TDT4280: Project 2
Parallalized 2D Poisson Solver

Introduction to Supercomputing

Neshat Naderi

April 19, 2017

Introduction

Poisson Problem

The two dimensional poisson problem is defined as:

$$-\nabla^2 u = f \qquad in \quad \Omega = (0,1) \times (0,1).$$

$$u = 0 \qquad\qquad on \quad \delta\Omega.$$

Where f is the right hand side function, u is the solution and Ω is the domain.
The solution is achieved by descretizing the problem in finite difference grid with $n+1$ points in each spatial direction, where the grid size is $h = \frac{1}{n}$.
The 5-point stencil method is applied in descretizing the Laplace operator(Ω).
Diagonalization and Discrete Sine Transform(DST) methods are choosen to solve the poisson problem. The diagonalization is applied first and then DST. By applying the DST method $O(n^2 logn)$ upperbound of floating points operations is gained.

The Parallel Poisson Solver

The goal of this project is implementing a parallalized 2D-poisson solver. The first step is parallalizing the given source code by MPI and OpenMP. Then the results of elapsed time is analysed and compared with different problem sizes. This report describes the methodology of the poisson solver and its performance efficiency. The goal is to partition the given square matrix of data into vectors and distribute those matrix segements to processes. So that the processes operates in parallel. A serial implementation of the poisson solver is handed out and the task is to implement a parallel version of that solver. The multiprocessing and multithreading strategy is described in more details in this section.

Parallel program using MPI and OpenMP

Modifications in the serial code is started by partitioning a square matrix($M \times M$) into vectors and distributing $\frac{M}{P}$ vectors to each process. P is the number of processes. The syncronization and communication between processes is provided by the message passing interface(MPI). MPI library is imported by including header file `mpi.h`.
In the next step the MPI environment is initialized by a given number of processes (`nproc`). OpenMP is a shared memory multiprocessing API that is used together with MPI. To apply multithreading with OpenMP, the parallel regions(loops) are marked with `#pragma omp parallel for`. So that loop iterations are splitted up among threads and so the threads are run paralleled.

Load balancing

In the case, the number of rows in the matrix is not divisible by the number of processes, the work per process should be balanced. In this situation, some processes is given $\frac{M}{p}$ vectors. Listing 1 shows that the leftover vectors $M \mod p$ are distributed to the last processes. So that they own $\frac{M}{p} + 1$ number of vectors each.

Listing 1: Load balancing

```
// load balancing, partition the matrix in vectors
for (size_t i = 0; i < nproc; i++) {
    local_columns[i] = columns;
    // Distribute the remaining columns to last processes
    if ( rest_columns && i >= last_process_index_start ) {
        local_columns[i] = local_columns[i] + 1;
        rest_columns-- ;
    }
    col_displacements[i] = displacement;
    displacement += local_columns[i];
}

for (size_t i = 0; i < nproc; i++) {
    counts[i] = local_columns[i] * local_columns[rank];
    displs[i] = chunk_size;
    chunk_size += counts[i];
}

if ( rank >= last_process_index_start && m % nproc ) {
    columns++ ;
}
```

Parallel Implementation of Transform Operation

In order to use a distributed memory programming model, the transpose function involves `all-to-all` communication. The transpose operation is parallalized in the source code and is called `parallel_transpose`. In this parallalized version, transpose operation is started by packing the data into sendbuffer. The all-to-all communication during transpose operation is done by `MPI_Alltoallv`. This function takes care of all communication. Each process distributed a segement operates on the part assigned to that process. Then the receiving data from each process is stored in recievebuffer. In the end, the recievebuffer is unpacked and the data is placed into the matrix. After the last step the given matrix is transposed.
The source code is shown in Listing 2.

Listing 2: Parallalized Transpose Operation

```
void parallel_transpose(int rank, int nproc, int m,
                        double *bt, double *b,
                        double *sendbuf, double *recvbuf,
                        int *counts, int *displs,
                        size_t *local_columns, size_t *col_displacements)
{
    #pragma omp parallel for schedule(static)
    for (size_t p = 0; p < nproc; p++) {
        for (size_t col = 0; col < local_columns[rank]; col++) {
            size_t r_index = col * m + col_displacements[p];
            size_t s_index = displs[p] + col * local_columns[p];

        // copy blocks of matrix into the send buffer
        memcpy( sendbuf + s_index, b + r_index, local_columns[p] *
                sizeof(double) );
        }
    }
    MPI_Alltoallv( sendbuf, counts, displs, MPI_DOUBLE, recvbuf,
                   counts, displs, MPI_DOUBLE, MPI_COMM_WORLD );

    #pragma omp parallel for schedule(static)
    for (size_t p = 0; p < nproc; p++) {
        for (size_t col = 0; col < local_columns[rank]; col++) {
            for (size_t c = 0; c < local_columns[p]; c++) {
                size_t r_index = c*local_columns[rank] + displs[p] + col;
                size_t s_index = m * col + col_displacements[p] + c;
                bt[s_index] = recvbuf[r_index];
            }
        }
    }
}
```

