Machine Problem 1

CS 484 - Parallel Programming

Due: February 23, 2024 @ 23:59

(Typeset on : 2024/02/08 at 19:21:05)

Introduction

Learning Goals

You will learn the following:

- How memory access patterns can affect the performance of serial algorithms.
- How to reason about the relative performance of two algorithms based on machine characteristics.
- An introductory OpenMP directive.
- How to calculate Speedup and Parallel Efficiency.

1 Assignment Tasks

The basic workflow of this assignment is as follows:

- Clone your turnin/mp1 repo
  Your repo should be visible in a web browser and clonable with git from
  `https://gitlab.engr.illinois.edu/sp24-cs484/turnin/<YourNETID>/mp1.git`
- Implement the algorithms / code for the various programming tasks.
- Build and test on Campus Cluster (see README.md) (It should pass all tests except "Cache.TileSizeChosen" , which you will handle later.)
- Push your complete, correct code.
- Do at least two rounds of benchmarking on Campus Cluster (`scripts/submit_batch.slurm`):
  1. To determine the best tile size for the Campus Cluster nodes.
  2. A final round to get the benchmark results you will use for your writeup.
- Push any benchmarking results that you wish to and final versions of code files. (At this point, you should pass all tests.)
- Write and push your writeup.

1.1 Matrix Transpose

This exercise introduces *tiling* as a way of improving cache performance. There is a function named `transpose_tiled` in `transpose/transpose.cpp`. Implement this function (matrix transposition) using the *tiling* concept from class.

For simplicity, we will only deal with square matrices and tiles, and you may assume that the matrices are optimally aligned in memory.

After implementing and verifying, benchmark this function on Campus Cluster (see Section 2). Then answer the following questions in your writeup:

1. For a square matrix of size $N = 2048$, plot the average execution times as the `tile_size` varies over $2, 4, 8, ...512$. Explain the trend in terms of the machine's cache specifications. (If you use the provided `scripts/batch_script.slurm`, you will have these results in `writeup/tile_results.txt` .) You can find details about the cache using the following command: '`getconf -a | grep CACHE`' . (On campus cluster, the run script will do this automatically and place the output in `writeup/cache_properties.txt`)

2. On the basis of the above results, alter the definition of `BEST_TILE_SIZE` in `transpose/best_tile_size.h` (and add/commit), re-run the batch script, reexamine the tiling results, and plot the execution time varying the matrix size N over $128, 256, 512, 1024, 2048, 4096, 8192$. Explain these new results in terms of the cache specifications of the machine on which you are benchmarking (Campus Cluster).

1.2 Matrix Multiplication

In this exercise, you will analyze the performance difference of two variants of matrix multiplication.

Implement the functions `multiply_tiled` and `multiply_transposed` in `mulitply/multiply.cpp` , verify their correctness, and answer the following questions.

1. Explain performance differences that you observe between the two variants, Is one clearly better than the other? Why or why not? (Note that the benchmarks will automatically use the same `BEST_TILE_SIZE` that you determined in exercise 1.1.

1.3 Basic OpenMP

In this exercise you will apply simple OpenMP directives to two for loops (wrapped by functions `squares_serial` and `primes_serial`) and analyze their effects on performance. You will use OpenMP pragmas to parallelized versions of these two loops inside the functions `squares_parallel` and `primes_parallel` located inside `openmp/openmp.cpp` .

1. Explain performance differences that you observe between the two variants. Is one clearly better than the other? Why or why not?

2 Testing and Benchmarking

The process for running unit tests and benchmarks is described in see Section 2.1.

Though we reserve the right to subject your code to any tests of correctness, benchmarks, or human code examination, these are essentially the tests that we will use for grading. In some cases, tests that cannot be provided without revealing an answer ("Cache.TileSizeChosen") have been redacted to only check if your submission is of the correct form, not to check the accuracy of your answer itself.

Below is the grading rubric based on expected speedups for each test. Keep in mind that you may need to run your code several times to find the best speedups. We will also be running your code several times to find the best speedups.

