UNIVERSITY OF HELSINKI
DEPARTMENT OF PHYSICS

TOOLS FOR HIGH PERFORMANCE COMPUTING

Final Project Report
Poisson Equation

Student: Caike Crepaldi

Professor: Antti Kuronen

December 2015

Contents

1 Introduction

In this problem we have the Poisson equation in two dimensions:

$$\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right) f(x,y) = g(x,y),\ (x,y) \in [0,1]^2 \tag{1}$$

We know the function g(x,y) and want to know the function f(x,y).

We have that the problem is discretized as such:

$$f_{i,j} = f\left(\frac{1}{N}, \frac{j}{N}\right),\ 0 \leq i,\ j \leq N \tag{2}$$

$$g_{i,j} = g\left(\frac{1}{N}, \frac{j}{N}\right),\ 0 \leq i,\ j \leq N \tag{3}$$

Our goal here is to implement a parallel program that solves the Poisson equation using the Successive Over Relaxation (SOR) method.

2 Algorithms

In this program we use the Successive Over Relaxation (SOR) method in order to solve the Poisson equation.

The SOR method is desribed below:

$$f_{i,j}(t+1) = (1-\gamma) f_{i,j}(t) + \frac{\gamma}{4}[f_{i+1,j}(t) + f_{i-1,j}(t+1) + f_{i,j+1}(t) + f_{i,j-1}(t+1)] - \frac{\gamma}{4N^2} g_{i,j} \tag{4}$$

The t is only an iteration index.

According to the assignment, the SOR means that the last updated value for the cross-neighboors of an element of the matrix is used to update this element. Therefore, perhaps the true form of the algorithm is a little different than what is shown above since, using this description, all iteration indeces should be (t+1) when talking about the cross-neighboors elements inside the square brackets in the equation. In order to use this method in a parallel code, we shall use the red-black algorithm.

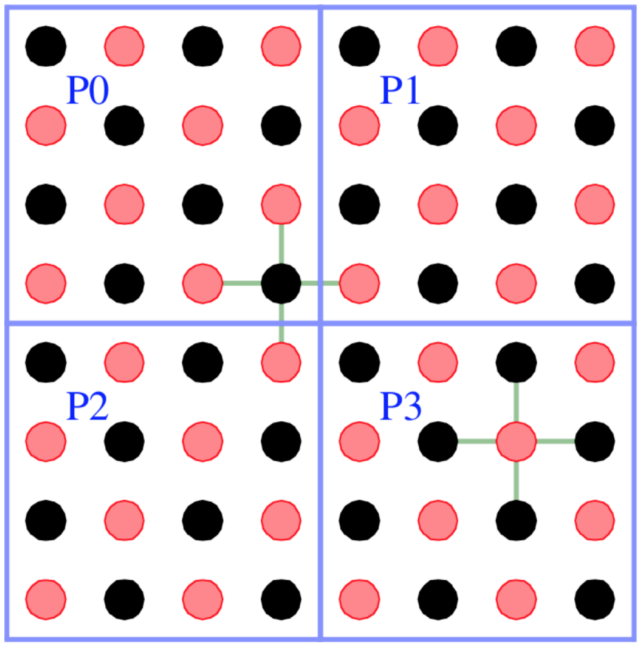

Figure 1: The black and red elements.

To use the red-black algorithm we work with two sub-lattice, one with red elements and another with black elements. A red element can be defined (using fortran language) as an element which:

$$mod(i+j,2) /= 0$$

In the same way, a black element can be defined (using fortran language) as an element which:

$$mod(i+j,2) == 0$$

To use the red-black algorithm, we must follow the steps below:

1. We update the red elements for the first time using guess values.
2. We then procede to update the black elements using the SOR method (note that the cross-neighboors of a black element are red elements and their values had already been updated in the previou step).
3. Next, we update the red elements using the SOR method (note that the cross-neighboors of a red element are black elements and their values had already been updated in the previou step).
4. We check if the method has converged. If it hasn't, then we go back to the step 2.