Performance Analysis

The parallel program is compiled and run on Lille supercomputer. Different combinations of $n = 2^k$, P and t are tested. The range for process values are $1 \leq P \leq 36$. In Table 1 some timing results are compared. Remaining test results are attached to the source code as TXT and CSV files which contain timings for executed program with cluster nodes of 1 and 2. Where n is the problem size, p is the number of processes, t is the number of threads, τ is the elapsed time, S_p is the speedup and η_p is the parallel efficiency achieved by the program.

n	**p**	**t**	τ (s)	u	S_p	η_p
512	1	1	4.742250e-01	6.249888e-02	-	-
512	4	12	1.085990e-01	6.249735e-02	8.155186e+00	2.038796e+00
512	8	4	2.005280e-01	6.249735e-02	4.416565e+00	5.520707e-01
512	16	8	7.256050e-01	6.249888e-02	1.220561e+00	7.628505e-02
512	36	18	5.074341e+00	6.249888e-02	1.745340e-01	4.848166e-03
1024	1	1	1.911259e+00	6.249972e-02	-	-
1024	2	8	2.153310e-01	6.249800e-02	1.750386e+01	8.751928e+00
1024	4	16	2.678350e-01	6.249781e-02	1.407256e+01	3.518139e+00
1024	16	8	7.882380e-01	6.249972e-02	4.781707e+00	2.988567e-01
1024	36	18	4.950787e+00	6.249972e-02	7.613179e-01	2.114772e-02
2048	1	1	8.084189e+00	6.249993e-02	-	-
2048	2	36	7.216190e-01	6.249981e-02	2.229966e+01	1.114983e+01
2048	3	16	1.002780e+00	6.249991e-02	1.604725e+01	5.349082e+00
2048	16	4	1.145253e+00	6.249993e-02	1.405092e+01	8.781825e-01
2048	4	2	1.347390e+00	6.249993e-02	1.194298e+01	2.985746e+00
2048	36	18	7.040044e+00	6.249993e-02	2.285761e+00	6.349336e-02
4096	1	1	3.439144e+01	6.249998e-02	-	-
4096	4	16	2.649360e+00	6.249995e-02	2.577316e+01	6.443291e+00
4096	12	36	2.726301e+00	6.249998e-02	2.504580e+01	2.087150e+00
4096	18	8	3.152753e+00	6.249998e-02	2.165802e+01	1.203223e+00
4096	2	3	6.361214e+00	6.249994e-02	1.073418e+01	5.367088e+00
4096	36	18	9.399781e+00	6.249998e-02	7.264253e+00	2.017848e-01
8192	1	1	1.457998e+02	6.250000e-02	-	-
8192	8	12	9.800463e+00	6.249999e-02	2.953165e+01	3.691456e+00
8192	8	16	9.939969e+00	6.249999e-02	2.911717e+01	3.639647e+00
8192	12	36	1.009025e+01	6.249997e-02	2.868351e+01	2.390293e+00
8192	36	4	1.565453e+01	6.250000e-02	1.848818e+01	5.135605e-01
8192	36	18	1.711667e+01	6.249999e-02	1.690888e+01	4.696912e-01
8192	4	2	2.051611e+01	6.250000e-02	1.410715e+01	3.526788e+00
8192	4	1	3.711160e+01	6.250000e-02	7.798743e+00	1.949686e+00
16384	1	1	6.174798e+02	6.250000e-02	-	-
16384	8	12	4.076668e+01	6.250000e-02	3.027954e+01	3.784942e+00
16384	16	6	4.088034e+01	6.250000e-02	3.019535e+01	1.887209e+00
16384	8	16	4.126072e+01	6.250000e-02	2.991698e+01	3.739623e+00
16384	6	36	4.144498e+01	6.249999e-02	2.978398e+01	4.963996e+00
16384	12	2	4.310132e+01	6.250000e-02	2.863940e+01	2.386617e+00
16384	36	4	4.914745e+01	6.250000e-02	2.511618e+01	6.976716e-01
16384	4	4	5.177619e+01	6.250000e-02	2.384100e+01	5.960250e+00
16384	2	3	1.125774e+02	6.250000e-02	1.096486e+01	5.482432e+00

Table 1: Timing results of parallel program on *Lille* supercomputer with $n = 2^k$, p processes and t threads. $f(x, y) = 2(y - y^2 + x - x^2)$ and $1 \leq p \leq 36$.

