Performance Studies for Multithreading in Matlab with Usage Instructions on hpc

Neeraj Sharma and Matthias K. Gobbert

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

{nsharma1,gobbert}@math.umbc.edu

Abstract

This report explores the use of multiple computational cores by multithreading in the software package Matlab on a compute node with two dual-core AMD Opteron processors. After testing the built-in functions of Matlab for a small test problem, we consider a classical test problem resulting from a finite difference discretization of the Poisson equation in two spatial dimensions. The results demonstrate that the use of more than one thread is often not very beneficial for Matlab code. This suggests that Matlab jobs should be limited to using one core by default to allow for the fastest throughput of the largest number of jobs. The comparison of solving the same problem with a code using the source code language C indicates that Matlab uses more memory and takes longer; this has to be contrasted with the productivity gains possible of programming in Matlab. This report also provides detailed information on how to run Matlab jobs in the UMBC High Performance Computing Facility.

1 Introduction

This report focuses on the concept of multithreading in the general-purpose software package Matlab for the purpose of taking advantage of multiple computational cores that are becoming common on today's computer chips. Section 2 explores the concept of multithreading in Matlab by tests based on one of Matlab's own demos. We study the performance of using 1 to 4 computational threads on one computational node with two dual-core processors.

The tests suggested by Matlab in its demo use eight representative computational operations applied to relatively small sample matrices. To repeat the scalability tests for a larger problem, we consider the classical test problem of solving the large sparse system of linear equations resulting from a finite difference discretization of the Poisson equation [2, 4, 5]. Section 3 reviews the finite difference discretization for the test problem in two spatial dimensions and summarizes the linear system and its properties. One of the most important properties of the system matrix is its symmetric positive definiteness, which justifies the use of the iterative conjugate gradient method in the following section. Section 4 summarizes the results of three convergence studies for the linear solvers. The studies' reported run times and memory usage observations demonstrate that the problem becomes computationally significant as the mesh resolution increases. These results bear out that direct linear solvers available in Matlab solve the problem faster if they are able to solve it in the available memory of the compute node. Specifically, Matlab's direct solver can solve the problem up to a mesh resolution of $2{,}048 \times 2{,}048$. The iterative conjugate gradient method is able to solve the problem on a finer mesh, namely up to $4{,}096 \times 4{,}096$. If a matrix-free implementation of the iterative method is used, it becomes possible to solve the problem up to a mesh with resolution $8{,}192 \times 8{,}192$ due to the significant memory savings. By comparing to [3], which solves the same problem with a program in the source code language C, we can demonstrate that for this example the C code is at least four times faster and uses a third of the memory of the Matlab version. This is qualitatively expected, because we are not in full control of how many auxiliary variables

are created in Matlab and has to be seen in the context of productivity gains associated with programming Matlab code.

In Section 5, this report collects detailed information on how to run Matlab on the cluster hpc in the UMBC High Performance Computing Facility (HPCF; `www.umbc.edu/hpcf`). This section provides the details for running the job on the head node for testing purposes as well as on one of the compute nodes for production runs. In this section, we also conduct a performance test for our test problem using 1 to 4 threads analogous to the studies in Section 2. These results from both Sections 2 and 5 indicate that using multiple computational cores might not be advantageous for Matlab code. This explains the default setup of Matlab on the cluster hpc in HPCF that runs Matlab using only one thread. Accordingly, users should use a submission script to the job scheduler that reserves only one core of a compute node for the Matlab job. This policy is designed to optimize the throughput of Matlab jobs on the cluster, that is, it allows as many Matlab jobs as possible to run simultaneously, which is likely to minimize the total run time of all jobs, even if an individual job could run faster using more than one core. It is possible for users to override these settings, but they should only do so after careful tests — analogous to the ones performed for this report — that demonstrate the benefit for the particular Matlab functions used in the code.

