CG Method for the Poisson Problem

Ashar Nadeem
Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County

Abstract

Using the HPCF, the Poisson Problem can be solved using the CG Method. Parallel performance studies can be run to test and analyze the performance of the solution, especially when dealing with low memories at very large test values. Scalability studies and their results showed excellent speedup for both blocking and non-blocking implementations, while efficiency was excellent at lower processes for all mesh values and also for large processes on the median range of mesh values. Results showed parallel programming increasing speed and efficiency across the tests run on the HPCF cluster, with blocking implementation being faster and more efficient.

1 Introduction

The overall goal of the study is based around using parallel programing to test the run time, efficiency, and memory usage for the Poisson equation. Poisson's equation is a partial differential equation that is used in mathematics and theoretical physics. We aim to solve this equation in our code and eventually speed up the run times through making the program parallel.

For our study in particular, we first start off by implementing the serial code first, and then we move on to the parallel code. This ensures that we can see that we are getting correct solutions first, and we can move on from there on making the code more efficient. We want to not only test the code speeding up but also the efficiency of this code as it is run across various combinations of nodes and processes. Running the code on the HCPF, we use the high_mem and develop partitions, running our code on up to 32 nodes. Both blocking and non-blocking solutions were implemented, and the differences between them are explored and a recommendation is made, which is that blocking is the better implementation for what we are trying to achieve.

The remainder of this report is organized as follows: Section 2 is the formal problem statement. Section 3 goes into the details of the numerical method, the implementation of the C code, and a high-level overview of how the program was made parallel. Section 4 shows the results of our study on the HPCF cluster with various resource allocations. Section 5 contains a very brief conclusion summarizing the actions taken and the very high-level conclusion reached after the end of the study. The last section is acknowledgements and references to material used throughout the writing of this report.

2 Problem Statement

Poisson's equation is a partial differential equation that is used in mathematics and theoretical physics, in areas such as heat flow or fluid dynamics. In order to solve the Poisson's problem, we can use the conjugate method. It can be used to solve any linear system for coefficient matrices that are symmetric. The first bullet point below represents the classical problem and its right-hand equation. The second bullet point has our boundary conditions,

- $f(x_1, x_2) = (-2\pi_2)(\cos[2\pi x_1] \sin^2[\pi x_2] + \sin^2[\pi x_1] \cos[2\pi x_2])$
- $-\Delta u = f$ in Ω, $u = 0$ on $\partial\Omega$

3 Implementation

The conjugate method is useful to us as we can use it to save memory, resulting in a matrix free implementation that proves useful at higher values of N when memory becomes an issue, unlike gaussian elimination. The solution is elegant in the sense that it only needs a few algebraic operations to achieve most of our tasks. We can call the approximate solution, residual, and search direction at each step. The majority of the work is matrix-vector multiplication.

MPI was used to make the code parallel. MPI_Wtime was used to keep track of the run time of our program. Functions were provided, edited, and used to solve the dot product of our matrix in parallel using MPI_Scatter and MPI_Allreduce. MPI_Send and MPI_Recv functions were used to send and receive messages between our processes as we spread out and load balance our calculations across all of our resources. MPI_Send was used before MPI_Recv in order to avoid a deadlock. We have an if statement for our rank 0 process so that we are able to use the send and receive data in the correct manner, with send before receive. A non-blocking implementation was explored with MPI_Isend and MPI_Irecv. This non-blocking returns immediately and does not wait for communication to finish, so we can post a receive and wait for a send.

4 Results

This section reviews the results of the performance studies ran on the HPCF cluster using the MPI code as well as some initial serial code for testing. To best be able to gather some useful information on our code, its speed, and its efficiency, the MPI code was ran for different mesh resolution values. For each mesh resolution for the meshes with N = 32, 64, 128, 256, 512, 1024, 2048, 4096, and 8192, the parallel version of the code was run on 1, 2, 4, 8, and 16 nodes with processes per node of 1, 2, 4, 8, 16, and 32.