Hybrid Model

The test results in Figure 1 show decreasing amount of elapsing time for $n = 16384$ and comparison of a pure distributed memory program and a hybrid program. When the number of processes are high, both models result in similar wall times. The hybrid model works better than the pure distributed model for most of process size selections. It shows more stability for various p size.

A good timing result is achieved by $P \in \{6, 12, 18\}$ and $t \in \{6, 3, 2\}$ for hybrid model documented in Table 4. In those three cases the job is well distributed and the program performs faster. So the program performs better when the number of processes are more than a single process. But too high process size shows poor performance also. As it is shown in Figure 1 the curve tends to increase for very high p.

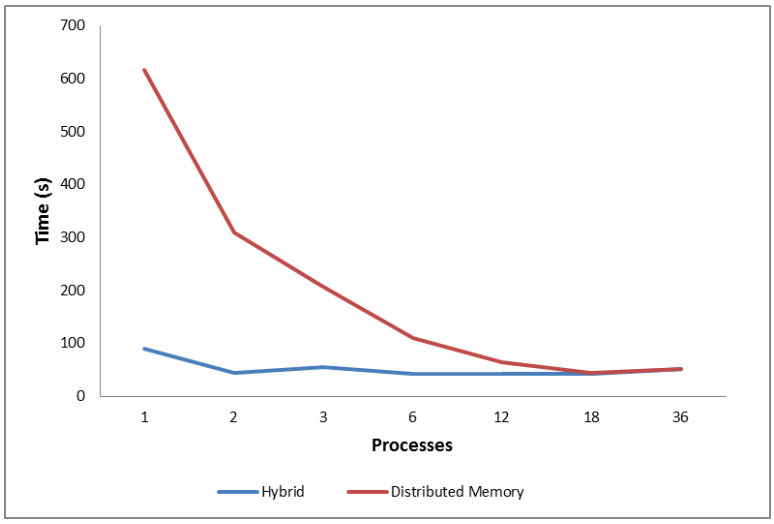


Figure 1: Hybrid model is compared to distributed memory parallel program for $n = 2^{14} = 16384$

p	t	Hybrid, τ	t	Distributed, τ
1	36	8.927343e+01	1	6.174798e+02
2	18	4.441231e+01	1	3.089000e+02
3	12	5.583350e+01	1	2.073900e+02
6	6	4.175794e+01	1	1.102293e+02
12	3	4.173763e+01	1	6.365200e+01
18	2	4.183185e+01	1	4.476399e+01
36	1	5.142002e+01	1	5.142002e+01

Table 2: Hybrid model is compared to distributed memory parallel program with only one thread. For hybrid model $p.t = 36$.

Figure 2 illustrates how the high process size affect the timing results for three different problem size n. The curve's increment tendency is because of the processes' communication overhead. All benchmark results in Table 3 are tested on 2 nodes.

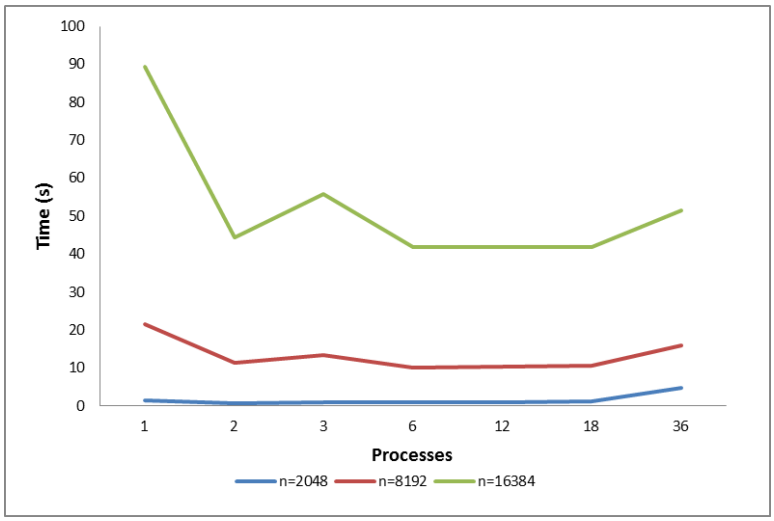


Figure 2: Decreasing of elapsing time in hybrid model.

