TMA4280 Super Computers

Jakob Hovland
Jørgen Grimnes
Assignment 6

Spring 2015

Abstract

This report presents a solution to homework project number six in TMA4280 Super Computers, spring 2015, at Norwegian University of Science and Technology. This report will discuss a solution strategy to implementing an efficient, parallel poisson solver by applying the Descrete Sine Transform.

1 The Poisson problem

This report will discuss a solution strategy to the 2D Poisson problem. The problem is mathematically defined as:

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in } \Omega = (0,1) \times (0,1) \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \tag{1}$$

The f is the known right hand side of the equation and u is the solution. We will present a solution strategy to solving equation 1 in the given domain $\Omega = (0,1)$ x $(0,1)$. To solve the equation numerically, we have to discretize the problem on a finite difference grid. We have implicitly discretized the problem to a regular grid with $n+1$ points in each spatial direction with grid spacing $h = \frac{1}{n}$.

$$\Delta = \frac{\partial^2 f}{\partial x^2} \frac{\partial^2 f}{\partial y^2} \tag{2}$$

We have applied the standard 5 point stencil when discretizing the Laplace operator 2 as illustrated in Figure 1. This operation yields an error of $\mathcal{O}(h^2)$.

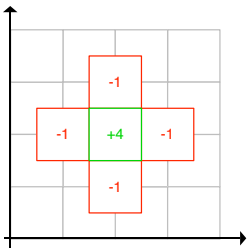

FIGURE 1: 5 point stencil

2 Designing the solution strategy

To solve the equation listed in 1 we have exploited the power of diagonalization and then applied the Descrete Sine Transform to calculate a solution in $\mathcal{O}(n^2 \log n)$ floating point operations.

2.1 Diagonalization. We define $U_{i,j}$ for $i, j = 1, ..., n-1$ as

$$U_{i,j} = \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,j} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ u_{i,1} & u_{i,2} & \cdots & u_{i,j} \end{pmatrix}$$

and then initialize a symmetrical matrix T:

$$T = \begin{pmatrix} 4 & -1 & 0 & \cdots & -1 & 0 & \cdots & 0 \\ -1 & 4 & \ddots & & & \ddots & & \vdots \\ 0 & \ddots & \ddots & & & & & 0 \\ \vdots & & & & & & & -1 \\ -1 & & & & & & & \vdots \\ 0 & \ddots & & & & & & 0 \\ \vdots & & & & & & & -1 \\ 0 & \cdots & 0 & -1 & \cdots & 0 & -1 & 4 \end{pmatrix}$$

We can now express our 5 point stencil equation as:

$$\begin{aligned} h^2 f_{i,j} &= 4u_{i,j} - u_{i+1,j} - u_{i-1,j} \\ &\quad - u_{i,j+1} - u_{i,j-1} \\ &= (TU + UT)_{i,j} \end{aligned} \tag{3}$$

For $i, j = 1, ..., n-1$. We can write this on the concensed form $(TU + UT) = G$ where

$$G = h^2 \begin{pmatrix} f_{1,1} & \cdots & f_{1,j-1} \\ \vdots & \ddots & \vdots \\ f_{i-1,1} & \cdots & f_{i-1,j-1} \end{pmatrix} \tag{4}$$

By taking advantage of the symmetrical property of the T matrix we specify $T = Q\Lambda Q^T$ where Q is an orthonormal matrix containing the normalized eigenvectors of T and Λ contains the corresponding eigenvalues along the diagonal. If we multiply $(TU + TU) = G$ with Q on the left and Q^T on the right, we can rewrite this as:

$$\Lambda Q^T U Q + Q^T U Q \Lambda = Q^T G Q$$

Which reveals a numerical approach to calculating U by sequentially computing $\widetilde{G} = Q^T G Q$, solving $\Lambda\widetilde{U} + \widetilde{U}\Lambda = \widetilde{G}$ for $\widetilde{U}$ and finally calculating $U = Q\widetilde{U}Q^T$. These calculations require $\mathcal{O}(n^3)$, $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ floating point operations respectively.

2.2 Applying the Discrete Sine transform. We can optimize the computation of $\widetilde{G} = Q^T G Q$ and $U = Q\widetilde{U}Q^T$ by taking advantage of the Discrete Sine transform. The DST of a vector $\vec{v}$ where $\vec{v} = [v_1, v_2, ...v_{n-1}]$ is defined as:

$$\widetilde{v} = \sum_{i=1}^{n-1} \vec{v}_i \sin\left(\frac{ij\Pi}{n}\right) \quad j \in 1, .., n-1 \tag{5}$$

Following 5 we can write this as $\widetilde{v} = S(\vec{v})$ and $\vec{v} = S^{-1}(\widetilde{v})$ where $S = \frac{2}{n}S^{-1}$.

