Parallelism

Lab 3

Gerard Bayego
Martín Dans

Grup: par3101
Curs: 2014-15
Fall semester

Question 5.1.1:

Tareador API:

Jacobi:

In the file solver-tareador.c we added, in the function relax_jacobi, inside the inner for:

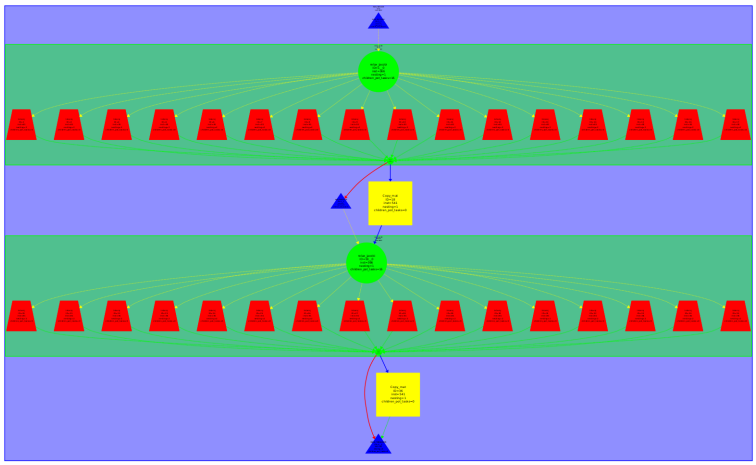
```
tareador_start_task("Internj");
utmp[i*sizey+j]= 0.25 * ( u[ i*sizey     + (j-1) ]+  // left
          u[ i*sizey     + (j+1) ]+  // right
          u[ (i-1)*sizey + j     ]+  // top
          u[ (i+1)*sizey + j     ]); // bottom
diff = utmp[i*sizey+j] - u[i*sizey + j];
tareador_disable_object(&sum);
sum += diff * diff;
          tareador_enable_object(&sum);
tareador_end_task("Internj");
```

Gauss-Seidel:

In the file solver-tareador.c we added, in the function relax_gauss, inside the inner for:

```
tareador_start_task("Interng");
unew= 0.25 * ( u[ i*sizey        + (j-1) ]+  // left
           u[ i*sizey        + (j+1) ]+  // right
          u[ (i-1)*sizey    + j     ]+  // top
           u[ (i+1)*sizey  + j     ]); // bottom
diff = unew - u[i*sizey+ j];
tareador_disable_object(&sum);
sum += diff * diff;
u[i*sizey+j]=unew;
tareador_end_task("Interng");
```

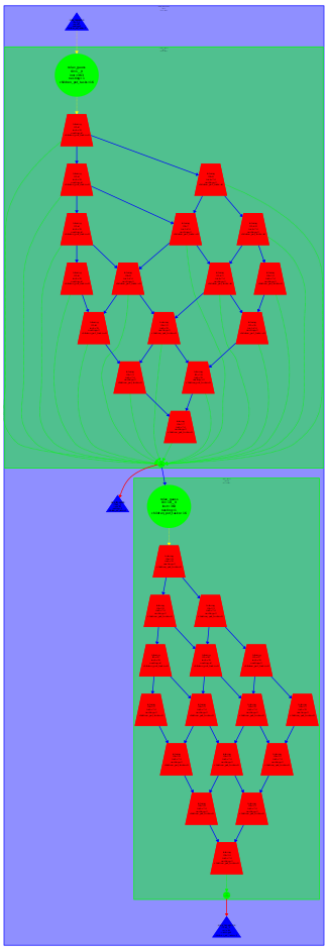
Jacobi:



The problem is that the variable sum where they store the partial summations is shared. This variable is accessed by every thread and that makes to tareador think that it is a dependency. Of course that's not our case since we only want to sum and the order doesn't matter.
We have to notice that even if it's parallelizable reading the variable and writing on it could produce a race condition between threads. To prevent that we can make that region critical, reduction(+:sum) or do partial summations for every thread and add all of them at the end.

Also copy_mat can be paralyzed, with a simple pragma omp for collapse 2, because didn't have any dependence.

Gauss-Seidel:



In the gauss case, each position depends of the element upside and left side, and this generates a lot of dependences.

When we parallelize with blocks, first we can compute block 0-0, after, we can compute 0-1 and 1-0 blocks. This parallelization creates a tree of dependences, that first increase the number of tasks to do, and after decrease to the end.

