLICIT NONE
   REAL (high), DIMENSION (0:,0:) :: u,v,p,tk,ed,gamma,t,tvis
   REAL (high), DIMENSION (0:,0:) :: uo,vo,tko,edo,gammao,to
   REAL (high), DIMENSION (0:,0:) :: uoo,voo,tkoo,edo0,gammaoo,too
   TYPE (sgrid), DIMENSION (:) :: szone
   INTEGER::nblock,itr
   INTEGER::xi,yj,iblock,sni,snj,si,sj

   DO iblock = 1,nblock
      sni = szone(iblock)%sni + 1
      snj = szone(iblock)%snj + 1
      DO sj = 0,snj
         DO si = 0,sni
            xi = szone(iblock)%i(si,sj)
            yj = szone(iblock)%j(si,sj)
            szone(iblock)%su(si,sj) = u(xi,yj)
            szone(iblock)%sv(si,sj) = v(xi,yj)
         END DO
      END DO
   END DO
IF (imodel /= 0) THEN
  DO sj = 0, snj
    DO si = 0, sni
      xi = szone(iblock)%i(si,sj)
      yj = szone(iblock)%j(si,sj)
      szone(iblock)%stk(si,sj) = tk(xi,yj)
      szone(iblock)%sed(si,sj) = ed(xi,yj)
      szone(iblock)%stvis(si,sj) = tvis(xi,yj)
    END DO
  END DO
END IF

IF (intermittency) THEN
  DO sj = 0, snj
    DO si = 0, sni
      xi = szone(iblock)%i(si,sj)
      yj = szone(iblock)%j(si,sj)
      szone(iblock)%sgamma(si,sj) = gamma(xi,yj)
    END DO
  END DO
END IF

IF (ltemperature) THEN
  DO sj = 0, snj
    DO si = 0, sni
      xi = szone(iblock)%i(si,sj)
      yj = szone(iblock)%j(si,sj)
      szone(iblock)%st(si,sj) = t(xi,yj)
    END DO
  END DO
END IF

IF (itr == 1) THEN
  DO sj = 1, snj-1
    DO si = 1, sni-1
      xi = szone(iblock)%i(si,sj)
      yj = szone(iblock)%j(si,sj)
      szone(iblock)%suo(si,sj) = uo(xi,yj)
      szone(iblock)%suoo(si,sj) = uoo(xi,yj)
      szone(iblock)%svo(si,sj) = vo(xi,yj)
      szone(iblock)%svoo(si,sj) = voo(xi,yj)
    END DO
  END DO
END IF

IF (imodel /= 0) THEN
  DO sj = 0, snj
    DO si = 0, sni
      xi = szone(iblock)%i(si,sj)
      yj = szone(iblock)%j(si,sj)
      szone(iblock)%stko(si,sj) = tko(xi,yj)
      szone(iblock)%sedo(si,sj) = edo(xi,yj)
      szone(iblock)%stkoo(si,sj) = tkoo(xi,yj)
      szone(iblock)%sedoo(si,sj) = edoo(xi,yj)
    END DO
  END DO
END IF
IF (intermittency) THEN
  DO sj = 0,snj
    DO si = 0,sni
      xi = szone(iblock)%i(si,sj)
      yj = szone(iblock)%j(si,sj)
      szone(iblock)%sgammao(si,sj) = gammao(xi,yj)
      szone(iblock)%sgammaoo(si,sj) = gammaoo(xi,yj)
    END DO
  END DO
END IF

IF (ltemperature) THEN
  DO sj = 0,snj
    DO si = 0,sni
      xi = szone(iblock)%i(si,sj)
      yj = szone(iblock)%j(si,sj)
      szone(iblock)%sto(si,sj) = to(xi,yj)
      szone(iblock)%stoo(si,sj) = too(xi,yj)
    END DO
  END DO
END IF

END IF

END SUBROUTINE break_velocity

SUBROUTINE combine_velocity(u,v,tk,ed,gamma,t,p,tvis,szone,nblock)
IMPLICIT NONE
REAL(high), DIMENSION(0:,0::) :: u, v, tk, ed, gamma, t, p, tvis
TYPE(sgrid), DIMENSION(:) :: szone
INTEGER :: nblock
INTEGER :: xi, yj, iblock, sni, snj, si, sj

DO iblock = 1,nblock
  sni = szone(iblock)%sni - 1
  snj = szone(iblock)%snj - 1
  DO sj = 2,snj
    DO si = 2,sni
      xi = szone(iblock)%i(si,sj)
      yj = szone(iblock)%j(si,sj)
      u(xi,yj)=szone(iblock)%su(si,sj)
      v(xi,yj)=szone(iblock)%sv(si,sj)
      p(xi,yj)=szone(iblock)%sp(si,sj)
    END DO
  END DO
  IF (imodel /= 0) THEN
    DO sj = 0,snj
      DO si = 0,sni
        xi = szone(iblock)%i(si,sj)
        yj = szone(iblock)%j(si,sj)
        tk(xi,yj)= szone(iblock)%stk(si,sj)
        ed(xi,yj)= szone(iblock)%sed(si,sj)
        tvis(xi,yj)= szone(iblock)%stvis(si,sj)
      END DO
    END DO
  END IF
END IF