We can further observe that the Q matrix can be described as a scaled DST. The Q matrix is composed by the normalized eigenvectors of T and the elements of the Q matrix can be described by:

$$q_{i,j} = \sqrt{\frac{2}{n}} \sin(\frac{ij\Pi}{n}) \tag{6}$$

Which is clearly a $\sqrt{\frac{2}{n}}$ scaled version of the DST transformation S. We can therefore denote the Q matrix as:

$$Q = \sqrt{\frac{2}{n}} S \tag{7}$$

By the orthonormal properties of Q we know that $Q^{-1} = Q^T$, and we can further specify equation 7:

$$Q = \sqrt{\frac{2}{n}} S = \sqrt{\frac{n}{2}} S^{-1} \tag{8}$$

If we use the Fast Fourier Transformation as popularized by Cooley and Tukey [1] we can calculate 5 in $\mathcal{O}(n \log n)$ floating point operations.

This enables us to calculate $\widetilde{G} = Q^T G Q$ and $U = Q\widetilde{U}Q^T$ in $\mathcal{O}(n^2 \log n)$ rather than $\mathcal{O}(n^3)$. The solution strategy has then been reformed to consist of the following tree steps:

1) $\widetilde{G}^T = S^{-1}((SG)^T)$

2) $\widetilde{u}_{i,j} = \frac{\widetilde{g}_{i,j}}{\lambda_i + \lambda_j}$

2) $U = S^{-1}(S(\widetilde{U}^T))^T$

3 The implementation

Our proposed solution strategy is parallelized with OpenMP and the MPI library. OpenMP is used for shared memory multiprocessing through multithreading while MPI provides the means of communicating between the different processes.

3.1 Load balancing. At any time, each process performs computations on a subset of the rows in the matrix. If the number of rows in the matrix, M, is not divisible by the number of processes, P, then $M\%P$ processes will be responsible for $\frac{M}{P}+1$ rows, and $P-M\%P$ processes will be responsible for $\frac{M}{P}$ rows. Thus, the program is perfectly load balanced, in the sense that all processes are responsible for an equal amount of rows, if $M\%P = 0$.

3.2 Matrix transposition. The transpose of the matrix is computed in three steps. First the matrix is packed into the send buffer. The packing is a standard out-of-place matrix transposition. Then a call to MPI_Alltoallv is used to distribute the columns among the processes. Finally the receive buffer is unpacked into the matrix.

4 The evaluation environment

4.1 Libraries. We've implemented two versions of the solver. We created two versions because we didn't manage to run the Fortran script on our Windows workstation. The first version we developed was based on the Intel MKL [2] implementation of the Discrete Sine transform. This yielded very good results and performed very well even when restricted to a single processor. When we compiled the solver on Kongull, we used the provided Fortran script to perform the DST as specified in the project description.

4.2 Kongull. Kongull is a high performance cluster computer consisting of 108 compute nodes with a total of 1344 cores. 96 nodes are HP DL165 G6 servers with two AMD Opteron model 2431 @ 2.40GHz 6-core (Istanbul) processors. 44 of the HP nodes has 24 GiB @ 800MHz memory, while the remaining HP nodes has 48 GiB @ 667 MHz.

The last 12 new nodes are Dell Intel Xeon based servers with Intel®Xeon®CPU E5-2670 @ 2.60GHz 8-core (Sandy Bridge) processors. All of the Intel based nodes has 32GB @ 1600 MHz memory.

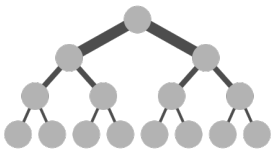

FIGURE 2: Fat Tree. The figure is from Wikipedia.

The cluster is networked as a *Fat Tree* as graphically represented in Figure 2. The thickness of the edges depict the connection speed.

4.3 Compiler. We are using the Intel C++ compiler and a CMake build script with the DCMAKE_BUILD_TYPE flag set to *Release*.

5 Results

The results presented in this report was generated by running the solver on Kongull. The solver were run with configurations ranging $p \in [1, 36]$ and $t \in [1, 12]$. The implemented solver showed to yield very promising speedup gains as presented in Figure 7.

5.1 The hybrid model compared to the pure distributed memory model with $n = 16384$ and $p * t = 36$. The hybrid model does not appear to offer a performance gain in. As depicted in Figure 3, the runtime of the solver is mainly dictated by the number of processors. If you were at a position to either chose 12 different processors or a single processor running 12 threads, the 12 processor computation would be faster.