Table 4.1 on Page 4 displays the results of the study run on 1 node. The study was running alone on the node and the metrics were measured to see speed and efficiency of the code. The table has columns for the mesh resolution, degrees of freedom, norm of the finite difference error, iterations, wall clock time in seconds, predicted memory, and actual memory. Results were obtained for up to mesh resolution N = 8192. The wall clock time is rounded to the nearest second in this case for the sake of clean results. Additionally, any run time below 1 second is simply represented by '< 1'. The norm of finite difference is rounded to 2 decimal places, also for the sake of clean, readable results. For the same purposes, memory values are rounded to the nearest integer. We can draw some conclusions about our study from our results. First of all, we can look at the predicted memory versus the actual memory and see that we almost always use more memory. Some brief math shows a range of 2% - 13% additional memory actually used. Additionally, the memory per iteration of N increases at a multiple of 4. At higher values of N, we would surely run into memory issues at we could possibly exceed the resources available to us at the HPCF. Secondly, our wall clock time seems to increase roughly tenfold (give or take) per increase in N value. That is to say that in seconds, we take roughly 10 times as long to run double the amount of mesh resolution. Looking at these values, it is clear that continuing to run this code in serial would for increasing mesh resolutions would offer scenarios where we would exceed our run time limits. Making this code parallel would be essential to save time and computational cost. Lastly, we can look to norms, and see that we are decreasing by roughly a factor of 2 each time.

Table 4.2 on Pages 4 and 5 shows the results of the observed wall clock time, in total seconds. The wall clock time is rounded to the nearest second in this case for the sake of clean results. Additionally, any run time below 1 second is simply represented by '< 1', for the same reasons. Immediately we can see that for any mesh below $N = 1024$, all results we obtained were below one second. This shows us that the program was executing almost immediately in these cases and the more visual and clear information would be obtained at greater values. Let us take for example the mesh $N = 8192$. We can observe that when we double the nodes used, our code runs roughly twice as fast. That is to say that we are halving the wall clock time every time we double the number of parallel processes. Similarly, we can see that this is fairly constant across different values of nodes and processes per node combinations. So, 2 nodes with 1 process per node gives us similar results to 1 node with 2 processes (though not exact). Lastly, another thing we observe is that by doubling the mesh resolution, we are roughly increasing the wall clock time by a factor of between 7 and 9. That is to say that it takes roughly 7 to 9 times more time to run the code every time we double the size of the mesh.

Table 4.3 on Page 6 shows a scalability study. We see our raw wall clock times in the first part of the table when the code is run on different numbers of processes (factors of 2). We then break down the observed speedup and observed efficiency for each increase in number of processes. For meshes below $N = 512$, we use NA to denote 'Not Available'. This is due to the run time being so small the calculations start throwing numerical errors during calculations, including essentially divide by zero errors. Thus, observed speedup and observed efficiency are shown for the higher mesh values. Furthermore, Figure 4.1 on Page 7 shows the observed speedup and efficiency in plot form for some data visualization. They simply summarize the information shown for observed speedup and efficiency in Table 4.3. We see some very interesting results from the plot. Up to $N = 4096$, the observed speedup is below or just at optimal for all values of p, and for values greater than $N >= 4096$, the observed speedup is much higher than optimal across all values of p. Efficiency is where the results get even more interesting. Up until $p = 16$, essentially all the meshes perform in an above optimal manner. After that, our highest and lowest N values, 512 and 8096 both tank to suboptimal efficiency. Our other meshes hover within a certain bounding box of optimal efficiency until $p = 256$, where the $N = 2048$ mesh goes suboptimal, and $N = 4096$ sharply rises. Meanwhile, $N = 1024$ stays constant afterwards. Looking at the results of our plots, it seems safe to say that the most optimal code in terms of tradeoff between speedup and efficiency would be $N = 4096$, and $p = 512$. Here we see both the most optimal speedup and the most efficiency.

