Problem set 6 - Shared memory computing and distributed memory computing.

Torje Hoås Digernes & Einar Johan Trøan Sømåen

April 16, 2013

1 Introduction

Poisson's equation is a partial differential equation with broad utility in electrostatics, mechanical engineering and theoretical physics.[**?**]

$$\delta^2\psi = f \tag{1}$$
$$\tag{2}$$

We are considering this problem on a scale from 0 to 1 in both x and y direction with homogenouos dirichelet conditions amongst the edges, set to zero in our case.

2 Solution strategies

Given that we have a supercomputer at our hands we could be tempted to just throw processing power on the problem until it disappears, but a simple solver based on Elementary linear algebra would be very slow compared to other solutions. A solver using gaussian elimination or LU-factorisation would have a vector of unknowns being n^2 long and require n^6 floating point operations. Given that each unknown only depends on its neighbours we have a banded matrix and can shave off n^2 and achieve the same result in n^4.

Using properties that is in our stencil, our discretisation of the Laplace operator, is itself transposed so we can use a three-point stencil: T, PU = TU + UT, which is the same as we use in two dimensions, both ways, just swapping what side we multiply with, then diagonalising T, putting us at $n3$ floating point operations. A further improvement on this method can be done replacing a vector matrix product with a Fast sine transform, a transpose and a fast inverse sine transform landing us at $\log(n) \cdot n^2$. We are not attempting to further explain as we did not quite understand the diagonalisation part.

3 The program

3.1 Components of a solution

Our program consists of three components that come together to produce a results to the problem at hand, a matrix that holds the actual data, and some necessary metadata, like row/column-count, a transpose-function for this matrix, that utilizes MPI for communication, and finally the provided (inverse) fast sine transform-functions.

3.1.1 The matrix

We can set the matrix as the minimum memory we need as we do not attempt to compress the information. The matrix is square and this results in it using n times the space a variable occupies. After finalising we saw that we should have linked our struct directly to the MPI session as it never is used without it and that would make the code somewhat more neat.

3.1.2 MPI and transposition

MPI is only used to infer the local dimensions of the matrix and more directly to do the communication during the transpose operation. We use what we consider the minimum practical space required for this, auxillary memory being the same size as the area where the matrix holds the data. To minimise the connections needed we serialise the data before sending so that all data going from process A and process B is in one sequence. After the data have been received in a process it will do a transpose locally, since it must be done at one point and after the transport step is the most convenient point to do it.

Technically we could have managed with less auxiliary memory, but we believe this would have resulted in longer running time, because we would need to do parts of the transposition before we can continue the communication and thus add much more complexity.

3.1.3 fast sine transform and inverse

As the fortran file was not quite readable for us, we neatly wrapped up the `fst_()` and `fstinv_()` functions in functions which allocates the auxillary memory needed and dispatches the function which does the math. These

wrappers are trivially parallelisable using *openMP*, the one thing to watch out for is the part where each paralell need its own auxillary space. Paralellising uses n times the size of a double for each parallel run at the same time, a low price to pay for the speed increase.

3.2 Putting it all together

By looking at the original program and figuring out what each part did we mapped that to what we had implemented and put it together the same way, just using our parallelised versions. The result looks more or less like this list, except for timers that we included.

- Initialise the matrix
- Initialise the eigenvalue vector to our operator
- apply the fast sine transform on the matrix
- transpose the matrix
- apply the fast inverse sine transform
- divide the matrix elements by the eigenvalues of the operator
- sine transform
- transpose the matrix
- inverse sine transform to get the answer

4 Errors

Subsituting the formula in place of f and comparing the result with the analytical answer yields us a difference we can compare to the expected convergence of the method. Table 1 shows that we have the convergence that we were after. To achieve this only the populate function and minus/campare funtion needed to be changed. As the bounds and edge conditions were left unchanged nothing more needed to be changed.

Table 1: Error in the differents resolutions lining up where we expect them to be. Multiplying the absolute error with the relative error correction, which matches a quadratic convergence in error, below gives near identical results for all resolutions. The function that were set in were -5 $\pi \sin(\pi x) \sin(2\pi y)$ and it was compared to $\sin(\pi x) \sin(2\pi y)$.