		τ (**n**)		
p	**t**	2048	8192	16384
1	36	1.375820e+00	2.159030e+01	8.927343e+01
2	18	7.417560e-01	1.123698e+01	4.441231e+01
3	12	1.034756e+00	1.350203e+01	5.583350e+01
6	6	8.255120e-01	1.013905e+01	4.175794e+01
12	3	9.261280e-01	1.024582e+01	4.173763e+01
18	2	1.100374e+00	1.054620e+01	4.183185e+01
36	1	4.865029e+00	1.600261e+01	5.142002e+01

Table 3: Hybrid model's timing results for different n and $p.t = 36$.

In order to illustrate how thread size for each process can influence the performance, a comparison benchmark is done for various thread numbers. Figure 3 shows timing differences for $n = 16384$ and that a single thread shows poor performance compared to multiple threads.

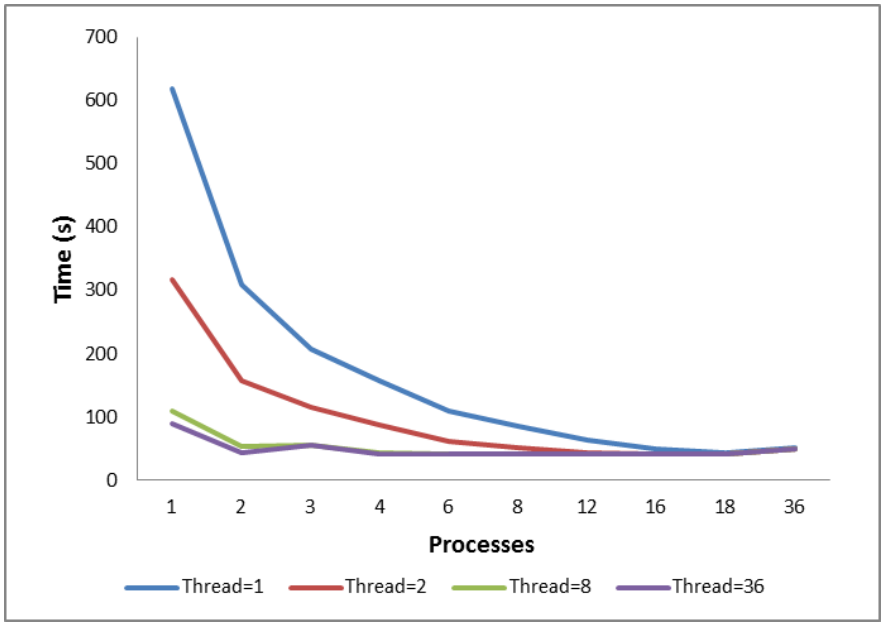


Figure 3: Comparison of elapsing time in hybrid model for fixed problem size $n = 16384$ and differing thread sizes $t \in \{1, 2, 8, 36\}$

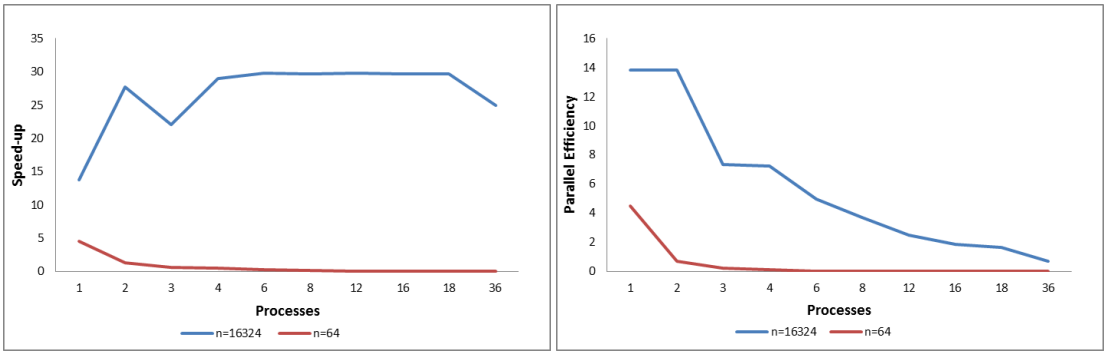
	τ			
p \ **t**	1	2	8	36
1	6.174798de+02	3.171956de+02	1.097055de+02	8.927343de+01
2	3.089000de+02	1.574979de+02	5.416437de+01	4.458945de+01
3	2.073900de+02	1.148720de+02	5.646032de+01	5.604048de+01
4	1.572596de+02	8.693238de+01	4.314681de+01	4.249215de+01
6	1.102293de+02	6.204062de+01	4.178181de+01	4.144498de+01
8	8.669200de+01	5.182738de+01	4.115636de+01	4.154543de+01
12	6.365200de+01	4.310132de+01	4.092585de+01	4.148660de+01
16	5.064756de+01	4.250397de+01	4.140603de+01	4.156513de+01
18	4.476399de+01	4.183185de+01	4.186483de+01	4.164100de+01
36	5.142002de+01	4.930061de+01	4.927885de+01	4.948230de+01

