ECE 420 LAB H1 Kirby Banman
Report 1 1173895
15 January 2015

Description of Implementation

This experiment discusses the performance increase in matrix multiplication from multithreading parallelization.

The parallel multiplication program (see Appendix Source `matr.c`) uses input and output matrices are read from files “data_input” and “data_output”, respectively. The program takes two inputs - the number of threads P and the matrix size N. In order for the partition scheme to work, the program verifies that P evenly divides N^2 and that P is a perfect square.

The multiplication function, `partition_mult`, takes matrix data (pointers to input matrices and output matrix) and partition data (top left x,y coordinate and partition size). This means an entire NxN output matrix is calculated if coordinate 0,0 and partition size N are used. This design is so that partition sizes are easily changed, and so that the function can be tested for correctness before any multithreading is implemented.

Each thread calculates its partition coordinate from the rank, thread count, and matrix size. Each thread calls `partition_mult` (above), which reads from certain rows and columns of the global input matrices, and writes results to a global output matrix. Threads are `join`ed, then the result is written to disk by the main thread.

All programs are compiled and run as needed by a makefile (See Appendix Source `makefile`). Matrix size and thread count can be passed to the `make run` action. Experimental data is gathered using a script (See Appendix Source `experiment.sh`).

Testing and Verification

The provided matrix generation and I/O code is used to generate a set of static matrices, and a few special cases are written by hand. A set of static test matrices is used so that the tests are exactly repeatable. (See Appendix Test Matrices).

The provided “serialtester.c” is used as a testing oracle, and its results are compared to the parallel version for the set of static test matrices described above. In the makefile discussed in the section above a `make test_run` is equivalent to a normal run, except a correctness test using the testing oracle is run on the results immediately afterward. The `make test` action will run all predesigned test cases.

ECE 420 LAB H1 Kirby Banman
Report 1 1173895
15 January 2015

Performance Discussion

The controlled variables in this experiment are the number of threads P and the matrix size N. The response variable is computation time. Using the constraints on P and N (P evenly divides N^2 and P is a perfect square), the following set of problem sizes and matrix sizes are chosen for testing so that all matrix sizes can be tested with each thread count:

Thread Count P

- 1
- 4
- 9
- 16
- 25

Matrix Size N

- 60
- 120
- 180
- 240
- 300
- 480
- 960

As is evident in Appendix Chart 1, speedup consistently increases as the thread count increases to 16, after which speedup decreases for most problem sizes. For all thread counts, speedup varied between 2.19 and 3.86, and efficiency varied between 0.887 and 0.088 (see Appendix Tables 7 to 13). Despite the relatively large speedup from thread counts of 9 and 16, the highest efficiency was obtained at 4 threads - the lowest level of parallelization.

The most significant speedup occurred for larger problem sizes. Though even for the large matrices, thread counts higher than 16 had very diminishing returns. This is evident from the plateau in the speedup chart (Appendix Chart 1) after 9 threads. Specifically, the largest speedup was 3.86, occurring for matrix size N = 180 at P = 16 threads. However, the average speedup was highest and most consistent for the largest problem size:

Average speedup $S_{N=960} = (3.55 + 3.60 + 3.65 + 3.59) / 4 = 3.60$

If we assume that 3.60 is a good approximation of maximum speedup (S_{MAX}), then we can use Amdahl's law to estimate the fraction of the problem that was parallelized (F):

$$F = 1 - S_{MAX}^{-1}$$
$$F = 1 - (3.60)^{-1}$$
$$F = 0.72$$

This means that about 72% of the matrix multiplication problem was parallelized, and 28% remained as serial overhead. This overhead may result from many causes, but there are two primary candidates:

- Computation necessary to split the problem. i.e. the calculation of coordinates partition coordinates for each thread.
- Thread creation overhead from the Pthreads library

- Memory cache misses. i.e. for the fully serial program, array reads and writes are in a strict pattern, which may take advantage of reference locality within the cache(s). However, the parallel program results in a much more scattered read/write pattern for the arrays, which is less likely to take advantage of cache locality.