Note that the updates must be done only in the elements that are not at the borders. The values of the elements of the borders must be ruled by the border conditions.

The temp and diff variables exist only to test the convergence of the method. The temp is a temporary variable that caries the difference between the old and new value of an element. The biggest temp of the matrix becomes the diff. Then, at the end of the iteration, the diff variable is compared to a convergence parameter (like 0.000001) and if $diff \leq 0.000001$ then the loop stops, since the method converged. Sometimes, it is expected to have a situation where the convergence does not happen, thus we count the number of iterations and, if this values exceeds the maximum number of iterations permited, we get out of the loop and warn the user about the method not being able to converge.

3 Parallelization

The parallelization of this code was done using OpenMP.

In this program, we use several do-loops that updates the values of the inner elements of the matrix f. Using the shared mamory features of OpenMP, we can use parallel do loops in order to do each loop, sharing the work with several threads.

This implementation is very straightfoward, mostly relying on the power of the omp parallel do, however, the most important point is to declare the variables f and diff as shared varibles, so each thread can access the values of different elements of the matrix and update the value of diff.

See more about the parallelization method in the "Apresentation of the code" section.

4 Apresentation of the code

The modules (sizes and gfunctionmod) have only 2 uses: define the function g_function that evaluates the functions g for each (x,y) and define parameters in order to use double precision. Note that the funcion g is null, so, in this case, the Poisson equation becomes the Laplace equation.

Now, in the main program, we start declaring the variables. In order to change the system size, we must change the value of the parameter N. After runing the code many times I finally got a value of γ (over relaxation parameter) that allows my code to converge faster. So I am using $\gamma \approx 1.938$ now.

Then we get the number of threads from the command line or, as default, use 4 threads. Then me print the values of N and gamma so the user can see the configuration of the method.

We can then start counting the time with the OpenMP wallclock function and then fill the matrix g with its values and fill the matrix f with zeros, in order to get rid of the trash stored in the memory. After that we can update f with the border conditions. Since this part has many loops, we can use the omp parallel do in order to divide the work between the several threads. This allows the code to run faster. See an example of this below.

```
66
67    !$ wt0=omp_get_wtime()
68
69    ! lets fill the g matrix first
70
71    do i=1,N
72        !$omp parallel do private(j)
73        do j=1,N
74            g(i,j)=g_function((1.0_rk*i)/N,(1.0_rk*j)/N)
75        end do
76        !$omp end parallel do
77    end do
```

After all of that we start the Red-black algorithm with the SOR method. Notice that we use the omp parallel do in order to divide the work in each loop between the threads. See the algorithm section in order to see how the algorithm works.

Lastly, after the method converged, the program exits the converge_loop and prints the Wallclock time on the screen.

5 Instructions

Like it was said before, the program accepts 1 command line argument. This command line argument is the number of threads. The default value for the number of threads is 4 (in case narg/=1).

In order to compile the program use the make command or read the Makefile and README files. There are 3 versions of the code, 1 parallel (using OpenMP and 2 serial). Information regarding the serial versions is in the README file in the ../src folder.

See examples of the program's output in the ../run folder.

Note: Remember to read the 2 README files (one in the ../src folder and the other in the ../run folder).

6 Scaling behavior

Running the poisson_omp program in my computer (4 cores), I got the following behavior (see figure 2). The time values are the mean value of each file timesN.txt (N=1...5). In those files are 25 wallclock times obtained through the script get_times.sh.

After plotting the ideal scaling ($y = y(1)/x$), the graph becomes as seen in figure 3.

We can see that, although the times are smaller than the serial version and decreases with the number of threads (until the number of threads is bigger than the number of physical cores), the behavior is far from the ideal.

In figure 4 we have the times for N=300, however, this time after running the program just once. This is the same for the graphs of N=364,N=442 and N=580 (figures 5,6,7).