	2048	1024	512	256	128	64
abs. error	6.6671e-07	2.6668e-06	1.0667e-05	4.2670e-05	1.7069e-04	6.8297e-04
rel. correction	1024	256	64	16	4	1
	6.8271e-04	6.8271e-04	6.8272e-04	6.8273e-04	6.8278e-04	6.8297e-04

5 Non-homogeneous Dirichelet boundary

We are going to hazard a guess that we can make a sloping surface between the boundaries and deduct the corresponding value from the matrix, resulting in us returning to a homogeneous Dirichelet boundary equal to 0, apply this solution to problem, and afterwards add the surface again.

6 Extension in both directions

As the h is bound to the steplength, changing the total length from 1 to L_x and L_y changes how we compute it, we would seemingly be getting to hs, h_y and h_x. We do also believe that this would have an effect on the eigenvalue vector, giving us 2, one for each direction.

7 Results

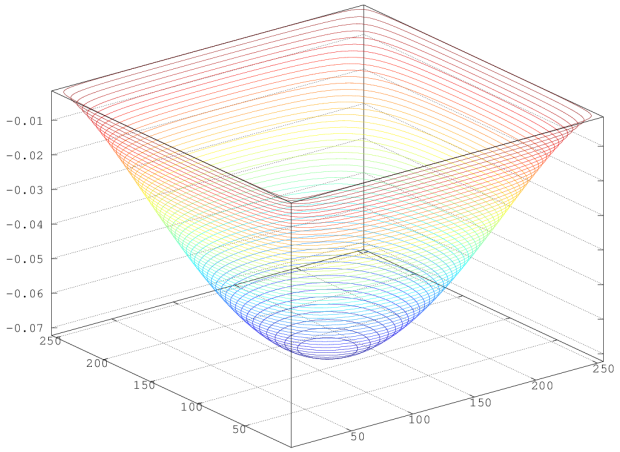

Figure 1: Putting the steplength squared,h^2 , into the matrix yielded this pyramidlike figure.

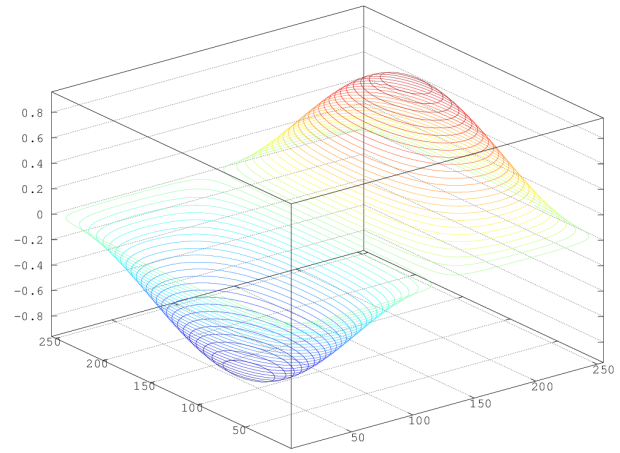

Figure 2: Seeding the matrix with one of the suggested functions, $(sin(\pi * x) \cdot sin(2\pi * y)) \cdot 5\pi^2$, multiplied by the steplength squared, yields this figure.

7.1 Speed

We did some testing with various problem sizes to see how fast our program ran, then we extrapolated the expected run time by using:

