OpenMP

Programming of Parallel Computers

R.G.A. Deckers

1. *Task 1*

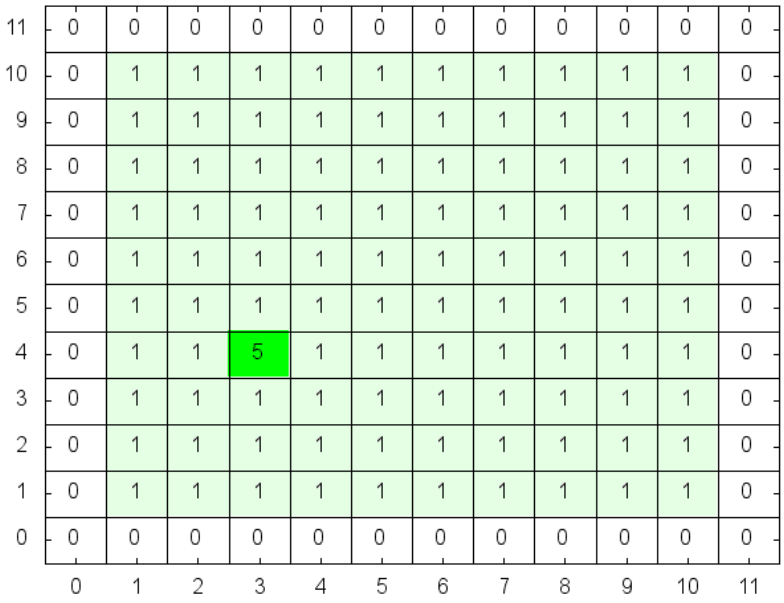

Figure 1: A simple print of an intial field.

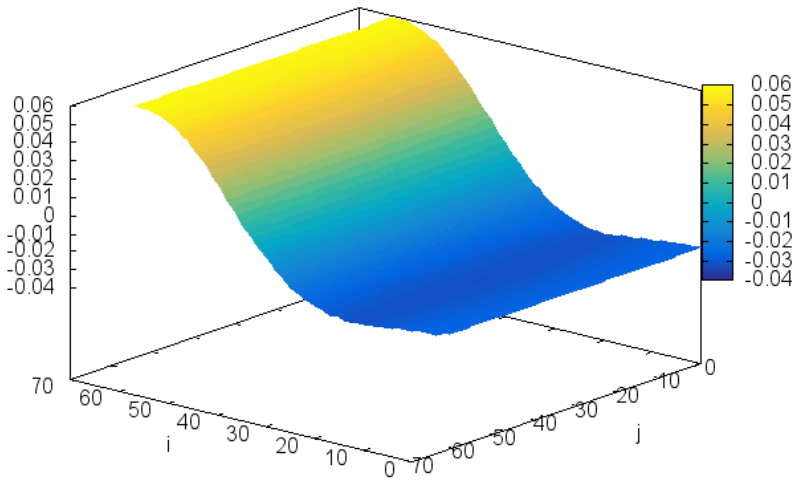

Figure 2: A solution for $f(x, y) = \sin(\pi x)$ on a 64×64 grid.

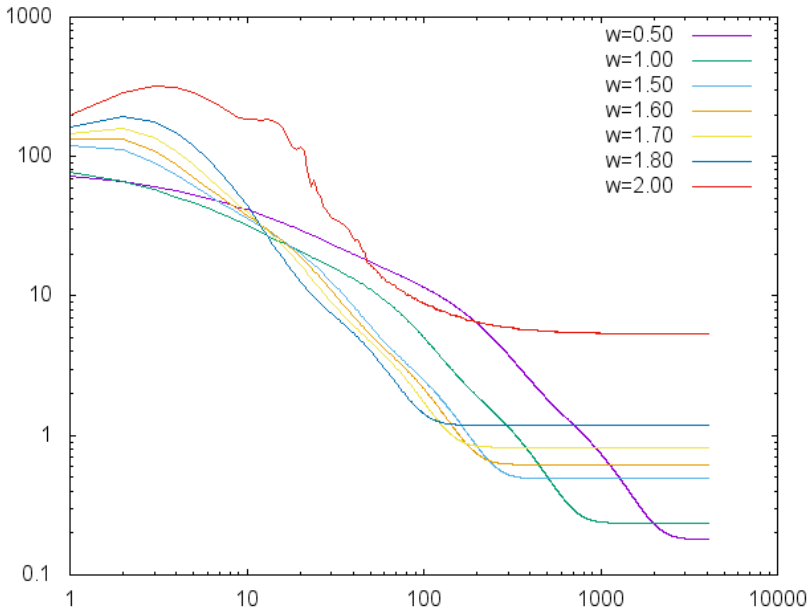

Figure 3: The residual as a function of the iteration count for several values of ω.