Question 5.2.1:

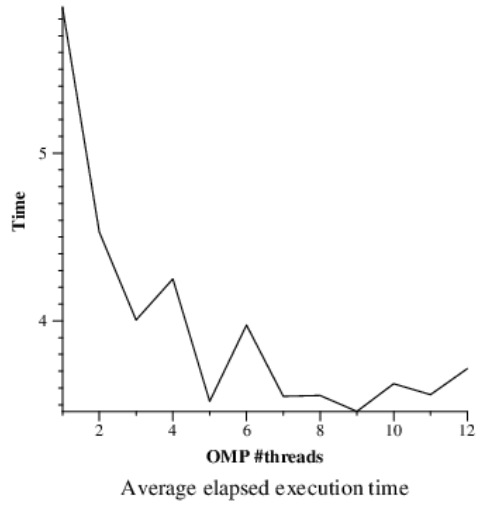
```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey) {
	double diff, sum=0.0;
	int howmany=omp_get_max_threads();

	#pragma omp parallel reduction(+:sum)
	{
	for (int blockid = 0; blockid < howmany; ++blockid) {
		int i_start = lowerb(blockid, howmany, sizex);
		int i_end = upperb(blockid, howmany, sizex);
		for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
			#pragma omp for nowait private(diff)
			for (int j=1; j<= sizey-2; j++) {
				utmp[i*sizey+j]= 0.25 * ( u[ i*sizey     + (j-1) ]+  // left
					u[ i*sizey     + (j+1) ]+  // right
					u[ (i-1)*sizey + j     ]+  // top
					u[ (i+1)*sizey + j     ]); // bottom
				diff = utmp[i*sizey+j] - u[i*sizey + j];
				sum += diff * diff;
			}
		}
	}
	}

	return sum;
}
```

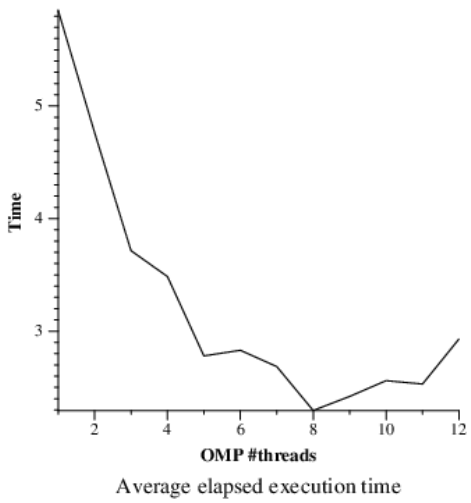
To parallelize jacobi, we added a omp parallel pragma with a reduction of sum variable, to avoid the dependence detected in the tareador, also we added a pragma omp for to create tasks of work of the most inner for of the function.

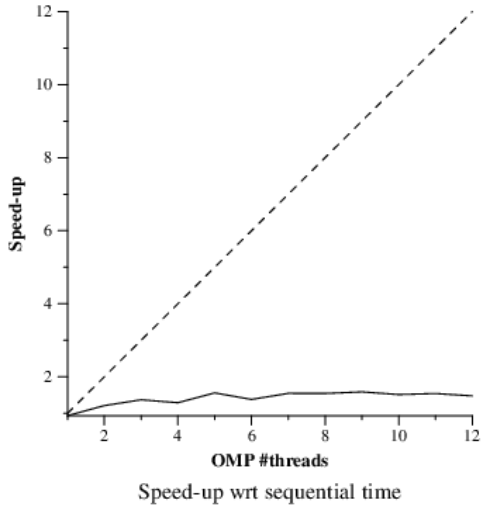
```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey) {
	#pragma omp parallel for collapse(2)
	for (int i=1; i<= sizex-2; i++){
		for (int j=1; j<= sizey-2; j++){
			v[ i*sizey+j ] = u[ i*sizey+j ];
		}
	}
}
```

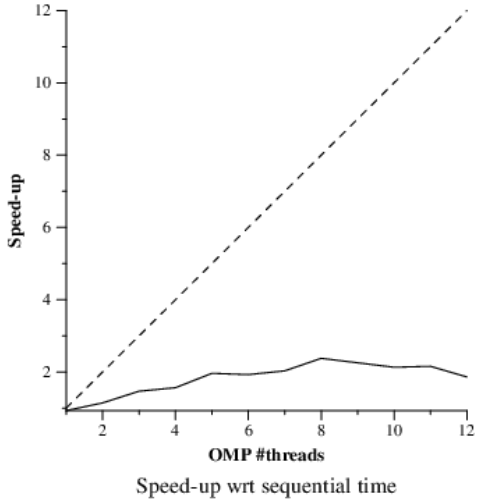
We also parallelized the copy mat with a simple omp for because no dependences exists, with collapse(2) to generate the max number of tasks possible and increase parallelism.

