University of Dublin

TRINITY COLLEGE

Dense Matrix Gauss-Jordan Inversion Auto-Optimisation on Multicore Systems

Ross Wynne
B.A.(Mod.) Computer Science
Final Year Project May 2007
Supervisor: Dr. David Gregg

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

_____ May 4th, 2007

Ross Wynne

Acknowledgements

I wish to thank David Gregg for supervising this project and for his helpful insight and encouragement throughout the year. My tutor, Donal O'Donovan, who taught me much of the mathematical knowledge that was used in this project. I also wish to thank Fiona van der Puil for her unwavering support and love throughout the years.

Abstract

This project examines if Gauss-Jordan Matrix Inversion on a dense matrix can be optimised by parallelisation of the core algorithm and investigates the relationship of speed up efficiencies using multi-core and single processor systems.

Table of Contents

Chapter 1

Introduction to Problem Description

1.1 Matrix Inversion

The inverse of a matrix is defined as $A.A^{-1}=I$ (where I is the identity matrix). A matrix is defined as invertible if and only if the determinate of the matrix is ≠ 0 and the matrix is square (i.e. NxN).

There are many various matrix inversion techniques such as Gauss-Jordan Inversion, Crammers Rule (co-factors), LU decomposition, singular value decomposition and blockwise inversion. However all but Gauss-Jordan Inversion and LU decomposition are inefficient for large matrices and as such are rarely used in applications such an image processing and scientific data processing.

The classical use of an inverted matrix is to solve:

$$A.x = b$$

where:
A is a matrix that represents the series of co-efficients of variables in a set of simultaneous equations.
x is the vector set of variables to be solved
and b is the vector set of solutions

By using matrix inversion the solution becomes a trivial matter of matrix-vector multiplication.

$$A^{-1}.A.x = A^{-1}b$$

$$\Rightarrow x = A^{-1}b$$

1.2 Gauss-Jordan Inversion

Gauss-Jordan Inversion is a two-stage process whereby Gaussian Elimination is performed first so as to reduce the matrix to 'reduced row echelon form' by using elementary row reduction so that the lower matrix (i.e. below the main diagonal) is filled with zeros. The second stage is where the upper triangle is similarly reduced to zeros. In the mean time a second matrix that starts as the Identity Matrix has each elementary row operation that was applied to the first matrix applied to it.

1.3 Example

Take a 3x3 matrix A and an identity matrix:

$$\begin{pmatrix}1 & 2 & 1\\ 2 & 9 & 2\\ 1 & 7 & 2\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}$$

$$\begin{pmatrix}1 & 2 & 1\\ 0 & 5 & 0\\ 0 & 5 & 1\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ -2 & 1 & 0\\ -1 & 0 & 1\end{pmatrix} R_2=R_2-2*R_1 \text{ AND } R_3=R_3-R_1$$

$$\begin{pmatrix}1 & 2 & 1\\ 0 & 1 & 0\\ 0 & 5 & 1\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ -0.4 & 0.2 & 0\\ -1 & 0 & 1\end{pmatrix} R_2=R_2/5$$

$$\begin{pmatrix}1 & 2 & 1\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ -0.4 & 0.2 & 0\\ 1 & -1 & 1\end{pmatrix} R_3=R_3-5*R_2$$

$$\begin{pmatrix}1 & 2 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}0 & 1 & -1\\ -0.4 & 0.2 & 0\\ 1 & -1 & 1\end{pmatrix} R_1=R_1-R_3$$

$$\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}0.8 & 0.6 & -1\\ -0.4 & 0.2 & 0\\ 1 & -1 & 1\end{pmatrix} R_1=R_1-2*R_2$$

The second matrix now contains the inverse matrix of A.

1.4 Conclusion

Gauss-Jordan Inversion is an $O(N^3)$ problem and hence any method that speeds up the computation of the problem is helpful. The ultimate goal of this project is to compute the task in a manner faster that previously available.

Chapter 2

Planning and Design

2.1 High Level Objectives and Aims

The aim of this design phase is to outline the high-level aim for what the proof-of-concept project is to achieve.

1. Produce an optimised version of a Gauss-Jordan Inversion algorithm suitable for parallelisation.
2. Produce a parallelised version of a Gauss-Jordan Inversion algorithm.
3. Compare the run timings of any such algorithm over a base-line comparison.
4. For various sized matrices find the most optimal algorithm based on dimensional size, dependent on the hardware of the system and compiler used.
5. Create a compiled function that takes into account the results of aim number 4 and could be called upon by an external program for processing of a square matrix.

2.2 Design Decisions

2.2.1 Programming language

To choose the right programming language for the project a set of requirements had to be met. They were:

1) Author's prior experience
2) Language's ability to deal with threading
3) Portability and Support
4) Fast code production
5) Environment that final generated function would be used in

Author's prior experience

The author has previously written algorithmic code in C, C++, Java, Eiffel and the M68k assembly language.

Language's ability to deal with threading

Both C and C++ have the ability to use OpenMP or POSIX threads. Java has its own threading model as well as POSIX threading.

Portability and Support

C, C++ and Java are widely supported on multiple hardware and operating system platforms. Other languages such as Eiffel and Hascal do not have large communities or companies pushing forward development.

Fast Code Production

Current C/C++ compilers such as gcc and the Intel compiler have an array of optimisation switches for various processors and for in-lining and loop unfolding. Java's compiler is only designed to produce bytecode that is translated into machine code at runtime and although Java's speed is catching up with C/C++ it still is slower.

Environment

The most common computation environment would be the one that is fast and portable and as such C or C++ would be the

obvious choices so that the final Gaussian algorithm will be compatible with other programs.

From these requirements it is obvious that the final choice of programming language was either C or C++. The author chose to use C++ since the memory declaration commands such as 'new' and 'delete' reduced the complexity of using malloc() to define arrays and matrices in memory as well as avoiding any memory leaks that may occur.

Decision: C++ is language of choice.

2.2.2 Mathematical Techniques

Three Gaussian based algorithms are used in this project with one acting as a base-line result. The base algorithm is a modified version of Numerical Recipes in C++'s *gaussj* function.

