Performance Studies for the Two-Dimensional Poisson Problem Discretized by Finite Differences

Jonas Schäfer

Fachbereich für Mathematik und Naturwissenschaften,

Universität Kassel

Abstract

In many areas, amongst others in Mathematics, one is supposed to deal with very large problems. In order to compute some of those problems fast processors and a lot of storage space are needed. So the idea is to focus on using different processes and dividing the problem into a lot of subproblems. This report describes the Poisson problem, a way to solve it numerically and introduces an implementation of a parallel algorithm for that. As a linear equation solver I chose the Conjugate Gradient (CG) Method. As a programming languange C and the MPI package were used. Tara of the High Performance Computing Facility, University of Maryland, Baltimore County (UMBC) was the cluster on which the studies were performed. This report also compares the studies with other reports to confirm correctness of the algorithm and efficient programming. As a result this special problem can be solved well with parallel computing resulting in good speed-up.

1 Introduction

Parallelizing algorithms can be a useful method to solve very large problems in a short amount of time. The idea behind it is simple: By using p processes, the parallel algorithm should optimally be p times quicker than the serial algorithm.

Since too much communication between processes is the reason why parallel algorithms may perform slowly, parallelizing is not always an option. The Jacobi-Method and the Gauss-Seidel-Method are good examples for linear equation solvers (both described in [3]). On the one hand, the computation of each vector component is independent of the other components in the Jacobi-Method, whereas in the Gauss-Seidel-Method the single components need to be computed in a specific order because of their dependencies. As a result it is fairly easy to find an algorithm for the Jacobi-Method, not for the Gauss-Seidel-Method.

However in this report a parallel version of the Conjugate Gradient (CG) Method will be presented in order to solve linear equation systems. Since we are looking at the discretization of the Poisson problem with Dirichlet boundary conditions, the Finite Difference Method leads to a system of linear equations with a symmetric, positive definite, sparse, highly structured matrix. Thus the CG Method is also an option as a linear equation system. The number of iterations and the performance of the CG Method in comparison to the Jacobi-Method and the Gauss-Seidel-Method suggest to actually use the CG Method. The goal was to find a parallel algorithm solving the Poisson problem with almost optimal seed-up qualities.

The studies for this report were performed at the University of Maryland, Baltimore County (UMBC) under the supervision of Dr. Gobbert as part of a semester abroad.

Technical details of the computing environment: The Cluster tara

Tara is an 86-node distributed-memory cluster at UMBC. It was purchased in 2009 by the UMBC High Performance Computing Facility (`www.umbc.edu/hpcf`) and it comprises 82 compute, 2 develop, 1 user and 1 management nodes. Each node features two quad-core Intel Nehalem X5550 processors (2.66 GHz, 8 MB cache), 24 GB memory and a 120 GB local hard drive, thus up to 8 parallel processes can be run simultaneously per node. All nodes and the 160 TB central storage are connected by an InfiniBand (QDR = quad-data rate) interconnect network. The cluster is an IBM System x iDataPlex. An iDataPlex rack uses the same floor space as a conventional 42 U high rack but holds up to 84 nodes, which saves floor space. More importantly, two nodes share a power supply which reduces the power requirements of the rack and make it potentially more environmentally friendly than a solution based on standard racks. For tara, the iDataPlex rack houses the 84 compute and development nodes and includes all components associated with the nodes in the rack such as power distributors and ethernet switches. The user and management nodes with their larger form factor are contained in a second, standard rack aling with the InfiniBand switch. The PFI 9.0 C compiler has been used to create the executables which were utilized in the report. The results are based on the MVAPICH2 implementation of MPI. Amongst other adjustments, the user can decide if his job uses nodes exclusively or if he accepts to share nodes. Sharing nodes means that different jobs can use different processes on one node at the same time.