Table 4: Timing results of hybrid model with different threads for $n = 16384$

Speed-up and Parallel Efficiency

The speed-up does not increase as much as the number of processes increases. The overhead of communication between processes does not allow the program perform much more efficiently. According to test results of the solver in Figure 4 that was executed with different number of nodes, processes and threads, it comes out that the speed-up starts decreasing significantly for 36 processes. This decrementing is due to communication overhead between large number of processes.

In the other hand, as the size of n increases the speed-up increases which makes parallel programs useful for large problem sizes. But for small sizes like $n = 64$ the speed-up S_p is too low and the curve is decreasing. Graphs in Figure 4a and 4b are also illustrating how the communication overhead does affect the speedups. The parallel efficiency is decreased quickly when the processes have large values like $p = 36$. So in order to make the program perform more efficiently, the number of processes must be scaled regarding the problem size. As the problem size increases, the achieved speed-ups also increase. But generally very large number of P is not prefered because of unnecessary communication between processors. The situation in Figure 4b is avoided by choosing a suitable value for P.



(a) Speed-ups

(b) Parallel Efficiency

Figure 4: Speedup and efficiency comparison when $n = 16384$ for different thread numbers. 2 cluster nodes.

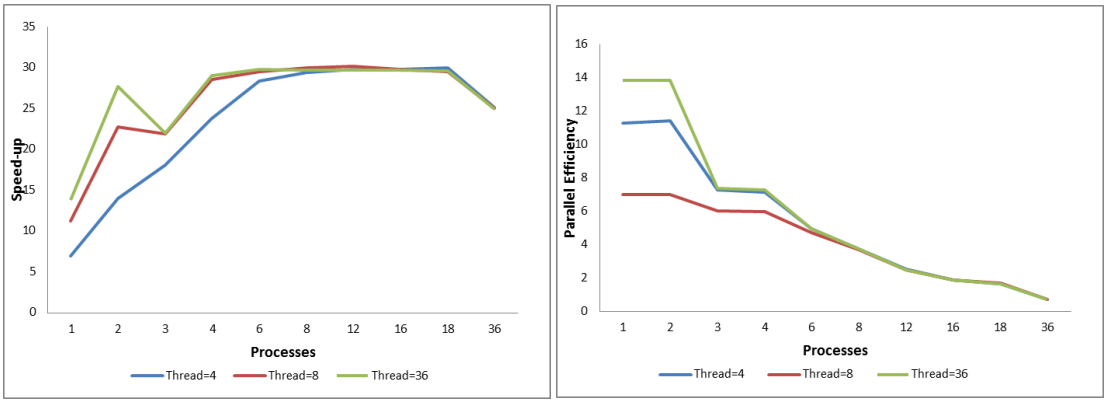
Speedup S_p		
p	$n = 16324$	$n = 64$
1	1.382714e+01	4.523502e+00
2	2.768359e+01	1.364565e+00
3	2.202687e+01	6.701942e-01
4	2.904998e+01	4.800000e-01
6	2.978398e+01	2.540307e-01
8	2.971196e+01	1.405619e-01
12	2.975409e+01	8.395664e-02
16	2.969788e+01	4.839688e-02
18	2.964377e+01	4.123720e-02
36	2.494621e+01	7.089781e-03