The *gaussj* function is quite a complex implementation of the Gauss-Jordan algorithm with numerous single letter variables which make it hard to comprehend. The code itself is not directly parallelisable and is intended to run serially. The author edited the function's matrix declarations to use *new* and *delete* commands as opposed to the proprietary matrix format that Numerical Recipes uses. By standardising the matrix format the eventual end timings will be able to be compared and contrasted.

The first of the algorithms written by the author to which *gaussjordanSerial* and *gaussjordanOMP* are based upon is the classic method for a Gauss-Jordan Inversion. The first part of the algorithm is the Gaussian Elimination technique. It requires two matrices, the first being the original matrix and the second is an Identity Matrix. The algorithm starts on the upper left entry of the first matrix and, if the entry is non-zero, then elementary row

operations are performed on the rows below it with the aim of creating a column of zeros below the lead diagonal. If the entry is zero then the whole row is swapped with a row below it that has a non-zero entry in that same column. If there happens to be no non-zero entries in the column then the matrix is singular and non invertible. Once the first column is complete the next entry on the lead diagonal will be processed.

The second part of the algorithm is a form of Gaussian Elimination in reverse, whereby the algorithm starts on the lower right element and uses elementary row operations on the rows above it to form a column of zeros above the lead diagonal. This continues until the matrix resembles an Identity Matrix.

All row operations upon the first matrix are replicated upon a second matrix that starts out as the Identity Matrix.

Once the algorithm has completed the inverse of the original matrix is to be found as the second matrix.

The second algorithm, to which *gaussjordanSerial_Combo* and *gaussjordanOMP_Combo* are based, attempts to do both the first and second stage of Gauss-Jordan Inversion at the same time. The algorithm starts exactly like the previous one but after the first column has been reduced to zeros (excluding the lead diagonal value) the algorithm then performs elementary row operations on the rows above the pivot[1] value.

[1] A pivot entry is the value on the lead diagonal of a matrix upon which the calculations for that column are being processed.

2.2.3. Algorithm Design

The design of each algorithm is crucial to optimising for parallelisation. As mentioned previously the *gaussj* function was complex and didn't lend itself to having distinct independent sections. By rewriting the algorithms from scratch it was possible to write these sections of code so as they were independent and hence parallelisable.

Additionally each section of the algorithmic code was designed to be composed of simple instructions such as additions or divisions and not function calls. By doing this it reduced the complexity of the machine code generated by the compiler and hence would require less processing cycles per instruction. However the more expensive processor instructions such as multiplication and division require sparing use, even more so in the case of division instructions since they take more cycles per instruction than multiplications.

> Decision: Write new Gauss-Jordan Inversion algorithms from scratch and section code for efficient parallelisation.

2.2.4 Algorithm Parallelisation

The next design decision was how to implement a parallelised algorithm. There are two standard methods to run sections of code in parallel. The first is by using POSIX threads or pthreads as they are more commonly known. This method requires each parallelised section to be encapsulated as a function. However that would have meant complicating the code further with more machine instructions in the final compiled version.

The second method for parallelising the program is by using OpenMP. It is a pragma based pre-processing language that is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify memory

parallelism in Fortran and C/C++ programs. It has been developed for various computers ranging from super-computers through to SMP servers and multicore desktops and all variations in between. It has a documented list of requirements that code must meet for OpenMP to be successful in parallelising 'for' loops which will be predominantly used in this algorithms. They are:

1. The loop variable must be a signed integer
2. The loop condition must not change throughout the processing of the loop.
3. That simple regular expressions are used in the loop condition such as >, <, >=, <=
4. The loop contains no jump commands for into or out of the loop.
5. Upon each iteration of the loop there is some form of incrementing or decrementing of a counter such that the loop does not run *ad infinitum* due to the loop condition never registering as false.

The author chose to use OpenMP as the method of choice for threading of the code. Its straightforward declarations allowed for more work to be done on optimising other areas of the algorithms.

Decision: OpenMP as API of choice

2.2.5 Sample Matrices

A range of matrix dimensions had needed to be chosen so that algorithms could be compared in a realistic manner. Hence all algorithms were run over a series so identical matrices from 3x3 through to 100x100 and sometimes larger dimensions if there was an interesting trend beyond. The matrix entries are created using a standalone program that generates a series of ten million random signed integer values, between -4999 and 5000, and then writes

them to a file. This data file is then available for providing test values for the project's algorithms.

Decision: Range 3x3 -> 100x100

2.2.6 Compilers

Once the language was chosen the decision had to be took on which compilers the program should be run under. For the sake of brevity and keeping the number of variables within the project to a manageable level it was decided by the author to use only 2 difference compilers. For the first choice G++ was an obvious choice since it is most widely ported and newer versions such as 4.2 now has OpenMP support. The second compiler was dependent on if it supported OpenMP and as such the Portland and Intel compilers were the primary choices. Unfortunately the Portland compiler cannot be used without having to pay for it where as the Intel compiler allowed for a fully featured evaluation without costing money.

Decision: G++ and Intel Compilers

Chapter 3

Implementation and Methods

3.1 Program Outline

The main program is split into 10 C++ files of which 5 are header files. There are 5 Gauss-Jordan algorithms, each made contained as a function. In addition to this there exists a 'wrapper' function, gaussjordan(), that is used in the final stage of the project when the best algorithms for a particular range of dimensions is established.

3.2 Optimisation

The optimisation of the gaussjordan() function is the crucial element to the project. Within it is the method for deciding upon which algorithm is used for a specific matrix dimension.
The determination of the value to which algorithm is used is a two stage process.

3.2.1 Stage One

Upon running the 'make all' command a default value for "GO_R" is read in by the compiler during pre-processing. This starting value is arbitrary since upon first running of the

program the exact crossover point between the various serial and parallel algorithms is undetermined since not all processors behave the same. Once the program is compiled and run it will cycle through the various dimensional sizes several times (as set by the "NO_AVG_VAL" variable in *main.h*). The range of dimensions is set by the "DIM_MIN_MAT" and "DIM_MAX_MAT" variables in *main.h*. Upon completion of the range of dimensions the function *stati()* will calculate the intersection point of the best serial and parallel algorithm. This is done by comparing the speedup ratios and selecting the first value where by the it and the following two speedup ratios are greater than or equal to one. Once this crossover value has been determined it is passed back to the main program whereby it is written out to the optimised.h file by the *writeout()* function.

3.2.2 Stage Two

The program is then recompiled for a second time and in doing so the *gaussjordan.o* object file now contains the optimal solution for finding the best algorithm.