2 The Poisson Problem

We consider the Poisson Equation in 2D with Dirichlet boundary conditions

$$\begin{aligned} -\Delta u(x,y) &= f(x,y) && \text{in } \Omega, \\ u(x,y) &= 0 && \text{on } \partial\Omega, \end{aligned}$$

on the unit square domain $\Omega = (0,1) \times (0,1)$ with $\partial\Omega$ being defined as the boundary of Ω. Δu denotes the Laplacian Operator $\Delta u(x,y) = u_{xx}(x,y) + u_{yy}(x,y)$.

The finite difference method approximates the solution at certain points, therefore we take a look at $x_i = ih, y_i = ih$, , $i = 1, \ldots, N$, with $h = \frac{1}{N+1}$ and get the points (x_i, y_j) for $i, j = 1, \ldots, N$. This results in the mesh

$$\Omega_h = \{(x_i, y_j) | i = 0, \ldots, N+1, j = 0, \ldots, N+1\} \subset \Omega \cup \partial\Omega$$

with $(N+2)^2$ points. u_{ij} denotes the approximation of $u(x_i, y_j)$. Because of the boundary condition there are only N^2 unknown approximations.

Using approximations for the second derivatives leads to

$$-\Delta u(x_i, y_j) \simeq \frac{-u_{i,j-1} - u_{i-1,j} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1}}{h^2}$$

for $i, j = 1, \ldots, N$. This can be transformed into a system of linear equations with matrix

$$A = \begin{pmatrix} \tilde{A} & -I & & \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & \tilde{A} \end{pmatrix} \in \mathbb{R}^{N^2 \times N^2},$$

with

$$\tilde{A} = \begin{pmatrix} 4 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix} \in \mathbb{R}^{N \times N} \text{ and } I = \begin{pmatrix} 1 & & & \\ & \ddots & & \\ & & \ddots & \\ & & & 1 \end{pmatrix} \in \mathbb{R}^{N \times N}$$

and vector

$$u = (u_1, \ldots, u_{N^2})^T, \; u_{i+Nj} = u_{ij}.$$

The matrix A is symmetric, positive definite, sparse and highly structured. There are several options to solve this linear equation system. A solver which takes advantage of those qualities would be the CG-Method for symmetric, positive definite matrices. The other properties of the matrix lead to the idea of an efficient, matrix-free implementation of the CG-Method for this particular equation system. Which means, instead of implementing the matrix A, we use a routine in order to compute the matrix-vector product.

So the question now is, how to implement a parallel version of the CG-Method. Since we don't need the actual matrix, we can focus on the vectors. If p processes are being used, a vector of size $n = N^2$ is divided into p sub-vectors with size $local\_n = \frac{n}{p}$. We assume that n is a multiple of p. Thus the first process has access to the subvector with components $1, \ldots, local\_n$, the second process has access to the subvector with components $local\_n + 1, \ldots, 2local\_n$ and so forth.

Additions and scalar multiplications of vectors can be performed locally, no communication between processes is necessary. That leaves three tasks:

- dot-product
- norm
- matrix-vector mulitplication

Obviously all of these problems require communication between the processes. The dot-product is the sum of every component of the first vector multiplied with the corresponding component of the second vector. The multiplications of components and $local\_n - 1$ additions can be performed locally. Finally we receive p different dot-products of the sub-vectors. Adding all these dot-products results in the actual dot-product of the original vectors. The MPI command MPI_Allreduce does exactly that and additionally sends the result to every process.

The norm is just the squareroot of the dot-product, so there is no use in implementing a specific routine.

The matrix-vector multiplication is the most complicated task to parallelize. Every subvector on each process has $local\_n = \frac{n}{p} = N\frac{N}{p}$ components describing a certain part of the 2D mesh, specifically $\frac{N}{p}$ number of rows of size N. This helps to visualize the amount of comminucation necessary. At most five mesh points can play a role in order to compute one component of the solution. In terms of the mesh, every point itself and his neighbours define the corresponding component of the solution. Every neighbour with respect to x is stored locally (inside one row), but the neighbour with respect to y is in the previous, respectively next row. Thus there are two rows per process which need data from a row stored on a different process: The first and the last row. At the same time, these rows play an important part on other processes. All in all, with few exceptions, every process has to send and receive two vectors of size N. Therefore every process stores four vectors for the CG-Method and two vectors to compute the parallel matrix-vector multiplication.