END IF

IF (intermittency) THEN
  DO sj = 0,snj
    DO si = 0,sni
      xi = szone(iblock)%i(si,sj)
      yj = szone(iblock)%j(si,sj)
      ...
xi = szone(iblock)%i(si,sj)
yj = szone(iblock)%j(si,sj)
gamma(xi,yj) = szone(iblock)%sgamma(si,sj)
END DO
END DO
END IF
IF (ltemperature) THEN
DO sj = 0,snj
  DO si = 0,sni
    xi = szone(iblock)%i(si,sj)
yj = szone(iblock)%j(si,sj)
t(xi,yj) = szone(iblock)%st(si,sj)
  END DO
END DO
END IF
END DO
END SUBROUTINE combine_velocity

SUBROUTINE internal_block (zone,intzone,number_zone,bsni,bsnj)
! ## THE CALL FOR THIS SUBROUTINE MUST BE PLACED JUST BEFORE THE LOOP WHERE
! CALC_FLOWFIELD IS CALLED BECAUSE I AM BREAKING THE VARIABLES UEAST AND VNORTH. IF THE
! CODE IS RESTARTED, THESE VALUES ARE READ JUST BEFORE THE CALL TO THE SUBROUTINE
! CALC_FLOWFIELD.#################
IMPLICIT NONE
TYPE(grid),POINTER,DIMENSION(:)::zone
TYPE(block),POINTER,DIMENSION(:)::intzone
INTEGER::number_zone
INTEGER:: ni,nj,i,zone,iblock,xi,yj
INTEGER:: sni,snj,snim1,snjm1,nblock,si,sj,xiblock,yiblock,xint,yint
INTEGER:: snnj,nnj,ighost
INTEGER::xblock,yblock,xextra,yextra,nni,ssni
INTEGER::xtemp,ytemp,block_calc
INTEGER::bsni,bsnj
ALLOCATE(intzone(number_zone))
DO izone = 1,number_zone
  ! Calculating the best block size
  ni = zone(izone)%ni
  nj = zone(izone)%nj
  IF ((sni > ni) .or. (snj > nj)) THEN
    WRITE(*,*)'ERROR:Subblock size should be less than grid size'
    WRITE(*,*)' Check zone number =',izone
    WRITE(*,*)'ni=',ni,'nj=',nj,'sni=',sni,'snj=',snj
    EXIT
  END IF
  ! Calculating the number of blocks in y direction
  nnj = nj + 2
  ssnj = snj + 2
  IF ( MOD((nnj-ssnj),(ssnj-4)) == 0 ) THEN
    yblock = ((nnj-ssnj)/(ssnj-4))+1
    yextra = 0
  ELSE
    WRITE(*,*)'ERROR: Subblock size should be less than grid size'
    WRITE(*,*)' Check zone number =',izone
    WRITE(*,*)'ni=',ni,'nj=',nj,'sni=',sni,'snj=',snj
    EXIT
  END IF
END DO
ELSE
  yblock = ((nnj-ssnj)/(ssnj-4))+1
  yextra = nnj - ( ssnj + ((yblock-1)*(ssnj-4)))
END IF

! Calculating the number of blocks in x directions
nni = ni+2
ssni = sni + 2
IF ( MOD((nni-ssni),(ssni-4)) == 0 )  THEN
  xblock = ((nni-ssni)/(ssni-4))+1
  xextra = 0
ELSE
  xblock = ((nni-ssni)/(ssni-4))+1
  xextra = nni - ( ssni + ((xblock-1)*(ssni-4)))
END IF

! Logic to decide what is to be done with the grid points in the end.
!If the number of grid points in the end are greater than a third of the block size
!then a new block is created.
!If not it is just added to the last block.
!============================================================================
IF(yextra > (snj/3)) THEN
  yblock = yblock + 1
END IF
IF (xextra > (sni/3)) THEN
  xblock = xblock + 1
END IF
!=============================================================================
nblock = xblock * yblock
ALLOCATE(intzone(izone)%szone(nblock))
intzone(izone)%nblock = nblock
intzone(izone)%xextra = xextra
intzone(izone)%yextra = yextra
intzone(izone)%xblock = xblock
intzone(izone)%yblock = yblock

iblock = 1
xi = 0
yj = 0
xint = 0
yint = 0

!I start sweeping through the grid one block at a time.
! The if structures are deciding the size of the block.

DO yiblock = 1,yblock
  IF ((yiblock == yblock) .and.(yextra /= 0))THEN
    IF (yextra <= (snj/3))THEN
      ytemp = snj
      snj = snj+yextra
    ELSE
      ytemp = snj
      snj = yextra
    END IF
  END IF
END IF

DO xiblock = 1,xblock
  IF ((xiblock == xblock).and.(xextra /= 0))THEN
    IF (xextra <= (sni/3))THEN
      xtemp = sni
      sni = sni+xextra
    ELSE
      xtemp = sni
      sni = xextra
    END IF
  END IF
END IF
xtemp = sni
sni = xextra
END IF
END IF