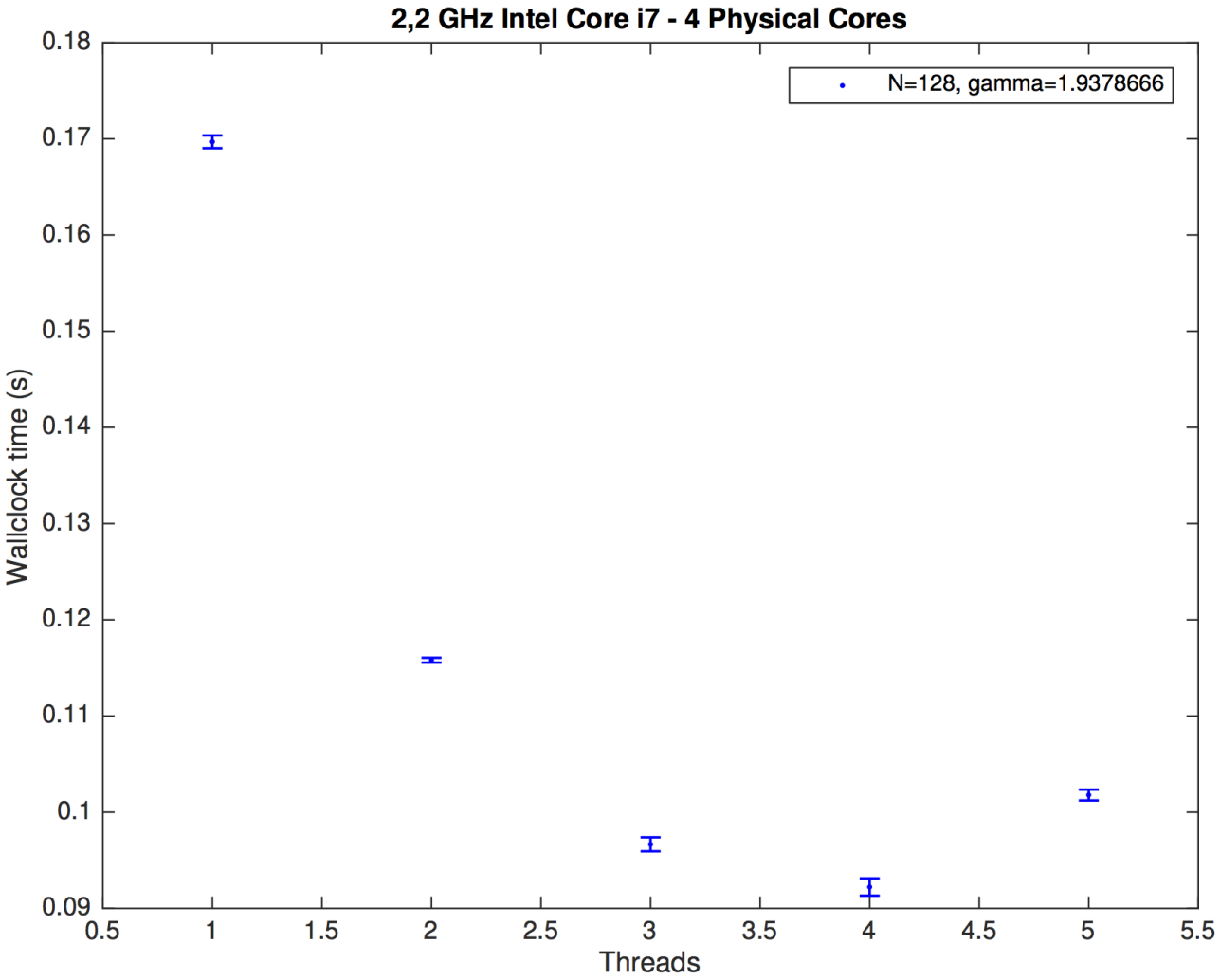

Figure 2: ./poisson_omp with N=128.

