Contents lists available at Science-Gate

International Journal of Advanced and Applied Sciences

Journal homepage: http://www.science-gate.com/IJAAS.html

Advanced efficient iterative methods to the Helmholtz equation

A. G. Shaikh [1,*], Wajid Shaikh [2], A. H. Shaikh [3], Muhammad Memon [1]

[1]Department of BS and RS, Quaid-e-Awam University of Engineering, Science and Technology, Nawabshah, Pakistan
[2]Department of Mathematics and Statistics, Quaid-e-Awam University of Engineering, Science and Technology, Nawabshah, Pakistan
[3]Department of Mathematics, Institute of Business Management, Karachi, Pakistan

ARTICLE INFO

Article history:
Received 10 December 2021
Received in revised form
24 March 2022
Accepted 12 April 2022

Keywords:
API
Fork join
Master thread
Parallel computing
OpenMP

ABSTRACT

Parallel computing has recently gained widespread acceptance as a means of handling very large computational data. Since iterative methods are appealing for large systems of equations, and they are the prime candidates for implementations on parallel architectures, We presented based on exploration, through virtual technology having 30 cores, in literature solutions of Helmholtz equation is available up to 12 cores by Jacobi method, here we increased the number of cores and virtual machine having 30 cores first time used to find the solution of Helmholtz equation, our findings are encouraging and found that parallel computing by OpenMP implementations is effective on current supercomputing as well as virtual machine platforms and that is an auspicious programming model to use for applications to be run on emerging and future platforms with accelerated nodes.

1. Introduction

Parallel computing means doing things simultaneously, we may be doing the same thing or something different simultaneously, or parallel computing (Eisenstat et al., 1983; Bogaerts et al., 2002). The idea of a single processor computer is fast becoming outmoded and gorgeous. Before taking an excise on Parallel Processing, first of all, let us look at the background of computations of computer software and why it is futile for the modern era. The performance of the computer is an increase in speed, that is if we use a single computer, it takes T amount of time to perform a job, then using two computers the same computers should cut the time, and, it takes to perform that same job in $\frac{T}{2}$ (half time of amount) and, whereas using four of the same computers should cut the time it takes to perform a job, in $\frac{T}{4}$, and so on, a kind of embarrassing parallel computing, however, in practical terms, this is impossible. The computer software was written conventionally for sequential processing. This meant that to evaluate a program, an execution divides the program into smaller instructions. These discrete instructions are then executed on the CPU of a computer one by one after one instruction is finished, the next will start, but dealing with a massive size of problems, it is impossible to do so, but need a fast solution for practical point of view, and the fastest solution iterative method is not always the best solution method (Ianculescu and Thompson, 2006; Operto et al., 2007). The change in the strategy; parallel computing played a role to adjust the strategies when it comes to computing. Meanwhile, it is impossible to improve the speed of a computer with the help of a single processor, which consumes unacceptable power causing heat issues, on the contrary, if an application is not running fast on a single computer machine; it will run even slower on new machines having many processors unless it takes the advantage of parallel processing. Nowadays, parallel computing is a dominant player in simulation analysis and scientific computing. Parallel computing refers to the process of breaking down the larger tasks into smaller, independent, usually similar tasks, that can be executed simultaneously by multiple processors and these processors will communicate through shared memory. The primary goal of parallel computing is to take the advantage of all available resources of the computer. OpenMP parallel Computing examines with the help of a virtual machine to solve the Helmholtz differential equation (Nabavi et al., 2007; Umetani et al., 2009; Zhu et al., 2010), by, the Jacobean Method iteratively with a different number

* Corresponding Author.
Email Address: agshaikh@quest.edu.pk (A. G. Shaikh)
https://doi.org/10.21833/ijaas.2022.06.020
Corresponding author's ORCID profile:
https://orcid.org/0000-0001-7367-993X

of cores, speed, and efficiency tested with a different number of processors.