intzone(izone)%szone(iblock)%sni = sni
intzone(izone)%szone(iblock)%snj = snj
ALLOCATE(intzone(izone)%szone(iblock)%sx(0:sni+1,0:snj+1))
ALLOCATE(intzone(izone)%szone(iblock)%sy(0:sni+1,0:snj+1))
ALLOCATE(intzone(izone)%szone(iblock)%i(0:sni+1,0:snj+1))
ALLOCATE(intzone(izone)%szone(iblock)%j(0:sni+1,0:snj+1))
ALLOCATE(intzone(izone)%szone(iblock)%swbx(1:3,1:sni-1,1:snj))
ALLOCATE(intzone(izone)%szone(iblock)%swfx(1:3,1:sni-1,1:snj))
ALLOCATE(intzone(izone)%szone(iblock)%swby(1:3,1:sni,1:snj-1))
ALLOCATE(intzone(izone)%szone(iblock)%swfy(1:3,1:sni,1:snj-1))
ALLOCATE(intzone(izone)%szone(iblock)%svol(0:sni+1,0:snj+1))
ALLOCATE(intzone(izone)%szone(iblock)%sdwall(0:sni+1,0:snj+1))
ALLOCATE(intzone(izone)%szone(iblock)%sae_x(0:sni,1:snj))
ALLOCATE(intzone(izone)%szone(iblock)%sae_y(0:sni,1:snj))
ALLOCATE(intzone(izone)%szone(iblock)%san_xt(0:sni,1:snj))
ALLOCATE(intzone(izone)%szone(iblock)%san_yt(0:sni,1:snj))
ALLOCATE(intzone(izone)%szone(iblock)%san_x(1:sni,0:snj))
ALLOCATE(intzone(izone)%szone(iblock)%san_y(1:sni,0:snj))
ALLOCATE(intzone(izone)%szone(iblock)%sae_xt(1:sni,0:snj))
ALLOCATE(intzone(izone)%szone(iblock)%sae_yt(1:sni,0:snj))
ALLOCATE(intzone(izone)%szone(iblock)%sinx(0:sni+1,0:snj+1))
ALLOCATE(intzone(izone)%szone(iblock)%su(0:sni+1,0:snj+1))
ALLOCATE(intzone(izone)%szone(iblock)%suo(1:sni,1:snj))
ALLOCATE(intzone(izone)%szone(iblock)%sv(0:sni+1,0:snj+1))
ALLOCATE(intzone(izone)%szone(iblock)%svo(1:sni,1:snj))
ALLOCATE(intzone(izone)%szone(iblock)%sp(0:sni+1,0:snj+1))
ALLOCATE(intzone(izone)%szone(iblock)%sueast(0:sni,1:snj))
ALLOCATE(intzone(izone)%szone(iblock)%svnoth(0:sni,1:snj))

IF (imodel == 0) THEN
  ALLOCATE(intzone(izone)%szone(iblock)%stk(0,0))
  ALLOCATE(intzone(izone)%szone(iblock)%sed(0,0))
  ALLOCATE(intzone(izone)%szone(iblock)%stvis(0,0))
ELSE
  ALLOCATE(intzone(izone)%szone(iblock)%stk(0:sni+1,0:snj+1))
  ALLOCATE(intzone(izone)%szone(iblock)%sed(0:sni+1,0:snj+1))
  ALLOCATE(intzone(izone)%szone(iblock)%stvis(0:sni+1,0:snj+1))
END IF

IF (intermittency) THEN
  ALLOCATE(intzone(izone)%szone(iblock)%sgamma(0:sni+1,0:snj+1))
ELSE
  ALLOCATE(intzone(izone)%szone(iblock)%sgamma(0,0))
END IF

IF (ltemperature) THEN
  ALLOCATE(intzone(izone)%szone(iblock)%st(0:sni+1,0:snj+1))
ELSE
  ALLOCATE(intzone(izone)%szone(iblock)%st(0,0))
END IF

!Generating a map between the grid and the block
DO sj = 0,snj+1
  DO si = 0,sni+1
    intzone(izone)%szone(iblock)%i(si,sj)= xi
    intzone(izone)%szone(iblock)%j(si,sj)= yj
    xi = xi + 1
  END DO
END DO
END DO
   xi = xint
   yj = yj + 1
END DO

iblock = iblock + 1

xi = (xiblock * (sni+1)) - (xiblock * 3)
xint = xi
yj = yint
IF ((xiblock == xblock) .and. (xextra /= 0)) THEN
   sni = xtemp
END IF
END IF
END DO ! xblock loop

xi = 0
yj = (yiblock * (snj + 1)) - (yiblock * 3)
yint = yj
xint = xi
IF ((yiblock == yblock) .and. (yextra /= 0)) THEN
   snj = ytemp
END IF
END IF
END DO ! yblock loop

! Copying all the grid parameters with the help of the map created above for navigation.
! From here onwards the logic is clean and clear.
DO iblock = 1,nblock
   sni = intzone(izone)%szone(iblock)%sni
   snj = intzone(izone)%szone(iblock)%snj
   DO sj = 0,snj+1
      DO si = 0,sni+1
         xi = intzone(izone)%szone(iblock)%i(si,sj)
yj = intzone(izone)%szone(iblock)%j(si,sj)
intzone(izone)%szone(iblock)%sx(si,sj) = zone(izone)%x(xi,yj)
intzone(izone)%szone(iblock)%sy(si,sj) = zone(izone)%y(xi,yj)
intzone(izone)%szone(iblock)%svol(si,sj) = zone(izone)%vol(xi,yj)
intzone(izone)%szone(iblock)%sdwall(si,sj) = zone(izone)%dwall(xi,yj)
intzone(izone)%szone(iblock)%sinx(si,sj) = zone(izone)%inx(xi,yj)
      END DO
   END DO
END DO

