Parallel Poisson Solver in Fortran

Nilas Mandrup Hansen, Ask Hjorth Larsen

January 19, 2010

1 Introduction

In this assignment the $2D$ Poisson problem (Eq.1) is to be solved in either C/C++ or FORTRAN, first in serial and then in parallel using OpenMP.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{1}$$

Consider the heat distribution in a square room. The A radiator is placed in the room with a radiation of $200°C/m^2$. Three of the room's walls are kept at $20°C$ while the last wall is kept at $0°C$, meaning that the boundary conditions of the Dirichlet type.

2 Serial computing (Jacobi & Gauss-Seidel)

In this question the problem is solved iteratively in serial. The two iteration methods used are the Jacobi method, (Eq. 2), and the Gauss-Seidel method, (Eq. 3).

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i-1,k}^{(k)} + u_{i+1,k}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)} + \Delta^2 f_{i,j}\right) \tag{2}$$

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i-1,k}^{(k+1)} + u_{i+1,k}^{(k)} + u_{i,j-1}^{(k+1)} + u_{i,j+1}^{(k)} + \Delta^2 f_{i,j}\right) \tag{3}$$

From the preceeding equations it can be seen that for the Jacobi update method, every new value is calculated by using the values of the previous iteration. The old values are taken from one matrix and written to a different one, requiring the allocation of two equally large buffers, T and U. After updating U according to Eq. (2), its contents are copied back to T explicitly before next iteration. We note that a pointer-swapping scheme is clearly more efficient, but didn't bother to implement that.

For the Gauss-Seidel update method, each new value is computed from its neighbours like in the Jacobi method, but then immediately written back to the *source* array. This means that if the elements are updated in memory-contiguous order, the next element will depend on already-updated values at $(i - 1, j)$ and

$(i, j-1)$, while the other two neighbours have not yet been updated. The Gauss-Seidel method is known to converge in half the iteration count compared to Jacobi, and does not require an additional buffer.

The solution to the Poisson problem is presented in Fig. 1 for the grid size $N^2 = 4096$. The placement of the radiator is clearly seen in the figure. The presented figure is based on the Jacobi method, and the Gauss-Seidel method is observed to yield the same result. The solution is assumed to be found when

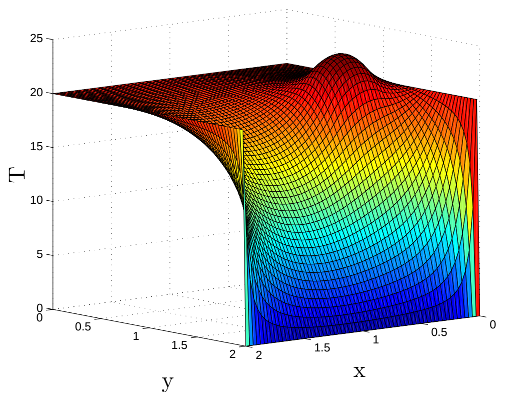

Figure 1: Plot of the temperature distribution, Grid size $N^2 = 4096$.

the stop criterion $||U - U_{old}||_F < \text{err}_{\text{max}}$ is reached, i.e. that the Frobenius norm of the difference between the current field and the one from previous iteration is smaller than the maximum error $\text{err}_{\text{max}} = 1e-7$.

The implementations have been tested on the grid sizes listed in Table 1 on p. 2).

Grid Size	Number of elements
8x8	64
16x16	256
32x32	1024
64x64	4096
128x128	16384
256x256	65536
512x512	262144

Table 1: Tested grid sizes.

The number of iterations as a function of number of elements is shown in Fig. 2. Observe that both methods yield a slope of 1 on log-log axes, which means that the complexity is $\mathcal{O}(n^2)$.

The Gauss-Seidel method converges twice as fast as the Jacobi method in terms of number of iterations, corresponding precisely to the vertical displacement between the two lines. Fig. 3 shows the wall clock time as a function

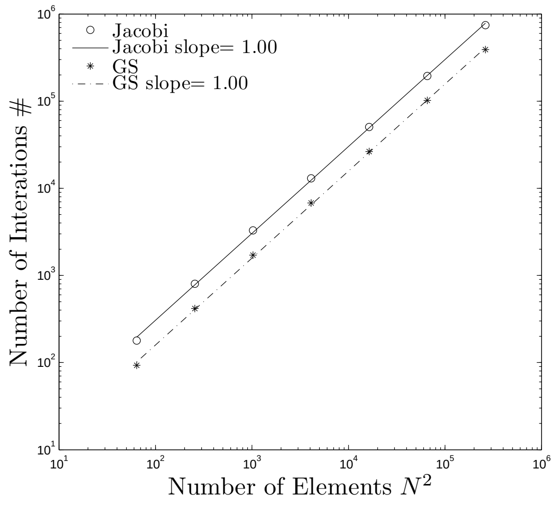

Figure 2: Number of iterations as a function of grid size N^2 on loglog.

