High precision arithmetic on multi-core processors

Jason Robert Finnegan
B.A.(Mod.) Computer Science
Final Year Project April 2011
Supervisor: Dr. David Gregg

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland
DECLARATION

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

______________________________________________        ________________________
Name            Date
Acknowledgements

I would like to thank David Gregg for all his support and encouragement during this project. Having him as my supervisor was one of the highlights of the project for me, and without him it would not have been possible. I would also like to thank my family and friends supporting me for the past four years.
Contents

1 Project Outline 3
1.1 Outline of the problem .................................. 3
1.2 Why this is the case ................................... 3
1.3 Hypothesis ........................................... 4
1.4 Benefit ............................................. 4
1.5 Goal ............................................... 5

2 Testing Methodology 6
2.1 Matrix Multiplication .................................. 6
  2.1.1 Naive Matrix Multiplication ....................... 7
  2.1.2 Block Matrix Multiplication ...................... 8
2.2 Data Types ......................................... 9
  2.2.1 More about GMP MPF ............................ 10

3 Implementation 11
3.1 Parallelisation ...................................... 11
3.2 Allocating the matrices ............................... 11
  3.2.1 "One big chunk" ................................ 11
  3.2.2 "Array of pointers" ............................ 12
3.3 Block size value .................................... 12

4 Results and Analysis 14
4.1 Overall ............................................ 14
4.2 Block versus Naive .................................. 14
  4.2.1 Float .......................................... 15
  4.2.2 Double ....................................... 16
  4.2.3 Quuble ........................................ 18
  4.2.4 Quad .......................................... 19
  4.2.5 GMP Multi-Precision Float .................... 21
4.3 High versus Standard precision ........................ 23
  4.3.1 Naive algorithm, sequential environment ....... 23
4.3.2 Naive algorithm, parallel environment . . . . . . . . . . 25
4.3.3 Block algorithm, sequential environment . . . . . . . . 26
4.3.4 Block algorithm, parallel environment . . . . . . . . . . 27
4.4 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
Chapter 1

Project Outline

1.1 Outline of the problem

Currently applications do not generally use precision above single and double precision floating point numbers, even if it would theoretically benefit the application. Graphical and scientific programs in general could benefit from using higher precision than they currently do, however do not choose to. The reason for this is because single and double precision arithmetic is supported in hardware, whereas higher precision is not, and is thus perceived as much faster than their higher precision alternatives. Thus, high precision arithmetic has very little support on the hardware side of things, as it is not perceived as important.

Programs which do provide the option for high precision data types generally use the GNU Multi-Precision Library, or GMP, as it is free and has been around for over 20 years, thus there is a lot of documentation and experience that can be availed of. Examples of such programs which use the GNU Multi-Precision Library to provide high precision arithmetic support include Mathematica[1] and Maple[2], both of which are computational algebra programs used in scientific, engineering and mathematical fields of computing.

1.2 Why this is the case

Hardware high precision arithmetic is virtually non-existent, due to the relatively low demand for it. Conversely, high precision software libraries are readily available and generally free, but are not used in many places where they could be.
The primary reason for this is because of the perceived massive slowdown associated with the switch to a precision higher than that supported in hardware. Where an arithmetic operation on a single or double precision floating point number may take only one or two cycles, the same operation on a quad precision floating point number may take tens of instructions. This leads to the perception that, for example, going from double precision to quad precision gives far more than a factor of 2 slow-down, for a factor of 2 increase in precision. In the past this has indeed been the case, since the limiting factor when determining the slow-down associated with arithmetic operations would have been the number of CPU instructions it took. However, the recent trend towards multiple core CPUs has provided a huge boost to the amount of processing power available to programmers.

1.3 Hypothesis

High precision arithmetic may no longer be the massive slow down it used to be, instead, it may now be an almost linear trade-off between precision and speed due to the limiting factor moving from processing power to memory bandwidth. Where in the past, the number of time a high precision operation might take would be primarily determined by the amount of time the processor took to execute the relevant arithmetic operations, on modern computers with massive amounts of processing power available that may no longer be the case. Instead, the time it takes to load the data from memory on which the operation is to be performed on may be a much larger part of the total time that operation takes, for example.

One of the aims of this project is to find out what the primary factor is in the amount of time it takes to execute a high precision operation compared with its lower precision counterparts.