DO sj = 1,snj
   DO si = 0,sni
      xi = intzone(izone)%szone(iblock)%i(si,sj)
yj = intzone(izone)%szone(iblock)%j(si,sj)
intzone(izone)%szone(iblock)%sueast(si,sj) = zone(izone)%ueast(xi,yj)
intzone(izone)%szone(iblock)%sae_x(si,sj) = zone(izone)%ae_x(xi,yj)
intzone(izone)%szone(iblock)%sae_y(si,sj) = zone(izone)%ae_y(xi,yj)
intzone(izone)%szone(iblock)%san_xt(si,sj) = zone(izone)%an_xt(xi,yj)
intzone(izone)%szone(iblock)%san_yt(si,sj) = zone(izone)%an_yt(xi,yj)
      END DO
   END DO
END DO

DO sj = 0,snj
   DO si = 1,sni
      xi = intzone(izone)%szone(iblock)%i(si,sj)
yj = intzone(izone)%szone(iblock)%j(si,sj)
intzone(izone)%szone(iblock)%svnoth(si,sj) = zone(izone)%vnoth(xi,yj)
intzone(izone)%szone(iblock)%san_x(si,sj) = zone(izone)%an_x(xi,yj)
intzone(izone)%szone(iblock)%san_y(si,sj) = zone(izone)%an_y(xi,yj)
intzone(izone)%szone(iblock)%sae_xt(si,sj) = zone(izone)%ae_xt(xi,yj)
intzone(izone)%szone(iblock)%sae_yt(si,sj) = zone(izone)%ae_yt(xi,yj)
   END DO
END DO

106
END DO
END DO

DO sj = 1, snj
  DO si = 1, sni-1
    xi = intzone(izone)%szone(iblock)%i(si,sj)
    yj = intzone(izone)%szone(iblock)%j(si,sj)
    intzone(izone)%szone(iblock)%swbx(1,si,sj) = zone(izone)%wbx(1,xi,yj)
    intzone(izone)%szone(iblock)%swbx(2,si,sj) = zone(izone)%wbx(2,xi,yj)
    intzone(izone)%szone(iblock)%swbx(3,si,sj) = zone(izone)%wbx(3,xi,yj)
    intzone(izone)%szone(iblock)%swfx(1,si,sj) = zone(izone)%wfx(1,xi,yj)
    intzone(izone)%szone(iblock)%swfx(2,si,sj) = zone(izone)%wfx(2,xi,yj)
    intzone(izone)%szone(iblock)%swfx(3,si,sj) = zone(izone)%wfx(3,xi,yj)
  END DO
END DO

DO sj = 1, snj-1
  DO si = 1, sni
    xi = intzone(izone)%szone(iblock)%i(si,sj)
    yj = intzone(izone)%szone(iblock)%j(si,sj)
    intzone(izone)%szone(iblock)%swby(1,si,sj) = zone(izone)%wby(1,xi,yj)
    intzone(izone)%szone(iblock)%swby(2,si,sj) = zone(izone)%wby(2,xi,yj)
    intzone(izone)%szone(iblock)%swby(3,si,sj) = zone(izone)%wby(3,xi,yj)
    intzone(izone)%szone(iblock)%swfy(1,si,sj) = zone(izone)%wfy(1,xi,yj)
    intzone(izone)%szone(iblock)%swfy(2,si,sj) = zone(izone)%wfy(2,xi,yj)
    intzone(izone)%szone(iblock)%swfy(3,si,sj) = zone(izone)%wfy(3,xi,yj)
  END DO
END DO

! Assigning the value of 1 to inx at all the block boundaries.
! (See ch. 4 of thesis for explanation)

DO si = 1, sni+1
  intzone(izone)%szone(iblock)%sinx(si,0) = 1
  intzone(izone)%szone(iblock)%sinx(si,1) = 1
  intzone(izone)%szone(iblock)%sinx(si,snj) = 1
  intzone(izone)%szone(iblock)%sinx(si,snj+1) = 1
END DO

DO sj = 1, snj+1
  intzone(izone)%szone(iblock)%sinx(0,sj) = 1
  intzone(izone)%szone(iblock)%sinx(1,sj+1) = 1
  intzone(izone)%szone(iblock)%sinx(sni,sj) = 1
  intzone(izone)%szone(iblock)%sinx(sni+1,sj) = 1
END DO

END DO ! iblock loop
END DO ! izone loop

CALL gen_real_bc(zone,intzone,number_zone,bsni,bsnj)

END SUBROUTINE internal_block

SUBROUTINE gen_real_bc(zone,intzone,number_zone,bsni,bsnj)
! It makes a copy of the real_bc type for each of the individual blocks.
IMPLICIT NONE
TYPE(grid), POINTER, DIMENSION(:)::zone
TYPE(block), POINTER, DIMENSION(:)::intzone
INTEGER::number_zone
INTEGER::bsni,bsnj

INTEGER:: izone,iblock,si,sj,ni,nj,ibc,dim,sibc,start,end
INTEGER:: n_real_bc_point,nblock,xblock,yblock,snj,sni
INTEGER:: bi,bj,sbi,sbj
CHARACTER(5)::bc_type
INTEGER,DIMENSION(:),ALLOCATABLE::i,j,i_n,j_n
CHARACTER(5),DIMENSION(:),ALLOCATABLE::sbc_type
CHARACTER(1),DIMENSION(:),ALLOCATABLE::sbc_number