3.3 Generalised Structure

The main.cpp code is written as follows:

1. Load text file of randomly generated numbers into an array
2. Dimension loop that cycles through the range of dimensions
    a. Repeat Loop 5 times or more to get averaged run timing
        i. Create matrices in memory and load values
        ii. Run each algorithm and time them individually
        iii. Delete matrices in memory
    b. Average the timings
3. Print out results and alter "*optimised.h*"

This is the primary function that generates timings and allows for comparisons to be established. The actual Gauss-Jordan Inversion algorithms are contained in two separate files, *gaussj.cpp* and *gaussjordan.cpp*, and are split so that the baseline function and authors created functions are kept distantly separate for easy of readability.

3.4 Implementation Decisions

There are a number of variables beyond the code that needed to be reduced so that it is manageable from a programming point of view. One of which was which optimisation level was to be passed to the compiler at runtime. From experimentation O2 level optimisation was superior to both O1 and O3 options. As such the implementation decision was that O2 optimisation was the default for the project for both compilers.

Additionally there was a choice to be made between the variable type of the matrices with either float or double being the realistic types. Under testing conditions both behaved in a similar way and so it was not a critical issue when it came to discussing the results and conclusion so it was decided that floats would be the default type. It is however possible to swap between floats and doubles by editing a single line in *main.h*.

Finally since this is a proof of concept rather than a full implementation it was decided that, given that there are 5 competing algorithms to compare for which is the best one given a specific dimension, the problem was minimised to choosing between the best two solutions (one serial and one parallel) that were found through experimentation. As such the *gaussjordan()* function chooses between *gaussjordanSerial_Combo()* and *gaussjordanOMP_Combo()*.

3.5 Problems Encountered

When trying to calculate the number of processing cycles each function took to complete it was found to be impossible to accurately use the Intel processor instruction, RDTSC. The RDTSC instruction works by counting the number of processor 'ticks' that goes by as a function is run. Since multi-core and SMP systems often transfer a running program and/or threads between processors any counting method relying on a single processor is likely to fail unless its affinity is set to a single processor, but in doing so setting an affinity it is relatively pointless to test a parallelised algorithm since threading on a single processor doesn't improve the performance.

3.6 Function Descriptions

gaussj():

This is the baseline function to which the other Gauss-Jordan Inverse functions are compared. It is from the text Numerical Recipes in C++ (2^{nd} edition)[1] which contains what would be best described as the defacto standard for numerical computing.

gaussjordanSerial():

This non-parallelised algorithm is written to follow the classical technique for Gauss-Jordan Inversion of reducing the lower triangle of the first matrix to zeros and then repeating the same steps on the upper triangle.

gaussjordanOMP():

This algorithm has the same code as the gaussjordanSerial() function except that it also contains OpenMP pragma statements. Both commands for optimising for-loops and for separating

sections, that can be run in parallel independently, are used in the code.

gaussjordanSerial_Combo():

Instead of being systematic in waiting for the first matrix's lower triangle to be reduced to zeros the code is set up as such that both upper and lower triangles are reduced via elementary row operations at the same time and not in the two stage process that the previous algorithms used.

gaussjordanOMP_Combo():

This algorithm is identical in structure to *gaussjordanSerial_Combo()* except that it has several OpenMP pragma statements placed in it to parallelise the function.

gaussjordan():

This is the wrapper algorithm that uses the "GO_R" value set in optimised.h to decide the point whereby it switches between using serial and parallel versions.

Chapter 4

Results and Analysis

To properly discuss the results a quantitative measuring scale must be established that will give a meaningful context to the timing data. As such both the physical timings are discussed as well as the speedup ratio of the baseline algorithm to the 4 other algorithms.

4.1 Definition of Speed Up

$$S_i = \frac{T_B}{T_i}$$

where:
T_B=timing of baseline algorithm over a specific matrix dimension
T_i=timing of test algorithm over the same matrix dimension
S_i=Speedup Ratio for the specific dimension

So:
When S<1 then the test algorithm is slower than the baseline algorithm
When S=1 then the test algorithm is no faster nor slower than the baseline algorithm
When S>1 then the test algorithm is faster than the baseline algorithm

Note: The 5 algorithms were run on various hardware ranging from a single core PII, a Core2Duo dual-core processor to a Core2Quad processor and represent.

