Assignment 2: Multithreading and Solving System of Linear Equations Algorithms

Tai Duc Nguyen
ECEC 622: Parallel Computer Architecture

May 6, 2020

Abstract

Knowledge from the previous assignment about multithreading's quirks and benefits is leveraged to develop a common algorithm used in solving system of linear equations called Iterative Jacobian Solver.

1 Experimental Setup

The Jacobi method is an iterative algorithm for determining the solutions of a strictly diagonally dominant system of linear equations ($A\boldsymbol{x} = B$). Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges. This algorithm is a stripped-down version of the Jacobi transformation method of matrix diagonalization.

The multithreaded version of this algorithm is implemented as follows:

```
Create a copy of matrix X called new_X
Initialize barrier sync

Create threads
Reserve memory for partial SSDs (sum of the squared differences)
Allocate memory on heap for threads' data

Start all threads. For each thread:
  while not DONE:
    Each thread is a assigned a row i;
      The vector dot product of A[i,:] with X is computed and stored in variable sum;
      new_X[i] = (B[i] - sum)/A[i,i];

      tmp_ssd = X[i] - new_X[i];
      prt_ssd += tmp_ssd * tmp_ssd;

      i += num_threads (striding method)
    partial_ssd[thread_id] = prt_ssd

    BARRIER SYNC

    if thread_id is 0:
      sum up all partial ssds;
      if sqrt(sum_ssds) < TOLERANCE:
```

```
          DONE = 1;

      BARRIER SYNC

      if not DONE:
        Swap X and new_X;

  Stop all threads
```

Note: running `make all` *will generate the report in* `rpt.txt`

2 Experimental Result

Matrix size (MxM)	Num Threads	Serial	Multithreading
512	4	1.779038	0.31103
512	8	1.77211	0.252212
512	16	1.799968	0.496907
512	32	1.818823	0.814503
1024	4	15.179658	2.184636
1024	8	15.00855	1.383655
1024	16	15.422613	1.875908
1024	32	14.882152	1.93245
2048	4	123.843689	7.284922882
2048	8	124.019871	6.20099355
2048	16	123.871231	6.19356155
2048	32	123.841112	7.3123149

Figure 1: Results of execution times in seconds of the serial vs the multi-threaded version using the pthread interface

3 Discussion

From Figure 1, it is apparent that the larger the problem size, multithreading provides increasingly significant improvements ($\approx$ 6x for 512, $\approx$ 10x for 1024 and $\approx$ 17x for 2048). This result can be improved with better memory management, better memory access pattern, and, better initial approximation of X (this can vastly reduce the number of iterations needed to reach the tolerance limit). From the data, it is also shown that 8 threads provide the best amount of speed up. Having more than 8 threads will increase the amount of overhead and make the program runs slower.