The most efficient way would be to communicate between processes while computing, since most of the needed data is stored locally. The non-blocking communication commands MPI_Isend and MPI_Irecv are perfect to use in this context. More information about the MPI commands can be found in [1].

The numerical theory (in [2] and [6]) predicts that the error will converge as $\|u - u_h\|_{L^\infty(\Omega)} \leq Ch^2$, as $h \to 0$, where C is a constant independent of h. As a result we get

$$\frac{\|u - u_{2h}\|_{L^\infty}}{\|u - u_h\|_{L^\infty}} \approx \frac{C(2h)^2}{Ch^2} = 4$$

by using half the step size. In other words, the error is supposed to get four times smaller with only half the step size. Thus this ratio is a practical criterion to prove the correctness of the implementation of the algorithm.

3 Convergence Study for the Model Problem

In order to test the algorithm of the Finite Difference Method, respectively the CG Method, we consider the Poisson problem described in Section 2 on the unit square $\Omega = (0,1) \times (0,1)$ with right-hand side function

$$f(x_1, x_2) = (-2\pi^2)(\cos(2\pi x_1)\sin^2(\pi x_2) + \sin^2(\pi x_1)\cos(2\pi x_2)).$$

The function $u(x_1, x_2) = \sin^2(\pi x_1)\sin^2(\pi x_2)$ is the known solution for this particular problem.

By using different mesh sizes, we can confirm the numerical theory regarding the Finite Difference Method and analyze the performance of the CG Method. Concerning the CG Method, as an initial guess we use the zero vector. The method terminates when the Euclidean vector norm of the residual is smaller than 10^{-6}.

Table 3.1: ν and N describe the mesh size, n the degrees of freedom, E_ν the finite difference error and R_ν an error fraction.

ν	$N = 2^\nu$	$n = N^2$	$E_\nu = \|u - u_h\|_{L^\infty(\Omega)}$	$R_\nu = \frac{E_{\nu-1}}{E_\nu}$
5	32	1,024	3.013e-3	N/A
6	64	4,096	7.781e-4	3.87
7	128	16,384	1.976e-4	3.94
8	256	65,536	4.980e-5	3.97
9	512	262,144	1.249e-5	3.99
10	1024	1,048,576	3.127e-6	3.99
11	2048	4,194,304	7.802e-7	4.01

Table 3.1 and Table 3.2 are the results of various one-process runs of the parallel code. Table 3.1 focuses on the error, whereas Table 3.2 shows performance results. Regarding Table 3.1, ν and N describe the mesh size ($N = 2^\nu$), n is the number of degrees of freedom, E_ν is the finite difference error and R_ν a fraction of errors. The error clearly decreases if finer mesh resolutions are being used. The last column of Table 3.1 confirms that, showing that the implementation of the algorithm computes as expected and the Finite Difference Method is a second order method.

Table 3.2 lists again the mesh sizes (described by ν or N) , the number of degrees of freedom, the number of iterations of the CG Method and the wall clock time, one column for hours, minutes and seconds and another column just for seconds. It shows that refining the mesh results in more iterations of the CG Method. The increasing number of degrees of freedom also suggests that the comlexity of computations increases per iteration by decreasing the step size. As a result the wall clock time increases dramatically. In order to prevent hours of wall clock time for large problems, parallelizing is a reasonable way to solve those problems.

These results are consistent to the results of [5, Table 3.1, Section 3.3.2], where the same problem is solved by Matlab's `pcg` function and the results are presented in a table similar to Table 3.1 and Table 3.2.

