Investigating the Use of pMatlab to Solve the Poisson Equation on the Cluster maya

Sarah Swatski (swatski1@umbc.edu)

Department of Mathematics and Statistics, University of Maryland, Baltimore County

Technical Report HPCF–2014–24, www.umbc.edu/hpcf > Publications

Abstract

Many physical phenomena can be described by partial differential equations which can be discretized to form systems of linear equations. We apply the finite difference method to the Poisson equation with homogeneous Dirichlet boundary conditions, which yields a system of linear equations with a large sparse system matrix. We implement pMatlab code which utilizes the conjugate gradient method to solve this system. We do not recommend the use of pMatlab at this time as we find that it is very limited, its implementation is highly complex and the results are inconsistent.

Key words. Parallel Computing, pMatlab, Finite Difference Method, Conjugate Gradient Method, Poisson Equation.

AMS Subject Classifications (2010). 65Y05, 65Y04, 65F05, 65M06, 35J05.

1 Introduction

As mathematical models become increasingly realistic, they also become increasingly complex and their simulations begin to take longer. Parallel computing allows one to divide the computations of such simulations among various processes so that each process can work on its piece of the problem simultaneously. As a result, parallel computing allows increasingly complex problems to be solved in reasonable amounts of time [8].

The Message Passing Interface (MPI) commands provide a library of routines which can be used for communication between processes. Typically, programmers must understand how their data is distributed among the processes as well as how to utilize MPI commands to communicate the necessary data between processes. This is usually done by writing parallel code in C which uses MPI commands for communication. However, many scientists and engineers utilize the high-level software, Matlab, due to its ease of use. They do not have the computer science background to be able to convert their complex Matlab code into parallel C code. But with more complex code, runtimes begin to require an excessive amount of time, and an alternative is needed. As a result of this conundrum, MIT Lincoln Laboratory created two libraries, pMatlab and MatlabMPI, which help users take advantage of parallel programming without leaving Matlab. They first developed MatlabMPI, which is a version of MPI commands in Matlab which use Matlab's input and output commands for communication. They then expanded upon this development creating pMatlab, which uses arrays which are distributed among the processes. When downloaded, pMatlab comes with

documentation which includes an introductory to parallel programming and the use of the software [7]. There is also a detailed book available that explains in detail the examples included in the download [6]. pMatlab does not require the user to understand the technicalities of MPI communication. "Just as message passing libraries like MPI allow programmers to focus on writing parallel code rather than networking code, global array libraries like pMatlab allow programmers to focus on writing scientific and engineering code rather than parallel code" [7, page 11]. Once the user declares a distributed memory array, they can work with the whole array rather than only working with the local pieces of each array. pMatlab checks whether communication is needed and if so, it determines how to communicate and executes the communication, all without commands from the user. pMatlab allows "any Matlab user to parallelize their existing program by simply changing and adding a few lines, rather than rewriting their program" [7, page 11]. Therefore, pMatlab allows users to reap the benefits of parallel programming with very little parallel programming experience.

We apply the finite difference method to the Poisson equation to produce a system of linear equations. We will then utilize the conjugate gradient method (without preconditioning for simplicity) to solve this system since this method was found to be the most efficient linear solver of methods tested in [10] for large mesh sizes. We note that the conjugate gradient method was implemented in serial in [10] (Matlab) and [9] (C) and both found that the method converges for our test problem. We will implement the method in pMatlab to see how the code and performance compares. We anticipate that the transition from serial Matlab to pMatlab code will be simple, and that we will be able to experience speedup when using pMatlab.

2 The Poisson Problem

In this report, we consider the classical test problem of the Poisson equation with homogeneous Dirichlet boundary conditions given by

$$-\triangle u = f \quad \text{in } \Omega, \tag{2.1}$$

$$u = 0 \quad \text{on } \partial\Omega, \tag{2.2}$$

with right-hand side function

$$f(x,y) = -2\pi^2 \cos(2\pi x)\sin^2(\pi y) - 2\pi \sin^2(\pi x)\cos(2\pi y),$$

on the two-dimensional unit square domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$. Here, the Laplace operator is defined by $\triangle u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ and $\partial\Omega$ in (1.2) represents the boundary of the domain Ω. This problem has a closed-form true solution of

$$u(x,y) = \sin^2(\pi x)\ \sin^2(\pi y). \tag{2.3}$$

Due in part to the known true solution, this problem is frequently used as a test problem in many textbooks [3–5, 11].

