TMA4280 - Problem set 6

Tor Holm Slettebak

10. april 2015

1 The poisson program

In the beginning I tried to take advantage of our professors framework of several poisson solvers, which uses vectors and matrices in Fortran format. This however turned out to be hard for me. When I tried to create a transpose operation in parallel I could not make it work. So I made a switched to the code distributed with the problemtext, which is written by Einar M. Ronquist.

1.1 The poisson equation

The program is to solve the Poisson equation

$$-\nabla^2 u = f, in \Omega = (0,1) \times (0,1) \tag{1}$$

$$u = 0 \text{ on } \partial\Omega \tag{2}$$

To solve this equation it needs to be translated to a numberical scheme. It needs discretization. A Taylor expansion on the equation gives us

$$u_{xx} \approx \frac{1}{h^2}(u(x+h,y) - 2u(x,y)0u(x-h,y)) \tag{3}$$

Doing this in the y-direction as well we get the discretization of the Poisson equaion in form of the 5 point stencil.

$$-u(x+h,y) - u(x,y+h) - u(x-h,y) - u(x,y-h) + 4u(x,y) = h^2 f(x,y) \tag{4}$$

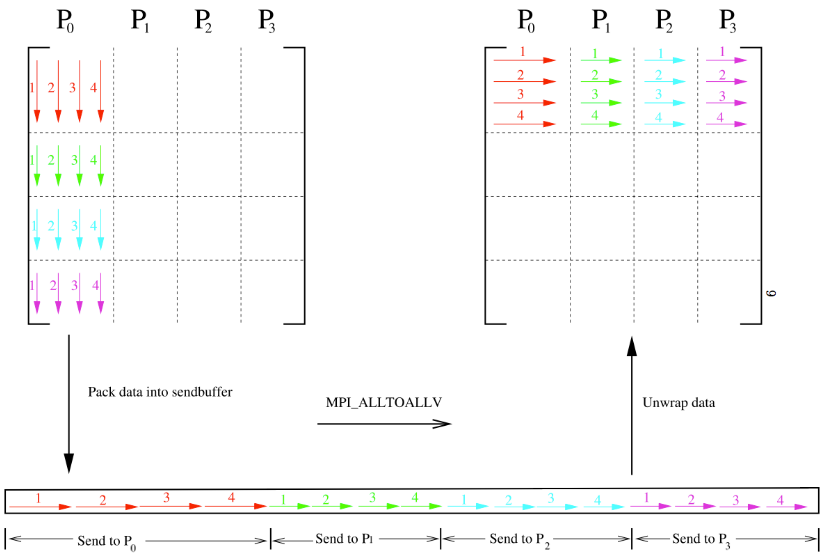
2 Parallelization

To parallelize the problem I chose to split up the problem in lesser parts, each solvable by it self. Each process generates it own matrix B, and modifies it through the program, before it shares it result with each of the other processes and the maximal error is calculated. During each process' execution the only communication, except for in assembly, between the threads are done in the transpose operation.

2.1 The transpose operation

Since each process has a part of the matrix b, a transpose will need to swap elements with other processes. This communcation is done through MPI_Alltoallv (see figure 1), a MPI call which sends a sendarray according to a count array and a displacement array. The countarray contains the number of elements to be sent to each process. The displacement arrays entry i specifies the displacement relative to the sendarray, entry i is sent to process i. The receive buffer is filled according to the same displacement and count array.

During the writing of this report, and after results had been obtained, I found that I had created a duplicate matrixAsVec operation. In the transpose operation I believe a less effective version of this has been used, a version not using memcpy(). This may have some small effect on runtime, but no real testing has been performed to confirm this.



Figur 1: Transpose operation through MPI_Alltoallv

2.2 The error computation in parallel

At computation of error there is also some need for paralellization. Generation of the problem's exact solution is done for each process', and the exact solution can therefore be compared with the solution found on each process. I here took advantage of our professor's common files, and blaslapack.

I created a matrixToVec() operation, which transforms the processes part matrix b to a vector, as well as the process' exact solution. After this, an axpy operation is performed on the solution v.s. the exact solution. This performs the following vector operation $y < -(-1.0 * solutionVector) + exactVector$. This generates the relative errors for this process' solution.

Then the maximal local error is found with blas routine idamax, which returns the element of largest magnitude. Now each process has found the max error in it's own part solution. The global max error is computed with a call of MPI_Reduce with MPI_MAX, which result in the maximal pointwise error.

For a time I also assembled the final solution matrix on the root process using MPI_Gatherv. I however found that this was unnecessary, when the results from the program is available in each process, and only the pointwise error is needed for report.

2.3 Use of OpenMP

To further increase the parallelization of the program OpenMP can be used. This enables the program's MPI processes to use more than one thread in it's execution. The main (perhaps only) use of extra threads in this program is when performing the fourier transformations. This means adding a

```
#pragma omp parallel for schedule(static)
```

before the various Fortran fourier transformations. This means that each MPI process can perform the fourier transforms with increased degree of parallelization. One drawback with use of OpenMP on these parts of the program is that the fortran procedures requires a buffer. This leads to the requirement of each thread having it's own buffer. This means an increase in memory requirement for the program, due to an increased size of buffer matrix z.