1.4 Benefit

There are several benefits of investigating the slow-down presented by high precision data types compared to single and double precision. Firstly, if the effects of using a higher precision are more transparent, both users and developers are able to make better educated decisions on whether or not to use high precision in place of standard precision, and can weigh its benefits with its costs more effectively.
Secondly, the perceived huge performance hit associated with the move to high precision arithmetic can be validated or invalidated, depending on the truth behind the hypothesis of the project. In the case that the hypothesis holds true, reassurance that using high precision will not result in massive performance loss may lead to increased use, and thus increased support and wider availability of tools.

1.5 Goal

The goals I set out to accomplish with this project were as follows:

- Determine the costs of using high precision over the standard precision supported by default.
- Find out what the primary factors in the cost of high precision over standard precision and their weight.
- Compare and contrast the resulting slowdown from different architectures.

Overall the primary goal is to prove or disprove the hypothesis outlined earlier, and to investigate the extent to which it holds true.
Chapter 2

Testing Methodology

2.1 Matrix Multiplication

The first thing that I needed to resolve was the arithmetic that I would generate my results from. I needed a common and non trivial operation that could be easily parallelised, and so matrix multiplication was an obvious choice. Matrix multiplication also had the added bonus of having a huge amount of research into different algorithms and their benefits, which helped immensely.

Some of the algorithms I looked into using included discrete fourier transform, sparse matrix multiplication and the Strassen algorithm for matrix multiplication, but I eventually settled on the relatively simple naive and block algorithms for matrix multiplication for several reasons. Before going in to why I chose to use these two algorithms them, first I will describe the algorithms themselves.
2.1.1 Naive Matrix Multiplication

First, a very simple version of the algorithm in pseudo code form:

```plaintext
for(i = 0; i < N; i++){
    for(j = 0; j < N; j++){
        for(k = 0; k < N; k++){
            result[i][j] = result[i][j] + ( matrixA[i][k] * matrixB[k][j] );
        }
    }
}
```

This algorithm essentially runs through each position in the result matrix, and does all the relevant calculations for that square before moving on to the next. The algorithm simply runs through each of the positions in the result matrix in this way, until it has passed through each position once. In the preceding graphic, the purple square represents the result position currently being calculated and the red and blue squares are the source squares.

A weakness of this algorithm is that even though the same red squares are used from step one to step two, the amount of data in total covered by the red squares can exceed the size of the cache for matrices of non trivial size. Thus by the time step one finishes and step two begins, the far left red square (the one which would have been used first in step one) may no longer be in cache and have to be loaded from memory again.
2.1.2 Block Matrix Multiplication

Again, a simplified version of the algorithm in pseudo code form

\[
\text{for}(ii = 0; ii < N; ii = ii + \text{Block\_Size})\
  \text{for}(jj = 0; jj < N; jj = jj + \text{Block\_Size})\
    \text{for}(kk = 0; kk < N; kk = kk + \text{Block\_Size})\
      \text{for}(i = 0; i < \text{minimum}(N, \text{Block\_Size}); i++)\
        \text{for}(j = 0; j < \text{minimum}(N, \text{Block\_Size}); j++)\
          \text{for}(k = 0; k < \text{minimum}(N, \text{Block\_Size}); k++)\
            \text{result}[i][j] = \text{result}[i][j] + (\text{matrixA}[i][k] * \text{matrixB}[k][j]);
\]

While initially looking very similar to the naive matrix multiplication algorithm, the block version is in fact quite different. The algorithm takes a ”block” of the first matrix, and then proceeds to use that block in conjunction with another block in the second matrix to work out partial results for all the result positions that correspond to the two selected blocks. After doing all the calculations with the first two selected blocks, the second block is moved and the same thing happens again.
The strength in this algorithm is in the fact that whenever a block of data is loaded into memory from matrix A, it will never leave cache until it has been used for every calculation it will be needed for. For the algorithm to work at its optimum, the block size must be as large as possible without being too big such that 3 blocks cannot fit into level 1 cache. Determining the block size involves running the algorithm several times in a given test environment with several different block sizes and observing which works best.

