19/Oct/2017

HPC - Serial Optimisation Report

Name: Shuai Ke Student no: 1609628

Introduction

This report will show some optimisations of the jacobi algorithm. These optimisations include adding compiler flags, using memory hierarchy, SMID, changing data type, loop fusion and changing compiler, which has a considerable improvement from original code.

Content

The runtimes of original code without any optimisations, shown in *Table-1*.

Data size	500*500	1000*1000	2000*2000	4000*4000
Run time	1.48 s	10.9 s	130 s	1180 s

Table-1

Optimisation 1 - Compiler flags

The first optimisation is adding compiler flags, I have referred the document of gcc compiler:

- -O, -O1: the compiler will try to reduce code size and execution time.
- -O2: turns on all optimization flags specified by -O.
- -O3: turns on all optimizations and uses more vectorisation algorithm.

In order to have a maximum improvement, I choose -O3 flag and the results shown in the *Table-2*.

Data size	500*500	1000*1000	2000*2000	4000*4000
Run time	0.49 s	3.29 s	51.8 s	500.9 s

Table-2

Optimisation 2 - Use memory hierarchy

The original code uses Column Major Order to express matrix, which leads to a low cache hit rate and many data reload since Row Major Order be used in C language. After using Row Major Order replace Column Major Order, the runtimes are shown in *Table-3.*

Data size	500*500	1000*1000	2000*2000	4000*4000
Run time	0.34 s	2.74 s	23.4 s	170.2 s

Table-3

Optimisation 3 - Undersized Improvement

This optimisation tries some improvements but not very useful because the optimisation is not in the core code. For example:

- Using multiple operation to replace division and square root.
- Calculating the row index in advance, like storing $row * N$ in a variable named $ridx$.

Optimisation 4 - SMID

This optimisation uses SMID to optimise program. SSE use 128 bit length register, which can operate two double variables once. AVX use 256 bit length register, which can operate four double variables once. When I used AVX, the program has a error, *segmentation fault*, and I use ***'_mm_malloc'*** function to align memory when dynamically applying array. The runtime results are shown in *Table-4*.

Data size	500*500	1000*1000	2000*2000	4000*4000
Run time	0.20 s	1.54 s	16.1 s	122.1 s

Table-4

Optimisation 5 - Data type and Loop fusion

This optimisation changes data type from double to float since $MAX\_INT \approx 2.1 * 10^9$ and $MAX\_FLOAT \approx 3.4 * 10^{18}$, the length of float is shorter than double which has a better computing performance. And it is useful to help compiler do vectorisation by removing the if statement ***'if (i != j)'*** and minus this item, which row equals to column, after loop. The program runs more faster than **Optimisation 4** since let the compiler do the vectorisation is more efficient.

Data size	500*500	1000*1000	2000*2000	4000*4000
Run time	0.10 s	0.73 s	5.28 s	53.09 s

Table-5

Optimisation 6 - Compiler choice

The last optimisation is changing compiler by icc-16.0.0 compiler.

Data size	500*500	1000*1000	2000*2000	4000*4000
Run time	0.09 s	0.57 s	4.15 s	46.28 s

Table-6

Additional optimisation - Algorithm

Compare Gaussian Elimination with Jocobi method, The Gaussian results are shown in *Table-7.*

Data size	500*500	1000*1000	2000*2000	4000*4000
Run time	0.05 s	0.37 s	3.54 s	32.8 s

Table-7

Summary

This experiment from some different aspects to optimise the original code, such as compiler, memory, vectorisation. Compare **optimisation 5** and **optimisation 3,** it is clear that improving critical code is more useful. And the **Additional optimisation** demonstrates that different algorithm will have considerable effectiveness during a large data set.