MCSC 6030G : High Performance Computing
Assignment 1: Matrix-Vector Product

Parikshit Bajpai
100693928

1 Introduction

Computing, in today's world, has radically changed since the days of Konrad Zuse who constructed the world's first fully automated, freely programmable computer with binary floating-point arithmetic in 1941 [1]. Zuse's great visions regarding the possible use of his device are now a reality, and computers, owing to their computational abilities at an incredible, ever-increasing speed, have become essential, not only in science and engineering but in all sectors of life [2].

As an essential tool for most research areas, both in academia and industry, computer-based simulations have become ever-present standard tools. However, even with the rapid increase in the performance of desktop machines, there still remain tasks with requirements on raw computational speed, storage, or main memory which require dedicated computational frameworks [3]. High Performance Computing, in general, refers to the practice of clustering computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business [4].

In times of stagnating single processor capabilities and increasing parallelism, there has been a growing focus on the performance and scalability, and the most sensible measure of performance in benchmarking is the *wallclock time*, also called *elapsed time* [3]. In order to reduce wallclock times, HPC systems rely on a few techniques such as parallelisation, compiler optimisation, and use of standard libraries.

Parallel processing refers to processing the program instructions by splitting them into multiple threads in order to reduce execution time. The different threads can perform different subtasks simultaneously thus reducing the time required for the execution. OpenMP is a shared memory parallel programming API that works on the fork-join model and consists of a set of compiler directives that a non-OpenMP-capable compiler would just regard as comments and ignore [5]. In context of parallelisation, apart from the wallclock time, the following parameters are of interest:

$$Speed\,Up = \frac{Serial\,Wall\,Time}{Parallel\,Wall\,Time} = \frac{S}{P} \tag{1}$$

$$Efficiency = \frac{Speed\,Up}{Number\,of\,Threads} = \frac{S}{n \times P} \tag{2}$$

Apart from parallelisation, optimising compilers has become an essential component of modern high-performance computer systems. In addition to translating the input program into machine language, the compilers analyse it and apply various transformations to reduce its running time or its size [6]. Amongst the popular compilers, GNU Gfortran has developed an open-source *gfortran* compiler capable of working on multiple architectures and diverse environments, and, Intel has developed the *ifort* compiler to deliver superior Fortran application performance and to boost Single Instruction Multiple Data (SIMD) vectorisation and threading capabilities [7].

2 Methodology

2.1 Objective

The present study is aimed at exploring the impact of code, compiler and execution on the wall time for matrix-vector multiplication problem. To this aim, compiler optimisation, parallelisation, and use of standard software library have been implemented and the wall times have been compared.

2.2 Machine Configuration

Manufacturer & Model: Lenovo ThinkPad Yoga 370
Processor: Intel Core i5 -7200U (4 processor cores)
Clock Rate: 2.50 GHz
RAM: 16 GB
Operating System: Ubuntu 18.04

2.3 Implementation

In order to obtain a reference performance benchmark, the standard double-loop code for matrix-vector multiplication was implemented without optimisation and parallelisation and compiled using the GNU compiler *gfortran*, and the matrix size for which the wall time obtained was of the order of 10 seconds was selected for further study. The matrix size corresponding to the aforementioned wall time was 16384×16384. Once the matrix size was fixed, compiler optimisation, parallelisation and use of standard library was implemented and the obtained wall times were compared with that obtained for the double loop without optimisation case. For this purpose, the following primary cases and their combinations were considered:

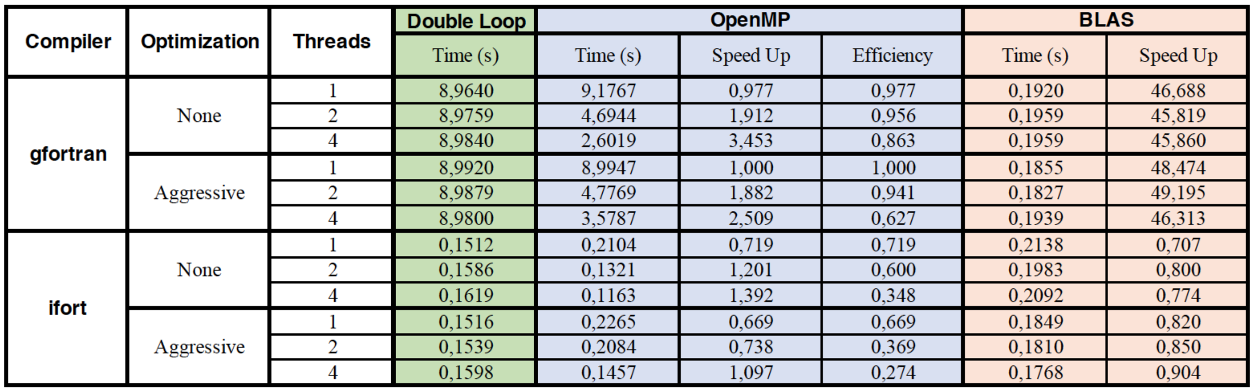
1. Choice of compiler: GNU *gfortran* or Intel *ifort*
2. Compiler optimisation: no optimisation (-o0) or aggressive optimisation (-o3)
3. Parallelisation: 1, 2 or 3 OpenMP threads
4. Matrix multiplication approach: double loop or BLAS standard library

