Xiaohui, Zhou
104-014-248

Lab Report for CS133 Lab1

mmul1:

a) **Method used:**
The first trick I found is instead of writing to C[][] in each iteration, we can have a local variable to store the value and then after iterations we can assign the value from the variable to c[][]. It will also eliminate the initial step.
A better trick was found by interchange j and k, which will significantly reduce the number of cache miss. However, this trick made the first one invalid and so only this trick is used.
The final and the best way I found is first reform the matrix B so that the value at B[i][j] will be stored at B2[j][i]. The reason why I did this is because then when do the multiplication, both value from A and B should be stored in the cache as they are both in the same rows instead of one column and one row.
I also tried different schedule options, and based on the results of running these options, I found guided schedule is the best option.

b) **performance result:**
Second Trick:
```
matrix size : 2048 x 2048 x 2048
Performance : 21.08 GFlop/s (37.1X)
Performance : 23.32 GFlop/s (44.3X)
Performance : 20.70 GFlop/s (30.1X)
Performance : 20.66 GFlop/s (33.9X)
Performance : 23.26 GFlop/s (41.0X)
```

Third Trick:
```
matrix size : 2048 x 2048 x 2048
Performance : 29.38 GFlop/s (47.6X)
Performance : 30.68 GFlop/s (44.8X)
Performance : 28.82 GFlop/s (42.8X)
Performance : 30.43 GFlop/s (45.6X)
Performance : 32.22 GFlop/s (50.1X)
```
The performance results I got have fluctuations and I believe that it is because of the processes from others students

c) **discussion:**
I believe that the way of dividing tasks i.e. scheduling in OpenMP, will affect the result but due to the constraints that I am not the only student who use the server, it is hard for me to get an accurate result.

mmul2:

a) **method used:**
I divided the whole matrix A and B into sub-matrixes with width 64. In every loop submatrix A2 and B2 will be copied and then, the results for multiplying A2 and B2 will be added into C.

b) **performance result:**
1024x1024 BLOCKSIZE 32
matrix size : 1024 x 1024 x 1024
Performance : 24.16 GFlop/s (36.3X)
Performance : 26.92 GFlop/s (39.5X)
Performance : 25.00 GFlop/s (38.1X)
Performance : 24.76 GFlop/s (49.8X)
Performance : 24.34 GFlop/s (35.9X)
2048x2048 BLOCK SIZE 64
matrix size : 2048 x 2048 x 2048
Performance : 31.47 GFlop/s (62.8X)
Performance : 31.11 GFlop/s (44.9X)
Performance : 31.85 GFlop/s (64.7X)
Performance : 30.22 GFlop/s (53.8X)
Performance : 31.09 GFlop/s (46.9X)
When copying the sub matrix, I did B2[i][j] = B[j][i] again to reduce cache miss, which increased the performance
When I changed the number of threads used, the result did not change much.
4096x4096 BLOCK SIZE 128
matrix size : 4096 x 4096 x 4096
Performance : 29.19 GFlop/s (0.0X)
Performance : 30.21 GFlop/s (0.0X)
Performance : 34.69 GFlop/s (0.0X)
Performance : 32.88 GFlop/s (0.0X)
Performance : 32.57 GFlop/s (0.0X)
For 0.0X: I only run the parallel part of the program in this scale because it will take a relatively very long time to finish the sequential one.

c) **discussion:**
The size of the sub-matrix should be chosen wisely for a better performance. I found that in general this block parallelization method has a similar performance with the third method I used in mmult1.

d) **challenges:**
What block size to choose