A notable anomaly occurred for problem size N = 120. The speedup for that problem size dropped significantly from 16 threads to 25 threads (See the red line in Appendix Chart 1). Other problem sizes saw either a minor increase or a minor decrease between those thread counts. Random chance or competition with other processes are unlikely causes, since the trend is fairly consistent across all 5 experimental runs.

Conclusion and Experiences

With regards to performance, as is evident from the speedup and efficiency charts (see Appendix Chart 1 and Chart 2), for most problem sizes:

- increasing from 9 to 16 threads results in the largest increase in speedup
- increasing from 4 to 9 threads results in the largest decrease in efficiency

Using Amdahl's Law for analysis, the estimated maximum speedup of 3.60 translates to a parallelized fraction of 72%. Several sources of the remaining 28% were identified, including calculation, IO, and OS overhead.

With regards to experimental quality, the constraints on thread count and problem size decreased the detail of results when comparing performance gains to thread count. It would have been ideal to use thread counts of 1, 2, 3, 4, … instead of 1, 4, 9, … . Also, too many problem sizes with too little range were used. It would be better to choose fewer problem sizes, starting with small matrices and jump up by orders of magnitude.

With regards to software quality, in each thread function, the partition coordinates are calculated from thread rank. This was done in order to minimize the amount of data copied to the local state of each thread. In retrospect, however, it is not clear that this is more performant than simply copying the data.

Getting each thread to calculate its partition coordinate was surprisingly difficult. Several iterations worked for some cases, but not for all. Difficult edge cases included:

- When the number of array elements was equal to the number of threads.
- A single thread.
- Arrays of size 1 and 2.