Table 4.4 on Page 9 shows the observed wall clock times when non-blocking MPI commands are used. Being asynchronous, the non-blocking implementation returns immediately while the synchronous blocking implementation waits for the communication to finish. The wall clock time is rounded to the nearest second in this case for the sake of clean results. Additionally, any run time below 1 second is simply represented by '< 1', This time, we choose to disregard values of N less than 1024, as they returned < 1 for all combinations of nodes and processes per node that we used. Thus, we are throwing away some junk data and focusing on the visible results. The non-blocking study was run for mesh resolutions 1024, 2048, 4096, and 8192. We see that implementing non-blocking MPI commands either returns the same result or one that is slightly higher. For the lower mesh values the run time was virtually identical, and as we made our mesh resolution larger and larger, we began to see the run time increase by an average of 5%-20%. The difference in run time is very noticeable in $N = 8192$ in particular and would only grow larger as N increases. Hence, we see that not only does blocking code not dead lock, but it is also much more efficient.

Table 4.1: Convergence Study of Finite Difference Method

N	DOF	Iterations	WCT	$\|\|u - u_h\|\|$	Predicted Memory	Actual Memory
32	1024	48	< 1	2.95e-03	32 KB	35 KB
64	4096	96	< 1	8.15e-04	128 KB	131 KB
128	16384	192	< 1	1.69e-04	512 KB	540 KB
256	65536	387	< 1	4.20e-05	2 MB	2 MB
512	262144	783	< 1	2.99e-06	8 MB	9 MB
1024	1048576	1581	7	8.13e-07	32 MB	36 MB
2048	4194304	3192	99	5.52e-07	128 MB	129 MB
4096	16777216	6452	841	1.52e-07	512 MB	515 MB
8192	67108864	13033	6942	2.83e-08	2 GB	2 GB

* WCT: Wall Clock Time (Seconds)
* DOF: Dimension of Linear System
* $||u - u_h||$: Norm of Finite Difference Error

Table 4.2: Wall Clock Time in Seconds – MPI Code

Mesh Resolution N X N = 32 X 32					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	< 1	< 1	< 1	< 1	< 1
2 processes per node	< 1	< 1	< 1	< 1	< 1
4 processes per node	< 1	< 1	< 1	< 1	< 1
8 processes per node	< 1	< 1	< 1	< 1	< 1
16 processes per node	< 1	< 1	< 1	< 1	< 1
32 processes per node	< 1	< 1	< 1	< 1	< 1
Mesh Resolution N X N = 64 X 64					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	< 1	< 1	< 1	< 1	< 1
2 processes per node	< 1	< 1	< 1	< 1	< 1
4 processes per node	< 1	< 1	< 1	< 1	< 1
8 processes per node	< 1	< 1	< 1	< 1	< 1
16 processes per node	< 1	< 1	< 1	< 1	< 1
32 processes per node	< 1	< 1	< 1	< 1	< 1
Mesh Resolution N X N = 128 X 128					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	< 1	< 1	< 1	< 1	< 1
2 processes per node	< 1	< 1	< 1	< 1	< 1
4 processes per node	< 1	< 1	< 1	< 1	< 1
8 processes per node	< 1	< 1	< 1	< 1	< 1
16 processes per node	< 1	< 1	< 1	< 1	< 1
32 processes per node	< 1	< 1	< 1	< 1	< 1
Mesh Resolution N X N = 256 X 256					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	< 1	< 1	< 1	< 1	< 1
2 processes per node	< 1	< 1	< 1	< 1	< 1
4 processes per node	< 1	< 1	< 1	< 1	< 1
8 processes per node	< 1	< 1	< 1	< 1	< 1
16 processes per node	< 1	< 1	< 1	< 1	< 1
32 processes per node	< 1	< 1	< 1	< 1	< 1