To apply the finite difference method, we first define a grid of mesh points with uniform mesh spacing $h = \frac{1}{N+1}$. Thus, our grid can be represented by $\Omega_h = \{(x_i, y_i) = (ih, jh), i, j =$

$0, \ldots, N+1\}$. We apply the second-order finite difference approximation (given as (8.1.3) on page 546 of [11]) to the x- and y-derivatives at all interior points of Ω_h. This results in equations that approximate the x- and y-derivatives of u at a given (x_i, y_j) which are explicitly stated as (3) and (4) in Section 3 of [2]. Substituting these results in (2.1) produces a system of equations for the approximation of the unknowns $u_{i,j} \approx u(x_i, y_j)$. We see that

$$-u_{i-1,j} - u_{i,j-1} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = h^2 f_{i,j} \quad i,j = 1, \ldots, N, \tag{2.4}$$

where $f_{i,j} = f(x_i, y_j)$ and note that the boundary conditions give $u_{0,j} = u_{i,0} = u_{N+1,j} = u_{i,N+1} = 0$. From (2.4), we see that the N^2 equations produced will be linear in $u_{i,j}$. Therefore this problem can be organized into $\mathbf{Au} = \mathbf{b}$, with dimension N^2 where $\mathbf{A} \in \mathbb{R}^{N^2 \times N^2}$ and $\mathbf{u}, \mathbf{b} \in \mathbb{R}^{N^2}$. Since the boundary values are provided, there will be exactly N^2 unknowns. For this system, we see that

$$\mathbf{A} = \begin{bmatrix} S & -I & & & \\ -I & S & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & S & -I \\ & & & -I & S \end{bmatrix} \in \mathbb{R}^{N^2 \times N^2}, \quad \text{where} \quad \mathbf{S} = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix} \in \mathbb{R}^{N \times N},$$

$\mathbf{I}$ is the $N \times N$ identity matrix, and $\mathbf{b}_m = h^2 f_{i,j}$ where $m = i + (j-1)N$. We note that the system matrix $\mathbf{A}$ is symmetric and positive definite as discussed in [10].

3 The Computing Environment

3.1 Hardware

The UMBC High Performance Computing Facility (HPCF) is the community-based, interdisciplinary core facility for scientific computing and research on parallel algorithms at UMBC. The current machine in HPCF is the 240-node distributed-memory cluster maya. The newest components of the cluster are the 72 nodes with two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory that include 19 hybrid nodes with two state-of-the-art NVIDIA K20 GPUs (graphics processing units) designed for scientific computing and 19 hybrid nodes with two cutting-edge 60-core Intel Phi 5110P accelerators. These new nodes are connected by a high-speed quad-data rate (QDR) InfiniBand network for research on parallel algorithms. All nodes are connected via InfiniBand to a central storage of more than 750 TB.

The calculations in this report were performed on the newest (2013) nodes with two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory. All calculations in this report were run on one node in the develop partition.

3.2 Matlab

"Matlab is a high-level language and interactive environment for numerical computation, visualization, and programming," as stated on its webpage www.mathworks.com. The software

was created by Cleve Moler, a numerical analyst at the University of New Mexico's Computer Science Department, and when the potential of this new software was recognized, Mathworks was established in 1983 [2]. The key features include its high-level language, interactive environment, mathematical functions, and built-in graphics. All calculations using serial Matlab in this report use Matlab Version R2014a (8.3.0.532) 64-bit (glnxa64).