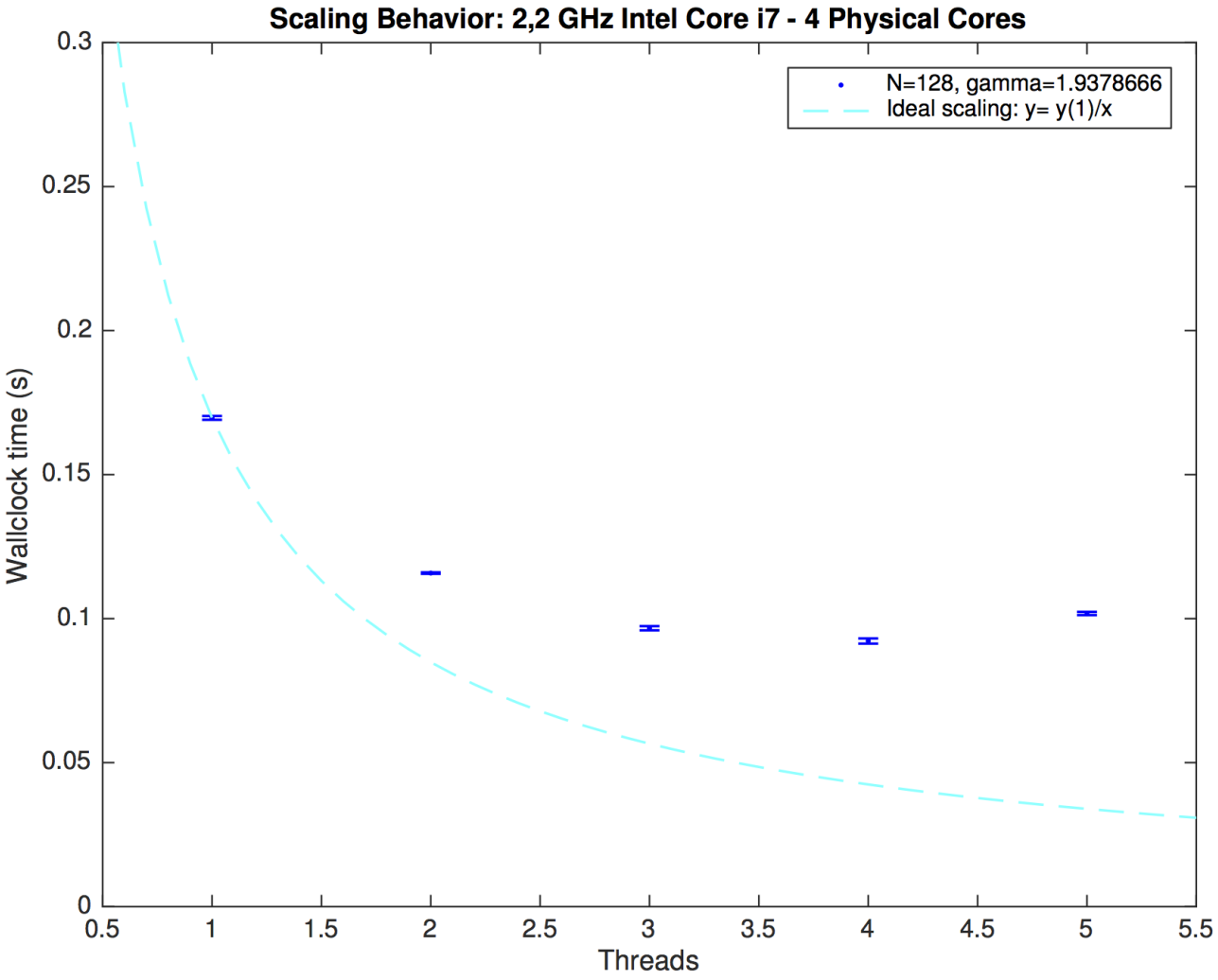

Figure 3: Ideal scaling compared with the actual one.