!LOGIC: Visit each block at a time, check if any of the block boundaries is in the
!real_bc of the original boundary
!if yes, then make a real_bc for it.
DO izone = 1,number_zone
   n_real_bc_point = zone(izone)%n_real_bc_point
   nblock = intzone(izone)%nbloc
   ni = zone(izone)%ni
   nj = zone(izone)%nj
   DO iblock = 1,nblock
      sni = intzone(izone)%szone(iblock)%sni
      snj = intzone(izone)%szone(iblock)%snj
      dim = n_real_bc_point
      ALLOCATE (i(dim))
      ALLOCATE (j(dim))
      ALLOCATE (i_n(dim))
      ALLOCATE (j_n(dim))
      ALLOCATE (sbc_type(dim))
      ALLOCATE (sbc_number(dim))
      sibc = 1
      !LEFT Boundary
      si = 1
      start = 1
      end = ni-1
      DO sj = 2,snj-1
         sbi=intzone(izone)%szone(iblock)%i(si,sj)
         sbj=intzone(izone)%szone(iblock)%j(si,sj)
         !                 DO ibc = start,end
         DO ibc = 1,dim
            bi=zone(izone)%real_bc(ibc)%i
            bj=zone(izone)%real_bc(ibc)%j
            IF ((bi == sbi) .and. (bj == sbj)) THEN
               i(sibc)=si
               j(sibc)=sj
               i_n(sibc)=si+1
               j_n(sibc)=sj
               sbc_type(sibc)=zone(izone)%real_bc(ibc)%bc_type
               sbc_number(sibc)=zone(izone)%real_bc(ibc)%bc_number
               intzone(izone)%szone(iblock)%sinx(si,sj) = 0
               intzone(izone)%szone(iblock)%sinx(si-1,sj) = 0
               sibc = sibc + 1
               EXIT
            END IF
         END DO
      END DO
      !TOP Boundary
      sj = snj
      start =end+1
      end = ni+nj
      DO si = 0,sni+1
         sbi=intzone(izone)%szone(iblock)%i(si,sj)
         sbj=intzone(izone)%szone(iblock)%j(si,sj)
         !                 DO ibc = start,end
         DO ibc = 1,dim
            bi=zone(izone)%real_bc(ibc)%i
            bj=zone(izone)%real_bc(ibc)%j
            IF ( ((bi == sbi) .and. (bj == sbj)) THEN
               i(sibc)=si
               j(sibc)=sj
               sibc = sibc + 1
            END IF
         END DO
      END DO
   END DO
END DO
i_n(sibc)=si
j_n(sibc)=sj-1
sbc_type(sibc)=zone(izone)%real_bc(ibc)%bc_type
sbc_number(sibc)=zone(izone)%real_bc(ibc)%bc_number
intzone(izone)%szone(iblock)%sinn(si,sj) = 0
intzone(izone)%szone(iblock)%sinn(si,sj+1) = 0
sibc = sibc + 1
EXIT
END IF
END DO
END DO
!RIGHT Boundary
si = sni
start = end + 1
end = end + (ni-2)
DO sj = 2,snj-1
  sbi=intzone(izone)%szone(iblock)%i(si,sj)
sbj=intzone(izone)%szone(iblock)%j(si,sj)
  DO ibc = start,end
    bi=zone(izone)%real_bc(ibc)%i
    bj=zone(izone)%real_bc(ibc)%j
    IF ( (bi == sbi) .and. (bj == sbj)) THEN
      i(sibc)=si
      j(sibc)=sj
      i_n(sibc)=si-1
      j_n(sibc)=sj
      sbc_type(sibc)=zone(izone)%real_bc(ibc)%bc_type
      sbc_number(sibc)=zone(izone)%real_bc(ibc)%bc_number
      intzone(izone)%szone(iblock)%sinn(si,sj) = 0
      intzone(izone)%szone(iblock)%sinn(si+1,sj) = 0
      sibc = sibc + 1
      EXIT
    END IF
  END DO
END DO
!Bottom Boundary
sj = 1
start = end + 1
end = n_real_bc_point
DO si = 0,sni+1
  sbi=intzone(izone)%szone(iblock)%i(si,sj)
sbj=intzone(izone)%szone(iblock)%j(si,sj)
  DO ibc = start,end
    bi=zone(izone)%real_bc(ibc)%i
    bj=zone(izone)%real_bc(ibc)%j
    IF ( (bi == sbi) .and. (bj == sbj)) THEN
      i(sibc)=si
      j(sibc)=sj
      i_n(sibc)=si
      j_n(sibc)=sj+1
      sbc_type(sibc)=zone(izone)%real_bc(ibc)%bc_type
      sbc_number(sibc)=zone(izone)%real_bc(ibc)%bc_number
      intzone(izone)%szone(iblock)%sinn(si,sj) = 0
      intzone(izone)%szone(iblock)%sinn(si+1,sj) = 0
      sibc = sibc + 1
      EXIT
    END IF
  END DO
END DO
! WRITE(*,*),iblock,sibc,dim,sni,snj
IF (sibc == 1) THEN
! Doing this as a precautionary measure
intzone(izone)%szone(iblock)%n_real_bc_point = 1
ALLOCATE(intzone(izone)%szone(iblock)%real_bc(1))

ibc = 1
intzone(izone)%szone(iblock)%real_bc(ibc)%bc_type = 'null'
intzone(izone)%szone(iblock)%real_bc(ibc)%bc_number = '*'
intzone(izone)%szone(iblock)%real_bc(ibc)%i = 1
intzone(izone)%szone(iblock)%real_bc(ibc)%j = 1
intzone(izone)%szone(iblock)%real_bc(ibc)%i_n = 1
intzone(izone)%szone(iblock)%real_bc(ibc)%j_n = 1

