CS546 “Parallel and Distributed Processing”

Homework 2
First name: Guillermo
Last Name: de la Puente
CWID: A20314328
Due date: 3/3/2014

Also included:
- gauss_pthread.c
- gauss_openmp.c

The provided algorithm in the file *gauss.c* has three loops, the outer two iterate through the rows and the inner one through columns. That's why the algorithm has a $O(n^3)$ complexity. To solve that, this is the solution proposed:

The outer loop presents loop-carried dependency. Every iteration requires the previous one to have finished. However, the two inner loops are good to parallelize because there's no dependency between iterations.

The middle loop calculates the multiplying factor for every row. Inside it, the third loop applies the multiplying factor to every element in the row. Any of these loops could be parallelized, but experiments have shown that it's more effective to parallelize the middle loop, the one that calculates the multiplying factor.

By the parallelization, for every iteration of the outer loop, the set of rows that are going to be processed is divided among threads. Every thread will process a subset of rows. When all are done, the next iteration of the outer loop comes. The new set of rows, which is one row smaller in size, is divided between the threads again.

Thanks to the barrier placed at the end of every iteration of the outer loop, the dependency problems are avoided. This barrier becomes the new bottleneck for the system.

Two programs have been developed with this parallel algorithm, gauss_pthread.c and gauss_openmp.c. Both of them use the same algorithm and their execution times are very similar.

These are some time measures taken. The values are approximated from multiple samples. The program has been run in an Intel Core i7 with 2.10 GHz each processor, inside a virtual machine that uses up to 4 processors.

N	Sequential	Parallel Pthread	Parallel OpenMP
500	180 ms	150 ms	80 ms
1000	1350 ms	600 ms	425 ms
2000	10,600 ms	3,500 ms	3000 ms
4000	84,000 ms	31,000 ms	30,000 ms

OpenMP provides better times than manual Pthread handling. OpenMP is using a guided distribution of workload in the parallelization, while Pthread is using static division of work, with every processor having to handle the same load. It doesn't matter much because time increases at the same rate that with the Pthread program because that has to do with the algorithm, not with the language specifics.

In this case, **the speedup is equal to 3** (4 processors).
As seen from the previous table, this speedup is approximately constant for N = 1000, N = 2000 and N = 4000. This means that **the algorithm is scalable**.

During the development of this algorithm, some alternatives have been tested. Parallelizing the inner loop (that applies the multiplying factor in every columns) proved not to be a better choice.

To test how spatial locality affects in rows but not in columns, the algorithm was tested but inverting rows for columns and vice versa. As expected, it took a lot more time for N big enough so that not all the array can be stored in cache.

In the Pthreads problem, the method for dividing the rows between threads was studied. When the number of rows divided by the number of threads isn't exact, there are multiple possibilities. The simplest one is to truncate the decimal result.
A better way is to round the result because the distribution of work is more uniform. That's how it's been finally implemented, although avoiding operating with floats or using the round() function.

```
numRows = (N - norm - 1);
from = ( 10*norm + ( ( threadid     * numRows) *10 / numWorkers ) + 5)/10;
to   = ( 10*norm + ( ( (threadid+1) * numRows) *10 / numWorkers ) + 5)/10;
```

In gauss_pthread.c is explained with more detail with comments.