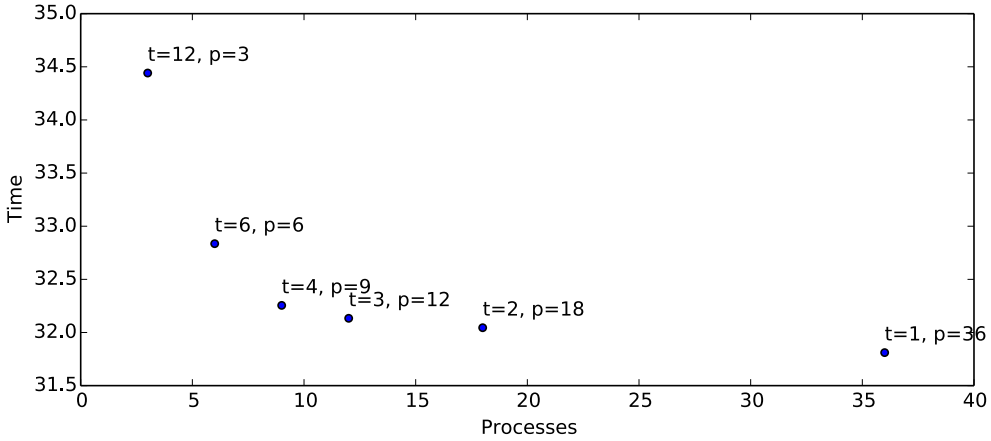

FIGURE 3: Comparison of the hybrid and the pure distributed model timings. t number of threads and p number of processes so that $p * t = 36$ with problem size $n = 2^{14}$ averaged over 30 runs.

n	$P = 36, T = 1$ error	$P = 18, T = 2$ error	$P = 3, T = 12$ error
16	$2.59 * 10^{-3}$	$2.59 * 10^{-3}$	$2.59 * 10-3$
64	$1.63 * 10^{-4}$	$1.63 * 10^{-4}$	$1.63 * 10-4$
256	$1.02 * 10^{-5}$	$1.02 * 10^{-5}$	$1.02 * 10-5$
1024	$6.37 * 10^{-7}$	$6.37 * 10^{-7}$	$6.37 * 10-7$
4096	$3.98 * 10^{-8}$	$3.98 * 10^{-8}$	$3.98 * 10-8$
16384	$3.59 * 10^{-9}$	$3.59 * 10^{-9}$	$3.59 * 10-9$

TABLE 1: Runs with different configurations and problem size $n = 2^{14}$ averaged over 30 runs. The error allways converges as expected.

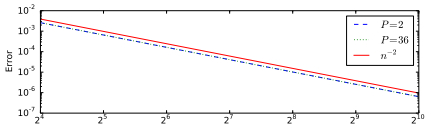

FIGURE 4: Pointwise error with two configurations compared to the projected error bound with problem size $n = 2^{14}$ averaged over 30 runs. The graphs indicate that the solver works correctly.

6 Strategy analysis

6.1 Numerical analysis of the algorithm. We have run the tests with:

$$f(x, y) = -30y^4x(x^5-1)-30x^4*y(y^5-1) \quad (9)$$

which has the solution:

$$u(x, y) = x(x^5 - 1) * y(y^5 - 1) \qquad (10)$$

for Equation 1. We tested the convergence rate of the error to confirm that the error followed the predicted bound $\mathcal{O}(h^2)$. As Figure 4 shows, the error converges correctly. An excempt of the numerical testes that were conducted is presented in Table 1.

6.2 Experimental analysis of the implementation. The solver should have a pointwise error proportional to $\frac{1}{n^2}$ since we are calculating a second order approximation to the Poisson problem. The experimental results presented in Figure 4 shows that the proposed solver follows the convergence on $\mathcal{O}(h^2)$ and thus does not suffer from convergence issues.

6.3 Timed runs. According to the theoretical analysis of the runtime bounds calculated in section 2, the solver is estimated to compute the results in $\mathcal{O}(n^2 \log n)$. In order to confirm that the solver computes the approximation within the analytical bounds, the solver were run 30 times on one process with 12 threads. According to the result presented in Figure 5, the solver evidently converges to a runtime complexity of $\mathcal{O}(n^2 \log n)$.