of grid size. The wall clock time increases equal amounts for both methods, meaning that the Jacobi method uses half the time per iteration compared to the Gauss-Seidel method for small problems. This we find both remarkable and unlikely, but nonetheless quite consistent. A possible explanation is that reading and writing to the same array makes it more difficult for the compiler to optimize.[1]

Finally Fig. 4 show the number of iterations per second. Here it can be seen that in general the Jacobi method manages more iterations per second than the Gauss-Seidel method for problems having a maximum size of about $512x512$ elements. One should remember that a grid of $512x512$ corresponds to approximately $2MB$, which is less than the cache size of the CPU ($8MB$), meaning that the Jacobi method is the fastest method. If one uses, e.g. a $1600x1600$ grid, then the memory requirement is approximately $20MB$. When the amount of data exceeds the cache size, it is expected that the Gauss-Seidel will do more iterations per second since it uses two grids, whereas the Jacobi method uses three grids. This was tested on a $1600x1600$ grid where the Gauss-Seidel method was approximately 10% faster that the Jacobi in total, i.e. still much slower per iteration.

[1] Indeed we have tested that the red-black Gauss-Seidel solver (to be discussed later), which doesn't read from and write to the same memory in immediate succession, uses slightly less time per updated element compared to the Jacobi solver.

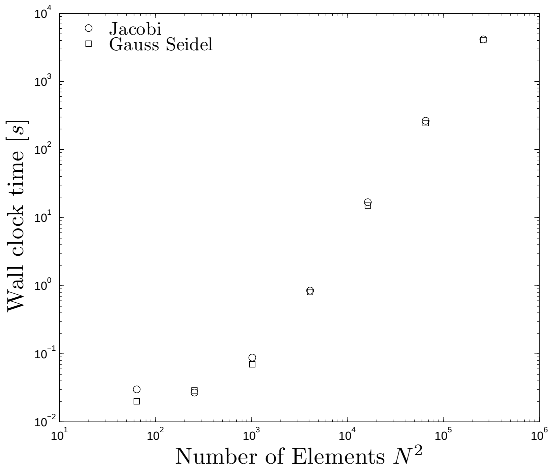

Figure 3: Wall clock time as a function of grid size N^2 on loglog.

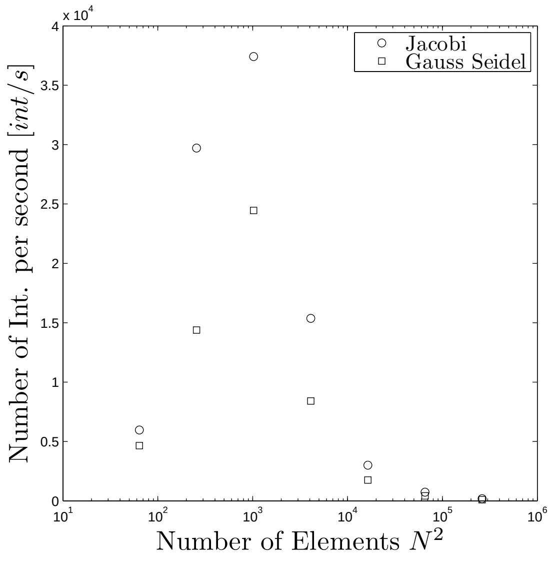

Figure 4: Number of iterations per second as a function of grid size N^2 on semilogx.

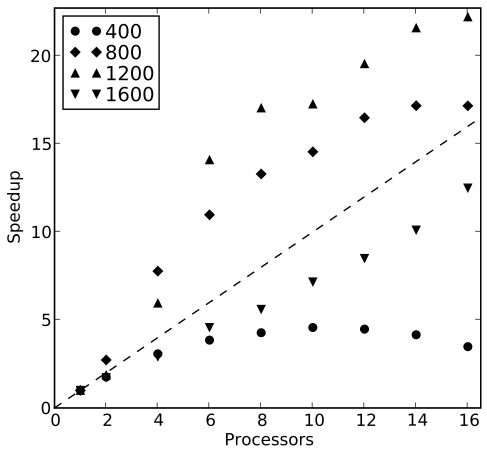

Figure 5: Speedup of Jacobi method as a function of process count for different problem sizes.

3 Parallelization of Jacobi solver

The Jacobi method is simple to parallelize: Each process is made responsible for a distinct slice of the output matrix by declaring the outermost loop as parallel. The elements on the domain boundaries of the input array are read concurrently by different processes, but this is fine as nothing is written back to that array. Some synchronization is required, since no thread should begin copying the contents of the output array back into the input array, but this is taken care of by the implicit barrier at the end of the parallel for loop.