2.1 Compiling and Testing

To compile, just run the script `compile_script.sh` in the scripts directory. To test and benchmark your code, just run the script `submit_batch.sh` in the scripts directory.

You will have output files in your writeup directory that will give you performance measurements, and your `slurm-<JOBID>.out` file will tell you which tests you passed or failed. We use Google Benchmarks for benchmarking your code. Feel free to create your own benchmarks in `tests/student_benchmarks.cpp` if you wish, but you do not have to. If you do use your own benchmarks, then uncomment the student benchmark run command at the bottom of `scripts/batch_script.pbs`.

2.2 Grading Rubric

2.2.1 Part 1: Matrix Transpose

Cache.TileSizeCorrect weight: 0.5
Transpose.TiledCorrect weight: 0.5

- `Speedup` $\geq$ `2.0: 1 point`
- `Speedup` $\geq$ `1.6: 0.5 points`
- `Speedup` $\geq$ `1.05: 0.25 points`

Total weight for part 1: 0.4

2.2.2 Part 2: Matrix Multiplication

Multiply.TiledCorrect weight: 0.5

- `Speedup` $\geq$ `2.0: 1 point`
- `Speedup` $\geq$ `1.2: 0.5 points`

Multiply.TransposedCorrect weight: 0.5

- `Speedup` $\geq$ `2.0: 1 point`
- `Speedup` $\geq$ `1.6: 0.5 points`

Total weight for part 2: 0.4

2.2.3 Part 3: OpenMP

OpenMP.SquaresCorrect weight: 0.5

- `Speedup` $\geq$ `3.0: 1 point`
- `Speedup` $\geq$ `2.0: 0.5 points`

OpenMP.PrimesCorrect weight: 0.5

- `Speedup` $\geq$ `2.0: 1 point`
- `Speedup` $\geq$ `1.5: 0.5 point`

Total weight for part 3: 0.2

3 Submission Guidelines

Your writeup must be a **single PDF file** that contains the tasks outlined above.

Please save the file as "`./writeup/mp1_<NetID>.pdf`*" , commit it to git, and push it to the* **main** *branch of your turnin repo before the submission deadline.*

You must also commit at least the following code files. These files, and only these files, will be copied into a fresh repo, compiled, and tested at grading time.

- `transpose/transpose.cpp`
- `transpose/best_tile_size.h`
- `mulitply/multiply.cpp`
- `openmp/openmp.cpp`

Nothing prevents you from altering or adding any other file you like to help your debugging or to do additional experiments. This includes the unit test and benchmark code. (Which will just be reverted anyway.)

It goes without saying, however, that any attempt to subvert our grading system through self-modifying code, linkage shenanigans, etc. in the above files will be caught and dealt with harshly. Fortunately, it is absolutely impossible to do any of these things unaware or by accident, so relax and enjoy the assignment.

Appendices

A Matrices

As far as this class is concerned, a **Matrix** is a *conceptually* 2D array of either real or complex numbers. This misdefinition does not do any justice to the beauty of linear algebra, but is sufficient for our needs.

This conceptually 2D array will often be stored as a 1D array, for example `double *mymatrix` . Unless otherwise stated, assume that the code uses the conventional storage for the language in use. This class uses mostly C/C++ so we will store matrices in **Row Major** order. `https://en.wikipedia.org/wiki/Row-_and_column-major_order`

The usual notation for a matrix is a capital letter. Subscripts indicate first row and then column.

Ex: $A_{i,j}$ is the element at row i and column j of matrix A.

When written as linear algebra, numbering of rows/columns usually starts at 1. This is why some languages (FORTRAN, MATLAB) choose to index their arrays starting at 1, rather than the 0 we are used to.

Let $A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$

Then $A_{2,3} = f$

This is stored in an array in C like this:

```
//The size of the matrix, must be supplied to any function that
//uses or manipulates the matrix.
int N = 3;
//matrix A as an array.
double A[] = {a,b,c,d,e,f,g,h,i};
```

See the wikipedia page `https://en.wikipedia.org/wiki/Row-_and_column-major_order` for more depth.