$$k = \frac{n_0^2 * \log n_0}{t} \tag{3}$$

$$t_n = \frac{n^2 \cdot \log n}{k} \tag{4}$$

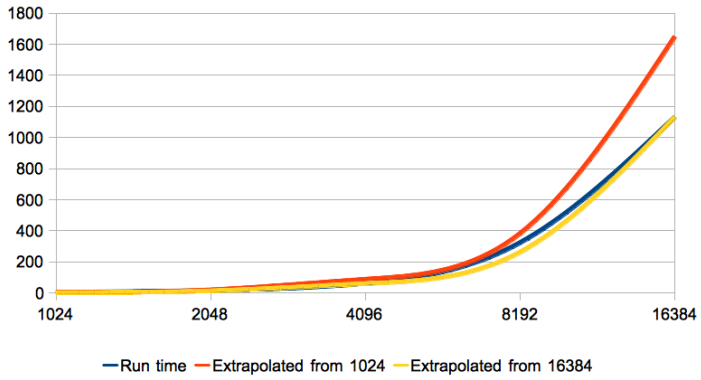

Figure 3: Run time for various problem sizes, compared to their extrapolation

N	Run time	Extrapolated from 1024	Extrapolated from 16384
1024	4,6	4,6	3,16
2048	16,76	20,24	13,89
4096	62,8	88,32	60,59
8192	323,67	382,72	262,55
16384	1130,97	1648,64	

Table 2: Run time compared to extrapolated run time

To show a n^2·log n extrapolation from the actual run time, as shown in figure 3, where we did an extrapolation from the run time of a problem set of size 1024 and up, and a similar extrapolation from the run time of a problem set of size 16 384. This shows that our runtime scales within the expected

values, although the extrapolation looks closer when basing it on 16 384 than when basing it on 1024. This is partially because the smaller problem sets fit better into cache, but also because the actual run time dominates less than the constant-factors at these sizes.

We also did a comparison with regards to how our program scales with the amount of processes it is run with, as shown in figure 5, our run time does indeed go down, with an odd increase in speed between 6 processes and 9 processes.

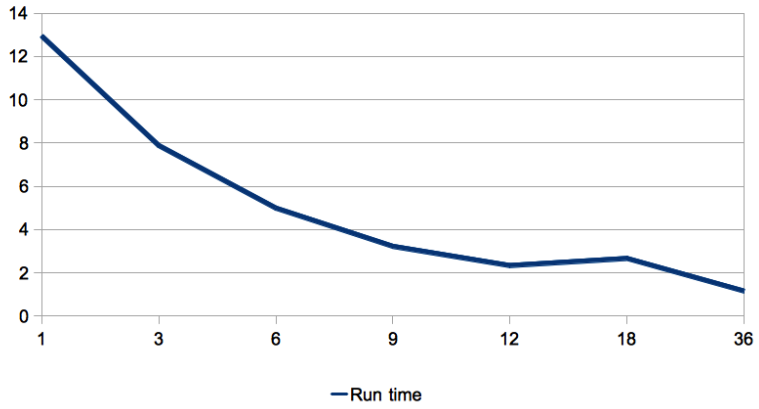

Figure 4: Run time for problem size 2048 with varying process-amount

Similarly we did a comparison for problem sizes of 2048 (shown in Figure 4, which showed a bit smoother a graph, mainly for the same reasons that the extrapolation from 1024 was weird, namely that we are quite a lot closer to the run times where the constant factors weigh more than the growth in problem size. This run also showed a similar local peak in the graph, around 18. The peaks shown in both these graphs might stem from variance in which node on Kongull runs our task, as, while the Kongull-home page does state that it is a homogenous cluster, results and rumour seems to indicate that it is at least in part heterogenous, as some nodes seem to give a doubling in floating point operation-speed compared to other nodes. Rerunning the tests provided different results, which leads us to thinking that the rumours are true. Either way, the results shown here are all from the same batch-job, prioritizing consistency before averaging out such errors.

A similar run was also run for problem sizes of 8192 (Figure 6).