Omitting convergence check, the update loop is[2]:

```
do n=1,maxiter
!$omp do
   do j=2, nx - 1
      do i=2, ny - 1
         U(i, j) = 0.25 * (T(i-1, j) + T(i+1, j) + T(i, j-1)&
            + T(i, j+1) + F(i, j))
      enddo
   enddo
!$omp end do
enddo
```

Figure 5 shows the scaling properties of the Jacobi method. For small matrices the performance is limited as each process has little work between the barriers. Larger matrices scale better as there is more work compared to the

[2]The array `F` contains a premultiplied factor of Δ^2

amount of synchronization. For sufficiently large problems, the matrices do not fit into the CPU cache size for low CPU counts, which hampers performance due to cache misses. Adding more CPUs makes the buffers fit into the cache, resulting in superlinear speedup for “medium-sized” matrices. Sufficiently large matrices do not fit into the cache either way and thus do not achieve superlinear speedup. On the contrary, the 1600 by 1600 matrix does not scale well in spite of the very little synchronization required. We believe the large number of CPUs combined with many cache misses strain the main memory bandwidth.

4 Parallelization of Gauss-Seidel solver

The Gauss-Seidel solver is not quite as easily parallelized, as information is read from *and* written back to the same array. Each process must therefore ensure that the neighbours of the element which it is about to update have been updated precisely the right number of times, which requires explicit synchronization if the update is performed in straight-forward memory-contiguous order.

A useful Ansatz would be to divide the problem into e.g. horizontal slices, each process having one slice. Process 1 will then start with the top slice and will, at some point finish updating it. At this point process 2 in the slice immediately below can start updating, since its upper boundary elements have now been updated. Meanwhile process 0 restarts, iterating the first slice a second time, but must not again update the lowest row before process 2 has finished its own top row. This way, each process can start updating slices further down the array in a pipeline-like fashion, requiring synchronized access to the top and bottom rows in each domain. This is called temporal blocking. In principle this can scale well since only neighbouring processes communicate, but some overhead must be expected. Also the method is rather complicated.

4.1 Red-black Gauss-Seidel

In place of temporal blocking we have implemented a so-called red-black scheme. In this scheme we perform one update by making two entire sweeps of the array: first we update half the points in a checkerboard pattern, where each update follows the same formula as the Jacobi method. Then we perform a second sweep, updating the other half of the points using the values calculated in the first sweep. While the first sweep uses purely “old” values, the second sweep uses purely “new” values such that on average a point is updated from two new and two old values, making this a Gauss-Seidel scheme.

The red-black method is parallelized simply by dividing the matrix into slices over the non-leading dimension, and giving each process one slice. The advantage is that during one sweep, every element is either written or read, but not both, so synchronization is required only between distinct sweeps.

This makes the red-black implementation both simple and highly scalable. A drawback is that we incur the overhead of looping over the same array twice,

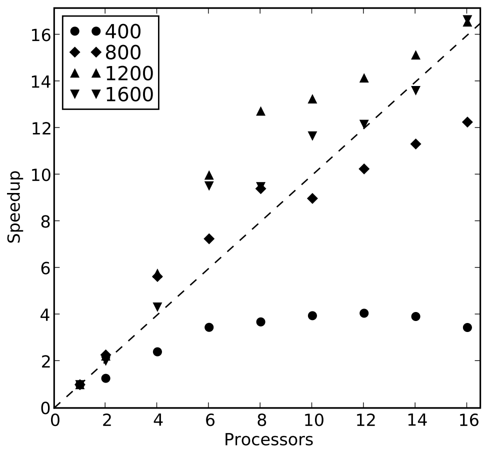

Figure 6: Speedup of the red-black Gauss-Seidel implementation as a function of process count for various problem sizes.

only touching half the elements each time, thus also taking less advantage of caching. Another, much more profound drawback is that the red-black scheme is limited to finite-difference stencils that are in fact red-black.

The code for the red-black scheme is seen here:

```
!$omp parallel shared(T, F, nx, ny, maxiter, interval, maxerr) &
!$omp   & private(n, i, j, istart, istop)
  do n=1, 2 * maxiter
!$omp do
     do j=2, nx - 1
        istart = 2 + mod(j + n, 2)
        istop = ny - 1
        do i=istart, istop, 2
           T(i, j) = 0.25 * (T(i-1, j) + T(i+1, j) + T(i, j-1) &
                + T(i, j+1) + F(i, j))
        enddo
     enddo
!$omp end do
  enddo
!$omp end parallel
```

The `istart` parameter alternates between the first and second row on each sweep, while the loop variable has a stride of 2.

Figure 6 shows the scaling properties of the Gauss-Seidel implementation. The plot looks quite similar to Figure 5. The 800 by 800 problem does (mostly) not scale superlinearly as in the Jacobi case, because with only two buffers allocated, the problem almost fits into the cache even in serial. Unlike for the Jacobi method, even the 1600 by 1600 problem scales superlinearly, confirming that the Gauss-Seidel method is considerably more buffer-friendly.