2. OpenMP

The OpenMP is a computer language; it works together with either standard Fortran or C/C++. It is comprised of a set of compiler directives that demonstrate the parallelism in the foundation code, along with a supporting library of subroutines existing to applications. Jointly, these directives and library routines are properly described by the application programming interface (API), now a day's OpenMP (Puzyrev et al., 2013), which allows sending "request" any number of threads of execution. This is not every time a suitable request. If the system has four to thirty processors available, and they're not busy doing other things or serving else one another, maybe four or more than four, threads are what we want. Moreover, if we run the same program by using one thread and four or more than four threads, we may find that using more than one thread, the machine may slow down, either because we don't actually have four processors or because the processors are also busy doing other things.

2.1. Fork join

OpenMP works based on the Fork-Join model for parallel execution in Fig. 1 that is all OpenMP parallel programs start with a single thread a master thread. The master thread executes instructions sequentially till interned in the parallel region, after interning in the parallel region the master thread creates a spawn of threads, several threads depending on the CPU possesses, (also the number of threads requested), and after completing the execution simultaneously all the threads will join the master thread (Ping and Ali, 2014), and all threads will terminate except master thread, again master create a threads spawn to execute the instruction and this process will continue till the final result of the execution as shown in Fig. 1.

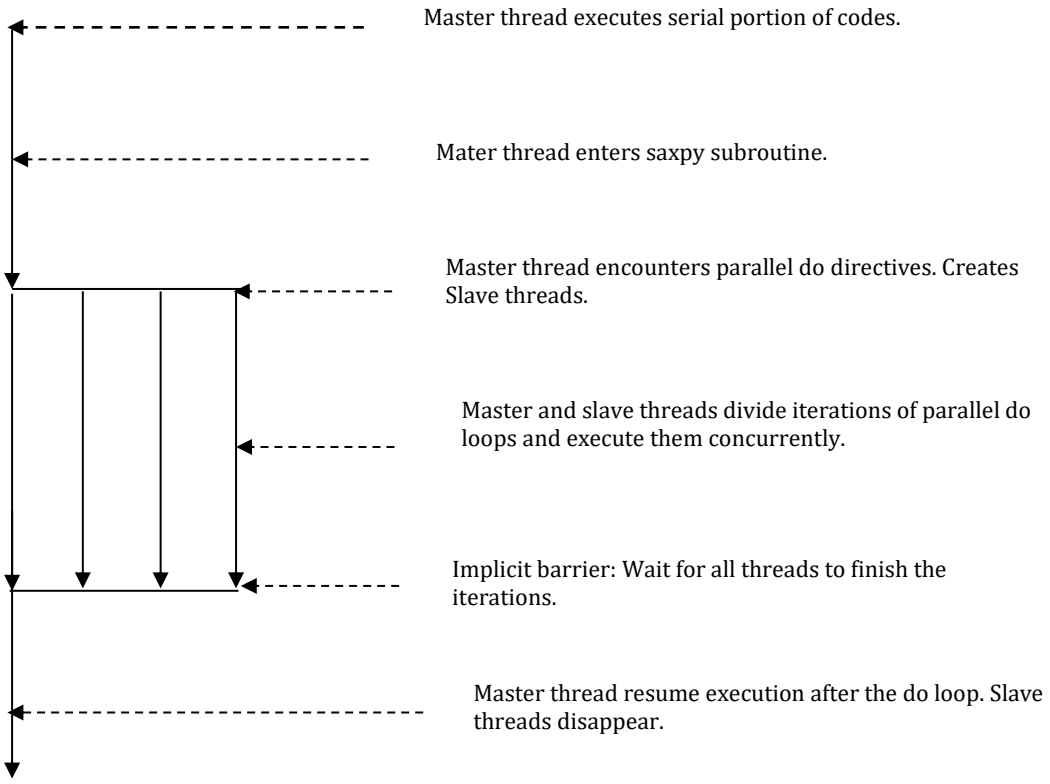

Fig. 1: Fork-join model with, 4 threads to execute the data

2.2. Shared memory

Multiple processors can operate independently but share the same resources of memory, or one large bank of memory, different computer cores acting on it, and each code is assigned threads of execution of a single program that acts on data. The changes in memory locations caused by one CPU are visible to all processors. Different CPUs communicate with each other through shared memory, in the language of Parallel computing cores are renowned as Threads.