Table 5: Speedup comparison between small and big n with $t = 36$

Parallel efficiency η_p		
p	$n = 16324$	$n = 64$
1	1.382714e+01	4.523502e+00
2	1.384180e+01	6.822826e-01
3	7.342288e+00	2.233981e-01
4	7.262495e+00	1.200000e-01
6	4.963996e+00	4.233845e-02
8	3.713995e+00	1.757024e-02
12	2.479508e+00	6.996386e-03
16	1.856117e+00	3.024805e-03
18	1.646876e+00	2.290956e-03
36	6.929504e-01	1.969384e-04

Table 6: Parallel efficiency comparison between small and big n with $t = 36$

Figure 5 illustrates a comparison of different thread sizes for fixed problem size $n = 2^{14}$. Similar behaviors compared to Figure 4 is shown here too.



(a) Speed-ups (b) Parallel Efficiency

Figure 5: Speedup and efficiency comparison when $n = 16384$ for different thread numbers. 2 cluster nodes.

Speedup S_p			
p \ **t**	4	8	36
1	6.981929e+00	1.125191e+01	1.382714e+01
2	1.400780e+01	2.278982e+01	2.768359e+01
3	1.810918e+01	2.186307e+01	2.202687e+01
4	2.384100e+01	2.860921e+01	2.904998e+01
6	2.837910e+01	2.954387e+01	2.978398e+01
8	2.942599e+01	2.999284e+01	2.971196e+01
12	2.977867e+01	3.016177e+01	2.975409e+01
16	2.983605e+01	2.981199e+01	2.969788e+01
18	3.004678e+01	2.948528e+01	2.964377e+01
36	2.511618e+01	2.504921e+01	2.494621e+01

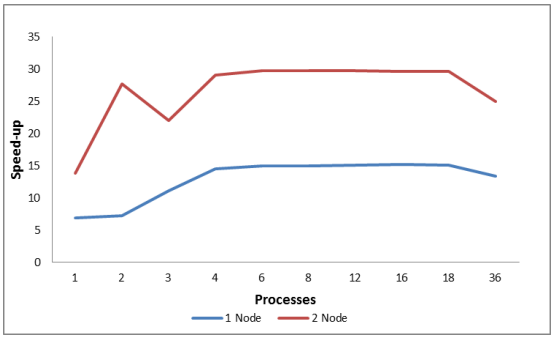
Table 7: Speedup report for $n = 16384$.

Parallel efficiency η_p			
p \ **t**	4	8	36
1	1.125191e+01	6.981929e+00	1.382714e+01
2	1.139491e+01	7.003900e+00	1.384180e+01
3	7.287691e+00	6.036394e+00	7.342288e+00
4	7.152303e+00	5.960250e+00	7.262495e+00
6	4.923978e+00	4.729850e+00	4.963996e+00
8	3.749105e+00	3.678248e+00	3.713995e+00
12	2.513481e+00	2.481555e+00	2.479508e+00
16	1.863249e+00	1.864753e+00	1.856117e+00
18	1.638071e+00	1.669266e+00	1.646876e+00
36	6.958114e-01	6.976716e-01	6.929504e-01

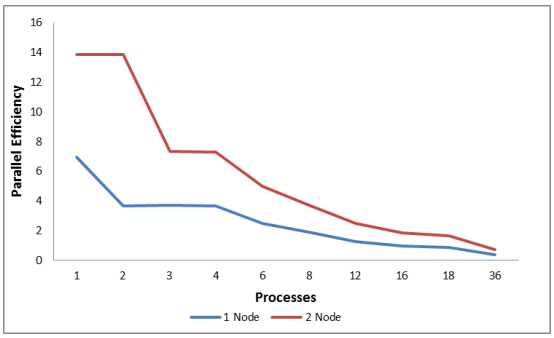
Table 8: Parallel efficiency report for $n = 16384$.

Performance Comparison for Different Node Size

In order to see the performance differences between node sizes, the parallel program is executed on Lille for both 1 node and 2 nodes for various P and fixed $t = 36$ to see if there is any communication overhead when applying 2 nodes. A better performance is achieved when using 2 nodes.