Mesh Resolution N X N = 512 X 512					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	< 1	< 1	< 1	< 1	< 1
2 processes per node	< 1	< 1	< 1	< 1	< 1
4 processes per node	< 1	< 1	< 1	< 1	< 1
8 processes per node	< 1	< 1	< 1	< 1	< 1
16 processes per node	< 1	< 1	< 1	< 1	< 1
32 processes per node	< 1	< 1	< 1	< 1	< 1
Mesh Resolution N X N = 1024 X 1024					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	8	4	2	< 1	< 1
2 processes per node	3	1	< 1	< 1	< 1
4 processes per node	1	< 1	< 1	< 1	< 1
8 processes per node	< 1	< 1	< 1	< 1	< 1
16 processes per node	< 1	< 1	< 1	< 1	< 1
32 processes per node	< 1	< 1	< 1	< 1	< 1
Mesh Resolution N X N = 2048 X 2048					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	104	48	19	7	4
2 processes per node	44	19	7	3	2
4 processes per node	21	10	4	2	1
8 processes per node	11	5	2	< 1	< 1
16 processes per node	9	3	1	< 1	< 1
32 processes per node	3	< 1	< 1	< 1	< 1
Mesh Resolution N X N = 4096 X 4096					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	876	420	202	101	38
2 processes per node	421	221	121	45	25
4 processes per node	219	155	59	29	11
8 processes per node	152	65	35	16	6
16 processes per node	88	34	19	7	3
32 processes per node	39	17	7	3	1
Mesh Resolution N X N = 8192 X 8192					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	7396	3690	1813	884	421
2 processes per node	3655	1827	855	441	206
4 processes per node	1819	904	462	203	99
8 processes per node	898	491	291	103	74
16 processes per node	443	250	122	85	51
32 processes per node	331	175	103	69	45

Table 4.3: Strong Scalability Study of MPI Code

Wall Clock Time (Seconds)										
N	p = 1	p = 2	p = 4	p = 8	p = 16	p = 32	p = 64	p = 128	p = 256	p = 512
32	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
64	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
128	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
256	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
512	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
1024	8	4	2	< 1	< 1	< 1	< 1	< 1	< 1	< 1
2048	104	48	19	7	4	3	< 1	< 1	< 1	< 1
4096	876	420	202	101	38	39	17	7	3	1
8192	7396	3690	1813	884	421	331	175	103	69	45
Observed Speedup										
N	p = 1	p = 2	p = 4	p = 8	p = 16	p = 32	p = 64	p = 128	p = 256	p = 512
32	1.00	NA	NA	NA	NA	NA	NA	NA	NA	NA
64	1.00	NA	NA	NA	NA	NA	NA	NA	NA	NA
128	1.00	NA	NA	NA	NA	NA	NA	NA	NA	NA
256	1.00	NA	NA	NA	NA	NA	NA	NA	NA	NA
512	1.00	1.90	2.96	12.33	18.50	24.67	37.00	74.00	74.00	148.00
1024	1.00	2.00	4.00	9.41	21.05	36.36	72.73	160.00	400.00	804.00
2048	1.00	2.17	5.47	14.86	26.00	34.67	105.05	140.54	260.00	400.00
4096	1.00	2.09	4.34	8.67	23.05	22.46	51.53	125.14	292.00	876.00
8192	1.00	2.00	4.08	8.37	17.57	22.34	42.26	71.81	107.19	164.36
Observed Efficiency										
N	p = 1	p = 2	p = 4	p = 8	p = 16	p = 32	p = 64	p = 128	p = 256	p = 512
32	1.00	NA	NA	NA	NA	NA	NA	NA	NA	NA
64	1.00	NA	NA	NA	NA	NA	NA	NA	NA	NA
128	1.00	NA	NA	NA	NA	NA	NA	NA	NA	NA
256	1.00	NA	NA	NA	NA	NA	NA	NA	NA	NA
512	1.00	1.01	0.84	0.92	0.95	0.57	0.39	0.21	0.12	0.06
1024	1.00	1.00	1.00	1.02	1.00	1.03	1.00	1.02	0.98	1.00
2048	1.00	1.09	1.37	1.86	1.63	1.08	1.14	1.18	1.14	1.04
4096	1.00	1.05	1.09	1.08	1.44	0.70	0.81	0.98	1.14	1.71
8192	1.00	1.00	1.02	1.05	1.10	0.70	0.66	0.56	0.42	0.32