2 Multithreaded Computation

It has recently become common even for general-purpose computer chips to contain multiple computational cores. Matlab can take advantage of multiple cores by multithreading its computations, as is motivated in one of Matlab's demos. This demo can be located from Matlab's Helpdesk under the Help pull-down menu as follows:

Help → Demos → MATLAB → Mathematics → Multithreaded Computation

Using the helper function `runAndTimeOps` and tests suggested in this demo, one can study the performance of using 1 to 4 threads by a script such as

```
[timeNthread functionNames]=runAndTimeOps; % run once to pre-load function
for numThreads = 1:4
   maxNumCompThreads(numThreads);
   [timeNthread functionNames]=runAndTimeOps;
   timeNthread
end
```

In this script, the function `maxNumCompThreads` sets the maximum number of computational threads; see its documentation by `doc maxNumCompThreads` or `help maxNumCompThreads`. The helper function `runAndTimeOps` times the wall clock time using `tic` and `toc` for eight representative computational operations in Matlab that might profit from multithreading: matrix multiplication `*`, the backslash operator `\`, the QR decomposition `qr`, the LU decomposition `lu`, element-wise sine function `sin`, element-wise array power `.^`, element-wise square root `sqrt`, and element-wise multiplication `.*`. Specifically, `runAndTimeOps` creates a 500×500 random matrix and averages the observed run times from 10 runs of each of the operations considered. The results are returned in the vector `timeNthread` that lists the average run time for each operation in units of seconds. The vector `functionNames` returns the names of the operations that were tested. The code of the helper function is printed in the demo, or you can see it by saying

```
>> more on; type runAndTimeOps; more off
```

We run this helper function once outside of the for loop in order to force pre-loading of the function before the actual timings in order to avoid timing the potential overhead associated with the initial loading of the function; see a note on this issue in the demo.

Increasing the number of computational threads should decrease the run time for these functions. Hence, we hope to improve the run time. Table 1 lists the results obtained using the script above. This job was performed on one compute node in the hpc cluster which contains two dual-core AMD Opteron 2.6 GHz (1 MB cache per core) processors, hence up to 4 threads can be run on the node. Each row in Table 1 lists the number of computational threads used to obtain the job run time. So increasing the number of computational threads each time should ideally decrease the job run time by a factor 2. Consider first whether the operations benefit from the use of two threads compared to one. Matlab's demo already mentions that "simple element-wise multiplication [`.*`] does not because it is a memory-bound operation," and hence the lack of speed improvement in the last column is expected. For the first seven columns in Table 1, it is apparent that the operations can be grouped into two categories: One category includes the functions that produce significantly faster results using two threads; these operations are matrix multiplication `*`, element-wise sine function `sin`, element-wise array power `.^`, element-wise square root `sqrt`, with the latter two enjoying nearly perfect speedup of a factor 2. The other category includes the operations that fall significantly short of the optimal value for speedup; these operations are the backslash operator `\`, the QR decomposition `qr`, the LU decomposition `lu`, which are in some cases only about 30% faster using two threads instead of one. This is corroborated and explained by the solution to a question in Matlab's technical support database: "For LU, QR and the backslash operator fall much short of the optimal value, because MATLAB calls the LAPACK library (which is single-threaded), which in turn calls the BLAS library (Basic linear Algebra Subroutines). The BLAS subroutines are the ones that are multithreaded. These functions will show performance improvement only because of multithreading in the BLAS level. Further, LAPACK may divide the input matrices into smaller sub-matrices, and make a BLAS call for each sub-matrix." This quote is from Matlab's technical solution available at `www.mathworks.com` under Support with Solution ID 1–7S5ULZ or directly at

`http://www.mathworks.com/support/solutions/en/data/1-7S5ULZ/`

For the results using 3 or 4 threads in Table 1, we notice some additional speedup for some operations tested. But the results are not systematic and do remain short of the optimal value of speedup for most cases.

Table 1: Observed run times in seconds for eight representative computational operations.