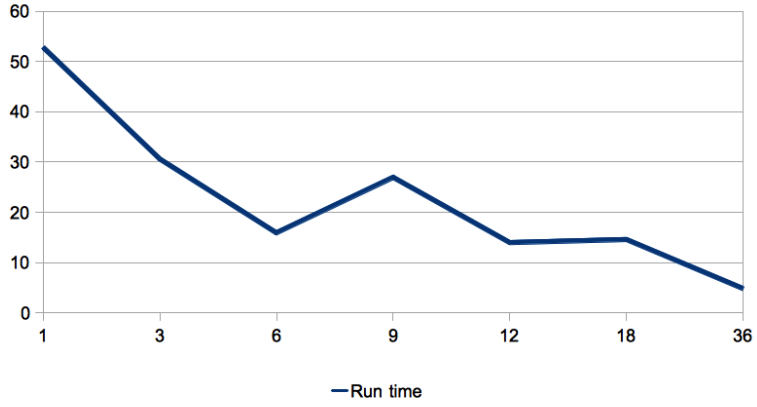

Figure 5: Run time for problem size 4096 with varying process-amount

NumProcs	Speed	SpeedUp	Parallell Efficiency
1	12,94	1	1
3	7,88	1,6421	0,547
6	4,99	2,5932	0,432
9	3,22	4,0186	0,44651
12	2,33	5,5536	0,46280
18	2,66	4,8647	0,27026
36	1,15	11,2522	0,31256

Table 3: Speedup and parallell efficiency for problem size 2048

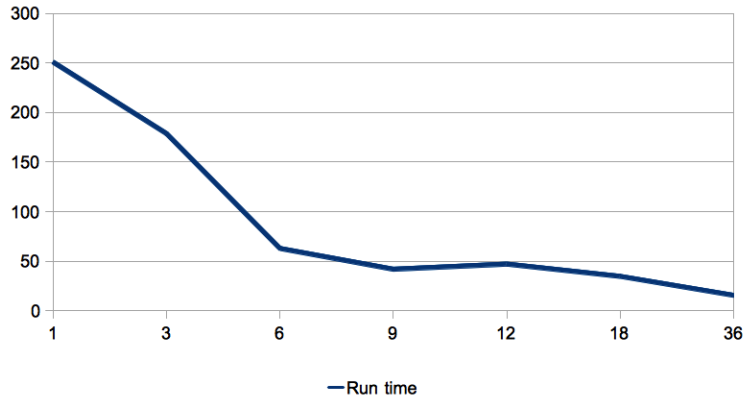

Figure 6: Run time for problem size 8192 with varying process-amount

NumProcs	Speed	SpeedUp	Parallell Efficiency
1	52,78	1	1
3	30,59	1,7254	0,575
6	15,4	3,3195	0,55325
9	26,95	1,4584	0,21760
12	13,98	3,7754	0,31462
18	14,59	3,6175	0,20097
36	4,84	10,7935	0,24982

Table 4: Speedup and parallell efficiency for problem size 4096

NumProcs	Speed	SpeedUp	Parallell Efficiency
1	250,8	1	1
3	178,74	1,4032	0,46772
6	62,89	3,9879	0,66465
9	41,92	5,9828	0,66476
12	47,12	5,3226	0,44355
18	34,74	7,2193	0,40107
36	15,58	16,098	0,44715

Table 5: Speedup and parallell efficiency for problem size 8192

8 Utilities, Compilers and Hardware

8.1 Libraries

Our program needs to be linked against the following libraries

- `libmpi` - to get MPI-functionality
- `libm` - to get the necessary math-functionality.

8.2 Compilers

On Kongull we used the following compilers to compile our program:

- `GCC 4.7.0`
- `GFortran 4.7.0`

Both of which had to be activated explicitly with `module load gcc/4.7.0`.
Locally we also did some testing, using these compiler-versions, with all warnings enabled to make sure that our code was as well written and portable as possible:

- `GCC 4.7.2`
- `GCC 4.2.1 (Apple LLVM)`
- `Clang 3.0`
- `Clang Apple LLVM 4.2 (based on LLVM 3.2svn)`
- `Gfortran 4.5.4`
- `Gfortran 4.7.2`

8.3 Kongull

The distributed memory computer we ran our program on is called Kongull, it consists of 93 compute nodes that each have the following specs[1]:

[1] `http://docs.notur.no/Members/hrn/kongull.hpc.ntnu.no/kongull-hardware-1/hardware`

- 24/48 GiB RAM
- 2x6 Core AMD Opteron 2431, each with the following specs:
  - 2.4 GHz
  - 6 x 128 L1 Cache
  - 6 x 512 KiB L2 Cache
  - 6 MiB L3 Cache

Each of these nodes has 2 CPUs, that share memory, thus we enjoy the combination of shared memory computing AND distributed memory while running on this cluster.