3. Root mean square

The Residual-Root-Mean-Square represents the square root of the second sample instant of the differences between predicted values and observed values. These deviations are called Residuals when the calculations are performed over the data, which were used for errors when computed out of the data. RMS is always non-negative, and a value of zero (almost impossible in real life) which indicates a perfect solution.

3.1. Model problem Helmholtz equation by Jacobi method

The solution of discretized Helmholtz Equation employing Parallel Computing with OpenMP. The two-dimensional region given is:

$$-1 \le x \le 1, \ and \ -1 \le y \le 1. \quad (1)$$

The region is discretized by a set of $M \times N$ grid points:
For,

$$0 \le i \le M-1, 0 \le j \le N-1, (\text{C/C++}). \quad (2)$$

The Helmholtz problem for the unknown function $u(x, y)$ is:

$$-U_{xx}(x,y) - U_{yy}(x,y) + K * u(x,y) = G(x,y). \quad (3)$$

The discretized differential equation becomes a set of a linear system of equations of the form:

$$A * U = G. \quad (4)$$

This linear system is then solved using a form of the Jacobi method with ω=1.5.

3.2. Solution of the Helmholtz equation

OpenMP version (This program ran in parallel).
Available Processors=30
Requested Thread=1
The region is [-1,1] x [-1,1].

The grid points in the x-axis direction M=10000, and grid points in the y-axis N=10000, therefore, the variables in the linear system $M \times N$=100000000. The constant wavenumber is taken K=100. The relaxation is:

ω=1.5, and the tolerance is, Tol=0.0E-12, Its-Max=50.

For the sake of simplicity, only two results are presented in the form of Tables 1 and 2. The results with different numbers of threads are presented in the form of Table 3 and Fig. 2.

Table 1: Iterations obtained requesting one thread by visual studio 2017 by (C/C++)

1 Residual RMS 5.601189e-11	26 Residual RMS 5.600965e-11
2 Residual RMS 5.601180e-11	27 Residual RMS 5.600956e-11
3 Residual RMS 5.601171e-11	28 Residual RMS 5.600947e-11
4 Residual RMS 5.601162e-11	29 Residual RMS 5.600938e-11
5 Residual RMS 5.601153e-11	30 Residual RMS 5.600929e-11
6 Residual RMS 5.601144e-11	31 Residual RMS 5.600920e-11
7 Residual RMS 5.601135e-11	32 Residual RMS 5.600912e-11
8 Residual RMS 5.601126e-11	33 Residual RMS 5.600903e-11
9 Residual RMS 5.601117e-11	34 Residual RMS 5.600894e-11
10 Residual RMS 5.601108e-11	35 Residual RMS 5.600885e-11
11 Residual RMS 5.601099e-11	36 Residual RMS 5.600876e-11
12 Residual RMS 5.601090e-11	37 Residual RMS 5.600867e-11
13 Residual RMS 5.601081e-11	38 Residual RMS 5.600858e-11
14 Residual RMS 5.601072e-11	39 Residual RMS 5.600849e-11
15 Residual RMS 5.601063e-11	40 Residual RMS 5.600840e-11
16 Residual RMS 5.601054e-11	41 Residual RMS 5.600832e-11
17 Residual RMS 5.601045e-11	42 Residual RMS 5.600823e-11
18 Residual RMS 5.601036e-11	43 Residual RMS 5.600814e-11
19 Residual RMS 5.601027e-11	44 Residual RMS 5.600806e-11
20 Residual RMS 5.601018e-11	45 Residual RMS 5.600801e-11
21 Residual RMS 5.601010e-11	46 Residual RMS 5.600807e-11
22 Residual RMS 5.601001e-11	47 Residual RMS 5.600854e-11
23 Residual RMS 5.600992e-11	48 Residual RMS 5.601055e-11
24 Residual RMS 5.600983e-11	49 Residual RMS 5.601834e-11
25 Residual RMS 5.600974e-11	50 Residual RMS 5.604793e-11

Computed solution L2 norm: 0.420073; Time taken by execution=22.166349; # Available processors=30# requested threads=30; The region is [-1,1] x [-1,1]