4.2 Dataset results

Serial Performance (timing) – Pentium II

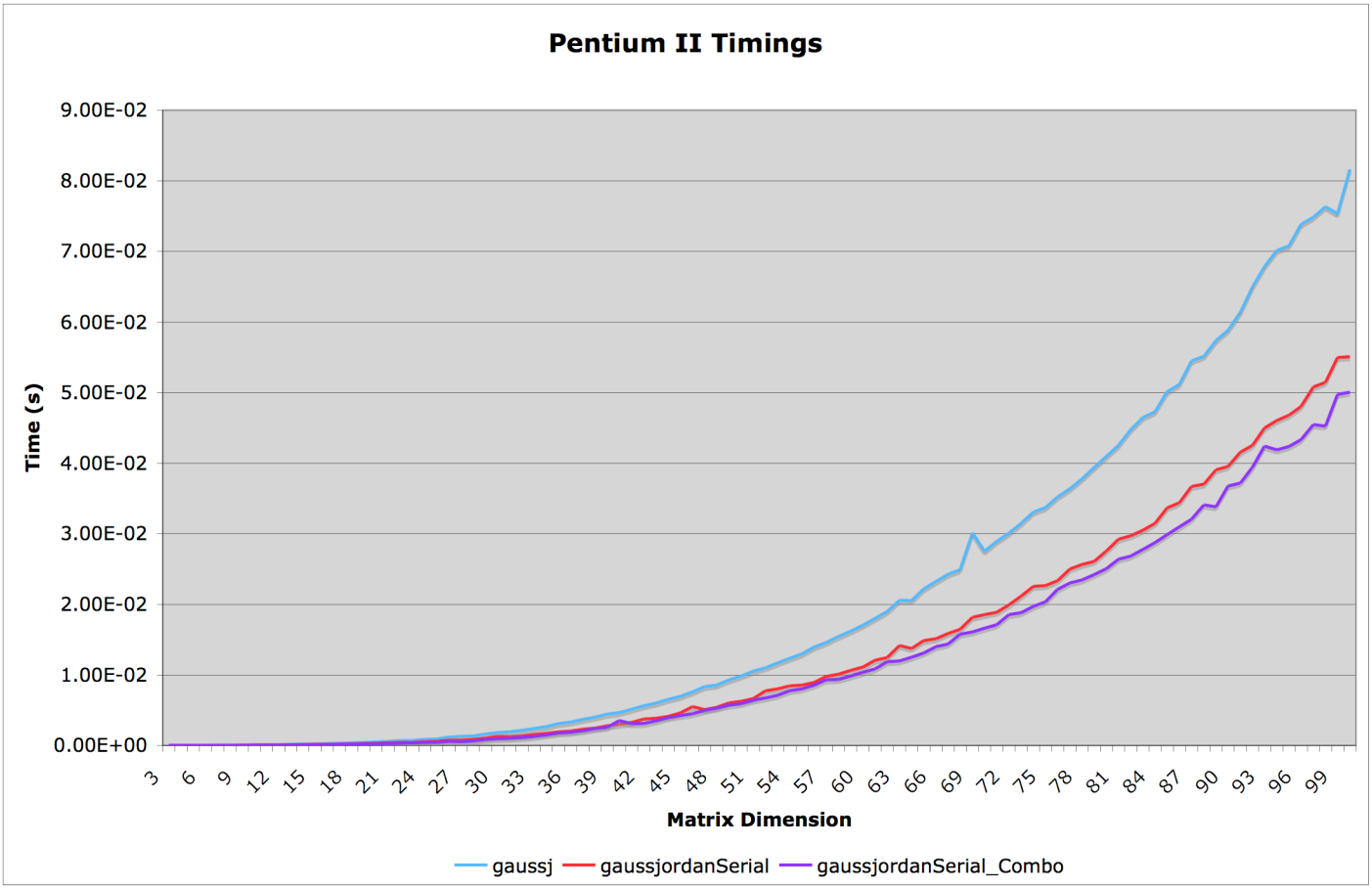

Figure1: *Timings taken on a single Pentium II 450Mhz w/ 512Kb cache using g++ version 3.4.4 and with the -O2 compile switch.*

As is clear from the timings graph the baseline algorithm takes longer than either of the two serial versions. This is especially evident at the 37 mark and larger. Overall the *gaussjordanSerial_Combo* seem to be marginally faster than the *gaussjordanSerial* algorithm.

Serial Performance (Speedup Ratio) – Pentium II

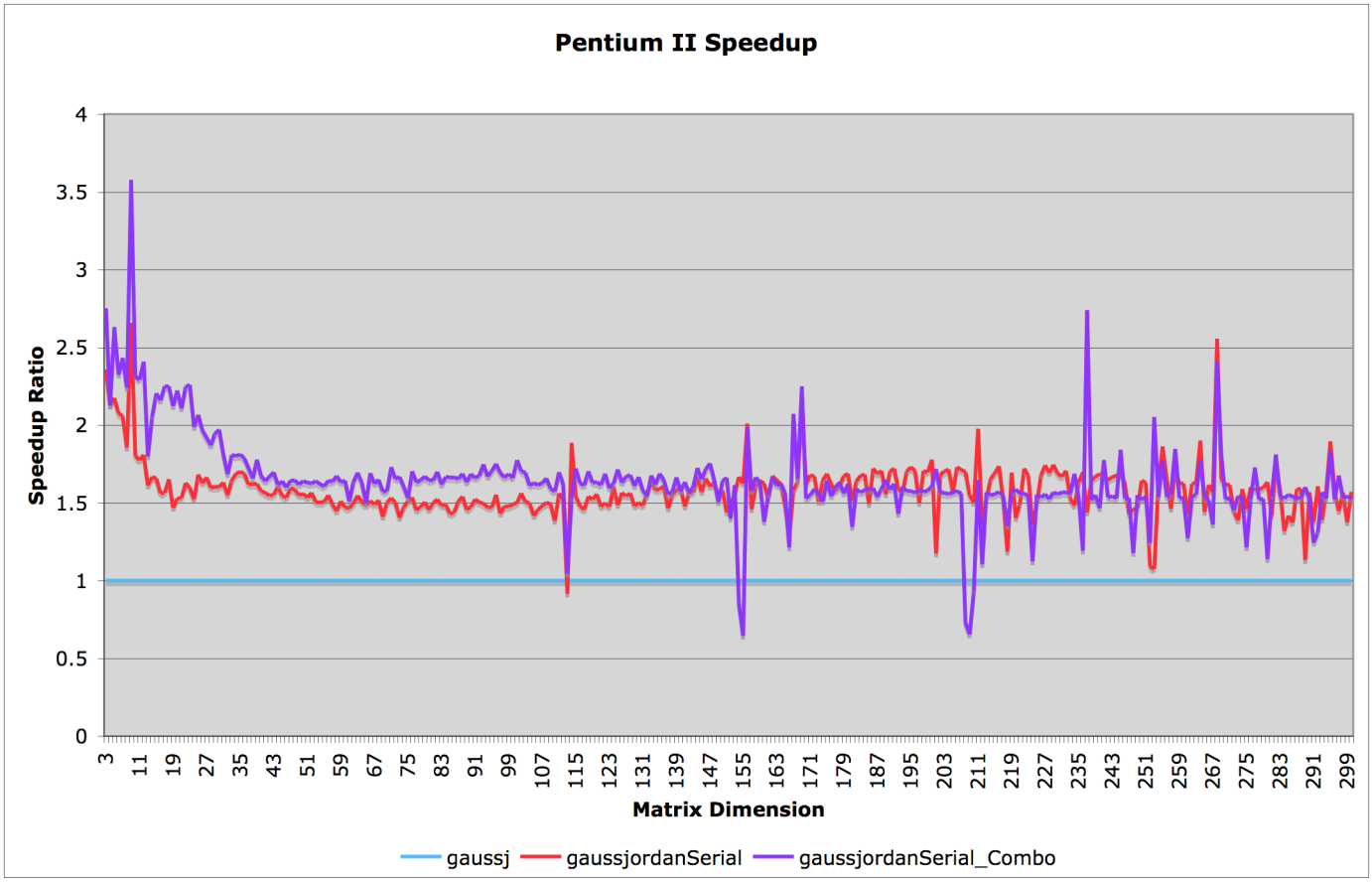

Figure1b: *Speedup based on a single Pentium II 450Mhz w/ 512Kb cache using g++ version 3.4.4 and with the -O2 compile switch.*

The trend that is visible is that at no time does the baseline algorithm register as the fastest method. Indeed both the gaussjordanSerial and gaussjordanSerial_Combo functions are between 1.6 and 2.5 times faster than gaussj over the range 3 through 300. However at 155 it is clear that something has changed with the implementation of the algorithm since there are some higher and lower peaks. It is probable that there is a hardware limitation that is slowing data from being sent to the processor in a timely fashion since at 151x151 a matrix would be approximately 700Kbits in size. The author would theorise that this is undoubtedly related to either front side bus speed or cache size.