The comparison with [4, Table 3.1, Section 3] shows differences in the number of iterations and in the error values. Each error value in Table 3.1 is slightly smaller and each run of the code needs between one or two iterations more then the runs of the corresponding code in [4, Table 3.1, Section 3]. The rerunning of the code in [4] resulted in the same number of iterations. But a one-process run with $N - 1$ instead of N in the code of this report resulted also in the same number of iterations as in [4, Table3.1, Section 3] for N. For this reason the N of [4, Table 3.1, Section 3] does not correspond to the N in Table 3.1.

4 Parallel Performance Studies

The idea behind parallel computing was already mentioned in Section 1: By using p processes, the parallel algorithm should be p times faster than the serial algorithm. In reality this is

Table 3.2: ν and N describe the mesh size, n the degrees fo freedom, iter the number of iterations. The last two columns describe the wall clock time in two different formats.

ν	N	$n = N^2$	#iter	wall clock time	
				HH:MM:SS	seconds
5	32	1,024	48	<00:00:01	<0.01
6	64	4,096	96	<00:00:01	0.01
7	128	16,384	192	<00:00:01	0.05
8	256	65,536	387	<00:00:01	0.32
9	512	262,144	783	00:00:03	2.56
10	1024	1,048,576	1581	00:00:28	27.97
11	2048	4,194,304	3192	00:03:40	219.80

not always the case but the parallel algorithms still perform better than serial algorithms. Thus a very important criterion is that using more processes should always leads to (not necessarily optimal) speed-up.

Table 4.1 lists the different wall clock times for certain problems. The table consist of various sub-tables each describing the Poisson problem for one particular mesh resolution. The different columns describe the number of nodes being used (1, 2, 4, or 8) and the rows show how many processes per node were used (again 1, 2, 4, or 8). Thus the table compares the wall clock times of the algorithm using between one and 64 processes. The nodes were used exclusively.

The table clearly shows that using more processes always decreases the wall clock time. Regarding speed-up we have to divide the results: First if the number of processes per node is fixed, the results indicate almost optimal speed-up. But if the number of nodes is fixed, we still observe speed-up, only not as good as in the other case. Nevertheless the results encourage that using parallel algorithms and various nodes is an efficient way of saving time (and storage), the more nodes the better.

In [4, Table 1.1, Section 1], where the table is presented in the same fashion as Table 4.1, the same problem using the same computing environment is described. Regarding the wall clock times, there are no dramatic differences. Seemingly the results in Table 4.1 are slightly better, but there are always single times worse for every mesh resolution. Additionally, as described in Section 3, the two codes do not have equal results. But having comparable wall clock times is still worth mentioning, especially because the code used for this report needed more iterations.

Table 4.2 shows the results of the same code, only with the sharing node feature instead of using nodes exclusively. Except for wall clock time, all results are equal and independent of how the nodes were used. Wall clock times in red font imply that this specific run of the code was slower then the corresponding run in Table 4.1, blue font implies that the run was faster. Table 4.3 uses the same idea of coloring and describes the actual differences between Table 4.2 and Table 4.1.

Both Table 4.2 and Table 4.3 point out that in most cases using shared nodes leads to a worse performance. Especially in Table 4.3 it becomes obvious how much of an impact

Table 4.1: Each sub-table lists the results of the studies (exclusive nodes) in terms of wall clock time (HH:MM:SS) for a specific mesh resolution. A column represents a certain number of nodes and a row represents the processes per node.