(a) Speed-ups

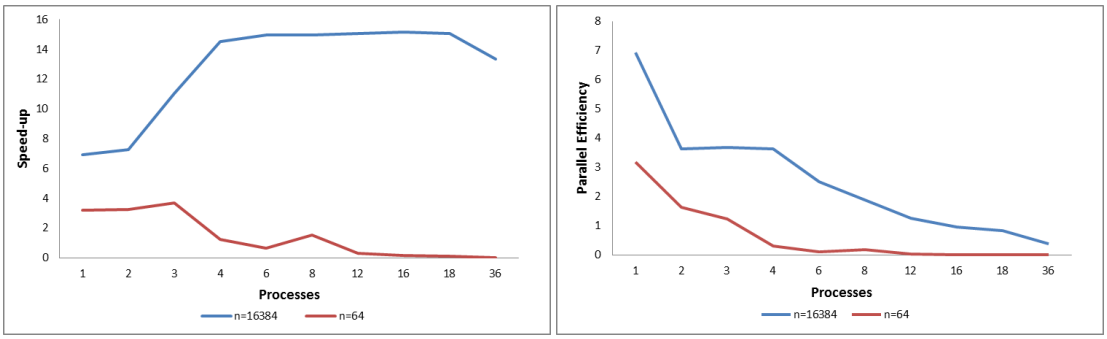


(b) Parallel Efficiency

Figure 6: Comparison of speedup and efficiency when $n = 16384$ for different numbers of cluster nodes.

	Parallel efficiency		**Speedup**	
p \ **node**	1	2	1	2
1	6.930640e+00	1.382714e+01	6.930640e+00	1.382714e+01
2	7.255147e+00	2.768359e+01	3.627574e+00	1.384180e+01
3	1.106023e+01	2.202687e+01	3.686744e+00	7.342288e+00
4	1.451790e+01	2.904998e+01	3.629474e+00	7.262495e+00
6	1.498800e+01	2.978398e+01	2.498000e+00	4.963996e+00
8	1.497342e+01	2.971196e+01	1.871678e+00	3.713995e+00
12	1.506168e+01	2.975409e+01	1.255140e+00	2.479508e+00
16	1.517134e+01	2.969788e+01	9.482085e-01	1.856117e+00
18	1.506886e+01	2.964377e+01	8.371587e-01	1.646876e+00
36	1.337315e+01	2.494621e+01	3.714765e-01	6.929504e-01

Table 9: Parallel efficiency comparison for 1 and 2 cluster nodes with fixed thread size $t = 36$.



(a) Speed-ups

(b) Parallel Efficiency

Figure 7: Speedup and efficiency comparison when $n = 16384$ for different process numbers. 1 cluster nodes.

Verification of Correctness

Figure 8 shows the convergence for function f.

$$f(x,y) = -\Delta u = 5\pi^2 sin(\pi x) sin(2\pi y)$$

The convergence test confirms the correctness of parallel poisson solver implementation. The test results are as expected. In table 10 the maximum pointwise error is calculated for fixed number of processes in each test.

n	Time	Error
16	1.212968e+00	3.208491e-01
32	1.330465e+00	1.785441e-01
64	1.351995e+00	9.374031e-02
256	1.392981e+00	2.426724e-02
512	1.329465e+00	1.220276e-02
1024	3.045029e+00	6.118655e-03
2048	3.907036e+00	3.063645e-03
4096	7.759986e+00	1.532901e-03
8192	2.534212e+01	7.667206e-04
16384	9.371541e+01	3.834278e-04

Table 10: Convergence test shows the error which is decreased to zero as the problem size is increased.$P = 18$, $t = 8$

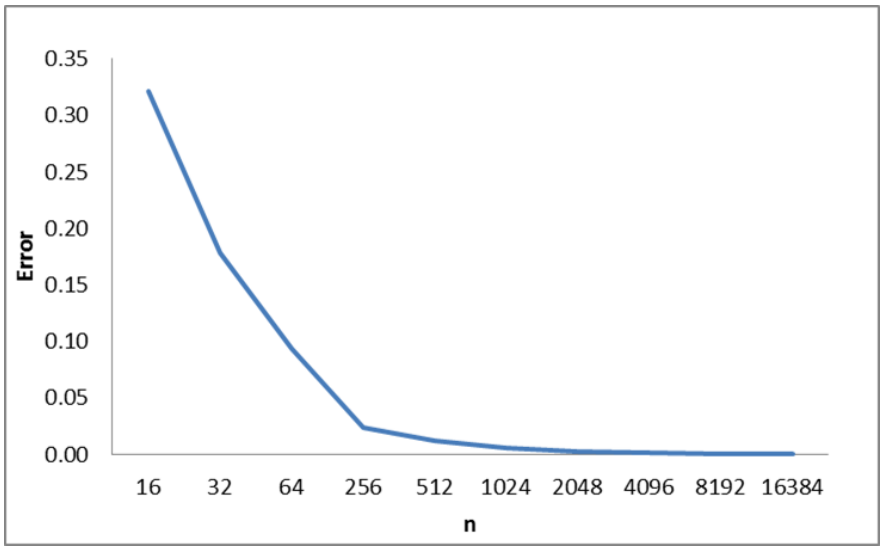


Figure 8: Convergence test showing decreasing of pointwise error when n increases