As a side note, the hardware-listings we read listed the machine as being homogenous, while our results differed enough that we are led to believe that Kongull is atleast somewhat heterogenous in it's CPU-setup.

9 Discussion

Our program seems to have some variance in it's runtime, timing the various processes show values between 0.5 and 9 seconds, this stems from the fact that the processes are using blocking communication for the transpose, so the slowest process will delay the faster processes at these points, effectively creating 2 bottlenecks in our runs (one for each of the matrix transposes).

Our program is dominated by the (inverse) fast-sine-transform-calls runtime-wise, which we have little possibility to optimise any further. We also use a large amount of small functions to do the various tasks that are required to set up and manage the matrix, however, none of these are close to any tight loops, and most of them will be optimised by the compiler so that inlining this functionality would not create any further runtime-benefits, but instead make the code base less manageable.

The decision to wrap the fortran-code into C-functions should also not have any effect on the runtime, for the same reason explained above, the functions will be link-time-optimised afterwards anyhow.

A Appendix: Transpose-source code

A.1 mpiMatrix_transpose

```
void mpiMatrix_transpose(struct mpiMatrix *matrix, struct mpi_com *uplink) {
  mpiMatrix_serialiseForSending(matrix, uplink);
  int *SRcounts = mpiMatrix_genCounts(matrix, uplink);
  int *SRdispl = mpiMatrix_genDispl(uplink, SRcounts);

  MPI_Alltoallv(matrix->data, SRcounts , SRdispl, MPI_DOUBLE,
                matrix->aux, SRcounts, SRdispl, MPI_DOUBLE,  uplink->comm);
  mpiMatrix_swapDataAux(matrix);
  mpiMatrix_deserialiseAfterReception(matrix);

  free(SRcounts);
  free(SRdispl);
}
```

A.2 mpiMatrix_serialiseForSending

```
void mpiMatrix_serialiseForSending(struct mpiMatrix *matrix,
                                   struct mpi_com *uplink) {
  size_t serialIndex = 0;
  for (size_t process = 0;
       process < (matrix -> height % uplink->nprocs);
       process++) {
    size_t startindex = 0;
    size_t vectorLength = (matrix -> height / uplink->nprocs + 1);
    for (size_t column = 0 ; column < matrix -> widthLocal ;
         column ++) {
      size_t matrixIndex = column * matrix -> height +
                           process * vectorLength + startindex;
      memcpy(& matrix->aux[serialIndex] , & matrix->data[matrixIndex],
             vectorLength * sizeof(double));
      serialIndex += vectorLength;
    }
  }
  for (size_t process = (matrix -> height % uplink -> nprocs);
       process < uplink->nprocs;
       process++) {
```

```
    size_t startindex = matrix -> height % uplink -> nprocs ;
    size_t vectorLength = (matrix -> height / uplink->nprocs);
    for (size_t column = 0 ; column < matrix -> widthLocal ;
         column ++) {
      size_t matrixIndex = column * matrix -> height +
                           process * vectorLength + startindex;
      memcpy(& matrix->aux[serialIndex] , & matrix->data[matrixIndex],
             vectorLength * sizeof(double));
      serialIndex += vectorLength;
    }
  }
  mpiMatrix_swapDataAux(matrix);
}
```

A.3 mpiMatrix_deserialiseAfterReception

```
void mpiMatrix_deserialiseAfterReception(struct mpiMatrix *matrix) {
  for (size_t index = 0;
       index < matrix -> height * matrix -> widthLocal ;
       index ++) {
    size_t newcoord = (index / matrix -> widthLocal) +
                      (index % matrix->widthLocal) * matrix->height;
    matrix->aux[newcoord] = matrix->data [index];
  }
  mpiMatrix_swapDataAux(matrix);
}
```