Mesh resolution $N \times N = 1024 \times 1024$, $n = 1{,}048{,}576$				
	1 node	2 nodes	4 nodes	8 nodes
1 process per node	00:00:28	00:00:14	00:00:06	00:00:03
2 processes per node	00:00:14	00:00:06	00:00:03	00:00:02
4 processes per node	00:00:09	00:00:03	00:00:02	00:00:01
8 processes per node	00:00:05	00:00:02	00:00:01	00:00:00
Mesh resolution $N \times N = 2048 \times 2048$, $n = 4{,}194{,}304$				
	1 node	2 nodes	4 nodes	8 nodes
1 process per node	00:03:40	00:01:53	00:00:58	00:00:29
2 processes per node	00:01:39	00:00:58	00:00:28	00:00:13
4 processes per node	00:01:08	00:00:39	00:00:17	00:00:07
8 processes per node	00:00:44	00:00:22	00:00:10	00:00:04
Mesh resolution $N \times N = 4096 \times 4096$, $n = 16{,}777{,}216$				
	1 node	2 nodes	4 nodes	8 nodes
1 process per node	00:29:58	00:14:49	00:08:03	00:03:55
2 processes per node	00:15:21	00:07:27	00:03:56	00:01:59
4 processes per node	00:08:38	00:05:04	00:02:32	00:01:36
8 processes per node	00:05:53	00:02:58	00:01:30	00:00:46
Mesh resolution $N \times N = 8192 \times 8192$, $n = 67{,}108{,}864$				
	1 node	2 nodes	4 nodes	8 nodes
1 process per node	03:53:37	02:00:05	01:08:02	00:31:53
2 processes per node	01:53:15	01:00:08	00:31:23	00:15:55
4 processes per node	01:07:39	00:38:09	00:19:41	00:10:21
8 processes per node	00:48:06	00:24:07	00:12:06	00:06:06

the sharing of nodes can have. Surprisingly the single-process run for $N = 8192$ is a clear exception, but other than that the results of the shared node studies are never more than ten seconds faster than the corresponding exclusive node studies. In fact, the exclusive node time for the single-process run is only 4.57% slower, while the shared node time for the same mesh resolution using two nodes with two processes per node is 56.52% slower. Additionally, most of the exclusive node times are clearly better. Note that the results confirm that for eight processes per node sharing does not affect the times for obvious reasons.

On the one hand if performance studies are supposed to be created, sharing nodes may have a negative impact on wall clock times. On the other hand sharing nodes can help finishing more jobs faster which can be helpful. But the results of this report only emphasize using exclusive nodes for actual performance studies.

Table 4.2: Wall clock times for the studies with shared nodes, each sub-table lists the results for a specific mesh resolution. Red color implies that the result is worse then the corresponding result of the exclusive node study, blue color that the result is better.

Mesh resolution $N \times N = 1024 \times 1024$, $n = 1{,}048{,}576$				
	1 node	2 nodes	4 nodes	8 nodes
1 ppn	00:00:28	00:00:14	00:00:06	00:00:03
2 ppn	00:00:16	00:00:08	00:00:03	00:00:02
4 ppn	00:00:11	00:00:05	00:00:02	00:00:01
8 ppn	00:00:05	00:00:02	00:00:01	00:00:00

Mesh resolution $N \times N = 2048 \times 2048$, $n = 4{,}194{,}304$				
	1 node	2 nodes	4 nodes	8 nodes
1 ppn	00:04:53	00:01:56	00:01:08	00:00:29
2 ppn	00:02:31	00:01:17	00:00:36	00:00:16
4 ppn	00:01:23	00:00:42	00:00:19	00:00:10
8 ppn	00:00:44	00:00:22	00:00:10	00:00:04

Mesh resolution $N \times N = 4096 \times 4096$, $n = 16{,}777{,}216$				
	1 node	2 nodes	4 nodes	8 nodes
1 ppn	00:36:21	00:18:18	00:09:47	00:03:56
2 ppn	00:20:07	00:10:15	00:04:35	00:02:17
4 ppn	00:11:19	00:05:58	00:02:50	00:01:31
8 ppn	00:05:54	00:02:58	00:01:30	00:00:46

Mesh resolution $N \times N = 8192 \times 8192$, $n = 67{,}108{,}864$				
	1 node	2 nodes	4 nodes	8 nodes
1 ppn	03:43:25	02:30:02	01:33:27	00:31:44
2 ppn	02:21:53	01:34:11	00:37:36	00:18:36
4 ppn	01:36:25	00:49:59	00:24:25	00:12:13
8 ppn	00:48:04	00:24:11	00:12:07	00:06:06

Acknowledgments

This report is the result of a semester abroad at UMBC and is also a preparation for my diploma thesis. I was able to be a Faculty Research Assistant at UMBC for the Fall semester 2011.

