Samuel Nyberg and Tobias Mattsson

Multiprocessor Programming DV2606

Parallel Gaussian Elimination (Task 10)

For this task we started from the template file *gaussianseq.c*

After refreshing our knowledge of how the gaussian elimination algorithm works we assumed that the elimination step of the algorithm would be the most suitable to parallelize. We moved the elimination step into a separate function, which made it possible to create new threads to call the function in parallel. We tried to balance the work between the threads by splitting the columns that would get eliminated into blocks or chunks for the threads to work on separately. We defined the chunks by the start and end columns for that specific chunk. These values got passed on as thread arguments into the elimination function. This ensures that the threads work isolated from each other and no mutex locks are required.

We tried to parallelize the division step, but quickly found saw that that resulted in too much overhead.

One thing that could have been improved would be to divide the work by cyclic partitioning instead of block partitioning, this would make sure that all threads are involved more equally.

Below is the measurements of speedup between the different test runs. As displayed, the Parallel version with 16 cores achieved a speedup of 1.62 over the original serialized code. Also, it had an speedup of 2.37 over the parallel code with only one core.

```
real    0m2.155s
user    0m13.388s
sys     0m1.224s
```

(Parallel with 16 cores)

```
real    0m5.099s
user    0m4.918s
sys     0m0.153s
```

(Parallel code with only 1 core)

```
real    0m3.486s
user    0m13.488s
sys     0m1.324s
```

(Serialized/original code)

Parrallel Quicksort implementation (Task 11)

For this task we used the templated from *qsortseq.c*

Quicksort is a sorting algorithm that works by splitting and partitioning parts to sort into a binary tree. A simple solution to parallelize the algorithm would be to create a new thread for each partitioning allowing until all threads are working, killing it when it is done and then spawn a new thread on a new partition. This solution would be simple but would create too much overhead when creating and destroying threads so often. Our solution to this problem was by implementing a thread pool. This group of always running threads would allow us to only create and join threads once, removing much of the overhead. The parallelization works by a thread starting in the top of the quicksort algorithm binary tree, if it finds a partition to sort on the left branch it adds this as a task to a task que. Then the thread continues down, sorting the right

branch. If it once again, on a lower depth finds another partition that goes to the left of the tree, it submits this task to the que and continues. Once a thread has reached the bottom of the branch it's currently working on, it leaves this be, and simply picks up another task from the que. The parallelization utilizes all available threads including the main thread until the que of tasks in empty, then it simply joins all threads and frees up memory to reduce the risk of potential faults. To further improve the algorithm and reduce overhead we decided to not que a task if it was below 50000 numbers in size, this was found through trial and error testing and gave a threshold from where the algorithm would run sequentially since this would be faster.

Measurements of the algorithm were taken according to instructions and ran on the jane lab computer. Running the algorithm sequentially returned a baseline speed of 10.318 seconds. Following running the algorithm parallel with a thread pool of 4 threads ran in 4.867 seconds resulting in a speedup of 2.11x. Further, running the parallel algorithm on 8 threads resulted in a time of 3.783 seconds and a speedup of 2.73x. Running on 16 threads resulted in an execution time of 3.070 seconds and a speedup of 3.36X. Lastly, running 20 threads resulted in a execution time of 2.345 seconds and a speedup of 4.4x.

The execution time of running the quicksort sequentially:

```
real    0m10.318s
user    0m10.230s
sys     0m0.084s
```

Running the algorithm parallel on 4 threads:

```
Running on 4 threads, min size: 50000
real    0m4.867s
user    0m14.565s
sys     0m1.782s
```

Running the algorithm parallel on 8 threads:

```
Running on 8 threads, min size: 50000
real    0m3.783s
user    0m18.986s
sys     0m3.972s
```

Running the algorithm parallel on 16 threads:

```
Running on 16 threads, min size: 50000
real    0m3.070s
user    0m20.812s
sys     0m12.454s
```

Running the algorithm parallel on 20 threads:

```
Running on 20 threads, min size: 50000
real    0m2.345s
user    0m9.561s
sys     0m11.664s
```