Question 5.2.2:







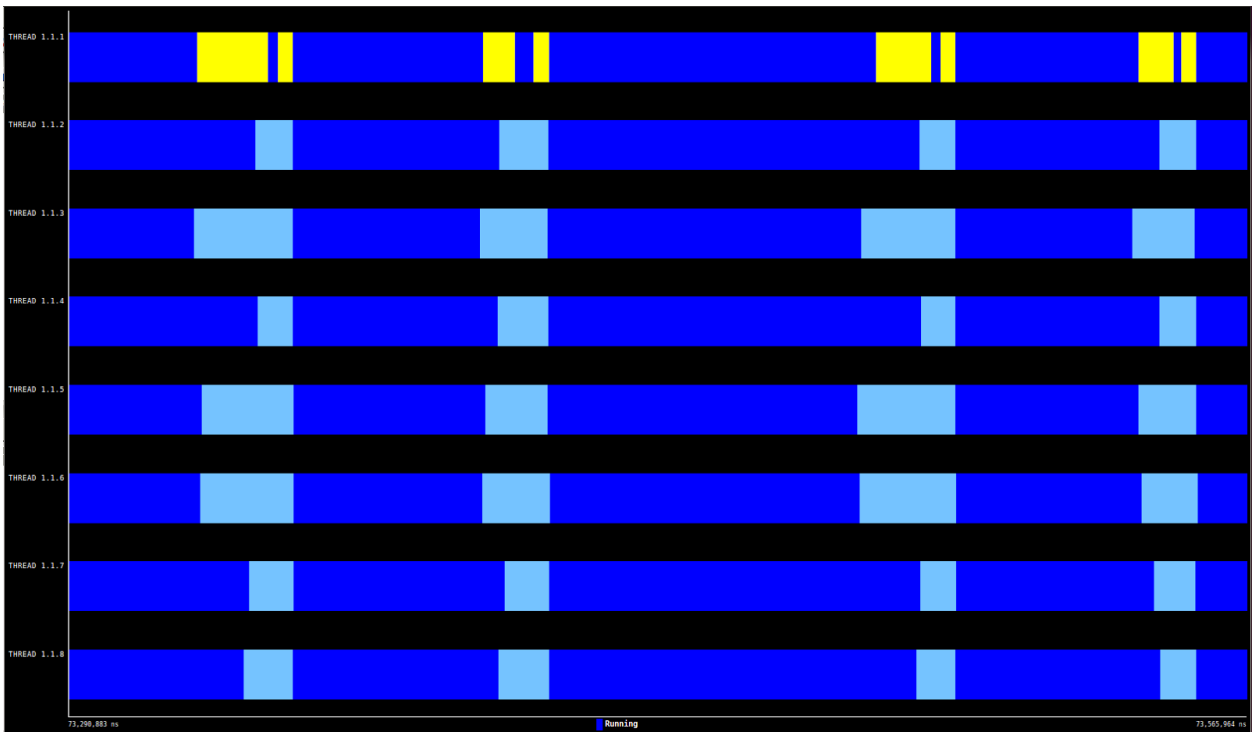


Plot before copy-mat parallelization

Plot after copy-mat parallelization

At left we have the speedup with only de relax function parallelized. We can see that the speedup obtained it's about 1.3. At right we can see the speed-up achieving 2 with 8 threads and then it starts decaying because the winnable time with the parallelization decays and the overhead increase.

With the copy-mat parallelization, we increase the speedup, because is fully parallelizable and needs small synchronization between tasks.