* Wall Clock Time (T_p), Observed Speedup (S_p) = $\frac{T_1}{T_p}$, Observed Efficiency (E_p) = $\frac{S_p}{p}$.

Figure 4.1: Strong Scalability Study of MPI Code

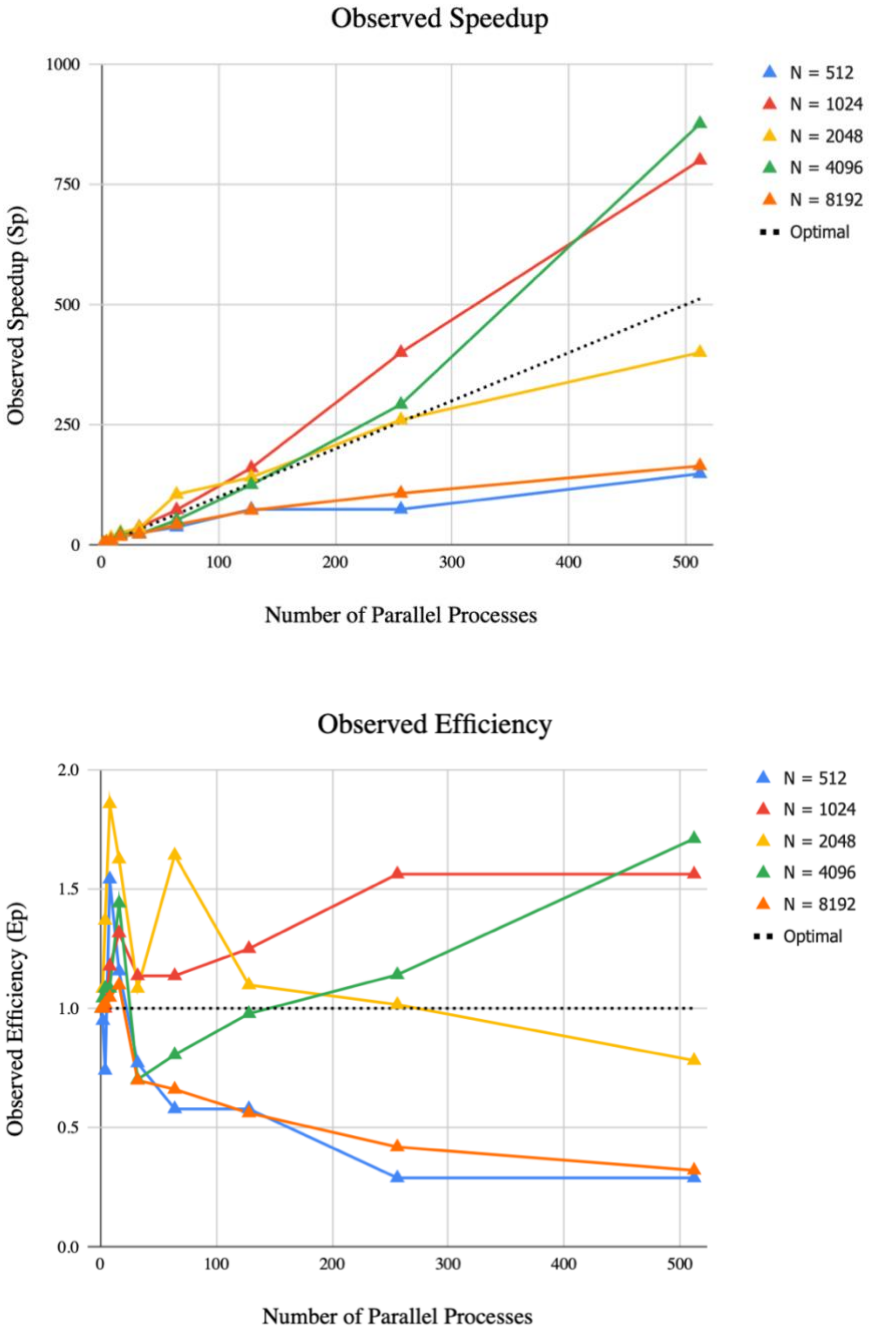

Table 4.4: Wall Clock Time in Seconds – MPI Code (Non-Blocking)

Mesh Resolution N X N = 1024 X 1024					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	9	4	3	2	< 1
2 processes per node	3	2	1	< 1	< 1
4 processes per node	1	< 1	< 1	< 1	< 1
8 processes per node	< 1	< 1	< 1	< 1	< 1
16 processes per node	< 1	< 1	< 1	< 1	< 1
32 processes per node	< 1	< 1	< 1	< 1	< 1
Mesh Resolution N X N = 2048 X 2048					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	105	46	22	9	4
2 processes per node	44	20	8	4	2
4 processes per node	23	10	4	3	1
8 processes per node	11	6	3	< 1	< 1
16 processes per node	9	3	1	< 1	< 1
32 processes per node	4	< 1	< 1	< 1	< 1
Mesh Resolution N X N = 4096 X 4096					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	890	421	203	102	41
2 processes per node	425	225	151	49	26
4 processes per node	221	153	69	32	12
8 processes per node	152	66	38	11	13
16 processes per node	89	33	20	9	5
32 processes per node	41	16	8	4	2
Mesh Resolution N X N = 8192 X 8192					
	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes
1 process per node	7582	3928	1756	911	455
2 processes per node	3720	1922	1006	451	237
4 processes per node	1937	953	470	254	111
8 processes per node	988	524	299	135	87
16 processes per node	462	273	139	92	53
32 processes per node	349	192	126	84	53

5 Conclusion