3.3 pMatlab

pMatlab and MatlabMPI are libraries developed by MIT Lincoln Laboratory to enable parallel programming with Matlab [6, 7]. pMatlab uses global array semantics, so "the programmer views an array as a single global array rather than multiple, independent arrays located on different processors" [7]. The program calls functions from the global array library which determine if and how data should be redistributed and perform the communication. pMatlab uses a new datatype, the distributed matrix or `dmat`, which can be created in 2, 3 or 4 dimensions. All `dmat` objects must be constructed by a constructor function which takes one new argument, a map, which tells pMatlab how and where the `dmat` should be distributed.

We downloaded the pMatlab version 2.0.15 zip file to a local computer from the webpage http://www.ll.mit.edu/mission/cybersec/softwaretools/pmatlab/pmatlab.html. We then copied this file onto maya and unzipped the file. The file contained a `startup.m` file to add the proper paths when starting Matlab. We edited the paths so that they were correct for the location at which we stored the files, and then copied the contents of this file into the current `startup.m` file in our home directory. As a result, Matlab will be able to find the files necessary for pMatlab no matter where we store other files. We then ran all of the examples provided with the download by running the `run.m` script in the `TestAll` directory under the `Examples` directory. We ran these using `PARALLEL=0` which runs serial Matlab code with no overhead and `PARALLEL=1` which sets up the distributed memory arrays. For the parallel case, we ran using both 1 and 2 processes per node. The examples all worked as expected. We then typed `help pMatlab`, which provided a list of commands that were supported by pMatlab. We found that the list of functions supported by pMatlab is very small, and even simple operations such as addition and the transpose operator are only supported for "special cases" which we determined to be when no overlap was used in the mapping. We noted that the examples provided with the code were specifically chosen because the necessary functions, such as `fft` for Fast Fourier transform, were supported by pMatlab. This led to the realization that the scope of pMatlab's capabilities was not nearly as broad as anticipated.

4 pMatlab Implementation

When the conjugate gradient method is implemented carefully, it requires only two inner products, three vector updates, and one matrix-vector product at each iteration [1]. Matlab code for this efficient implementation is provided with [2]. We began our attempt to implement the conjugate gradient method in pMatlab by attempting to adapt this Matlab code for use with pMatlab. We knew that Matlab's conjugate gradient function `pcg` was not supported by pMatlab. However, we realized that once the preconditioning and error checking are removed

from this function, all that is left are basic arithmetic operations. We intended to make an edited version of `pcg` and transform this into pMatlab. However, we soon discovered that only a small number of Matlab functions are supported by pMatlab. Functions used in the code provided with [2] like `ndgrid`, `reshape`, `max` and the numerous functions these functions call are not supported by pMatlab. The number of functions that needed to be transformed into pMatlab compatible functions kept growing as we investigated the code.

After some thought, we determined that rather than using the Matlab code as our guide, we should use the parallel C code we wrote for [9] as our guide. The first step we needed to take was to convert the `main.c` file described in [9] to the pMatlab file `driver_cg.m`. To do this, we first had to create a flag `PARALLEL` which is set to 1 when performing parallel computations and 0 when the parallelization of pMatlab is not used. For our purposes, we set `PARALLEL=1` for all runs. We then added some test output that displayed a greeting from all processes to ensure that our processes were allocated as expected. We then created our map object, required by pMatlab to distribute our vectors. We created a serial version of our map, `Cmap=1` so that in case `PARALLEL=0`, the code will still function. We then set up our column vector map, `Cmap = map([Np 1], {}, 0:Np-1)`. The first part of this map, the grid, says that the first dimension of our vectors will be divided into Np blocks and the second dimension (which is one since we are only dealing with vectors) will be divided into one block. The second part of the map, the distribution, which we set to `{}` says that we wish our dimension to be distributed in a block fashion. This means that each process will be working with one contiguous block of data. The third part of the map, the processor list, which we set to `0:Np-1` tells pMatlab that we wish the ranks of the processes to be `0` through `Np-1`. We leave out the optional fourth argument for maps, the overlap, since many of the functions that are supported by pMatlab do not support the use of overlap. We note, however, that if overlap were supported, it would hide the communication during the processes when computing the matrix-vector product in the conjugate gradient method.