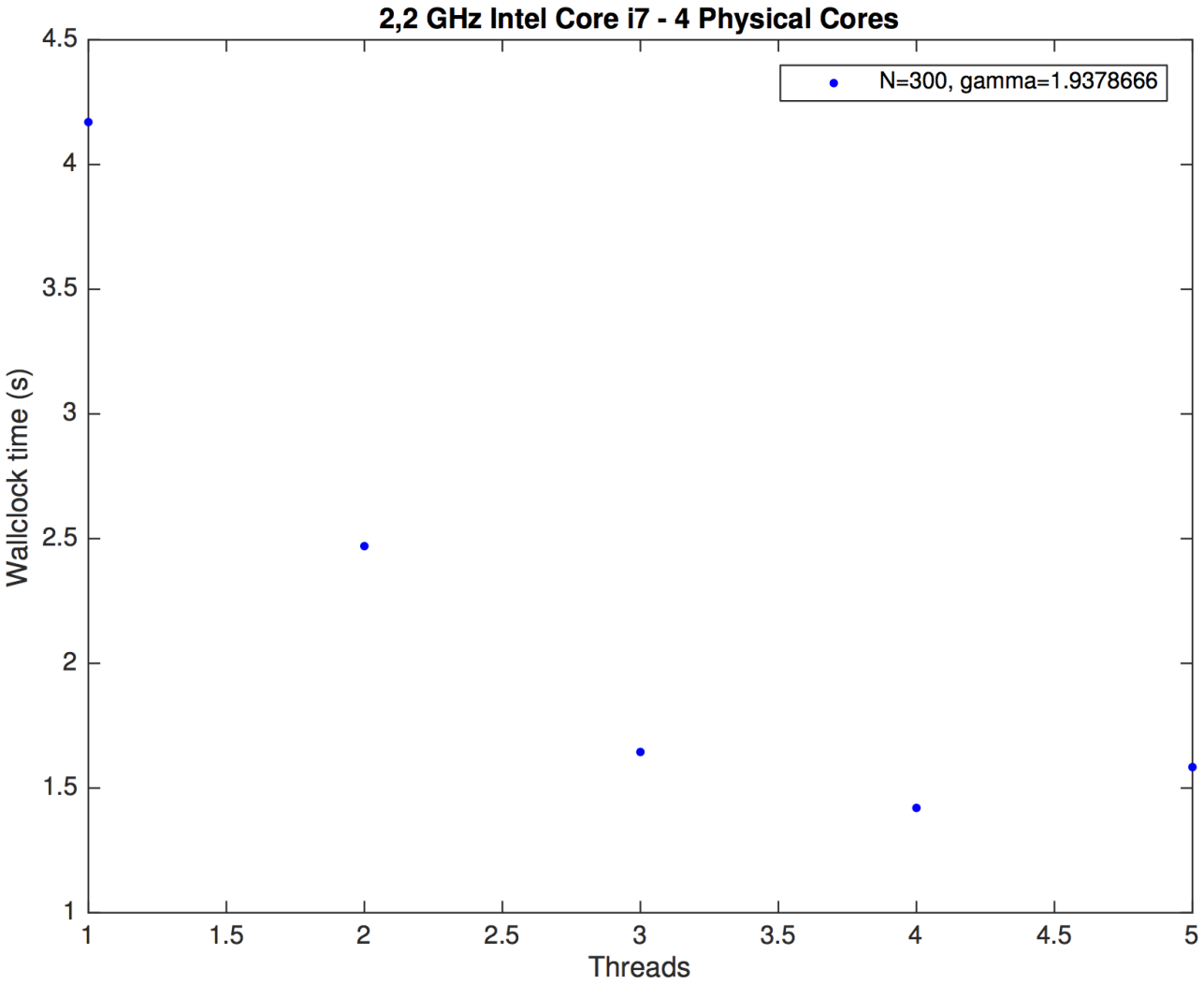

Figure 4: ./poisson_omp with N=300.