To reaffirm that the solver's runtime complexity converges correctly, we ran a huge selection of configurations as presented in Figure 6. These runs were not averaged over multiple run instances so they contain a fair amount of noise. By interpreting the graphs in Figure 6 we can observe that when the solver is run with

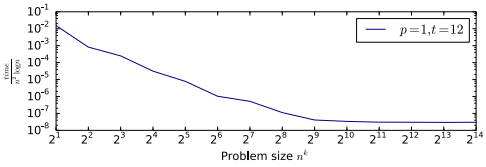

FIGURE 5: Verifying the analytical bounds on the runtime complexity. The solver ran with the a configuration of $P = 1, T = 12$

$p * t >> 1$ on small problem sizes n, the solver is acctually slower that computing the approximation in serial. This is an expected result, since the communication between the processes will dominate the compution time.

6.4 Speed-up and efficiency. The speed-up S_p from multiprocessing a program is defined as the runtime on a single processor $\tau_{p=1}$ divided by the runtime of executing the program on p processors, and the parallel efficiency η is defined as the expected speedup divided by the number of processes used.

$$S_p = \frac{\tau_1}{\tau_p} \tag{11}$$

$$\eta_p = \frac{S_p}{p} \tag{12}$$

The acheived speedup S_p will allways be lower than the naïve expected speedup due to communication overhead between the different computation components. We will therefore expect that the parallel efficiency will drop as we increase the number of processes p utilized to solve the problem.

By restricting the cluster computer to only use a single computation node the required level of communication is reduced and we experienced that the speedup S_p improved. Figure 7 shows how the acheived spedup S_p tightly follows the ideal speedup, when computing the result on a single node with 12 threads.

The divergence between the achieved speedup and the ideal speedup is reduced as the problem set n increases. This may be an indication that the bottleneck of our solver is the communication speed.

6.5 Modifying the function $f(x, y)$. Our proposed solver can solve the Poisson problem for different functions f, by modifying the source code. The change in f will result in a modified version of the G matrix as specified in Equation 4. To specify a new f in the solver, you will have to modify the function *f(double x, double y)* which is specified in the *main.cpp* file.

The rest of the code does not need to be modified, except for the analytical solution u for f if you wish to analyse the error convergence of the calculations. The code for providig parallel computation does not need to be modified.

6.6 Bottlenecks. The runtime of our proposed solution is bounded by the transpose operation and communication speed of the network. Imroving the transpose operation will yield better timing results and will further improve the speedup gained from using the hybrid version of the program.

7 Conclusion

The proposed solver has been evaluated to calculate the correct results according to the theoretical analysis of the error estimates from the analytically solvable seed function. The solver also perform very well when run with a high degree of parallelism as documented in Figure 7. The most efficient speedup was gained when using multiple processors rather than spawing OpenMP threading on a single processor.

The solver is bottlenecked by the necessity to communicate with all other processes during the transposition. This bottleneck is especially dominate when working with small problem sizes, as is illustrated in Figure 6. This is due to the runtime overhead of communicating between the processors.

The both the analytical and the experimental tests showed that the solver is working correctly.

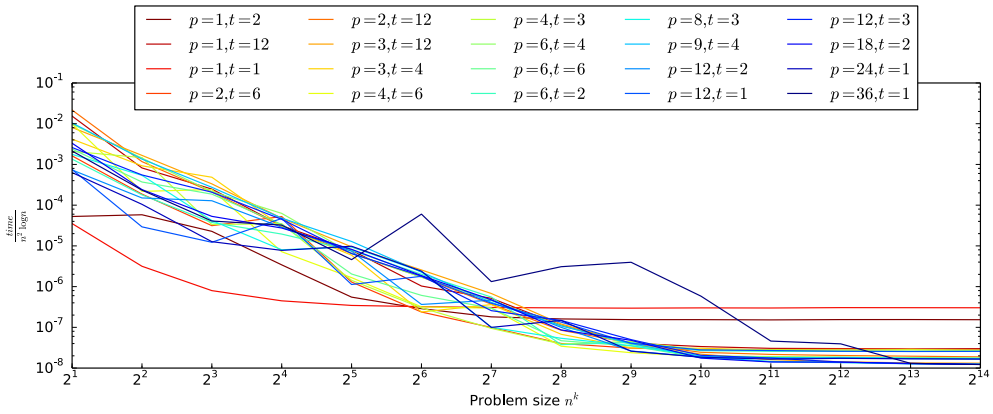

FIGURE 6: Verifying the analytical bounds on the runtime complexity.

