Parallel Computing – Assignment #01

Matrix Multiplication using OpenMP

Student: Chiang-Heng Chien (簡江恆)
ID: 60475001H
Advisor: Prof. Cheng-Hung Lin

Abstract

In this assignment, several methods are employed to challenge the performances of a simple matrix multiplication algorithm using Visual Studio C++ and OpenMP, including “transpose method”, “array dimension reduction method”, “loop reduction method”, “multiple blocks method”, and “tile-wise parallelization”. The reasons why the above-mentioned methods can accelerate the required computational time are briefly provided. The corresponding performances are compared to the serial version and the conventional design of OpenMP, where the experimental results show that the method is the fastest one among the all. To firmly validate the results, 20 independent runs are conducted to reach a statistical conclusion.

Pre-requisitions

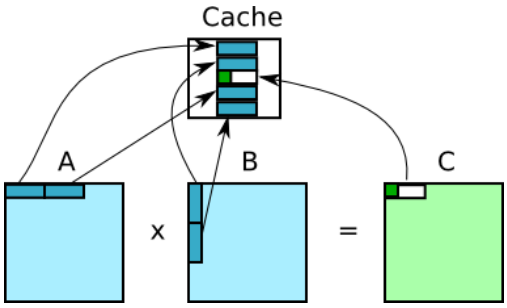
This assignment provides the user to perform matrix multiplication **C**=**AB** using arbitrary dimensions, where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{B} \in \mathbb{R}^{n \times p}$, and $\mathbf{C} \in \mathbb{R}^{m \times p}$. However, for multiple-blocks method and tile-wise parallelization one, **A** and **B** are square matrices.

Brief Descriptions of the Methods Used in the Assignment

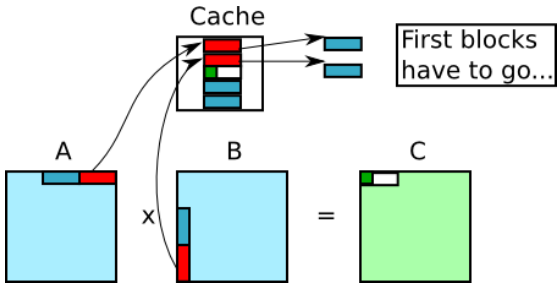
In this section, I will cover descriptions about three methods including “array dimension reduction method”, “loop reduction method” and “tile-wise parallelization”. For array dimension reduction method, I aim to reduce the dimension of array from 2 to 1. This is due to the fact that 1D approach offers better memory locality, less allocation and deallocation overhead. Note that in my program, the matrix B is prior to be transposed before using array dimension reduction method. (I take the reference from “http://stackoverflow.com/questions/17259877/1d-or-2d-array-whats-faster”) As for “loop reduction method”, since the outer loop of the simple matrix multiplication algorithm may write into the same matrix **C**, compiler adds additional locks on it. Therefore, a reduction is used in this assignment. Last one is the tile-wise parallelization, which makes optimal use of cache memories. The basic illustration is

described below:

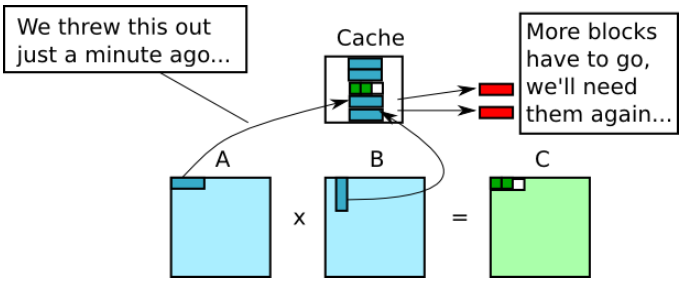
When the innermost loop of matrix multiplier reads entire rows/columns in sequence, the cache gradually fills up with data:



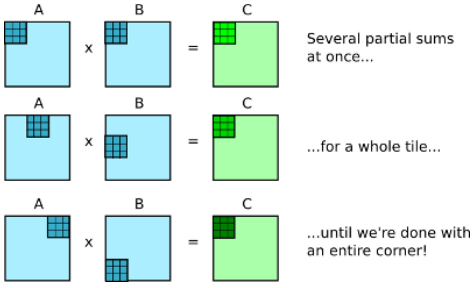
However, cache size is limited, so if the rows are really long, the cache must throw away what it loaded initially, to make room for new stuff,



When the program reaches the end of a row/column pair and start the next one, it is necessary to require some data that were recently in the cache. Since they've been thrown out, the program has to wait for them to come back from memory again:



This is a tragic waste of effort, because matrix multiplication is just a massive addition of a pile of products - it doesn't matter much which order they are added up in. If we work out partial sums for a tile of results at a time, it'll be ok as long as all the right products contribute to the right sums in the end,



It will still cause some re-loading, but there's less, because if tiles fit into caches

where whole matrices cannot, the program will get more of the overall work done per cache: values in a tile go into more than one result value before they are evicted.

Experimental Results

The experiments are performed on a laptop with an i5 CPU of 2.67 GHz and 4.00 GB RAM. It is noteworthy that 20 independent runs are conducted, and the following experimental results are the average computational time obtained from the variable "clock_t t1 = clock();". Table I shows the results, where **A** and **B** are square matrices with size 1000x1000, and the block size as well as the tile size are identically 100x100.

Table I. Experimental Results of Matrix Multiplication using OpenMP

Method	Computational Time (s)
Serial	1.7
Transpose Method	0.59
Array Dimension Reduction Method	0.51
Loop Reduction Method	0.4
Multiple Blocks Method	0.38
Tile-Wise Parallelization	0.37

Table II shows the experimental results, where **A** is the size of 4000x3000, and **B** is the size of 3000x1000, respectively. This experiment aims to challenge the performances when the numbers of columns are lesser than the rows. If so, it is expected that the effect using parallel computing will be clearer than Table I.

Table II. Experimental Results of Matrix Multiplication using OpenMP

Method	Computational Time (s)
Serial	7.16
Transpose Method	3.07
Array Dimension Reduction Method	2.92
Loop Reduction Method	1.09

From table I, we can see that "Tile-Wise Parallelization" method is much faster than any other methods, and compared to the sequential version, it accelerates 5-6 times of computational efficiency. As for table II, we can see that loop reduction method is faster, almost 7 times faster than the sequential one.