Serial Performance (Speedup) – Core2Duo

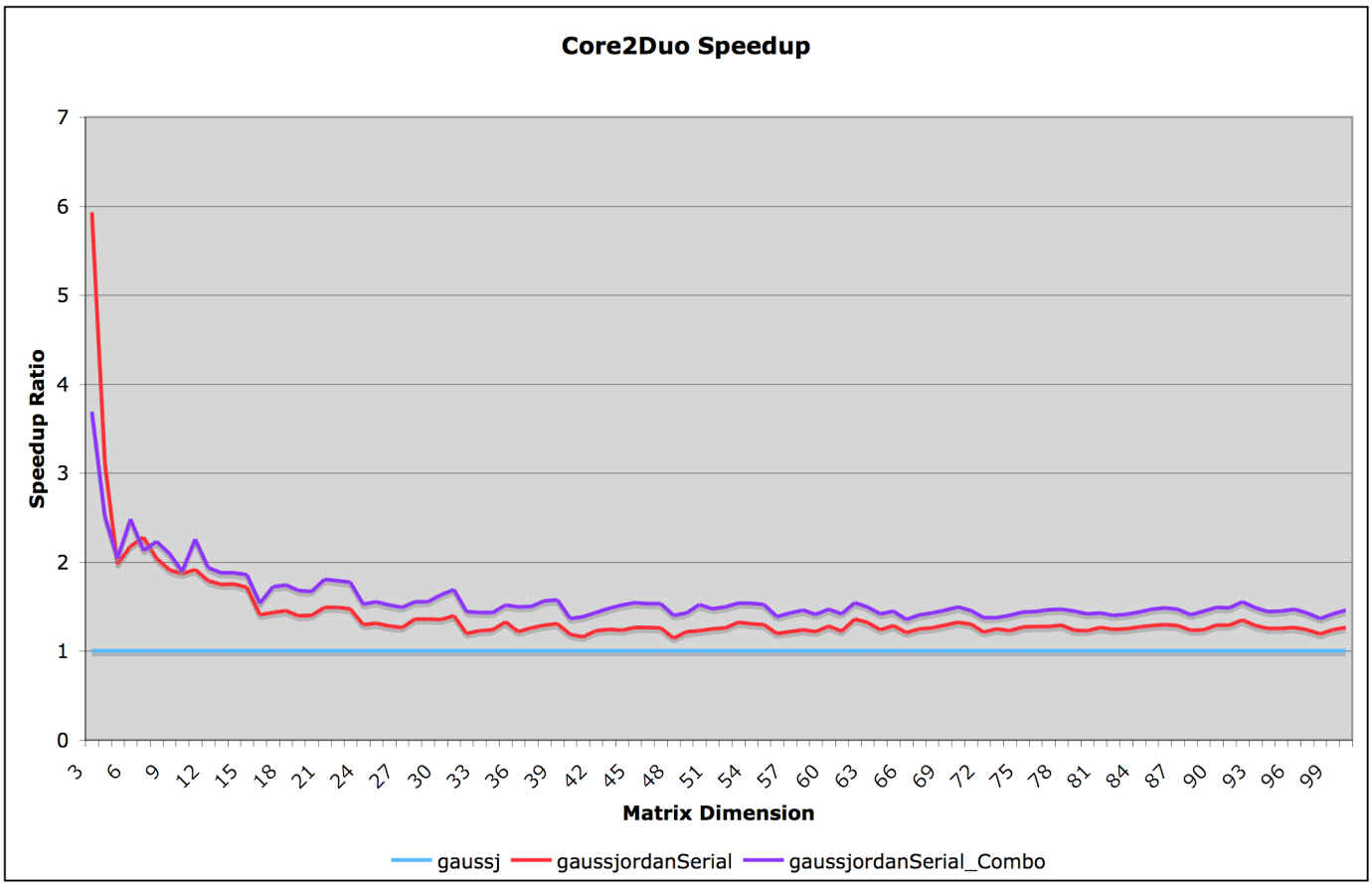

Figure2a: *Timings taken on a Core2Duo processor 2.33Ghz w/ 4Mb cache using g++ version 4.0.1 and with the -O2 compile switch with OpenMP flag off.*

Again the choice of processor does little to change the speedup ratio's implications that the *gaussj()* function is still the worst performing Gauss-Jordan Inversion algorithm for serialised use compared with the two other algorithms.
Also the *gaussjordanSerial_combo()* algorithm is still faster of the two algorithms written by the author.