	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	1,956,461,685 ns	7,582,648 ns	550,872,804 ns	19,392,822 ns	1,925 ns
THREAD 1.1.2	1,992,848,077 ns	7,582,856 ns	-	4,478,046 ns	-
THREAD 1.1.3	1,911,595,600 ns	7,583,201 ns	-	3,976,916 ns	-
THREAD 1.1.4	1,996,906,513 ns	7,583,551 ns	-	4,487,374 ns	-
THREAD 1.1.5	1,873,689,144 ns	7,583,883 ns	-	3,945,966 ns	-
THREAD 1.1.6	1,959,650,452 ns	7,584,126 ns	-	4,505,341 ns	-
THREAD 1.1.7	1,810,722,263 ns	7,584,436 ns	-	4,098,135 ns	-
THREAD 1.1.8	1,898,682,981 ns	7,584,811 ns	-	4,490,576 ns	-
Total	15,400,556,715 ns	60,669,512 ns	550,872,804 ns	49,375,176 ns	1,925 ns
Average	1,925,069,589.38 ns	7,583,689 ns	550,872,804 ns	6,171,897 ns	1,925 ns
Maximum	1,996,906,513 ns	7,584,811 ns	550,872,804 ns	19,392,822 ns	1,925 ns
Minimum	1,810,722,263 ns	7,582,648 ns	550,872,804 ns	3,945,966 ns	1,925 ns
StDev	59,716,862.58 ns	714.17 ns	0 ns	5,002,212.70 ns	0 ns
Avg/Max	0.96	1.00	1	0.32	1

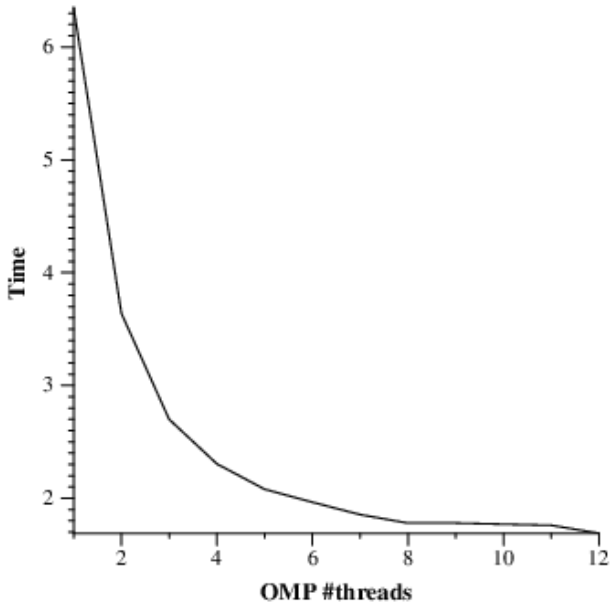
As we can see in the two captures of paraver, we have a lot of overhead caused for the synchronization and creation of tasks, that reduces the total speedup of the program.

Question 5.3.1:

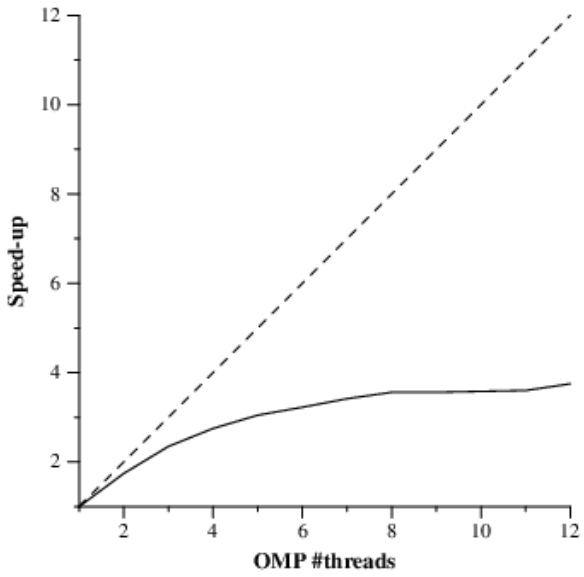
```
double relax_gauss (double *u, unsigned sizex, unsigned sizey) {
   double unew, diff, sum = 0.0;
   int howmany = omp_get_max_threads();
   int numBlocs = 8;
   int blocsProcesats[howmany];
   for(int i = 0; i < howmany; ++i) blocsProcesats[i] = 0;
   #pragma omp parallel for schedule(static) private(diff,unew) reduction(+: sum)
   for(int i = 0; i < howmany; i++){
        int ii_start = lowerb(i,howmany,sizex);
        int ii_end = upperb(i,howmany,sizex);
        for (int j = 0; j < numBlocs; j++){
             int jj_start = lowerb(j,numBlocs,sizey);
             int jj_end = upperb(j,numBlocs,sizey);
             if(i > 0){
               while(blocsProcesats[i-1]<=j){
                  #pragma omp flush
               }
             }
             for(int ii = max(1,ii_start); ii<= min(sizex-2, ii_end); ii++){
                for(int jj= max(1,jj_start); jj<= min(sizey-2, jj_end); jj++){
                   unew = 0.25* (u[ii * sizey + (jj-1)] +  // left
                            u[ii * sizey + (jj+1)] +  // right
                            u[(ii-1) * sizey + jj] +  // top
                            u[(ii+1) * sizey + jj]); // bottom
                   diff = unew - u[ii * sizey + jj];
                   sum+= diff*diff;
                   u[ii*sizey + jj]  = unew;
                }
             }
             ++blocsProcesats[i];
             #pragma omp flush
        }
   }

   return sum;
}
```