Possible speedup with use of OpenMP will be discussed in result section(INCLUDE REFERENCE TO IT).

3 Kongull

Kongull has 108 compute nodes, each having 2 processors. This is divided in 96 nodes using an AMD Opteron processor with 6 cores, and 12 nodes using Intel(R) Xeon(R) processors. Depending on which rack of the compute nodes are being used, the node has 24 GiB/node or 48 GiB/node. This means that each processor has at least 12 GiB of memory (create reference to https://www.hpc.ntnu.no/display/hpc/Kongull+Hardware Compute-nodes). I believe that our student projects in TMA4280 only have access to the section having 24 GB memory per node.

To compile and run the program I have used Intel's compilers for Fortran and C, ifortran and icc. Cmake was used to perform the compilation with these compilers. This generates a build system with portability and simplicity. The program uses MPI, OpenMP and BLAS/Lapack. In addition support for C99 has been added for simplicty of writing code. All of these are installed on Kongull.

4 Verification of correctness

To verify that the code works correctly, I have performed a convergence test. To obtain a correct error estimate an exact solution were computed. The exact solution has been entered as

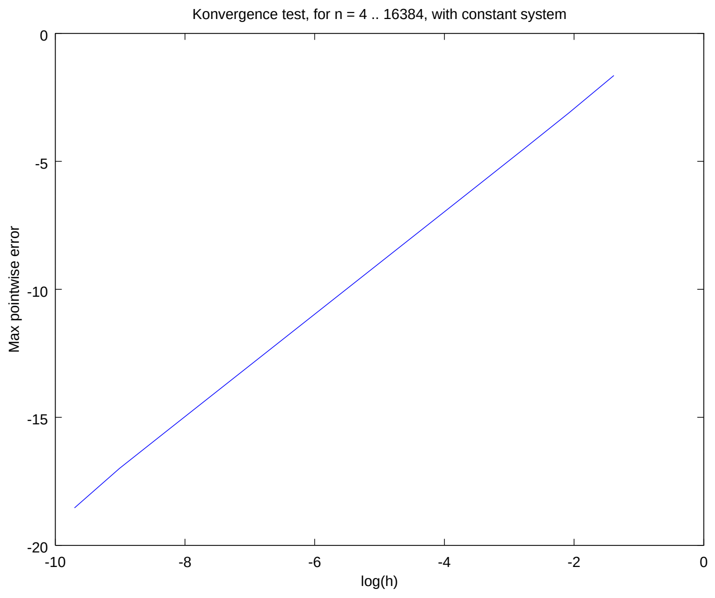
$$u(x, y) = sin(\pi x) \cdot sin(2\pi x) \tag{5}$$

Which satisfies the homogeneous boundary conditions $u = 0$ on $\partial\Omega$. If we then evalueate $-\nabla^2 u$ which should be equal to

$$u(x, y) = 5\pi^2 \cdot sin(\pi x) \cdot sin(2\pi x) \tag{6}$$

The error gained from comparison of the exact and computed solution is then used to check whether the program performs correctly.

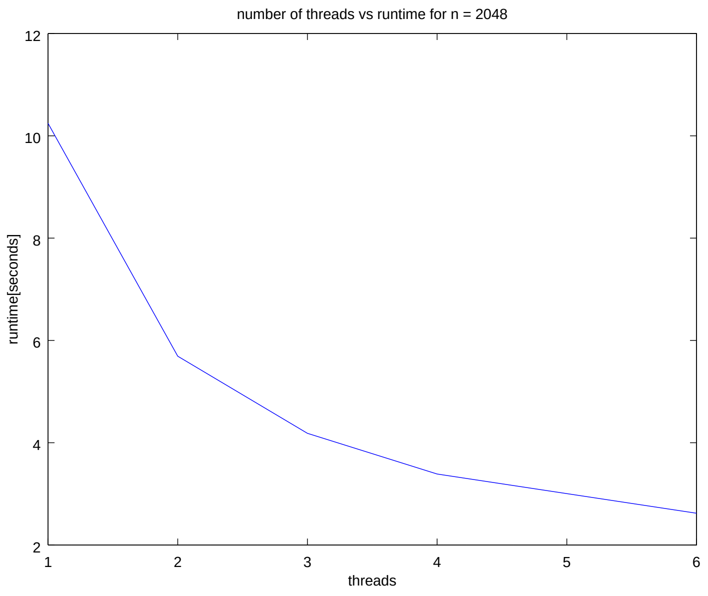
This test has been performed with 3 nodes with 2 processors each having 6 cores. Assigned number of threads per MPI process was 6, which perhaps is not ideal, but nevertheless. The program was executed with this setup for a number of problem sizes from $n = 4816...16384$, and the errors for each problem size is compared with $h = (1/n)^2$. This then was processed in a octave plot using loglog, resulting in the following plot 2 where the gradient $a = 2.0309$ is just about two. The graph is also just about completely linear. This verifies the correctness of the solution.