ELSE
intzone(izone)%szone(iblock)%n_real_bc_point = sibc-1
ALLOCATE(intzone(izone)%szone(iblock)%real_bc(sibc-1))

DO ibc = 1,sibc-1
intzone(izone)%szone(iblock)%real_bc(ibc)%bc_type = sbc_type(ibc)
intzone(izone)%szone(iblock)%real_bc(ibc)%bc_number = sbc_number(ibc)
intzone(izone)%szone(iblock)%real_bc(ibc)%i = i(ibc)
intzone(izone)%szone(iblock)%real_bc(ibc)%j = j(ibc)
intzone(izone)%szone(iblock)%real_bc(ibc)%i_n = i_n(ibc)
intzone(izone)%szone(iblock)%real_bc(ibc)%j_n = j_n(ibc)
END DO
END IF

DEALLOCATE(i)
DEALLOCATE(j)
DEALLOCATE(i_n)
DEALLOCATE(j_n)
DEALLOCATE(sbc_type)
DEALLOCATE(sbc_number)
END DO
END SUBROUTINE gen_real_bc

4. Create the subroutine scalc_flowfield

SUBROUTINE scalc_flowfield (ni, nj, nghost, u, v, p, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tko, edo, gammao, to, tviso, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
ueast, vnoth, uo, vo, tk, ed, gamma, t, tvis,
REAL (high), ALLOCATABLE, DIMENSION (:, :) :: vis_n, tau1_n, tau2_n, tau22_n, 
gam tk n, q tk x n, q tk y n, gam ed n, & 
& q ed x n, q ed y n, gam gamma n, q gamma x n, q gamma y n, gam t n, q t x n, 
q t y n
REAL (high), ALLOCATABLE, DIMENSION (:, :) :: uen, ven, pen, tken, eden, ten, 
gammaen
REAL (high), ALLOCATABLE, DIMENSION (:, :) :: upp, vpp, au, av
REAL (high), ALLOCATABLE, DIMENSION (:, :) :: f1, f2, eddtk, gen, s
REAL (high), ALLOCATABLE, DIMENSION (:, :) :: ae, aw, an, as, ap, rhs, dpdn
! new structure
TYPE (struct_aewnsp), POINTER, DIMENSION (:,:) :: aewnsp
!! end of new structure
REAL (high), ALLOCATABLE, DIMENSION (:, :) :: du, dtime

!Gilbert
REAL (high), ALLOCATABLE, DIMENSION (:, :) :: t11_cubic, t12_cubic,t22_cubic
!Gilbert
!
INTEGER :: nim1, njm1, iter, i, j
REAL (high) :: sumu, sumu_struct, sumv, summ, sumtk, sumed, sumgamma, sumt,
xstr, tufstr, duref, umove, vmove

!=====================================================================================  
==============================================
IF ( .NOT. intermittency) sumgamma = 0.
IF ( .NOT. ltemperature) sumt = 0.
IF (imodel == 0) THEN
  sumtk = 0.
  sumed = 0.
END IF
snim1 = sni - 1
snjm1 = snj - 1
!
ALLOCATE (vis e(0:sni, snj))
ALLOCATE (tau11 e(snim1, snj))
ALLOCATE (tau12 e(snim1, snj))
ALLOCATE (tau22 e(snim1, snj))
ALLOCATE (vis n(sn1, 0:snj))
ALLOCATE (tau11 n(sn1, snjm1))
ALLOCATE (tau12 n(sn1, snjm1))
ALLOCATE (tau22 n(sn1, snjm1))
ALLOCATE (uen(0:sn1, 0:snj))
ALLOCATE (ven(0:sn1, 0:snj))
ALLOCATE (pen(0:sn1, 0:snj))
!
ALLOCATE (up(sni, snj))
ALLOCATE (vp(sni, snj))
ALLOCATE (au(sni, snj))
ALLOCATE (av(sni, snj))
ALLOCATE (ae(sni, snj))
ALLOCATE (aw(sni, snj))
ALLOCATE (an(sni, snj))
ALLOCATE (as(sni, snj))
ALLOCATE (ap(sni, snj))
ALLOCATE (rhs(sni, snj))

!! new structure allocation
ALLOCATE (aewnsp(sni,snj))
!! end of allocation for new structure
ALLOCATE (dpdn(sni, snj))
ALLOCATE (du(sni, snj))