Parallel Performance (Timing) – Core2Duo

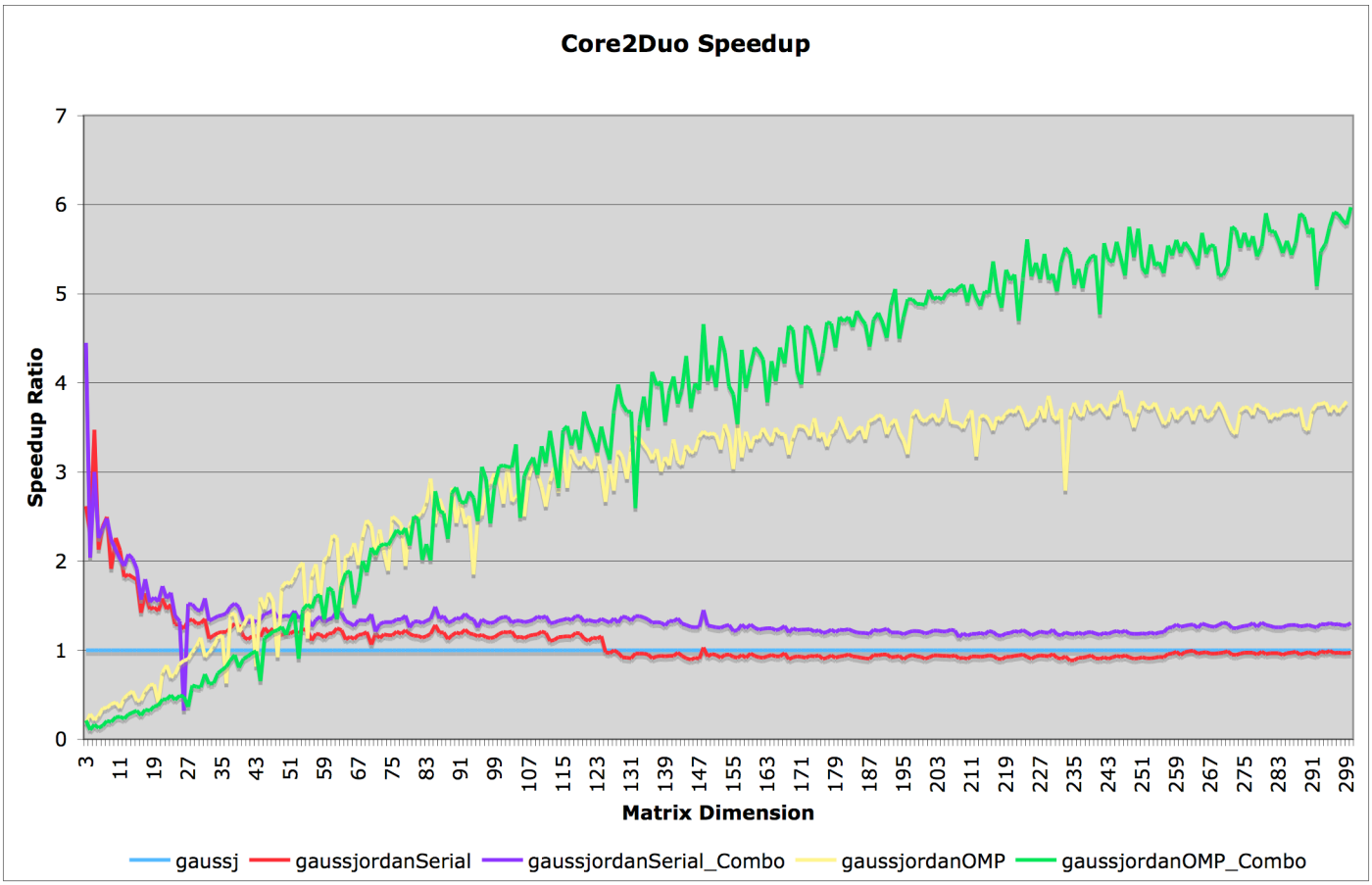

Figure 2b: *Timings taken on a Core2Duo processor 2.33Ghz w/ 4Mb cache using Intel compiler version 9.1.039 with the -O2 compile switch and OpenMP flag On.*

Interesting events occur when examining the performance of the parallel algorithm in relation to the serial algorithm. At a dimension of 43 it is clear that the gaussjordanOMP algorithm is faster than either of the three serial implementations. The gaussjordanOMP_Combo algorithm becomes the faster than the serial versions at approximately 53. Before wither of these points the obvious optimal algorithm is the gaussjordanSerial_Combo. It is interesting to note that the serial combination algorithm is continually faster than the two stage version but when comparing parallel algorithms it is clear that for values before 83 the parallel

combination algorithm has a lower speedup ratio than the two stage version.

In fulfilling this project it was required to create a 'wrapper function' that generally performed the fastest algorithm at each dimension. As such an overall *gaussjordan()* function was used.