For this algorithm, the parallelization strategy is more complex than Jacobi, caused for the complex dependences.
We have to divide the matrix to give equally rows to all threads, after we are going to divide the matrix space of each thread in blocks. With this decomposition we have secured the dependence left to right, but we have to secure also the up dependence. To do this we created the blocsProcesats vector of the size of num_threads. Here with a pooling strategy of pooling we wait the dependences needed, doing flush.

Question 5.3.2:

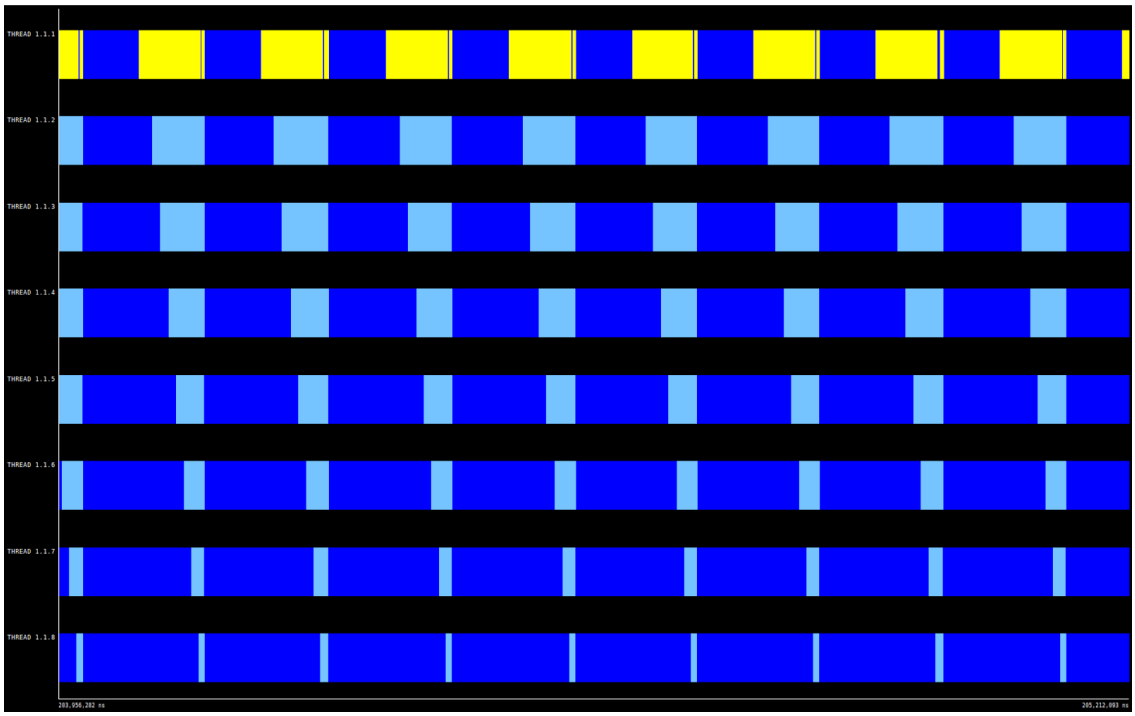


Average elapsed execution time



Speed-up wrt sequential time

As we can see gauss has a better speedup compared to jacobby, but is not a big increase. The increase is produced because is more efficient the gauss version, because we didn't need to copy matrix, and we save in the overhead of the program. But has more computation the gauss version.