Number of threads	`*`	`\`	`qr`	`lu`	`sin`	`.^`	`sqrt`	`.*`
1	0.0604	0.0426	0.0579	0.0311	0.0059	0.0482	0.0512	0.0016
2	0.0335	0.0330	0.0361	0.0234	0.0033	0.0244	0.0259	0.0017
3	0.0240	0.0303	0.0289	0.0206	0.0031	0.0164	0.0174	0.0016
4	0.0199	0.0280	0.0256	0.0184	0.0030	0.0190	0.0207	0.0014

3 The Test Problem

3.1 The Problem

The tests in the previous section used several sample operations suggested by Matlab to study the performance improvements associated with multithreading on computational nodes with more than one computational core. Starting in this section, we study this question specifically for a classical test problem given by the numerical solution with finite differences for the Poisson problem with homogeneous Dirichlet boundary conditions [1, 2, 4, 5], given as

$$\begin{aligned} -\triangle u &= f \qquad \text{in } \Omega, \\ u &= 0 \qquad \text{on } \partial\Omega. \end{aligned} \tag{3.1}$$

Here $\partial\Omega$ denotes the boundary of the domain Ω while the Laplace operator is defined as

$$\triangle u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

This partial differential equation can be used to model heat flow, fluid flow, elasticity, and other phenomena [5]. Since $u = 0$ at the boundary in (3.1), we are looking at a homogeneous Dirichlet boundary condition. We consider the problem on the two-dimensional unit square $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$. Thus, (3.1) can be restated as

$$\begin{aligned} -\tfrac{\partial^2 u}{\partial x^2} - \tfrac{\partial^2 u}{\partial y^2} &= f(x,y) \qquad \text{for } 0 < x < 1, \quad 0 < y < 1, \\ u(0,y) = u(x,0) = u(1,y) = u(x,1) &= 0 \qquad \text{for } 0 < x < 1, \quad 0 < y < 1, \end{aligned} \tag{3.2}$$

where the function f is given by

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

The problem is designed to admit a closed-form solution as the true solution

$$u(x,y) = \sin^2(\pi x) \sin^2(\pi y).$$

3.2 Finite Difference Discretization

Let us define a grid of mesh points $\Omega_h = (x_i, y_j)$ with $x_i = ih, i = 0, \ldots, N+1, y_j = jh, j = 0, \ldots, N+1$ where $h = \frac{1}{N+1}$. By applying the second-order finite difference approximation to the x-derivative at all the interior points of Ω_h, we obtain

$$\frac{\partial^2 u}{\partial x^2}(x_i, y_i) \approx \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2}. \tag{3.3}$$

If we also apply this to the y-derivative, we obtain

$$\frac{\partial^2 u}{\partial y^2}(x_i, y_i) \approx \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2}. \tag{3.4}$$

Now, we can apply (3.3) and (3.4) to (3.2) and obtain

$$-\frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2} - \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2} \approx f(x_i, y_j). \tag{3.5}$$

Hence, we are working with the following equations for the approximation $u_{ij} \approx u(x_i, y_j)$:

$$\begin{aligned} -u_{i-1,j} - u_{i,j-1} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} &= h^2 f_{i,j} \qquad i,j = 1,\ldots,N \\ u_{0,j} = u_{i,0} = u_{N+1,j} &= u_{i,N+1} = 0 \end{aligned} \tag{3.6}$$

Using (3.6), we obtain a linear system $Au = b$ of N^2 equations. Since we are given the boundary values, we can conclude there are exactly N^2 unknowns. In this linear system, we have

$$A = \begin{bmatrix} S & T & & & \\ T & S & T & & \\ & \ddots & \ddots & \ddots & \\ & & T & S & T \\ & & & T & S \end{bmatrix} \in \mathbb{R}^{N^2 \times N^2}, \text{ where}$$

$$S = \begin{bmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{bmatrix} \in \mathbb{R}^{N \times N} \text{and } T = \begin{bmatrix} -1 & & & & \\ & -1 & & & \\ & & \ddots & & \\ & & & -1 & \\ & & & & -1 \end{bmatrix} \in \mathbb{R}^{N \times N}$$

and the right-hand side vector $b_k = h^2 f_{i,j}$ where $k = i + (j-1)N$. The matrix A is symmetric and positive definite; hence, there exists a unique solution to the linear system. One of the things to consider to confirm the convergence of the finite difference method is the finite difference error. The finite difference error is defined as the difference between the true solution u and the approximation on the grid points u_h. Since the solution is sufficiently smooth, we expect the finite difference error to decrease as N gets larger and $h = \frac{1}{N+1}$ gets smaller. Specifically, the finite difference theory predicts that the error will converge as $\|u - u_h\|_{L^\infty(\Omega)} \leq Ch^2$, as h is sufficiently small, where C is a constant independent of h.