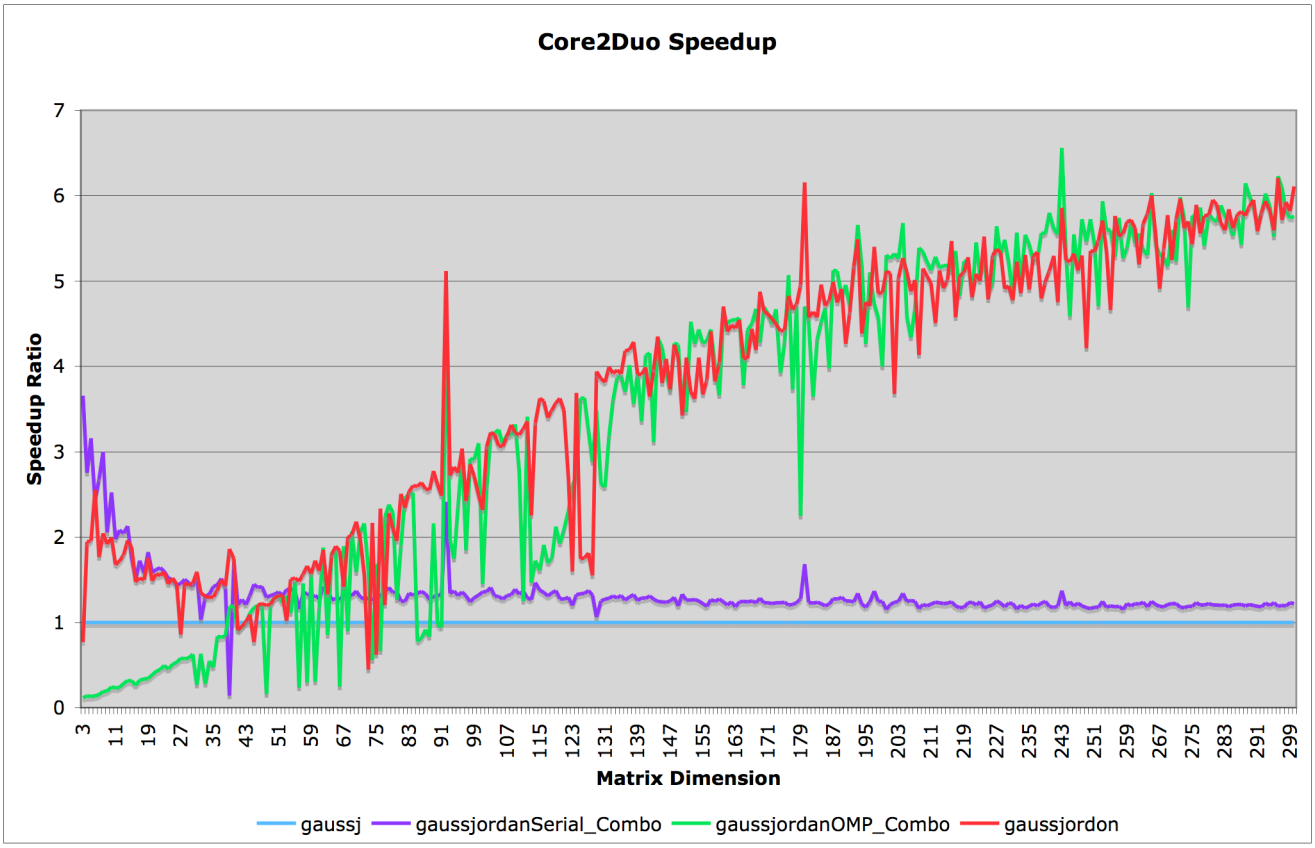

Figure 2c: *Timings taken on a Core2Duo processor 2.33Ghz w/ 4Mb cache using Intel compiler version 9.1.039 with the -O2 compile switch and OpenMP flag On.*

When examining the general trend it is clear that the *gaussjordan()* wrapper uses the most optimal algorithm based on dimension size. As such it has used the intersection point found on a prior run to optimally change algorithm implementation.

The various peaks and troughs are probably cache-misses and a also a result of running the computation on a busy system.

Parallel Performance (Timing) – Core2Quad

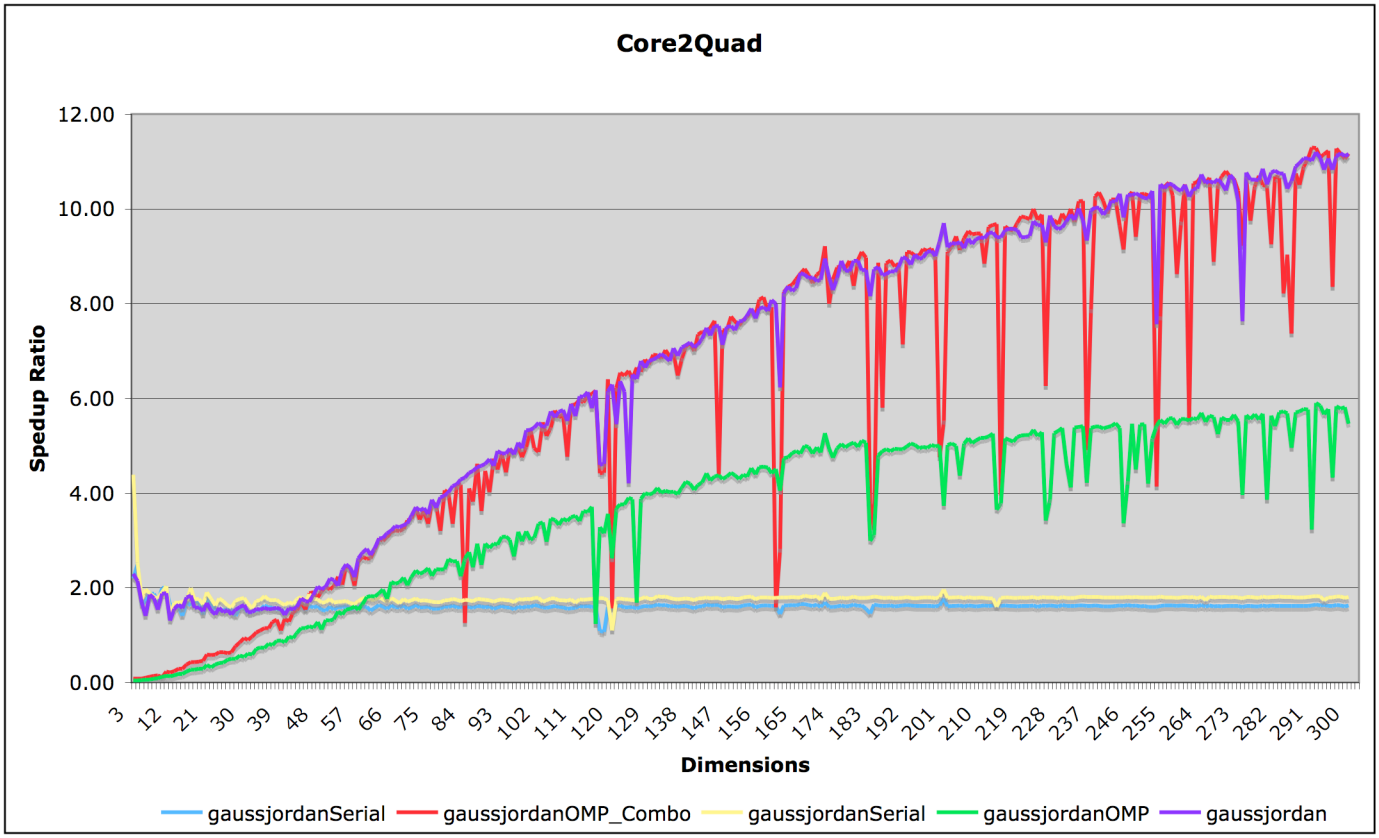

Figure 3a: *Timings taken on a Core2Quad processor 1.86Ghz w/ 4Mb cache using Intel compiler version 9.1.046 with the -O2 compile switch and OpenMP flag On.*

Again the values follow a similar pattern to the previous example except for the fact that the parallel combination algorithm is faster than the other parallel version over all values.