! IF (imodel == 0) THEN
ALLOCATE (gam_tk_e(0:0, 1))
ALLOCATE (q_tk_x_e(1, 1))
ALLOCATE (q_tk_y_e(1, 1))
ALLOCATE (gam_ed_e(0:0, 1))
ALLOCATE (q_ed_x_e(1, 1))
ALLOCATE (q_ed_y_e(1, 1))
ALLOCATE (gam_tk_n(1, 0:0))
ALLOCATE (q_tk_x_n(1, 1))
ALLOCATE (q_tk_y_n(1, 1))
ALLOCATE (gam_ed_n(1, 0:0))
ALLOCATE (q_ed_x_n(1, 1))
ALLOCATE (q_ed_y_n(1, 1))
ALLOCATE (tken(0:0, 0:0))
ALLOCATE (eden(0:0, 0:0))
ALLOCATE (f1(1, 1))
ALLOCATE (f2(1, 1))
ALLOCATE (eddtk(1, 1))
ALLOCATE (gen(1, 1))
ALLOCATE (s(1, 1))
ELSE
ALLOCATE (gam_tk_e(0:sni, snj))
ALLOCATE (q_tk_x_e(snim1, snj))
ALLOCATE (q_tk_y_e(snim1, snj))
ALLOCATE (gam_ed_e(0:sni, snj))
ALLOCATE (q_ed_x_e(snim1, snj))
ALLOCATE (q_ed_y_e(snim1, snj))
ALLOCATE (gam_tk_n(sni, 0:snj))
ALLOCATE (q_tk_x_n(sni, snjm1))
ALLOCATE (q_tk_y_n(sni, snjm1))
ALLOCATE (gam_ed_n(sni, 0:snj))
ALLOCATE (q_ed_x_n(sni, snjm1))
ALLOCATE (q_ed_y_n(sni, snjm1))
ALLOCATE (tken(0:sni, 0:snj))
ALLOCATE (eden(0:sni, 0:snj))
ALLOCATE (f1(sni, snj))
ALLOCATE (f2(sni, snj))
ALLOCATE (eddtk(sni, snj))
ALLOCATE (gen(sni, snj))
ALLOCATE (s(sni, snj))
ALLOCATE (t11_cubic(sni, snj))
ALLOCATE (t12_cubic(sni, snj))
ALLOCATE (t22_cubic(sni, snj))
END IF
IF (intermittency) THEN
ALLOCATE (gam_gamma_e(0:sni, snj))
ALLOCATE (q_gamma_x_e(snim1, snj))
ALLOCATE (q_gamma_y_e(snim1, snj))
ALLOCATE (gam_gamma_n(sni, 0:snj))
ALLOCATE (q_gamma_x_n(sni, snjm1))
ALLOCATE (q_gamma_y_n(sni, snjm1))
ALLOCATE (gammaen(0:sni, 0:snj))
ELSE
ALLOCATE (gammaen(0:0, 0:0))
ALLOCATE (gam_gamma_e(0:0, 1))
ALLOCATE (q_gamma_x_e(1, 1))
ALLOCATE (q_gamma_y_e(1, 1))
ALLOCATE (gam_gamma_n(1, 0:0))
ALLOCATE (q_gamma_x_n(1, 1))
ALLOCATE (q_gamma_y_n(1, 1))
END IF
IF (ltemperature) THEN
ALLOCATE (gam_t_e(0:sni, snj))
ALLOCATE (q_t_x_e(snim1, snj))
ALLOCATE (q_t_y_e(snim1, snj))
ALLOCATE (gam_t_n(sni, 0:snj))
ALLOCATE (q_t_x_n(sni, snjm1))
ALLOCATE (q_t_y_n(sni, snjm1))
ALLOCATE (ten(0:sni, 0:snj))
ELSE
ALLOCATE (ten(0:0, 0:0))
ALLOCATE (gam_t_e(0:0, 1))
ALLOCATE (q_t_x_e(1, 1))
ALLOCATE (q_t_y_e(1, 1))
ALLOCATE (gam_t_n(1, 0:0))
ALLOCATE (q_t_x_n(1, 1))
ALLOCATE (q_t_y_n(1, 1))
END IF
ALLOCATE (dtime(1, 1))
CALL cal_property (su, sv, sp, tk, ed, t, gamma, sni, snj, snim1, snjm1, nghost, imodel, &
& sdwall, uen, ven, pen, tken, eden, ten, gammaen, vis_e, vis_n, tau11_e, tau12_e, tau22_e, tauu22_n, tvis, fl, f2, svol, sae_x, sae_y, san_xt, san_yt, &
& sae_xt, sae_yt, san_x, san_y, gam_tk_e, gam_tk_n, q_tk_x_e, q_tk_y_e, &
& q_tk_x_n, q_tk_y_n, gam_ed_e, gam_ed_n, q_ed_x_e, q_ed_y_e, q_ed_x_n, q_ed_y_n, gam_gamma_e, gam_gamma_n, &
& q_gamma_y_e, q_gamma_x_e, q_gamma_y_n, q_gamma_x_n, gam_t_e, gam_t_n, q_t_x_e, q_t_y_e, q_t_x_n, q_t_y_n, t11_cubic, t12_cubic)
CALL cal_u (su, uo, uoo, dtime, sv, sueast, svnoth, sae_x, &
& sae_y, san_x, san_y, sx, sy, vis_e, vis_n, &
& svol, tau11_e, tau12_e, tau11_n, tau22_n, sp, sni, snj, snim1, snjm1, nghost, &
& swfx, swbx, swfy, swby, sinx, &
& upp, au, n_real_bc_point, real_bc, aewnsps, dpdn, du, sumu, iblock)
CALL cal_v (sv, vo, voo, dtime, sueast, svnoth, sae_x, sae_y, &
& san_x, san_y, sx, sy, vis_e, vis_n, svol, &
& tau12_e, tau22_e, tau11_n, tau22_n, tvis, sp, sni, snj, snim1, snjm1, nghost, &
& swfx, swbx, swfy, &
& swby, sinx, vpp, av, n_real_bc_point, real_bc, aewnsps, dpdn, du, sumv, iblock)
!IN the subroutine cont, I have commented the part of the code that uses boundary
conditions.
CALL cont (su, sv, sp, au, av, sae_x, sae_y, san_x, &
& san_y, sae_xt, sae_yt, san_xt, san_yt, sni, snj, snim1, snjm1, &
& nghost, swfx, swbx, swfy, swby, sx, sy, &
& pen, upp, vpp, sueast, svnoth, n_real_bc_point, real_bc, svol, sinx, &
& aewnsps, du, summ, duref, umove, vmove, izone, global_zone_number, iblock)
DEALLOCATE (vis_e)
DEALLOCATE (tau11_e)
DEALLOCATE (tau12_e)
DEALLOCATE (tau22_e)
DEALLOCATE (vis_n)
DEALLOCATE (tau11_n)
DEALLOCATE (tau12_n)
DEALLOCATE (tau22_n)
DEALLOCATE (uen)
DEALLOCATE (ven)
DEALLOCATE (pen)
!
DEALLOCATE (upp)
DEALLOCATE (vpp)
DEALLOCATE (au)
DEALLOCATE (av)
!
DEALLOCATE (ae)
DEALLOCATE (aw)
DEALLOCATE (an)
DEALLOCATE (as)
DEALLOCATE (ap)
DEALLOCATE (rhs)
!! deallocating the structure
DEALLOCATE (aewns)
!! end of deallocation
DEALLOCATE (dpdn)
DEALLOCATE (du)
DEALLOCATE (gam_tk_e)
DEALLOCATE (q_tk_e)
DEALLOCATE (q_tk_y_e)
DEALLOCATE (gam_ed)
DEALLOCATE (q_ed_x)
DEALLOCATE (q_ed_y)
!
DEALLOCATE (gam_tk_n)
DEALLOCATE (q_tk_n)
DEALLOCATE (q_tk_y_n)
DEALLOCATE (gam_ed_n)
DEALLOCATE (q_ed_x_n)
DEALLOCATE (q_ed_y_n)
!
DEALLOCATE (tken)
DEALLOCATE (eden)
DEALLOCATE (f1)
DEALLOCATE (f2)
DEALLOCATE (eddtk)
DEALLOCATE (gen)
DEALLOCATE (s)
!
DEALLOCATE (gam_gamma_e)
DEALLOCATE (q_gamma_x_e)
DEALLOCATE (q_gamma_y_e)
DEALLOCATE (gam_gamma_n)
DEALLOCATE (q_gamma_x_n)
DEALLOCATE (q_gamma_y_n)
DEALLOCATE (gammaen)
!
DEALLOCATE (ten)
DEALLOCATE (gam_t_e)
DEALLOCATE (q_t_x_e)
DEALLOCATE (q_t_y_e)
DEALLOCATE (gam_t_n)
DEALLOCATE (q_t_x_n)
DEALLOCATE (q_t_y_n)
!
DEALLOCATE (dtime)
END SUBROUTINE scalc_flowfield