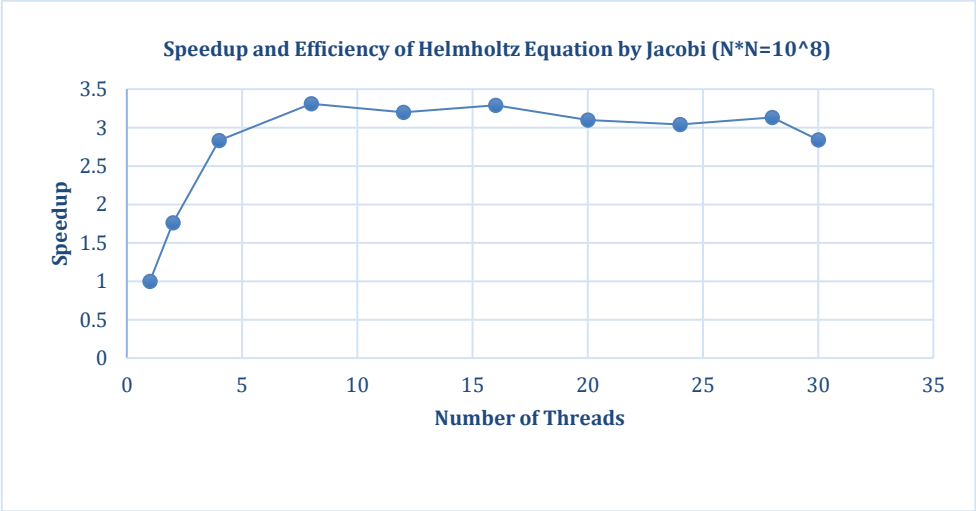

Fig. 2: Speedup with an increasing number of cores

Table 2: Iterations obtained requesting and thirty cores (threads) by visual studio 2017 by (C/C++)

1 Residual RMS 5.601189e-11	26 Residual RMS 5.600965e-11
2 Residual RMS 5.601180e-11	27 Residual RMS 5.600956e-11
3 Residual RMS 5.601171e-11	28 Residual RMS 5.600947e-11
4 Residual RMS 5.601162e-11	29 Residual RMS 5.600938e-11
5 Residual RMS 5.601153e-11	30 Residual RMS 5.600929e-11
6 Residual RMS 5.601144e-11	31 Residual RMS 5.600920e-11
7 Residual RMS 5.601135e-11	32 Residual RMS 5.600912e-11
8 Residual RMS 5.601126e-11	33 Residual RMS 5.600903e-11
9 Residual RMS 5.601117e-11	34 Residual RMS 5.600894e-11
10 Residual RMS 5.601108e-11	35 Residual RMS 5.600885e-11
11 Residual RMS 5.601099e-11	36 Residual RMS 5.600876e-11
12 Residual RMS 5.601090e-11	37 Residual RMS 5.600867e-11
13 Residual RMS 5.601081e-11	38 Residual RMS 5.600858e-11
14 Residual RMS 5.601072e-11	39 Residual RMS 5.600849e-11
15 Residual RMS 5.601063e-11	40 Residual RMS 5.600840e-11
16 Residual RMS 5.601054e-11	41 Residual RMS 5.600832e-11
17 Residual RMS 5.601045e-11	42 Residual RMS 5.600823e-11
18 Residual RMS 5.601036e-11	43 Residual RMS 5.600814e-11
19 Residual RMS 5.601027e-11	44 Residual RMS 5.600806e-11
20 Residual RMS 5.601018e-11	45 Residual RMS 5.600801e-11
21 Residual RMS 5.601010e-11	46 Residual RMS 5.600807e-11
22 Residual RMS 5.601001e-11	47 Residual RMS 5.600854e-11
23 Residual RMS 5.600992e-11	48 Residual RMS 5.601055e-11
24 Residual RMS 5.600983e-11	49 Residual RMS 5.601834e-11
25 Residual RMS 5.600974e-11	50 Residual RMS 5.604793e-11

Computed U L2 norm: 0.420073; Time taken by execution=65.262351

Table3: Comparative time taken by different numbers of threads

Speedup and efficiency of Helmholtz differential equation by Jacobi with C/C++ Time taken by different threads, with dimensions N*N=10^8										
# Threads	#1	#2	#4	#8	#12	#16	#20	#24	#28	#30
Time(S)	65.26	37.18	23.10	19.69	20.41	19.86	21.07	21.48	20.86	22.16
Speedup	1	1.76	2.83	3.31	3.20	3.29	3.10	3.04	3.13	2.84
Efficiency	100	88	71	41	27	21	16	13	11	09