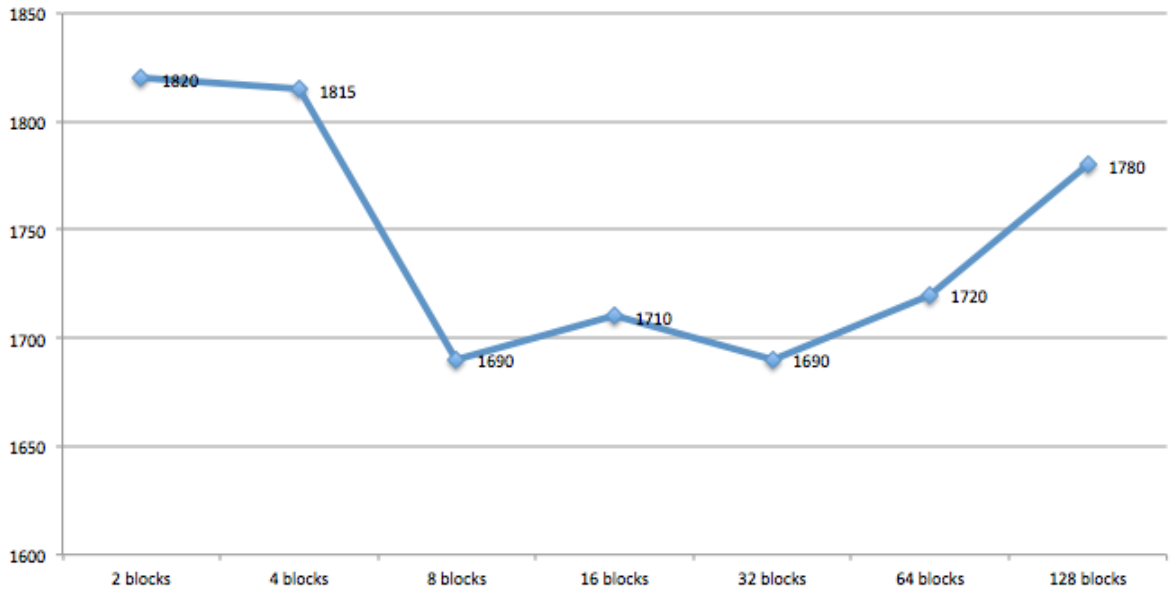
	Running	Not created	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	908,212,301 ns	8,930,145 ns	939,765,143 ns	6,574,724 ns	1,950 ns
THREAD 1.1.2	1,021,196,026 ns	8,930,473 ns	-	1,770,386 ns	-
THREAD 1.1.3	1,134,696,933 ns	8,931,105 ns	-	1,621,126 ns	-
THREAD 1.1.4	1,256,848,206 ns	8,931,840 ns	-	1,728,531 ns	-
THREAD 1.1.5	1,369,870,672 ns	8,932,583 ns	-	1,593,603 ns	-
THREAD 1.1.6	1,482,877,669 ns	8,933,343 ns	-	1,619,535 ns	-
THREAD 1.1.7	1,598,620,873 ns	8,934,018 ns	-	1,566,429 ns	-
THREAD 1.1.8	1,697,615,432 ns	8,934,688 ns	-	1,603,774 ns	-
Total	10,469,938,112 ns	71,458,195 ns	939,765,143 ns	18,078,108 ns	1,950 ns
Average	1,308,742,264 ns	8,932,274.38 ns	939,765,143 ns	2,259,763.50 ns	1,950 ns
Maximum	1,697,615,432 ns	8,934,688 ns	939,765,143 ns	6,574,724 ns	1,950 ns
Minimum	908,212,301 ns	8,930,145 ns	939,765,143 ns	1,566,429 ns	1,950 ns
StDev	261,104,296.94 ns	1,558.78 ns	0 ns	1,632,220.02 ns	0 ns
Avg/Max	0.77	1.00	1	0.34	1

As we can see if we compare with the gauss profile, the synchronization time is less, and the working time is a little bit larger, and with this two things we can justify the small speedup.

Question 5.3.2:

In the gauss algorithm is so important to find the point with the optimum ratio between computation and synchronization. To do this we fixed the executions with 8 threads and we do various executions with different parameters.

We decided to do executions with 2, 4, 8, 16, 32, 64, 128 blocks, to see the effect of computation and synchronization (blue line is the execution time).



With the results we can see that the optimal value is between 8 and 32 blocks, because has the smaller execution times. With small values there are a lot of computation per thread and with large values we have a lot of synchronization time.