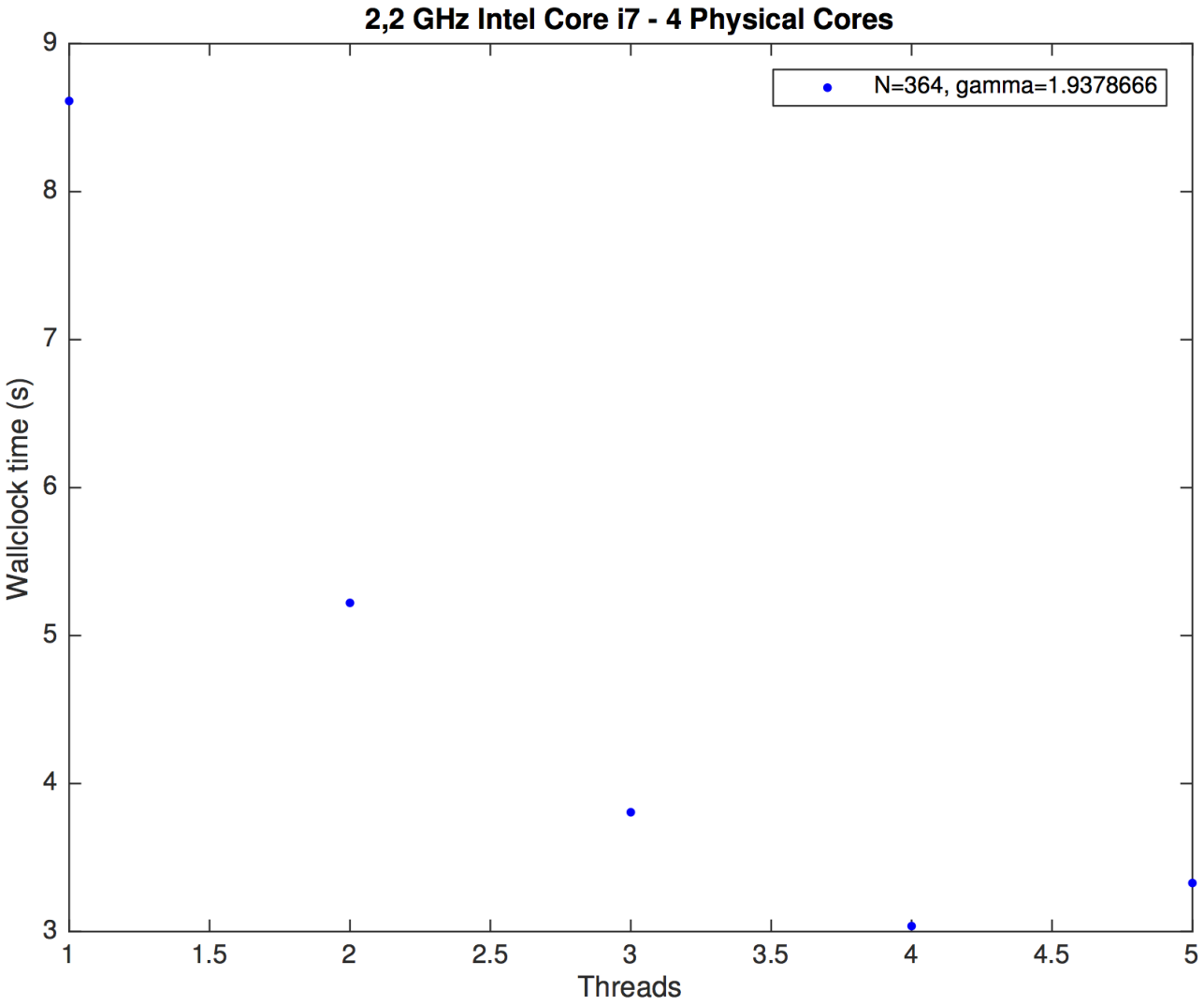

Figure 5: ./poisson_omp with N=364.