Over a larger dataset:

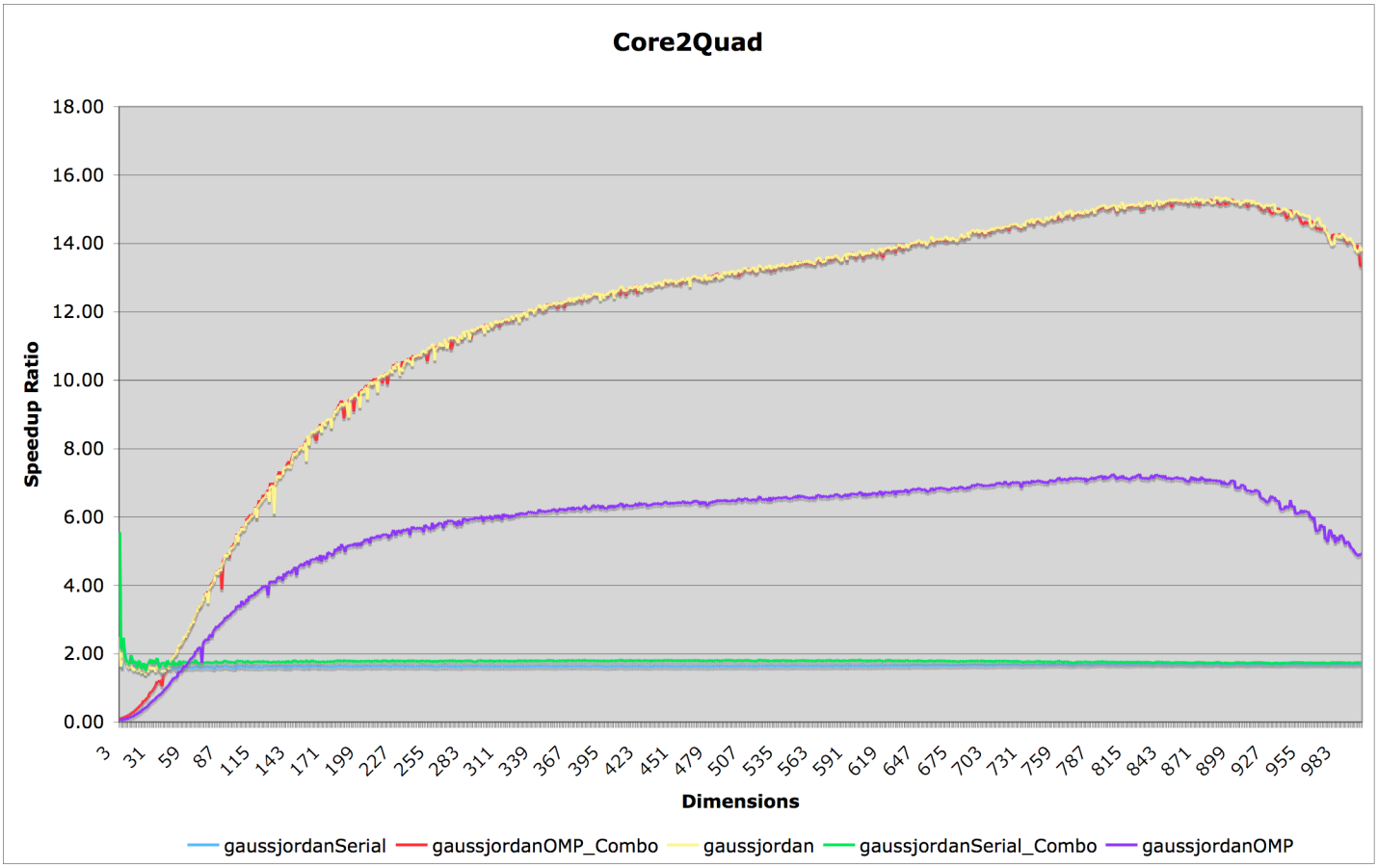

Figure 3a: *Timings taken on a Core2Quad processor 1.86Ghz w/ 4Mb cache using Intel compiler version 9.1.046 with the -O2 compile switch and OpenMP flag On.*

What is very interesting to see here is that both parallel algorithms seem to have peek efficiencies approximately at 870 for the *gaussjordanOMP()* and at 899 for the *gaussjordanOMP_Combo()*. This implies that beyond this point the use of computing clusters may be a good way of continuing the remarkable speedup ratios.

Chapter 5

Summery

5.1 Conclusion

In conclusion the project has succeeded in its general aims of:

1) Creating a faster Gauss-Jordan Inversion algorithm compared to the baseline algorithm
2) Creating a parallelised Gauss-Jordan algorithm that produces significant speedups to warrant use in an everyday computation environment
3) Finding the most optimal algorithm for an inversion relative to dimensional size for specific hardware and processor.
4) Producing an object function that is usable in external applications that require the best solution for the hardware it is run under.

5.2 Future Work

Work that would be of interest would be rewriting the main algorithms using SSE commands. Vectorisation of the values you significantly improve performance especially when using floats where 4 32bit values could be packed in a 128bit SSE native value. Also running the parallelised code over a cluster could probably increase the maximum speed up efficiencies of each algorithm over very large matrices.

Bibliography

[1] Numerical Recipes in C++ (Second Edition)
http://www.amazon.com/exec/obidos/ASIN/0521750334/numericalrecipes
[2] Elementary Row Operations defined
http://en.wikipedia.org/wiki/Elementary_matrix_transformations
[3] Anton, Rorres: Elementary Linear Algebra with Applications, 9th Edition
[4] Inside the Intel Compiler
http://www.linuxjournal.com/article/4885
[5] Intel Compiler Documentation
http://www.intel.com
[6] Information on algorithmic speedup evaluation
http://en.wikipedia.org/wiki/Speedup
[7] SSE Documentation
http://developer.apple.com/documentation/Performance/Conceptual/Accelerate_sse_migration/migration_sse_C/chapter_3_section_2.html
[8] Timing in C code
http://rabbit.eng.miami.edu/info/functions/time.html
[9] Introduction to the Streaming SIMD Extensions in the Pentium III
http://x86.ddj.com/articles/sse_pt2/simd2.htm
[10] Introduction to SSE Programming
http://www.codeproject.com/cpp/sseintro.asp
[11] Gauss-Jordan Elimination
http://en.wikipedia.org/wiki/Gauss-Jordan_elimination
[12] Row Reduction on Matrices
http://www.mcraefamily.com/mathhelp/MatrixReducedRowEchelonForm.htm

[13] OpenMP Website
http://www.openmp.org/
[14] An introduction to OpenMP without agonizing pain – Dr David Gregg
[15] Cell Programming Introduction
http://www.blachford.info/computer/Cell/Cell0_v2.html
[16] Introduction to Cell Programming
IBM Course (TCD)