A.1 Simple Operations

The simplest matrix operations are **multiplication by a scalar** (usually matrices are uppercase and scalars lowercase)

$$[bA]_{i,j} = bA_{i,j}$$

and **matrix addition**

$$[A + B]_{i,j} = A_{i,j} + B_{i,j}$$

.

B Matrix Transposition

One simple matrix operation is *transposition* you "transpose a matrix", "take the transpose of a matrix", *etc.* by flipping its columns and rows.

That new matrix is denoted A^T. (Some authors use A', but I personally avoid this.)

Thus, $[A^T]_{i,j} = A_{j,i}$.

Using the same matrix from the previous section,

$$B = A^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

C Block Matrices and *Tiling*

A *block matrix* or *partitioned matrix* is just a convenient way to write out linear algebra matrices to consider their construction/form, for proofs, and to ease notation for other operations. `https://en.wikipedia.org/wiki/Block_matrix`

This notation does not imply any difference in the matrix, **nor does it imply any difference in how it is stored in memory** (though in a linear algebra textbook, it might imply a difference in subscript notation, the subscripts referring to the blocks rather than the individual elements. More on this later.)

In general, the blocks don't even have to be the same size. However, blocks on one row or one column of blocks must have the same number of rows/columns, respectively.

Thus, we could form the matrices:

$W = \begin{bmatrix} a & b \\ d & e \end{bmatrix}$

$X = \begin{bmatrix} c \\ f \end{bmatrix}$

$Y = \begin{bmatrix} g & h \end{bmatrix}$
$Z = \begin{bmatrix} i \end{bmatrix}$
and say that

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} W & X \\ Y & Z \end{bmatrix}$$

Then

$$B = A^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} = \begin{bmatrix} W^T & Y^T \\ X^T & Z^T \end{bmatrix}$$

C.1 Tiling

In their simplest form, *tiling* algorithms find some way to decompose the matrix into equally sized, usually square, blocks. It is often faster to handle an entire tile at once than to proceed through the underlying 1D array linearly.

D Matrix Multiplication

One way to think of matrices is as being composed of *vectors*. A vector is a 1D list of numbers, which can also be thought of as a matrix with either a number of colums or number of rows equal to one.

D.1 Dot Product

The dot product of two vectors *of the same length* $\vec{x} = (x_1, x_2, x_3, ..., x_n)$ and $\vec{y} = (y_1, y_2, y_3, ..., y_n)$ is defined as

$$\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} x_i y_i$$

D.2 Matrix Multiplication

Matrix multiplication is defined as:

$$[AB]_{i,j} = Row_i(A) \cdot Col_j(B) = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

Thus, we can multiply any pair of matrices A and B so long as the number of columns of A is equal to the number of rows of B.

D.3 Block Matrix Multiplication

Recall that you could think of a block matrix as a matrix of matrices. In that mode of thinking, a row or column is a vector whose elements are matrices.

In that case, (so long as all the elements are shaped appropriately for multiplication) matrix multiplication still follows the above rules, except that each of the multiplications is a matrix multiplication, itself following the normal rules for matrix multiplication.

For example: $A = \begin{bmatrix} P & Q \\ R & S \end{bmatrix}$ $B = \begin{bmatrix} W & X \\ Y & Z \end{bmatrix}$

Where P, Q, R, S, W, X, Y, Z are all square submatrices each of the same shape.

Then

$$AB = \begin{bmatrix} PW + QY & PX + QZ \\ RW + SY & RX + SZ \end{bmatrix}$$

Note that there can be any number of rows or columns of submatrices, we have just chosen 2×2 for this example.

D.4 Tiled Matrix Multiplication

Tiled Matrix multiplication takes advantage of the cache properties of the machine by applying the block matrix multiplication algorithm to square blocks all of the same size, called *tiles*. By doing the underlying scalar multiplications in an order that makes temporal locality match spatial locality, this algorithm can be significantly faster than the naïve algorithm.