We compute the step size h, $n = N^2$ and the size of the local blocks l_N and l_n as scalars. In order to create a dmat object, one must allocate it using one of the few supported functions. For our code, we choose to allocate our vectors using `zeros`. We allocate $\mathbf{u}, \mathbf{r}, \mathbf{p}$ and $\mathbf{q}$ using `zeros` with our map `Cmap`. Therefore, these arrays are distributed across the processes. We then define the entries of $\mathbf{x}$ and $\mathbf{y}$, although we do not send our map since we do not want these to be distributed arrays. Next, we define the vector $\mathbf{r}$ to be the right-hand side $\mathbf{b}$ initially using a `for` loop. We note that even though $\mathbf{r}$ is a dmat object, we do not need to work with its local part due to the fact that pMatlab hides this process and takes care of the arithmetic we supply. Since we will need communication for the matrix-vector product, we set the values `idleft` and `idright` based on the current process. To prevent an issue when we are the first or last process, we use the modulus function. As a result, `idleft` for process `0` becomes process `Np-1` and `idright` for process `Np-1` becomes process `0`. We then use `tic` and `toc` to time a call to the conjugate gradient method function `cg.m`. The function `driver_cg.m` ends with a display of relevant information including the timing results and an iteration count that we can use to check for accuracy.

We begin the `cg.m` function by computing the norm of the residual vector $\mathbf{r}$. Since the norm function is not supported by pMatlab, we accomplish this by setting `n2bd` to be the square root

Listing 1: Code for converting a 1×1 dmat to a double available on all processes.

```
% Initialize for communication
global pMATLAB;
tag = 1;
pMATLAB.comm = MatMPI_Save_messages(pMATLAB.comm,1);

% Compute the norm of b, convert from a dmat to a double and then
% distribute to all processes
n2bd = sqrt(r'*r);
n2b = local(n2bd);
if Pid == 0
  for dest = 1:Np-1
    SendMsg(dest, tag, n2b);
  end
end
if Pid ~= 0
  n2b = RecvMsg(0, tag);
end
```

of the transpose of $\mathbf{r}$ times $\mathbf{r}$. However, we realize that in pMatlab, since $\mathbf{r}$ is a dmat object, the result in `n2bd` is a 1×1 dmat object stored on process 0. Due to this, we cannot use this value as a scalar, which is needed. To overcome this, we must utilize communication to send this value to all other processes. This requires setting up a communicator, accessing the local part on process 0, and using the pMatlab commands `SendMsg` and `RecvMsg` which use Matlab MPI communication. The implementation for this is shown in Listing 1. We note that the software is intended to replace the need for communication and the documentation cautions against using these functions [6], but we see no other way to get around the problem that a 1×1 dmat is not available on all processes. We can then use the resulting `n2b` available on all processes to check whether the right-hand side vector is all zeros, and if so, set the solution to zero and exit the method. If the right-hand side vector is not all zeros, we compute the relative tolerance. We then use our `Ax` function to compute the matrix-vector product $\mathbf{q} = \mathbf{Au}$ without ever storing the matrix $\mathbf{A}$. Since $\mathbf{r}$ and $\mathbf{q}$ are dmat objects with the same map, we can subtract them as $\mathbf{r} = \mathbf{r} - \mathbf{q}$. We then calculate `rhod` to be the inner product of $\mathbf{r}$ with itself. Since this results in another 1×1 dmat object, we must implement communication similar to the above to convert this to the scalar `rho` available on all processes. We can then compute the square root of rho, which gives the norm of $\mathbf{r}$. We check whether the initial guess is good enough, and exit the function if this is true.

We now enter the main `while` loop of the `cg` function. At each iteration, we update the iteration count, update $\mathbf{p}$, and compute $\mathbf{q} = \mathbf{Ap}$. We then compute the inner product of $\mathbf{p}$ and $\mathbf{q}$ and again use the communication method to convert this to a scalar available on all processes. We update $\mathbf{r}$ and recompute `rho` by using an inner product and communication to