At the conclusion of this study, we were to successfully implement parallel code in C to solve the Poisson problem using the conjugant gradient method. The parallel code was tested for accuracy to ensure correct results were returned. Both blocking and non-blocking implementations of the code were carried out and tested upon. There was a definite decrease in time taken to run the program as additional resources on the HCPF were allocated to the program, and we saved large amounts of time in calculating very large mesh values. When non-blocking was compared to blocking MPI commands, it was found that non-blocking took about the same time or longer, depending on the size of the mesh resolution. However, with large mesh resolutions, the blocking code was up to 20% faster, making it the dominantly better solution to our problem. Further studies could include exploring the effects of running the program on a different cluster, using 32 nodes on the HPCF, or pushing the boundaries of our resources and testing for mesh resolutions greater than 8192. Some more material on the topic is linked below.

- University of California, Berkley – CS 240
- University of California, Berkley – CS 267
- University of Pittsburgh – MATH 2071

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 nos. CNS-0821258, CNS-1228778, and OAC-1726023) and the SCREMS program (grant no. DMS-0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

References

1. Carlos Barajas and Matthias K. Gobbert. Strong and Weak Scalability Studies for the 2-D Poisson Equation on the Taki 2018 Cluster. Technical Report HPCF–2019–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2018 (link)
2. Mei Yin, Different General Algorithms for Solving Poisson Equation, Nanjing University of Science and Technology, 1999 (link)
3. James Demmel, Solving the Discrete Poisson Equation using Jacobi, SOR, Conjugate Gradients, and the FFT, University of California Berkley, 1996 (link)