4 Results

The computations for with this project are performed using Matlab R2008b on the distributed-memory cluster cluster hpc in the UMBC High Performance Computing Facility (HPCF). Each node of hpc has two dual-core AMD Opteron 2.6 GHz processors and at least 13 GB of memory.

4.1 Gaussian Elimination

Let us begin solving this linear system via Gaussian elimination. We know that this is easiest approach for solving linear systems for the user of Matlab, but it may not necessarily be the best method for large systems. In the process of solving this problem, we create Table 2, which lists the mesh resolution N, where $N = 2^\nu$ for $\nu = 1, 2, 3, \ldots, 13$, for mesh of size $N \times N$; the number of degrees of freedom (DOF) which is dimension of the linear system N^2; the norm of the finite difference error; the ratio of consecutive error norms; the observed wall clock time in HH:MM:SS; and the observed memory usage in MB.

The norms of the finite difference errors clearly go to zero as the mesh resolution N increases. The ratios between error norms for consecutive rows in the table tend to 4, which confirms that the

Table 2: Convergence results for Gaussian Elimination include the mesh resolution N, the number of degrees of freedom (DOF), the finite difference norm $\|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors, the observed wall clock time in HH:MM:SS, and the observed memory usage in MB.

N	DOF	$\|u - u_h\|$	Ratio	Time	Memory
32	1,024	3.0128e-3	N/A	<00:00:01	1,577
64	4,096	7.7812e-4	3.8719	<00:00:01	1,609
128	16,384	1.9766e-4	3.9366	<00:00:01	1,670
256	65,536	4.9807e-5	3.9685	<00:00:01	1,607
512	262,144	1.2501e-5	3.9843	00:00:04	1,779
1,024	1,048,576	3.1313e-6	3.9922	00:00:19	2,520
2,048	4,194,304	7.8362e-7	3.9960	00:01:43	5,437
4,096	out of memory				
8,192	out of memory				

Table 3: Convergence results for conjugate gradient method include the mesh resolution N, the number of degrees of freedom (DOF), the finite difference norm $\|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors, the number of iterations, the observed wall clock time in HH:MM:SS, and the observed memory usage in MB.

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time	Memory
32	1,024	3.0128e-3	N/A	48	<00:00:01	1,577
64	4,096	7.7811e-4	3.8719	96	<00:00:01	1,645
128	16,384	1.9765e-4	3.9368	192	<00:00:01	1,697
256	65,536	4.9797e-5	3.9690	387	00:00:03	1,661
512	262,144	1.2494e-5	3.9856	783	00:00:30	1,754
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:04:17	1,897
2,048	4,194,304	7.8019e-7	4.0075	3,192	00:43:40	2,803
4,096	16,777,216	1.9366e-7	4.0288	6,452	06:03:32	6,165
8,192	out of memory					

Table 4: Convergence results for conjugate gradient method include the mesh resolution N, the number of degrees of freedom (DOF), the finite difference norm $\|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors, the number of iterations, the observed wall clock time in HH:MM:SS, and the observed memory usage in MB.

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	Time	Memory
32	1,024	3.0128e-3	N/A	48	<00:00:01	1,592
64	4,096	7.7811e-4	3.8719	96	<00:00:01	1,653
128	16,384	1.9765e-4	3.9368	192	<00:00:01	1,625
256	65,536	4.9797e-5	3.9690	387	00:00:06	1,684
512	262,144	1.2494e-5	3.9856	783	00:00:51	1,685
1,024	1,048,576	3.1266e-6	3.9961	1,581	00:07:14	1,690
2,048	4,194,304	7.8019e-7	4.0075	3,192	01:05:02	2,004
4,096	16,777,216	1.9366e-7	4.0287	6,452	08:45:22	3,217
8,192	67,108,864	4.7375e-8	4.0878	13,033	65:28:25	7,764

finite difference method for this problem is second-order convergent with errors behaving lik h^2, as predicted by the finite difference theory. By looking at this table, it can be concluded that Gaussian elimination runs out of memory for $N = 4{,}096$. Hence, we are unable to solve this problem for N larger than 2,048 via Gaussian elimination. This leads to the need of another method to solve larger systems. Thus, we will use an iterative method known as the conjugate gradient method to solve this linear system.