There were several reasons for choosing these two algorithms for the majority of my testing. Primarily, having two algorithms with the same number of operations but different memory bandwidth strategies means comparisons between the two can result in interesting observations. For example if they both perform similarly for a certain data-type and hardware set up, one can deduce that memory bandwidth is not an issue in the scenario. Another reason is they are both easily parallelisable, which was essential in being able to unlock the CPU rich environment in which the majority of my tests had to be run in.

2.2 Data Types

To properly test the effects of increasing the precision of increasing the precision of the data being worked with, several different data types are required. The data types I chose to use were

- Float: Serves as the base single precision (4 bytes) floating point data type.
- Double: Doubles would make up the base case for most of my tests, as it is the maximum precision supported in hardware. It is the standard double precision (8 bytes) floating point data type and is the most commonly used when floating point arithmetic is required.
- Float128 (Quad): A quadruple precision (16 bytes) floating point data type, provided by the GNU C compiler that works on x86, x86-64 and Itanium CPUs. It is provided by using software emulation provided with libc6 (GNU C Library).
- GMP MPF: The GNU Multi-Precision library Multi-Precision Float is the floating point data type provided by GMP[3]. It has a variable amount of precision, but defaults to quadruple precision until needing
to grow larger. The maximum precision provided is only limited by available memory, but for testing purposes I limited them to certain sizes depending on the test.

- Quuble: This is a data type I made myself, it has the same size as a quadruple precision data-type (16 bytes) yet only performs arithmetic as if it was a double precision data type. The purpose of this is to act as a control sample, when testing between data types instead of having to change both the size of the data and the amount of arithmetic being done; instead only one of these has to change if desired.

2.2.1 More about GMP MPF

GMP is part of the GNU project, and thus information on its internals is easily accessible. The GMP mpf is a different data type than what was initially expected. When the variable is initially allocated in memory, it allocates 24 bytes of space which includes a pointer which will point to the first "limb" of the structure, and other meta data about the data type. This is important because the memory used for the actual calculations and storing of values is not allocated until the mpf is initialised using functions provided by GMP. In addition, the memory allocated for storing the number itself can grow and shrink depending on the amount of precision required, adding or subtracting limbs as needed. As long as this is taken into account when the results are assessed it should not be a problem however.
Chapter 3

Implementation

3.1 Parallelisation

To gain access to the massive amounts of CPU available in modern architectures, parallelisation is a requirement. To for my project I looked at both POSIX Threads (Pthreads)[5] and Open Multi-Processing (OpenMP)[4] before deciding to use Pthreads for all of my testing. The reasons for this were mainly I have much better experience using Pthreads over OpenMP, and I feel Pthreads allow greater control over the parallelisation; whereas with OpenMP direct control over threads is more limited and it is overall less transparent.

3.2 Allocating the matricies

Allocating the memory which would be used to store the matricies was an important part of the project, as it would have an effect on almost everything else done in testing. There were two ways I went about allocating the a matrix.

3.2.1 ”One big chunk”

This method of storing the matrix basically just allocates the total amount of space that would be required for the entire matrix, and then splits it up into rows on the user side of things. For example, in this scheme of allocating the matrix, referencing the fourth element in the second row would be done as follows;

matrix[N*1 + 3];
Where N is the length of the row. This method allows better locality of the data, as it is guaranteed to be contiguous in memory, and also means the matrix structure takes up less space overall since there is only one pointer (to the start of the matrix). The trade-off is that one multiplication and addition has to be done every time data is accessed.

3.2.2 "Array of pointers"

This is the more commonly used method of storing a matrix; first enough space is allocated to store a pointer for each row, and then each row is allocated and assigned to one of those pointers. It functions similar to a 2 dimensional array, an example of accessing the fourth element in the second row in this scheme would simply be;

```
matrix[1][3];
```

This method eliminates the offset calculations that the previous method needed, however it comes with the cost of extra space required to hold the extra pointers required, and potentially much worse locality of the data in memory.

After testing each method in every algorithm and environment I was using, I found the differences between them to be negligible. The results for the "Array of pointers" method had slightly more variance than the "One big chunk" method so I decided to use the latter in all of my tests for that reason, even though I planned on running the tests enough times to eliminate variance anyway.

3.3 Block size value

For the blocking matrix multiplication, finding the optimal block size for each data type in both parallel and sequential environments was critically important. The block size has to be big enough that data is loaded as few times as possible, but small enough that it can fit in cache along with a block of the second matrix and a block of the result matrix. Having a too large block size would result in the blocking algorithm essentially just not working.