4. Conclusion

The main Server with the help of a virtual Technology having 30 (Thirty Processors) is applied, to solve the Helmholtz Equation, by the Jacobi method, and relaxation parameter "1.5" with OpenMP Parallel Computing, to examine with a different number of cores. It is notable that as we increase the number of Threads/Processors/Cores performance also increases, but huge increase in the number of Cores may get either no any improvement in performance or may decrease the performance, it means an increase number of processors is not giving a guarantee of good performance sometimes increases the number of processors put extra burden on computation, resulting slow down the performance, this shows from Fig. 2, we started with simply 1 processor, 2 processors, then 4 processors to solve same Helmholtz equation, using Visual Studio 2017 in C/C++, after that 8 processors each times an increase the number of processors getting increases in the performance, but when we further increase the number of cores resulting decrease the speed (performance), rather than increase the performance, it put an extra burden on CPU causes hindrance in execution, so an increase the number of cores will increases the performance, but generally it is not true, that is an increase the number of cores will increase the performance of computation. Few results with one thread and thirty threads in terms of Residual Roots Mean Square in Tables 1 and 2 and found satisfactory.

Acknowledgment

The authors would like to thank all researchers who participated in this study and gratefully acknowledge the support of the Quaid-e-Awam University of Engineering, Science and Technology Nawabshah Pakistan.

Compliance with ethical standards

Conflict of interest

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

References

Bogaerts A, Neyts E, Gijbels R, and Van der Mullen J (2002). Gas discharge plasmas and their applications. Spectrochimica Acta Part B: Atomic Spectroscopy, 57(4): 609-658. https://doi.org/10.1016/S0584-8547(01)00406-2

Eisenstat SC, Elman HC, and Schultz MH (1983). Variational iterative methods for nonsymmetric systems of linear equations. SIAM Journal on Numerical Analysis: Peer-Reviewed Journal, 20(2): 345-357. https://doi.org/10.1137/0720023

Ianculescu C and Thompson LL (2006). Parallel iterative solution for the Helmholtz equation with exact non-reflecting boundary conditions. Computer Methods in Applied Mechanics and Engineering, 195(29-32): 3709-3741. https://doi.org/10.1016/j.cma.2005.02.030

Nabavi M, Siddiqui MK, and Dargahi J (2007). A new 9-point sixth-order accurate compact finite-difference method for the

Helmholtz equation. Journal of Sound and Vibration, 307(3-5): 972-982. https://doi.org/10.1016/j.jsv.2007.06.070

Operto S, Virieux J, Amestoy P, L'Excellent JY, Giraud L, and Ali HBH (2007). 3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study. Geophysics, 72(5): SM195-SM211. https://doi.org/10.1190/1.2759835

Ping TW and Ali NHM (2014). Higher order rotated iterative scheme for the 2D Helmholtz equation. In the AIP Conference Proceedings: 21st National Symposium on Mathematical Sciences, AIP Publishing LLC, Penang, Malaysia, 1605: 155-160. https://doi.org/10.1063/1.4887581

Puzyrev V, Koldan J, de la Puente J, Houzeaux G, Vázquez M, and Cela JM (2013). A parallel finite-element method for three-dimensional controlled-source electromagnetic forward modelling. Geophysical Journal International, 193(2): 678-693. https://doi.org/10.1093/gji/ggt027

Umetani N, MacLachlan SP, and Oosterlee CW (2009). A multigrid-based shifted Laplacian preconditioner for a fourth-order Helmholtz discretization. Numerical Linear Algebra with Applications, 16(8): 603-626. https://doi.org/10.1002/nla.634

Zhu J, Ping XW, Chen RS, Fan ZH and Ding DZ (2010). An incomplete factorization preconditioner based on shifted Laplace operators for FEM analysis of microwave structures. Microwave and Optical Technology Letters, 52(5): 1036-1042. https://doi.org/10.1002/mop.25111