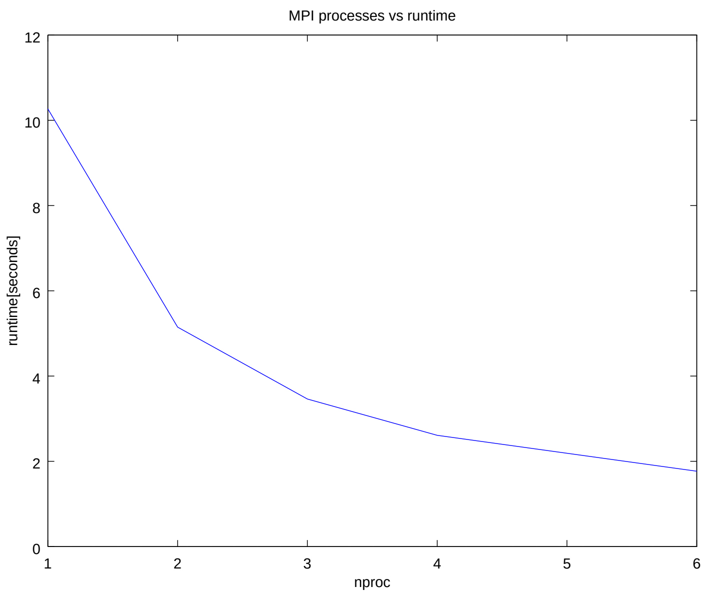
Figur 2: loglog plot of h, max pointwise error

5 Hybrid v.s. Pure distributed memory model

To check the effictiveness of the OpenMP threads in combination with MPI, I performed a test where $p \cdot t = 36$, and a constant problem size of $n = 16384$. The pure MPI solution performes quite a lot better, about 10 seconds faster than an execution with 6 MPI processes and 6 threads. This is because the main part of my solution builds on MPI. Sacrificing MPI-processes for threads leads to a smaller part of the problem size being parallelized. The threads only contribution is to perform the fast fourier routines in parallel. In addition the transpose operation will be slower because it purely depends on number of MPI processes and the problem size. However, tests performed with only one MPI process and a increasing number of threads show a good increase in speedup, a doubling of threads almost decreases runtime by a factor of 2. This proves that the threads does increase performance.



Figur 3: Threads vs runtime, for one MPI process



Figur 4: MPI processes vs runtime, for one thread

Similarly for an equivalent setup with MPI processes. We see almost similar results, it seems however that for this relatively small problem size, the improvement of MPI over OpenMP is very little. But from tests on bigger problem sizes we see that this grows more relevant.

6 Speedup and parallel efficiency

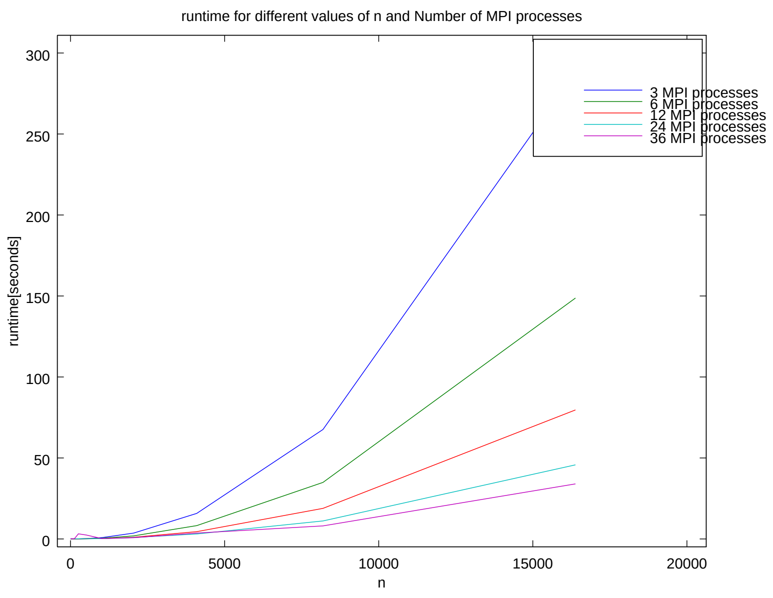
The speedup for the program is calculated with

$$S_p = \frac{T_1}{T_p} \tag{7}$$

T_1 being the execution time for the sequential version of the program, T_p is the time it takes to run the program in parallel using p processors. Calculating this speedup gives us the possibility to calculate an estimate of the parallel efficiency

$$\eta_p = \frac{S_p}{p} \tag{8}$$

The general speed for the program when executed with equal input, but for a various number of MPI processes, can be seen in the following plot[fig:5].



Figur 5: runtime for different number of processes, with one thread