Finding the block size for each data type involved running scripts that ran a specific algorithm in parallel or sequential and varied the block size up and down until one was found to be consistently the best for that environment. This had to be done for each variation of each algorithm on every machine I
did tests on, as the cache sizes did not remain constant and thus neither did the block sizes.
Chapter 4

Results and Analysis

4.1 Overall

There are two major ways in which I analysed the results from the tests. In one I compared the effectiveness of the naive algorithm with the block algorithm for specific data types in different environments, in order to determine whether memory bandwidth was an issue in that environment and also to get a rough idea of how much the CPU bottleneck was affecting the performance. In the other I looked at the results from different datatypes and compared their relative performance in the different environments, in order to gauge how well high precision arithmetic was performing in relation to its standard precision counterpart.

4.2 Block versus Naive

The first major area of interest was the total speed up given by going from the naive algorithm to the block algorithm, for all the different data types. The results of this would give a rough outline of how CPU intensive arithmetic in a given environment on that data type was; a large speed up from naive to block would mean that the amount of memory bandwidth available was having an effect, and no speed up from naive to block would show that the arithmetic was limited almost entirely by CPU availability. This test was done for every data type in both environments (parallel and sequential) on the same machine (a 4 core @ 2.13GHz machine with a level 1 cache size of 4096KB on each) and the results were as follows;
4.2.1 Float

For floats, the expected decrease in time taken when going from naive to block algorithm is low when parallelised, and almost non-existent for sequential programs. The reasoning behind this is that floats are rather small in memory and thus there will not be that many cache misses that could be avoided by better memory management in a sequential program. In the parallel version, there should be a small but noticeable increase in performance, as better memory management between threads will result in improved performance even when the amount of memory being accessed is low.

As predicted, substituting the block algorithm for the naive algorithm had barely any effect on the time taken to multiply square matrices of floats in a sequential environment.
Also as predicted, a slight increase in the performance.

4.2.2 Double

Doubles are similar to floats, in that they both take very few CPU cycles to do basic arithmetic on; however since the block algorithm mainly increases the performance in the area of memory access, the results are expected to be fairly different. Since doubles are essentially twice the size of floats and thus take up quite a large amount of space in memory, there should be a significant performance increase in both the sequential and parallel environments for doubles. Another reason for the expected increase in performance is that the major bottleneck - especially in the parallel environment - should be memory bandwidth related instead of CPU, as doubles are relatively efficient in terms of how much processing power they require.
The increase in performance when using the block algorithm over the naive algorithm, particularly the parallel result, confirms the widely accepted theory that when available CPU is enough to reduce the CPU bottleneck, the bottleneck shifts to memory. It also helps to confirm that the block algorithm does indeed improve memory access times significantly.
4.2.3 Quuble

Since a quuble is essentially a double that takes twice as much memory, it follows that any performance increases shown by doubles by changing to the blocked algorithm should be emulated twofold in the same tests but on quubles. If this holds true it essentially allows us to confirm beyond reasonable doubt that the performance increase provided by going from the naive to block algorithm is caused by alleviating the memory bandwidth bottleneck.
As shown, increase in performance for the quuble data type is almost double the increase for the double data type in both environments. This is a good result in that it lets us look at any performance increases provided by the block algorithm as almost entirely an alleviation of bandwidth related slowdowns.

### 4.2.4 Quad

For the quadruple precision data type, little to no speed improvement is expected in the sequential environment, as the amount of available processing power combined with the fact that quadruple precision arithmetic is very CPU intensive means that memory bandwidth optimisation will do very little to affect the performance. However, if the hypothesis of this project holds true, in the parallel environment where processing power is more abundant and CPU ceases to be the only bottleneck, the block algorithm should provide some noticeable performance gains.
As expected there was almost no improvement in performance by swapping to the blocked version of the algorithm in a sequential environment.