5. Define the integer variable \( iblock = 1 \) in the subroutine \textit{calc\_flowfield}.

6. Add the variable in the call of \textit{cal\_u}, \textit{cal\_v}, \textit{cont} and also define the variable in those subroutines.

7. Add an if statement in \textit{cal\_u}, \textit{cal\_v} and \textit{cont} where \( sumu, sumv \) and \( summ \) are initialized to zero.

8. In the subroutine \textit{cont}, at the point where a value is assigned to the value of \( duref \), add \( iblock = 1 \) into the if statement.

9. Define the variable \textit{szone} of type \textit{sgrid} in the main program.

10. Insert a call for \textit{internal\_block} subroutine just before the beginning of the main solution loop i.e. just after \textit{read\_restart}

11. Add the call for \textit{scalc\_flowfield} along with the if statement.
REFERENCES


60. S. Kadambi and J. C. Harden, “Accelerating CFD applications by improving cached data reuse”, ssst, p. 120, 27th Southeastern Symposium on System Theory (SSST’95), 1995.


77. G95 Project, http://www.g95.org


VITA

Anand B Palki was born on December 22, 1980 in Thane, India. He received a bachelors’ degree from Visweshwariah Technological University, Karnataka, India in July of 2003. Currently he is completing work on a Masters degree in the Department of Mechanical Engineering at the University of Kentucky.

Professional Positions Held

Software Release Engineer, August 2003 - July 2004
SCT Software Solutions,
Bangalore, Karnataka, India.

Research Assistant, Summer 2005 - Fall 2006
UK Cluster Fluid Dynamics Group,
Department of Mechanical Engineering, University of Kentucky, Lexington, KY

Thermal Engineer, October 2006
Vogt Power International,
Thermal Design Group, Louisville, KY.

Scholastic Honors

Kentucky Graduate Scholarship, Fall 2004 - Fall 2006
University of Kentucky, Lexington, KY.

Papers and Conferences

A.Palki, S. Gupta and R.P. Lebeau. - “Use of Cache-Friendly Blocking to Accelerate CFD Codes on Commodity Hardware”, 44th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, USA

S.C. Huang, A.Palki, S.Gupta, T. Hauser, R.P. Lebeau – “Parallel Performance Evaluation of Different Cluster Architectures for CFD Applications”, Accepted for Annual Dayton-Cincinnati Aerospace Science Symposium, Dayton, Ohio, USA