First I would like to thank Dr. Meister for giving me the opportunity to study abroad in Baltimore by introducing me to Dr. Gobbert, helping me to get all the necessary information and so on. Then I would like to thank Dr. Gobbert for all the work he put into getting me to UMBC, making sure I get my visa, introducing me to staff and students at UMBC and teaching me important things about MPI, cluster tara etc. He was my contact person at UMBC and helped me a lot with this report. Many thanks also go to the International Office of the Universität Kassel for the PROMOS stipend, especially because of all the changes to the PROMOS program and the uncertainties resulting from that. Additionally I thank all

Table 4.3: Difference between corresponding wall clock times of Table 4.2 and Table 4.1.

Mesh resolution $N \times N = 1024 \times 1024$, $n = 1{,}048{,}576$				
	1 node	2 nodes	4 nodes	8 nodes
1 ppn	00:00	00:00	00:00	00:00
2 ppn	+00:02	+00:02	00:00	00:00
4 ppn	+00:02	+00:02	00:00	00:00
8 ppn	00:00	00:00	00:00	00:00

Mesh resolution $N \times N = 2048 \times 2048$, $n = 4{,}194{,}304$				
	1 node	2 nodes	4 nodes	8 nodes
1 ppn	+01:13	+00:03	+00:10	00:00
2 ppn	+00:53	+00:19	+00:08	+00:03
4 ppn	+00:15	+00:03	+00:02	+00:03
8 ppn	00:00	00:00	00:00	00:00

Mesh resolution $N \times N = 4096 \times 4096$, $n = 16{,}777{,}216$				
	1 node	2 nodes	4 nodes	8 nodes
1 ppn	+06:23	+03:29	+01:44	+00:01
2 ppn	+04:46	+02:48	+00:39	+00:18
4 ppn	+02:41	+00:54	+00:18	−00:05
8 ppn	+00:01	00:00	00:00	00:00

Mesh resolution $N \times N = 8192 \times 8192$, $n = 67{,}108{,}864$				
	1 node	2 nodes	4 nodes	8 nodes
1 ppn	−10:12	+29:57	+25:25	−00:09
2 ppn	+28:00	+34:03	+06:13	+02:41
4 ppn	+28:46	+11:50	+04:44	+01:52
8 ppn	−00:02	+00:04	+00:01	00:00

the people at UMBC being involved in my admission.

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant no. CNS–0821258) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `www.umbc.edu/hpcf` for more information on HPCF and the projects using its resources.

References

[1] Peter S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, Inc., 1997

[2] Dietrich Braess, *Finite Elements*, Cambridge University Press, third edition, 2007

[3] Andreas Meister, *Numerik linearer Gleichungssysteme: Eine Einführung in moderne Verfahren*, Vieweg+Teubner, third edition, 2007

[4] Andrew M. Raim & Matthias K. Gobbert, *Parallel Performance Studies for an Elliptic Test Problem on cluster tara*, Technical Report HPCF-2010-2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010

[5] Matthew W. Brewster & Matthias K. Gobbert, *A Comparative Evaluation of Matlab, Octave, FreeMat, and Scilab on tara*, Technical Report HPCF-2011-10, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2011

[6] Arieh Iserles, *A First Course in the Numerical Analysis of Differential Equations*, Cambridge Texts in Applied Mathematics, Cambridge University Press, second edition, 2009