Disscussions

The source code of Poisson solver must be modified in order to deal with different functions. If function $f(x, y)$ is changed, the only modification in the code should be on the `rhs(real x, real y)`. The convergence test for different f must be implemented separately as well.

Efficiency Improvement

The transpose operation is the part of implementation that slows down the program. It requires a lot of communication between processes. In addition, there are two nested loops in the `prallal_transpose` function. The performance could be more efficient if this function operates faster. Dynamic programming could be tried in order to boost the transpose algorithm.

Non-homogeneous Dirichlet Boundary Conditions

According to the course's lecture notes, the solution for non-homogeneous Dirichlet boundary can be used in order to solve it for this problem. To include the boundary points we have to add a new vector b to the right hand side of the equation to express the system as $Au = g$ and so the left hand side will be exactly the same. So the new equation will be defined as $G = h^2 f + b$.

$$\mathbf{G} = h^2 \begin{bmatrix} f_{1,1} & \cdots & f_{1,n-1} \\ \vdots & \ddots & \vdots \\ f_{n-1,1} & \cdots & f_{n-1,n-1} \end{bmatrix} + \begin{bmatrix} u_{0,0} & \cdots & u_{0,n} \\ \vdots & 0 & \vdots \\ u_{n,0} & \cdots & u_{n,n} \end{bmatrix}$$

The Poisson solver represented in this report assumes that $u = 0$. The program assumption is that every other point out of the given boundary is equal zero . In order to support non-homogeneous Dirchlet boundary conditions where $u \neq 0$, the boundaries should be specified and stored. So the points out of its domain must be checked in order to find non-zero points. This checking operation could be done within the right hand side function. Modifications in `rhs` function could improve the program to deal with non-homogeneous Dirchlet Boundary conditions.

Rectangular Matrices

The parallel solver allows only square matrices. The source code could be improved in a way that it operates on rectangular matrices too. The length in both x-direction and y-direction in this project is defined as $L = 1$. Consequently $h = \frac{1}{n}$. One improvement could be assuming distinct step size in each direction. By defining h_x and h_y the problem is easily solved.

$$h_x = \frac{L_x}{n}$$

$$h_y = \frac{L_y}{n}$$

So the domain would be as following,

$$\Omega = (0, L_x) \times (0, L_y)$$

.

Appendix

Environment and Compiler Information

Modules
openmpi/2.0.1
gcc/6.3.0

cmake version 2.8.12.2

Compiling Environment	
Kernel name	linux
Nodename	lillelogin1.foreman.hpc.ntnu.no
Kernel release	3.10.0514.10.2.el7.x86_64
Kernel version	#1 SMP Fri Mar 3 00:04:05 UTC 2017
Machine hardware	x86_64
Processor type	x86_64
hardware platform	x86_64
Operating system	GNU/Linux

Table 11: Compiler details

CPU Information

Lille CPU Node Information	
processor	0
vendor_id	GenuineIntel
cpu family	6
model	79
model name	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
stepping	1
microcode	0xb00001d
cpu MHz	1199.945
cache size	25600 KB
physical id	0
siblings	10
core id	0
cpu cores	10
apicid	0
initial apicid	0
fpu	yes
fpu_exception	yes
cpuid level	20
wp	yes
flags	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflushdts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch ida arat epb pln pts dtherm intel_pt tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdseed adx smap xsaveopt cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local
bogomips	4400.30
clflush size	64
cache_alignment	64
address sizes	46 bits physical, 48 bits virtual
power management	

Table 12: Node information for processor 0

Memory Details

	total	used	free	shared	buffcache	available
Memory	65689948	987516	62491276	68360	2211156	64007736
Swap	524284	0	524284			

Table 13: Memory information

procs		memory				swap		io		system		cpu				
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa	st
0	0	0	62490864	216252	1994904	0	0	0	0	0	2	0	0 100	0	0	

Table 14: Virtual machine information

References

http://thebb.github.io/TMA4280/notes.pdf