4.2 Conjugate Gradient Method

Now, we will use an optimal implementation of the conjugate gradient method to solve the Poisson problem where the initial guess is the zero vector and the tolerance is 10^{-6} on the relative residual of the iterates. This optimal implementation is based on Matlab's `pcg` function and has the same functionality, but it saves four out of ten vectors by re-using some auxiliary vectors and it only requires one matrix-vector product per iteration compared to two in `pcg`. Once again, we create a table which lists the mesh resolution; dimension of the linear system; norm of the finite difference error; convergence ratio; `#iter`, which is the number of iterations it takes the method to converge; the run time; and memory usage. The finite difference error shows the same behavior with ratios of consecutive errors approaching 4 as for Gaussian elimination; this confirms that the tolerance on the relative residual of the iterates is tight enough.

The comparison of Table 3 and Table 2 shows that we are able to solve the problem on a mesh with a finer resolution that by Gaussian elimination, namely up to $N = 4{,}096$. Now, let us see if we can improve the conjugate gradient method to solve this problem for $N = 8{,}192$.

4.3 Matrix-free Implementation of the Conjugate Gradient Method

The system matrix A accounts for the largest amount of memory used by the conjugate gradient method. Hence, to solve the problem for larger meshes, we can use a so-called matrix-free implementation of the method that avoids storing A. The only place, where A enters into the iterative method is the matrix-vector multiplication $q = Ap$. Hence, for a matrix-free implementation, instead of supplying A itself, the user supplies a function that computes the matrix-vector product $q = Ap$ directly for a given input vector p without ever storing a system matrix A. Thus, the matrix-free implementation returns a vector q for the input vector p by performing the component-wise product on p and the matrix A using the knowledge of the components of A. Thus, we can expect to save memory significantly. Now, we will use this matrix-free implementation to solve the problem and report the results in Table 4. First, we must compare these results in Table 4 to Table 3 to confirm the convergence of this method. The need of comparison of these two tables leads to reporting the same quantities in both tables. In order to guarantee the convergence, we must compare the finite difference error, convergence order, and iteration count for each value of N. The comparison of the table concludes that these parameters match exactly, hence the matrix-free method accurately converges.

By eliminating the storage of A, we have boosted our memory savings, which can be confirmed by the comparison of the last column in Table 3 and Table 4. For instance, the memory usage for $N = 4{,}096$ in the matrix method is 6,165 MB where as the matrix-free method only requires 3,217 MB. This leads to a savings of about half of the memory. By looking at the time column of both tables, we note that the matrix-free implementation of the conjugate gradient method takes longer to converge. But we use significantly less memory each iteration in matrix-free implementation than

in the matrix-based method. Thus, the significant difference is that we are able to successfully solve the problem for $N = 8{,}192$ using the matrix-free implementation.

Since [3] solves exactly the same test problem with the matrix-free implementation of the same version of the conjugate gradient method, we can compare the results of Table 4 to Table 2 in [3]. The finite difference errors and iteration counts are nearly identical, confirming the correctness of the implementations. Both the Matlab and C codes are able to solve the problem up to $N = 8{,}192$ within the available memory. The time to solution in Matlab is at least four times the wall clock time reported in Table 2 of [3] with C. Matlab's memory consumption in Table 4 points to a baseline memory usage of over 1.5 GB. Taking this into account and considering only the portion of memory used beyond this baseline, it appears that Matlab uses at least three times as much memory for each case than the C code. Matlab can be expected to have higher memory usage than C, because we are not in full control of how many auxiliary variables are created. Thus, this memory usage for C is much smaller than Matlab.

5 Technical Details of Running the Jobs on hpc

The computations reported in the previous section were in fact performed in two different ways. One of them is to use the head node of the cluster hpc and the other one is to use one of the compute nodes of the cluster. We point out that using a compute node is required for production runs by the usage policies of the cluster, so the presentation of both methods is intended to demonstrate the techniques for both ways only, which can be useful in the context of testing and preparing for using a compute node. Additionally, the final subsection includes results of multithreading tests for the Poisson equation test problem that are similar to those in Section 2 for Matlab's built-in operations.