Listing 2: Code for obtaining the vectors **gl** and **gr**.

```
% Communication
global pMATLAB;
taga = 2;
tagb = 3;
pMATLAB.comm = MatMPI_Save_messages(pMATLAB.comm,1);
l_u = local(u);
if (Np > 1)
  SendMsg(idright, taga, l_u(N*(l_N-1)+1:N*(l_N-1)+N));
  SendMsg(idleft, tagb, l_u(1:N));
  gl = RecvMsg(idleft, taga);
  gr = RecvMsg(idright, tagb);
end

% Set gl and gr to zero in case of no next process
% (MPI_PROC_NULL in C)
if (Pid == 0)
  gl = zeros(N, 1);
elseif (Pid == Np-1)
  gr = zeros(N, 1);
end

% Take care of case when Np=1 (Ensure gl=gr=zeros)
if (Np == 1)
  gr = zeros(N, 1);
end
```

convert this to a scalar available on all processes. Finally, we update the solution vector **u**. After the iterations of the loop are complete, we set the flag and relative residual and end the function.

We now write the matrix-free implementation of the matrix-vector product $\mathbf{q} = \mathbf{Ap}$ in the function `Ax.m`. Since the matrix-vector product requires the vectors **gl** and **gr**, we begin by communicating these vectors. If we have no next process, we set the entries of these vectors to zero so no changes occur when using them. The code is shown in Listing 2. We then realize that to update the vector **q**, we cannot use **q** directly for several reasons. First, the vectors $\mathbf{gl}, \mathbf{gr}$ and **q** have different lengths, and thus we cannot add/subtract their elements due to the limitations of pMatlab. Second, using `for` loops to control the indices of the entries of the vector **q** is not possible. Therefore, we obtain the local part of **q** by setting `l_q = local(q)` and then we operate on $\mathbf{l_q}$ as before. When we are finished operating on $\mathbf{l_q}$, we put the entries back into the dmat object **q** by `put_local(q, l_q)`.

Finally, we create the file `run_cg.m`, which is a one line Matlab script to call the function

`driver_cg`. We do this since the pMatlab `run.m` script we will write cannot handle a call to a function that requires input parameters. We then write the pMatlab run script `run.m` in which we set the number of processes `Ncpus`, the name of the script file we want to run `run_cg` and the machine we want to run on `cpus`. We set `cpus` to `{}` to indicate that we want to run on the local machine. This file concludes with the line that actually runs the code, `eval(pRUN(mFile, Ncpus, cpus))`. To run the code, we use the `salloc` command to gain access to a compute node, use the `ssh` command to access that node, and open Matlab and run our code.

As you can see from our implementation, the fact that many functions are not supported by pMatlab leads to many extra `for` loops to control entries of vectors. The fact that an inner product results in a 1×1 dmat object that must be sent to the other processes using a `SendMsg` and `RecvMsg` also adds less efficient communication at each iteration, as the more efficient `Broadcast` operation was used in the C code. According to the documentation, a `Broadcast` is available in pMatlab, but our attempts to implement this were not successful. The advantages of using Matlab (e.g., vectorization) are outweighed by the costs of this extra communication and `for` loops. This makes the pMatlab code inefficient.

5 Results

For all runs of our code in this report, we supply a tolerance of 10^{-6}, a maximum number of iterations of 99,999, and the zero vector for the initial guess of the solution. The results of running the serial Matlab code, with no pMatlab overhead are shown in Table 5.1 (a). From the results of Table 5.1 (a), we see that for such small N values, the runtimes are fractions of a second. The results of running the pMatlab code are shown in Table 5.1 (b). We note that for many values of N, the results of all runs when $p > 1$ were unpredictable. Frequently, the code would stall during the communication and the run would not complete. Other times, the run would finish, producing the results seen above. We note that no changes were made to the code between these runs, therefore the different outcomes are disturbing. The fact that there is no timing result for $N = 16$ with $p = 2$ is a result of the fact that this run stalled every time it was run. We also tried to run $N = 8$ with $p = 4$, but this run also stalled every time it was run. The level of unreliability of the performance of this code makes its use very

N	Time (sec.)
4	0.005
8	0.004
16	0.008

(a) Serial Matlab

N	$p = 1$	$p = 2$
4	0.07	2.63
8	0.19	3.26
16	0.45	N/A

(b) pMatlab

Table 5.1: Observed wall clock time (in seconds), (a) for serial Matlab and (b) for pMatlab for various values of N using both one and two processes per node. The case of $N = 16$ and $p = 2$ stalled during communication during all runs.

unsettling. As a result of this, we are using very small N values as we anticipate problems when N is increased.