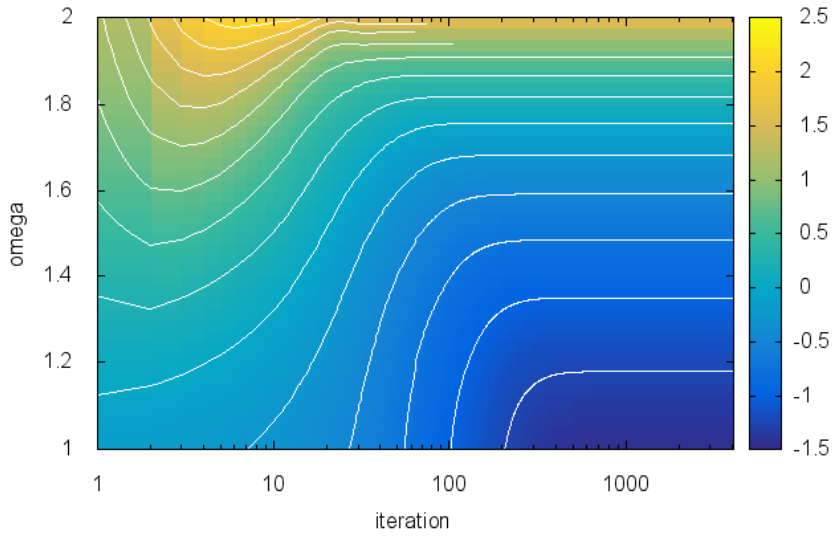

Figure 4: The residual as a function of the iteration count for several values of ω in a contour plot.

2. *Task 2*

Looking at figure 4 and 3 we can see that surprisingly smaller values of ω perform better. Small values of ω correspond to reducing the effect of overshooting so that is most likely what is happening here.[1] Still, looking at figure 3 we can clearly see that higher values of ω (around 1.5) initial descend steeper, so the best moethod would probably be to use a heuristic which gradually reduces ω as the slope of the residual curve flattens out. In any case, the tolerance of 10^{-12} as shown in the assignment's flowchart seems to be rather optimistic.

[1] or we have a bug somewhere.

3. *Task 3*

I have parallelized the solver at three points in the code. First, at the red and black loops for updating the points, and secondly when calculating the L2-norm of the residual, and third when subtracting the mean from each point. The speedup as a function of the threads are plotted in figure 5. The results are somewhat lackluster. This could be due to the way the code is written. Wile most functions are inlined into the main "run" function it may still fork/join too often to get enough speed-up, manually inlining all the functions into a single parallel block does perform somewhat better, supporting this hypothesis. A solution might be to do multiple red-black iterations at a time however these will still need to sync in between runs. False sharing was looked into but the static scheduler should split the work in equal sized continous chunks reducing it as much as reasonably possible.

We might also have been too impatient and not used a large enough grid size to see the real benefits. :)

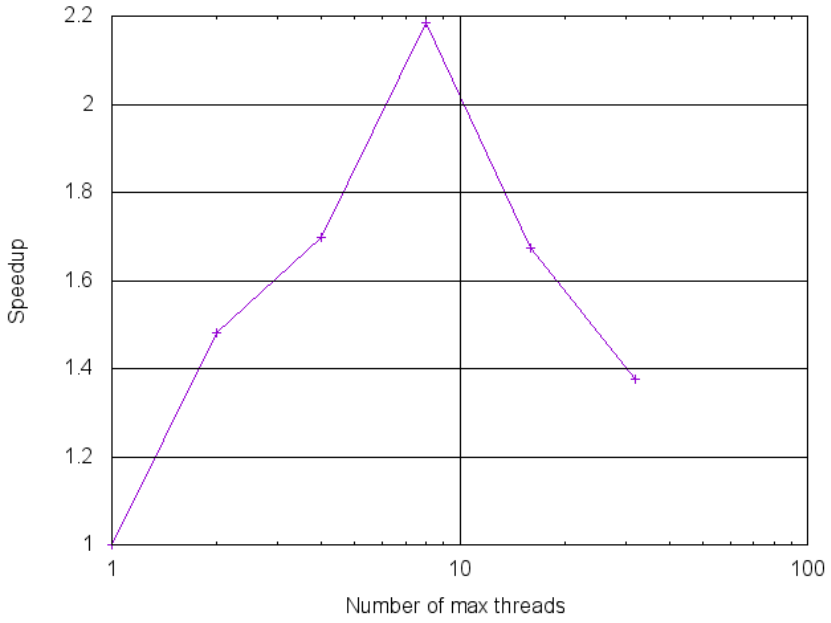

Figure 5: The speedup of our algorithm for various maximum thread counts.

4. *Thoughts on OpenMP*

OpenMP looks like an effective way of quickly getting a large speed-up for little work. The compiler oriented structure should also lead itself to highly optimized vendor implementations (Intel has its own runtime implementation it seems). However, as we have noticed in Task 3 it can lead to large blocks of code if one wants to reduce the overhead of fork-joining negating some of the readability benefits over other methods of parallelizing. The lack of more fine-grained control can also make it less efficient both in work-time and run-time I suspect if the problems become more complex (such as recursive heuristic based tasks or heavilly data-layout optimized code).