5.1 Running Matlab on the Head Node

The head node is the one that you can log in to. It may be used to perform short Matlab jobs or for testing, though production runs should take place on a compute node; see the following subsection. To perform the computations on the head node in the background, we enter the following command at the Linux prompt:

```
nohup matlab -nodesktop -nodisplay < driver_master.m >& driver_master.log &
```

Here, `matlab -nodesktop -nodisplay` starts Matlab without its graphical desktop and without graphics output. The Linux shell command `< driver_master.m` re-directs the contents of the script file `driver_master.m` into Matlab as stdin. In turn, the shell commands `>& driver_master.log` re-directs both stdout and stderr to the file `driver_master.log`. The ampersand `&` at the end of the line is what puts Matlab into the background, and the `nohup` at the beginning of the line gives the Linux prompt back as well as allows one to close the shell or to log out without the sub-shell that Matlab run in to hang up.

The memory observations are performed using the Linux command `top`, which displays the information about the CPU usage. We mainly focused on the `VIRT` column, which refers to the total virtual memory used by the task and includes all code, data, and shared libraries. Our task was to record the highest value observed while the program was running for each N.

5.2 Running Matlab on a Compute Node

The jobs on compute node are performed by submitting them to the cluster via qsub script and run whenever the compute nodes are available. For a Matlab job on a compute node, the default qsub script includes the lines

```
#!/bin/bash
#PBS -N 'MATLAB'
#PBS -o 'qsub.out'
#PBS -e 'qsub.err'
#PBS -W umask=007
#PBS -q low_priority
#PBS -l nodes=1:ppn=1
cd $PBS_O_WORKDIR
matlab -nodesktop -nodisplay < driver_master.m
```

The line `#!/bin/bash` is always the first line the qsub script which tells Linux to use the shell /bin/bash to execute the qscript. The second line `-N` creates a name for the job which shows up in the output of qstat. `-o 'qsub.out'` sends the stdout of the job into `qsub.out` and `-e 'qsub.err'` is used to store stderr in `qsub.err`. `-W` defines the permissions of new file. In `umask=007`, set to the default for users of hpc. The next line is used to set the queue in which the job will run, using the default `low_priority` queue here. Later in the script, `$PBS_O_WORKDIR` is used to change the current working directory to the directory from which you ran qsub. Finally, `matlab -nodesktop -nodisplay < driver_master.m` runs Matlab and re-directs the standard input form `driver_master.m`.

The crucial line `-l nodes=1:ppn=1` requests one compute node with one processor per node ("ppn") for the job, which is the default for Matlab on hpc. Corresponding to the fact that up to three other jobs could run on the compute node with two dual-core processors, this job should be limited to 3 GB of memory out of the total of 13 GB available on a compute node on hpc. Notice that this is the default setup, which is exactly justified by the results observed in this report, where we observe that it is most efficient for the throughput of Matlab jobs to use only one core per node.

If a job is expected to need more than 3 GB of memory or if tests for a particular Matlab job demonstrate that it is advantageous to use more than one core, then the user needs to modify the crucial line in the qsub script to `-l nodes=1:ppn=4` and set the maximum number of computational threads to 4 by putting the call `maxNumCompThreads(4)` in the beginning of the Matlab script that is re-directed in the `matlab` line of the qsub script. For consistency and since some cases required more than 3 GB of memory, we used `ppn=4` and the number of computational threads set by calling `maxNumCompThreads` to the desired number for each run.

To observe memory when running a job on a compute node, use first the command `qstat -n` to list the nodes on which your job is running. Once we discover the node on which our job is running, log into the node using `ssh node___` during the duration of the job. After logging into the node, we just used the Linux `top` command on the compute node to observe the memory usage.

By comparing the results obtained by running the jobs on compute node and head node, we conclude that the results are identical.