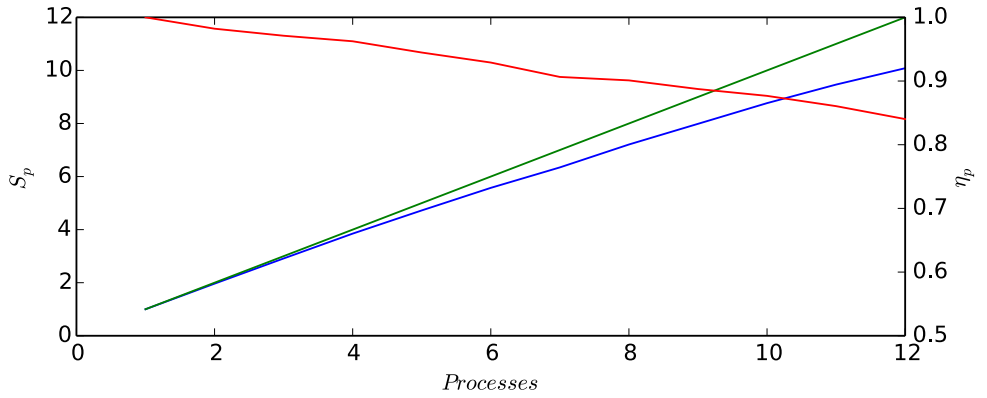

FIGURE 7: Comparing the achieved speedup S_p with the ideal speedup. The acheived speedup is satisfyingly close to the ideal speedup. The red line shows the how the efficiency of increasing parallelization of the program. Run with $P = 1, T = 12$.

8 Future work

8.1 Non-homogeneous Dirichlet boundary conditions. This solution strategy was designed to solve the Poisson problem with homogeneous Dirichlet boundary conditions, which does not require the program to store the boundary values. The MKL library on the other hand requires the boundary to be specified, and therefore the program already offers some support modifying the solver to support non-homogeneous Dirichlet boundary conditions. To solve the Poisson problem with non-zero boundary values we simply have to modify the right hand side of Equation 1.

8.2 Solving for the domain $\Omega = (0, L_x) \times (0, L_y)$. The proposed solver may also handle arbitrarily sized domains. The discretization of our domain follows $x_i = i * h_x$ for $i = 1, ..., n$ where h_x is the step size in the x direction and $h_x = \frac{L_x}{n}$. The step size in the y direction is specified as $h_y = \frac{L_y}{n}$.

In the implemented solver, $L_x = 1$ and $h_x = \frac{1}{n}$. Since $h_x = h_y = h$ in the currently implemented solver, we could combined the calculations in x and y direction into the single Equation 3. In order to solve the Poisson problem in the domain $\Omega = (0, L_x) \times (0, L_y)$ we would instead have to represent $f_{i,j}$ by the equation in Equation 13.

$$f_{i,j} = -\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} - \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2} \tag{13}$$

9 Code snippets

The transpose function is listed in Figure 8.

References

[1] James Cooley, John Tukey, *An algorithm for the machine calculation of complex Fourier series.* https://web.stanford.edu/class/cme324/classics/cooley-tukey.pdf, 1965

[2] Intel®, *Intel Math Kernel Library.* https://software.intel.com/en-us/intel-mkl

[3] NTNU HPC GROUP, *Kongull Hardware.* https://www.hpc.ntnu.no/display/hpc/Kongull+Hardware

```
void transpose(int rank, int size, int n, int* rowsPerProcessor, int* sendAndRecieveCounts, int*
sendAndRecieveOffsets, int* columnOffsets, double* matrix, double* sendBuffer, double* recieveBuffer)
{
    // Pack to send buffer
    #pragma omp parallel for schedule(static)
    for (int i = 0; i < rowsPerProcessor[rank]; i++)
    {
        for (int j = 0; j < n; j++)
        {
            int matrixIndex = i * n + j;
            int sendBufferIndex = j * rowsPerProcessor[rank] + i;
            sendBuffer[sendBufferIndex] = matrix[matrixIndex];
        }
    }

    MPI_Alltoallv(sendBuffer, sendAndRecieveCounts, sendAndRecieveOffsets, MPI_DOUBLE, recieveBuffer,
sendAndRecieveCounts, sendAndRecieveOffsets, MPI_DOUBLE, MPI_COMM_WORLD);

    // Unpack from recieve buffer
    #pragma omp parallel for schedule(dynamic)
    for (int p = 0; p < size; p++)
    {
        for (int i = 0; i < rowsPerProcessor[rank]; i++)
        {
            for (int j = 0; j < rowsPerProcessor[p]; j++)
            {
                int matrixIndex = i * n + columnOffsets[p] + j;
                int recieveBufferIndex = sendAndRecieveOffsets[p] + i * rowsPerProcessor[p] + j;
                matrix[matrixIndex] = recieveBuffer[recieveBufferIndex];
            }
        }
    }
}
```

FIGURE 8: The transpose function