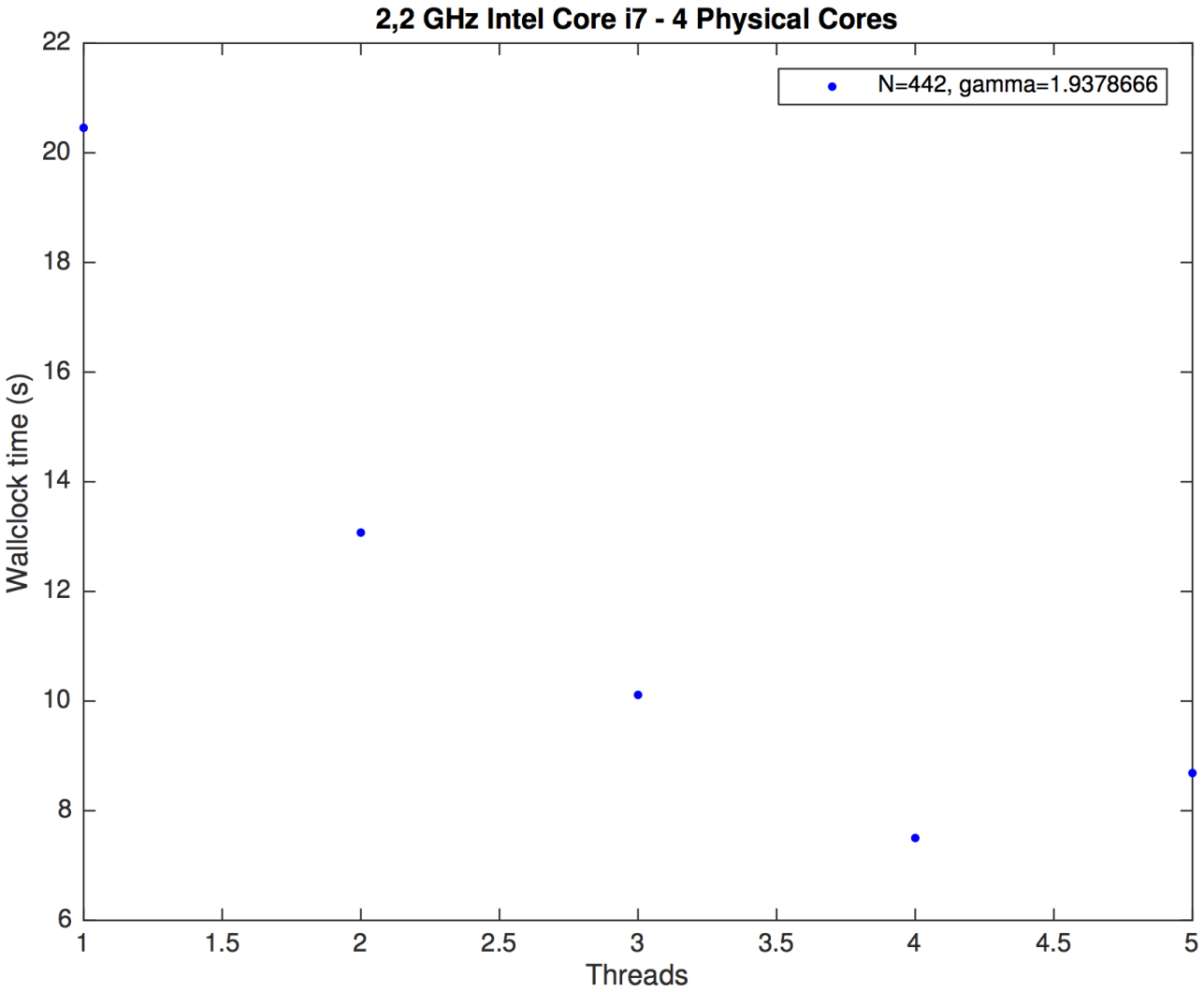

Figure 6: ./poisson_omp with N=442.