5.3 Multithreading for the Test Problem

To perform the multithreading test on the Poisson problem, we set the number of computational threads in the script file and run it on compute node using the qsub script stated earlier with a minor change. This minor change includes changing the processor per node to 4. So we request a group of four processor cores which is just requesting the entire compute node. This allows Matlab to use as may cores needed to perform the job and also ensures that your job is the only one running on that machine. Hence, we obtain the time results as well as the memory observation for the Gaussian elimination, conjugate gradient, and the matrix-free implementation of the conjugate gradient method. Table 5 lists the mesh resolution N, the observed wall clock time in HH:MM:SS, and the observed memory usage in MB for different number of computational threads each time as well as the three different methods. Table 5 leads us to conclude that the change in number of computational threads hardly impacts the timing and memory results. Thus, the results are very similar to each other and four processors are not required to run this type of job.

Table 5: Observed wall clock times in HH:MM:SS and observed memory usage in MB by number of threads used.

(a) Gaussian elimination

	1 thread		2 threads		3 threads		4 threads	
N	Time	Memory	Time	Memory	Time	Memory	Time	Memory
32	<00:00:01	1,598	<00:00:01	1,585	<00:00:01	1,525	<00:00:01	1,604
64	<00:00:01	1,593	<00:00:01	1,593	<00:00:01	1,528	<00:00:01	1,526
128	<00:00:01	1,555	<00:00:01	1,530	<00:00:01	1,533	<00:00:01	1,589
256	<00:00:01	1,618	<00:00:01	1,582	<00:00:01	1,522	<00:00:01	1,593
512	00:00:03	1,767	00:00:03	1,696	00:00:03	1,645	00:00:03	1,858
1,024	00:00:18	2,481	00:00:15	2,432	00:00:14	2,410	00:00:14	2,534
2,048	00:01:41	5,300	00:01:21	5,329	00:01:11	5,395	00:01:09	5,487

(b) Conjugate gradient

	1 thread		2 threads		3 threads		4 threads	
N	Time	Memory	Time	Memory	Time	Memory	Time	Memory
32	<00:00:01	1,542	<00:00:01	1,541	<00:00:01	1,558	<00:00:01	1,557
64	<00:00:01	1.546	<00:00:01	1,540	<00:00:01	1,535	<00:00:01	1,549
128	<00:00:01	1,504	<00:00:01	1,549	<00:00:01	1,590	<00:00:01	1,504
256	00:00:04	1,601	00:00:04	1,593	00:00:04	1,578	00:00:04	1,678
512	00:00:35	1,613	00:00:31	1,614	00:00:29	1,687	00:00:30	1,648
1,024	00:04:28	1,801	00:05:08	1,823	00:03:53	1,852	00:05:52	1,814
2,048	00:43:27	2,676	00:42:57	2,505	00:42:00	2,476	00:43:01	2,493

(c) Matrix-free implementation

	1 thread		2 threads		3 threads		4 threads	
N	Time	Memory	Time	Memory	Time	Memory	Time	Memory
32	<00:00:01	1,557	<00:00:01	1,532	<00:00:01	1,547	<00:00:01	1,516
64	<00:00:01	1,523	<00:00:01	1,526	<00:00:01	1,542	<00:00:01	1,521
128	<00:00:01	1,544	<00:00:01	1,545	<00:00:01	1,504	<00:00:01	1,530
256	00:00:06	1,536	00:00:05	1,625	00:00:06	1,622	00:00:06	1,583
512	00:00:48	1,574	00:00:37	1,638	00:00:56	1,576	00:00:47	1,618
1,024	00:07:27	1,653	00:06:48	1,618	00:07:10	1,655	00:07:22	1,654
2,048	01:10:30	1,813	00:53:40	1,990	00:52:02	1,939	01:08:57	1,958

Acknowledgements

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] Kevin P. Allen and Matthias K. Gobbert. Coarse-grained parallel matrix-free solution of a three-dimensional elliptic prototype problem. In Vipin Kumar, Marina L. Gavrilova, Chih Jeng Kenneth Tan, and Pierre L'Ecuyer, editors, *Computational Science and Its Applications—ICCSA 2003*, vol. 2668 of *Lecture Notes in Computer Science*, pp. 290–299. Springer-Verlag, 2003.

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

[3] Matthias K. Gobbert. Parallel performance studies for an elliptic test problem. Technical Report HPCF–2008–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2008.

[4] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 1996.

[5] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley, second edition, 2002.