As this result shows, when doing high precision arithmetic in a processor rich environment, the performance can be significantly improved by optimising memory accesses. What this shows, is that there is certainly some truth to the hypothesis that the amount of processing power available on modern multi-core architectures shifts the bottleneck associated with high precision
The extent to which multi-precision floats (henceforth MPF) improve with changing the algorithm from naive to blocking is hard to estimate as they can grow during execution and thus take far more memory than expected. Another factor adding to the difficulty of predicting the results for MPF data types is that they have extremely optimised algorithms for arithmetic, so the assumption that CPU will be the major bottleneck in the sequential environment may not hold true. Still, the prediction is that going from naive to blocking algorithms will result in nearly no speed up for the sequential version, and a significant speed up in the parallel environment, similar to the quadruple precision data type.

Surprisingly, MPF showed a noticeable increase in performance even when run in a sequential environment. What this means is that even only using a
single core of CPU, the processing power available is not the only factor in the performance of MPF arithmetic.

MPF showed huge improvement in a parallel (CPU rich) environment when optimised for memory access. This leads to the conclusion that in this environment for MPFs the CPU bottleneck is not the only factor, or even the dominant factor in the time involved with high precision arithmetic.
4.3 High versus Standard precision

The second major area of interest in analysing the results is when comparing how the higher precision data types performed with regard to the standard precision used in most cases where some degree of precision is required. These results will give indication as to the factor of slow down to be expected with a switch to higher precision.

4.3.1 Naive algorithm, sequential environment

Absolute:
As expected in a sequential environment the slowdown going from double to any high-precision data type is huge. A slow down factor of around 25 for going from double to quad precision roughly mirrors increase in the number of CPU instructions required to calculate the results for higher precision. The GMP MPF is more than two times faster than its quadruple precision equivalent, even though they both provide the same precision in this case. This can be attributed to the heavily optimised functions GMP MPFs use for arithmetic. Interestingly the GMP is only around a factor of 11 slower than the double precision equivalent, this is less than would be expected if CPU instructions was the only factor in the performance of the data types.
4.3.2 Naive algorithm, parallel environment

When comparing the results from the parallelised version of the naive algorithm it becomes obvious that something other than the increase in required instructions is effecting the slow down when going from standard to high precision. For the MPF data type, the slow down for arithmetic on a square
matrix of size 150 is only a factor of 6, whereas the slow down factor of a square matrix of size 300 is 25. Similarly for the quad data type, the slow down factor varies significantly with the size of the matrix.

### 4.3.3 Block algorithm, sequential environment

**Absolute:**

![Block - Sequential Absolute](image)

**Relative:**

![Block - Sequential Relative](image)
In the sequential environment, the block algorithm gave a speedup to everything but the quad data type, and thus the quad data type performed even worse relatively than it had in the naive environment. Since the quuble result was not affected in the same way, we can safely attribute this to the extra instructions required for quad data types and not to the extra memory bandwidth required, as if the memory bandwidth was a significant factor, then the quuble data type would have had similar problems. The MPF performed similarly to how it had in the naive algorithm, still maintaining the factor of 11 slow down compared to double.

### 4.3.4 Block algorithm, parallel environment

- **Absolute:**

![Block - Parallel](image)

- **Matrix Size:**

- **Time taken:**

- **Data Types:**
  - DOUBLE
  - QUUBLE
  - QUAD
  - MPF
Using the block algorithm in a parallel environment, we can see a significant increase in the relative performance of both the MPF and quad data types until matrix size 512, where the MPF matrices cease to fit in to level 1 cache and thus result in a large spike in the time taken. Prior to the matrix reaching a size of 512, the MPF data type is performing at around a factor of 8 slow down over double, which is a much better ratio than the number of instructions each one takes. The quad data type also shows reasonable relative increases in performance, for the lower matrix sizes.

### 4.4 Conclusions

Overall the results show that high precision data types are still a rather heavy investment. Although the performance can be increased by parallelising and thus providing more CPU, this does not yet overcome the increase in instructions required, only lessen the effect of them. The GMP MPF performed significantly better than the Quad, despite taking more space in memory; the reasons for this are because improving the efficiency of the arithmetic functions has a much bigger effect on the overall performance than increasing the amount of memory bandwidth required.
Bibliography

[1] Wolfram Mathematica - notes on internal implementation


[3] GNU Multi-Precision manual for floating point data type
    http://gmplib.org/manual/Floating_point-Functions.html

[4] OpenMP (Open Multi-Processing)
    http://www.openmp.org/

[5] PThreads
    POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)