Aleksandar Aleksandrov

ECEC 413

Gaussian Elimination Report

Multithreaded Implementation

The goal of this assignment is to parallelize the serial code for the gaussian elimination algorithm. The original algorithm consists of two steps:

1. Division step – divide all the elements of the current row by its pivot element
2. Elimination step – multiply and subtract all the following rows so that all elements under the pivot become zero

To parallelize this algorithm the two steps, have to be parallelized independently. Therefore, the worker function for the threads has one outer for loop that goes through all the rows in the matrix and then inside has two different algorithms that are parallelized on every iteration.

Thread Data Structure:

```
typedef struct thread_data_s {
    int tid;
    int num_threads;
    int offset;
    int chunk_size;
    Matrix *A; //Pointer to the source matrix
} thread_data_t;
```

For this assignment the thread data structure is simpler. It only consists of the thread id, the number of threads, the offset, chunk size and a pointer to the original matrix. All the changes are directly made to the source matrix. For barriers semaphores are used similarly to the Jacobi pthread project.

The Worker Function

The worker function starts off by casting the thread data and declaring the variables used in the loops:

```
    thread_data_t *thread_data = (thread_data_t *)args;
    int i,j,k;
```

Then it moves to the outer for loop which goes over all the rows in the matrix:

```
for(k = 0; k<thread_data->A->num_rows; k++)
```

At the beginning of the loop a barrier is placed to make sure that all the threads are done with the previous elimination step before moving into the division step of the next row. The division step is simple, the current row is chunked up between the threads and every thread divides the elements within its chunk with the pivot element. The pivot element index is always:

(Number of columns*k +k), where k is the current row index

The pivot element is not divided within the loop because all threads use it to divide their own element. An if statement is placed in the thread logic to avoid dividing the pivot. After the row is divided, a barrier is placed to wait for all the threads to complete their division step and thread 0 sets the pivot element to 1.

Now the function moves to the elimination step of the Gaussian Algorithm. This time instead of chunking elements in a specific row, all the rows **after** the divided row are chunked up between the threads. Therefore, the for-loop boundaries are as follows:

```
for (i = (thread_data->offset + k + 1 ); i <
(thread_data->chunk_size+thread_data->offset + k + 1); i++)
```

By setting the starting point of the loop to be the thread's offset plus K plus 1, it is assured that the threads never perform elimination on the row which was divided in the previous step.

$$A' = \begin{bmatrix} 1 & \frac{5}{3} & \frac{2}{3} \\ 2 & 3 & 1 \\ 1 & 2 & 2 \end{bmatrix}$$

Figure 2 - division step for k = 0

$$A' = \begin{bmatrix} 1 & \frac{5}{3} & \frac{2}{3} \\ 0 & -\frac{1}{3} & -\frac{1}{3} \\ 0 & \frac{1}{3} & \frac{4}{3} \end{bmatrix}$$

Figure 1 - elimination step for k = 0

Since with this algorithm with every new iteration the threads have less items to go through, an if statement has to be added to make sure that the threads do not attempt to index outside of the matrix bounds:

```
if((thread_data->A->num_columns * i + j) <
thread_data->A->num_rows*thread_data->A->num_rows)
```

After the elimination step is completed over the specific chunk, all the elements under the pivot within that chunk are set to 0.

The algorithm then repeats.

Performance

Grid Dimension	Single Thread	4 threads	8 threads	16 threads	32 threads
512x512	0.04	0.04	0.05	0.06	0.09
1024x1024	0.31	0.27	0.26	0.22	0.25
2048x2048	2.55	2.03	1.67	1.14	1.28
4096x4096	21.54	17.41	10.25	8.61	7.72

For smaller grid dimensions the multithreaded algorithm performs worse than the single threaded algorithm. This is due to the larger overhead brough by using multiple threads. However, for grid sizes 2048x2048 and 4096x4096 the multithreaded implementation starts to give better performance improvements. With 32 threads for the 2048 sized grid the algorithm performs almost twice as fast. For a grid dimension of 4096 even with 8 threads the algorithm is already twice faster and with 32 threads is 3 times faster. For the larger grids increasing the number of threads increases the performance significantly, however for smaller grids the larger overhead created by more threads leads to worse performance.