Appendix

Chart 1 - Speedup for matrices of size 60, 120, 180, 240, 300, 480, 960

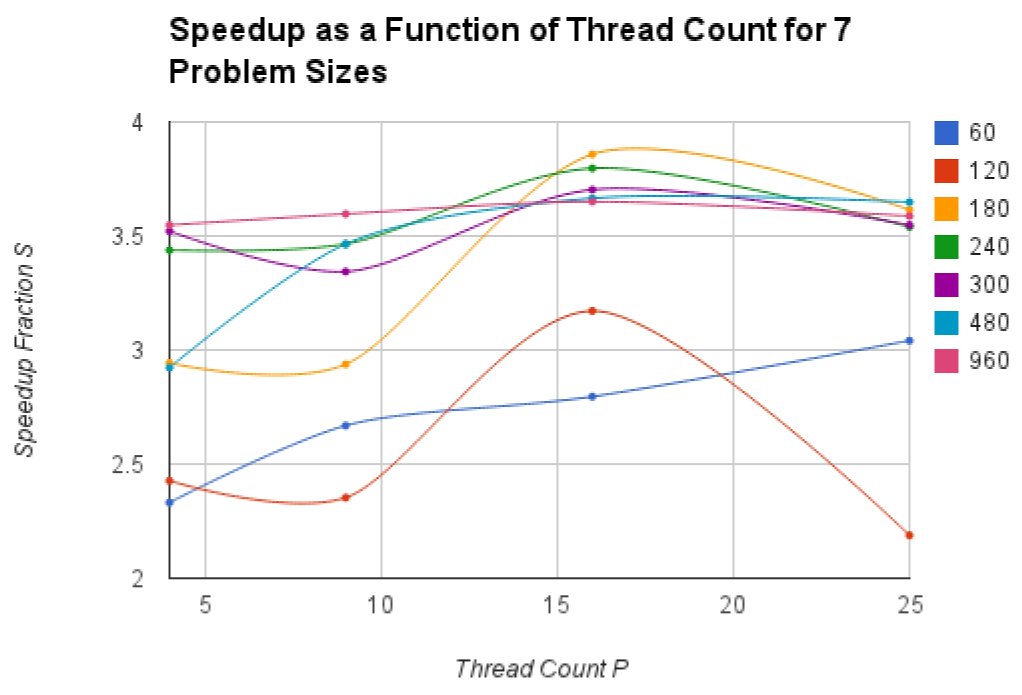
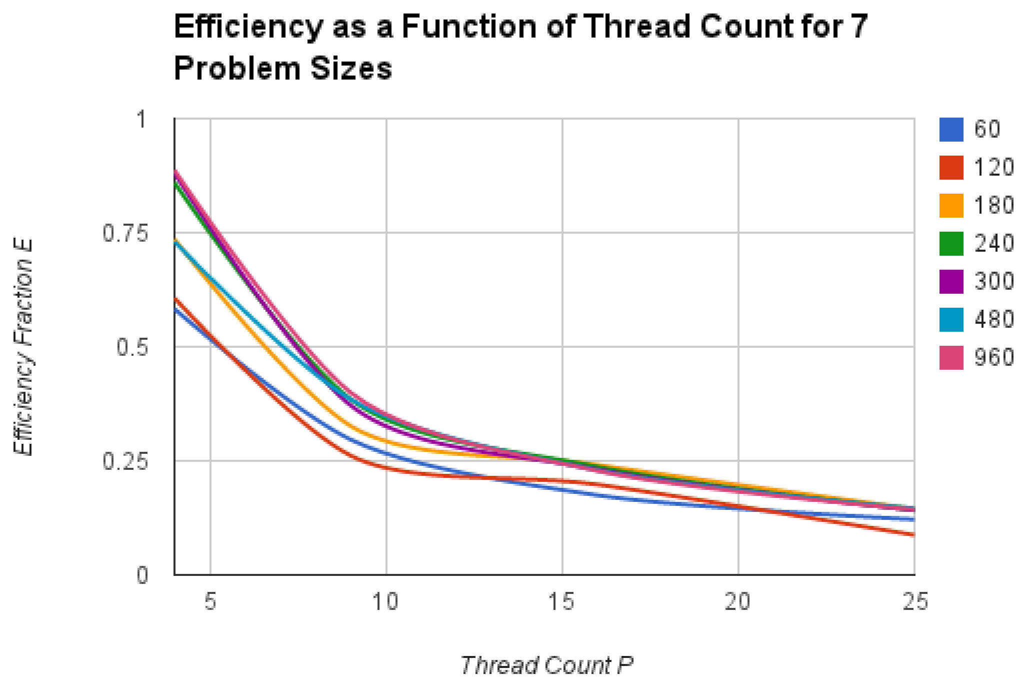


Chart 2 - Efficiency for matrices of size 60, 120, 180, 240, 300, 480, 960



ECE 420 LAB H1
Report 1
15 January 2015

Kirby Banman
1173895

Test Matrices 1 - All zeros

```
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0

0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0
```

Test Matrices 2 - All ones

```
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1

1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
```

Test Matrices 3 - Both Identity

```
1    0    0    0    0    0    0    0
0    1    0    0    0    0    0    0
0    0    1    0    0    0    0    0
0    0    0    1    0    0    0    0
0    0    0    0    1    0    0    0
0    0    0    0    0    1    0    0
0    0    0    0    0    0    1    0
0    0    0    0    0    0    0    1

1    0    0    0    0    0    0    0
0    1    0    0    0    0    0    0
0    0    1    0    0    0    0    0
0    0    0    1    0    0    0    0
0    0    0    0    1    0    0    0
0    0    0    0    0    1    0    0
0    0    0    0    0    0    1    0
0    0    0    0    0    0    0    1
```

Test Matrices 4 - Single Identity

```
1    22    -32    9
33    -98    1    8
0    0    33    7
1    32    83    342

1    0    0    0
0    1    0    0
0    0    1    0
0    0    0    1
```