ENCE464 Group Assignment 2

Tim Hadler - 44663394
Hassan Ali
Department of Electrical and Computer Engineering
University of Canterbury

October 10, 2020

1 Introduction

Numerical methods are often used to solve problems that are difficult or impossible to solve analytically. Often numerical methods repeat loops for a large number of iterations. For large problems, or for problems that require high accuracy, it may be desirable to optimise the speed of the numerical method. Optimising the speed of a numerical method can be done by a number of ways. The code itself that implements the method can be optimised to reduce the amount of time spent in repeated loops, and to make efficent use of the computers cache memory. Splitting the method into threads can also speed up the performance of numerical methods by executing multiple instructions on the same core at the same time. In this report, the Jacobi relaxation for a 3-D source distribution is optimized by code and cache optimisation, and multithreading. An analysis is done on how these techniques affect the overall time performance of the algorithm.

2 Background

2.1 Caches

Caches are a type of memory used to speed up the average memory access time by storing commonly used copies of values. The cache memory offers a small amount of SRAM, which can be accessed by a CPU much faster than main memory (DRAM). When a compiler optimises a programs execution, it will predict which memory locations will be accesed and store the associated values in a cache. A compiler predicts which values will be used by applying the concept of locality. Temporal locality refers to the concept that recently read memory locations are likely to be read again. Spatial locality refers to the concept that data/instructions near memory locations already in use are likely to be read in the future. These concepts lead to the opportunity to increase the efficiency of a program by optimising the way the code accesses memory. This applies particularly to implemenations of numerical methods, which often include several for loops.

2.2 Threading

3 Methods

This project involved taking an implementation of the Jacobi relaxation numerical method, and optimising its efficiency using cache, profilling, and code analysis. Multithreading was used to optimise the usage of the computers cores.
The project was performed on a University CAE computer with the following architecture overview:

- GenuineIntel x86_64
- 12 cores
- 2 threads per core
- 900 MHz CPU speed

- 32Kb Level 1 data and instruction caches
- 256Kb Level 2 cache
- 12Mb Level 3 cache

3.1 Cache optimisation

For each iteration of the Jacobi relaxation method, every element of a given 3-D source distribution is fetched from memory. Since the compiler stores memory values in caches based on spatial locality, the efficiency of the numerical method can be optimised by considering the order the values of the source-distribution is fetched. In C a 3-D matrix is stored in memory by flattening it into an array. The matrix values are stored row-by-row. For the source distribution maytrix, the value at a particular i, j, k position is access by:

$$distribution[((k * ysize) + j) * xsize + i] \tag{1}$$

Where xsize and ysize are the lengths of the distribution in the x and y direction respectively. The original implementaion of Jacobi relaxation, iterated over the y, then the z, then the x directions. To make better use of the memory caches, the revised implementation iterates over the x, then the y, then the z directions. This implementation accesses memory more linearly, and takes advatage of the compiler cache optimisation based on spatial locality.
The utilisation of the caches for each method implementation is compared by looking at the average memory access time, which is calculated using Equation (2).

3.2 Threading

3.3 Code optimisation

Something that can significantly slow the process of a numerical algorithm is conditional statements within loops. A numerical method used to solve a problem with boundary cases might have if statments within loops to check when the current point is on a boundary. However, the majority of times the loop is executed, the current point will not be on a boundary. To remove the wasted time on checking if the current point is on a boundary, we can deal with the boundary cases outside the loop, or within a different loop. The revised implementation solves for points on each side of the 3-D distribution in seperate loops. The corner points are handled individually. After this, the remaining points within the boundaries can be iterated over within a loop without any conditional statements.

4 Results

4.1 Cache optimisation

4.2 Threading

4.3 Code optimisation

5 Conclusion