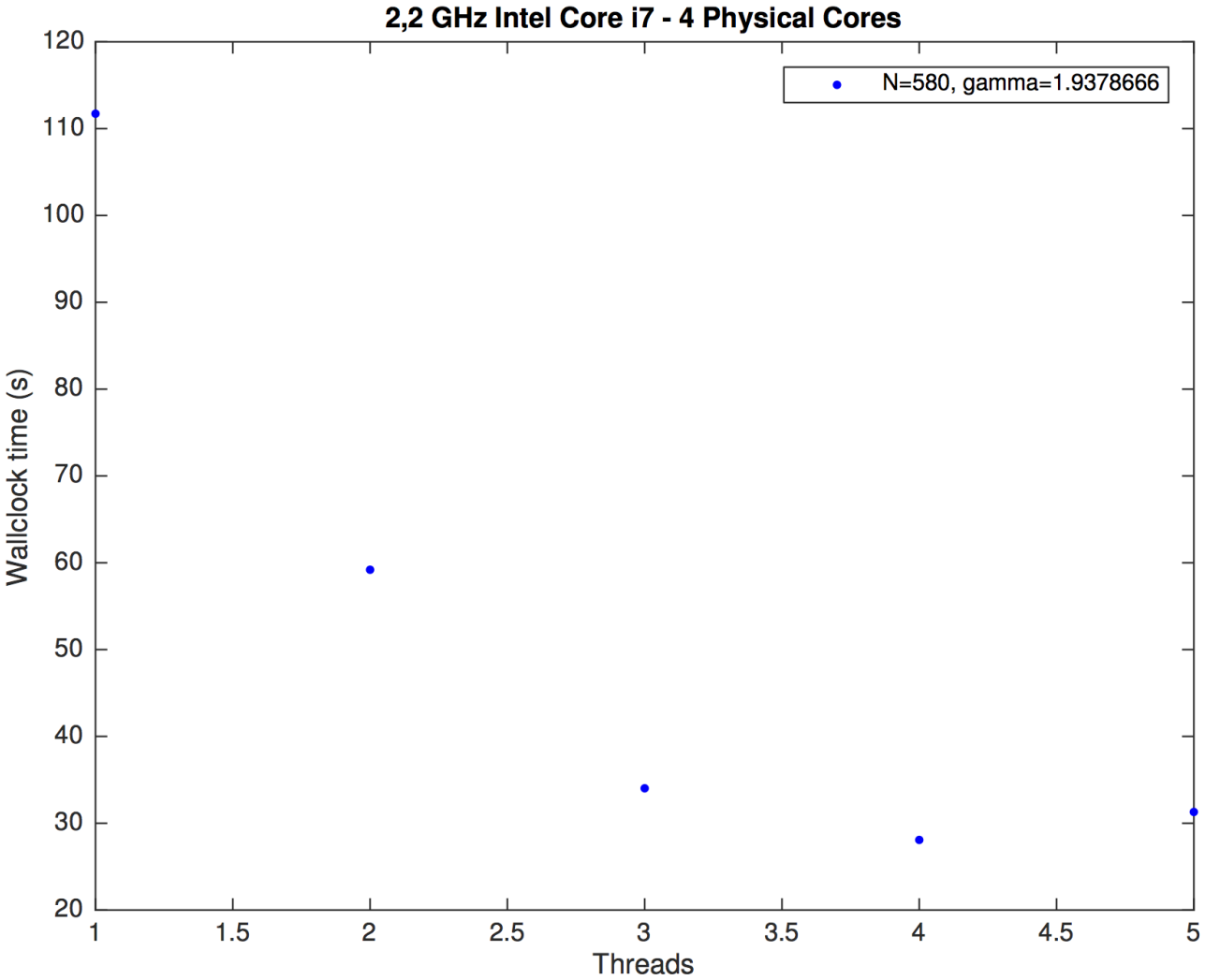

Figure 7: ./poisson_omp with N=580.

7 Conclusions

The SOR and Red-Black algorithms were implemented successfully in the poisson_omp parallel code. The parallel version is faster than the serial one (serial version, with N=128, takes 0.16900 s and the parallel version, with N=128 and 2 threads, takes 0.11753 s).

The wallclock times in the parallel version decreases with the number of threads until the number of threads surpasses the number of physical cores of the CPU. The scaling behavior is reasonable but is still far from the ideal scaling as seen in the lectures and exercises. However this was expected since I got similar results while studying the scaling behavior of other OpenMP codes while doing the weekly exercises.