We see from Table 5.1 that the results with pMatlab take longer than the results with serial Matlab code. We suspect that the timing differences between the serial Matlab code and the pMatlab code run on one process stem from the overhead the pMatlab code performs when the option is set as `PARALLEL=1`. However, the drastic increase in runtime from 1 to 2 processes is very unexpected. For such small N values, this likely stems from the fact that we require communication that takes more time. However, even if we increase N sufficiently, we suspect that this code would still perform very poorly compared to the optimized serial Matlab code. We also note that all runs produced correct results, as the iteration counts were checked to validate our results.

6 Conclusions

The documentation provided with pMatlab makes the claim that the library is easy to use and allows users unfamiliar with parallel programming to experience its benefits without significant work. However, from the discussion in Section 4, we see that transforming optimized Matlab code into pMatlab code is very challenging. This is a result of the fact that many important Matlab functions are not compatible with pMatlab, and the pMatlab dmat constructs are difficult to control. When we have to convert parallel C code into pMatlab code, we lose the ability to take advantage of vectorization in Matlab. This also requires knowledge of parallel algorithms, and pMatlab was created to avoid this requirement. From the results in Section 5, we see that the performance of the pMatlab code is poor compared to the serial Matlab code due to the extra measures required to ensure it would work.

We suspect that the transformation from parallel C code to MatlabMPI code may be more straightforward. However, using MatlabMPI defeats the purpose of pMatlab, as a background in parallel programming is required to use MatlabMPI. Also, this defeats the purpose of using Matlab since its vectorization capabilities are not used in this implementation. The few working examples of pMatlab code provided with the software are all designed to use the small number of functions that are supported by pMatlab, and thus represent a limited domain with which one can use pMatlab. Further, since no active development has been seen since 2013, it does not appear that additional functions are being converted to be compatible with pMatlab and it is likely that this software will be quickly out of date as Matlab undergoes upgrades. Therefore, we do not recommend the use of pMatlab to Matlab programmers at this time.

Acknowledgments

This report was done as a project for Math 627, Introduction to Parallel Computing, during the Fall 2014 semester. I would like to thank the instructor, Dr. Matthias Gobbert, for his guidance on this assignment as well as my classmate, Jonathan Graf, with whom I worked on testing pMatlab code on the HPCF cluster maya. I would also like to acknowledge financial

support as a Research Assistant with the Center for Interdisciplinary Research and Consulting (CIRC) at UMBC.

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 nos. CNS–0821258 and CNS–1228778) 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] Kevin P. Allen. Efficient parallel computing for solving linear systems of equations. *UMBC Review: Journal of Undergraduate Research and Creative Works*, 5:8–17, 2004.

[2] Ecaterina Coman, Matthew W. Brewster, Sai K. Popuri, Andrew M. Raim, and Matthias K. Gobbert. A comparative evaluation of Matlab, Octave, FreeMat, Scilab, R, and IDL on tara. Technical Report HPCF–2012–15, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2012.

[3] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[4] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*, volume 17 of *Frontiers in Applied Mathematics*. SIAM, 1997.

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

[6] Jeremy Kepner. *Parallel MATLAB for Multicore and Multinode Computers*. SIAM, 2009.

[7] Hahn Kim, Julia Mullen, and Jeremy Kepner. Introduction to Parallel Programming and pMatlab v2.0, 2009. Supplied with pMatlab v2.0.15 Download.

[8] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[9] Sarah Swatski. A comparison of the parallel performance of blocking and non-blocking mpi communication for the poisson equation on the cluser maya, 2014. Report.

[10] Sarah Swatski, Samuel Khuvis, and Matthias K. Gobbert. A comparison of solving the Poisson equation using several numerical methods in Matlab and Octave on the cluster maya. Technical Report HPCF–2014–10, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.

[11] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley, third edition, 2010.