The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations [8] and LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS) [9]. The wall times obtained based on the above strategies used individually and in conjunction were then used to calculate the speed up and efficiency of parallelisation based on equation 1 and equation 2 respectively.

Ibrahim Guiagoussou, Marcos Machado, Celina Desbiens and myself worked together on building the codes, executing and interpreting the results.

3 Results and Discussion

The source code was run for a number of different cases presented above and the obtained results have been presented in Figure 1. The results highlight that the wall times obtained using the Intel compiler are significantly less than those obtained using the GNU compiler. This can attributed to the optimisation of Intel compilers for systems using Intel processors and the high-level techniques used by the compiler to reduce stalls and produce codes that execute in the fewest possible number of cycles [7]. Parallelisation was implemented using OpenMP directives and we observe a significant



Compiler	Optimization	Threads	Double Loop	OpenMP			BLAS	
			Time (s)	Time (s)	Speed Up	Efficiency	Time (s)	Speed Up
gfortran	None	1	8,9640	9,1767	0,977	0,977	0,1920	46,688
		2	8,9759	4,6944	1,912	0,956	0,1959	45,819
		4	8,9840	2,6019	3,453	0,863	0,1959	45,860
	Aggressive	1	8,9920	8,9947	1,000	1,000	0,1855	48,474
		2	8,9879	4,7769	1,882	0,941	0,1827	49,195
		4	8,9800	3,5787	2,509	0,627	0,1939	46,313
ifort	None	1	0,1512	0,2104	0,719	0,719	0,2138	0,707
		2	0,1586	0,1321	1,201	0,600	0,1983	0,800
		4	0,1619	0,1163	1,392	0,348	0,2092	0,774
	Aggressive	1	0,1516	0,2265	0,669	0,669	0,1849	0,820
		2	0,1539	0,2084	0,738	0,369	0,1810	0,850
		4	0,1598	0,1457	1,097	0,274	0,1768	0,904

Figure 1: Wall time [s], speed up and efficiency for the considered cases.

reduction in the wall times as the number of threads was increased. However, we also notice that the speed up did not increase linearly when the number of threads was increased from 2 to 4. Such a non-linear increase can, most probably, be attributed to increase in overheads as the number of threads is increased. Use of the BLAS library with *gfortran* compiler resulted in the most significant increase in speed up. However, the wall times using OpenMP and BLAS concurrently were higher than the wall times obtained using them individually. For the sake of brevity, these results, therefore, have not been presented. They can, however, be viewed by running the shell script *run.sh* in the repository.

Interestingly, when using BLAS library, we observe that the wall times achieved using *gfortran* compiler were lower than those obtained using *ifort* compiler. In fact, when compared with the double loop code compiled with the relevant compiler, the speed up obtained using BLAS and *gfortran* compiler was around two orders of magnitude higher than the speed up obtained using BLAS and *ifort*.

4 Conclusion

In general, the impact of compiler optimisation, parallelisation, and use of standard library on the performance is evident from the obtained results. The results of the different cases reveal that, as foreseen, the Intel *ifort* compiler is more efficient than the GNU *gfortran* compiler. Furthermore, we observe that both parallelisation and use of standard library significantly speeds up the code. The tests highlight the importance of the aforementioned ways of improving the performance of codes developed for HPC and show how these can be used individually and in conjunction with each other to achieve the best possible performance from the available. computing device.

References

[1] R. Rojas and U. Hashagen, eds., *The First Computers–History and Architectures*. History of Computing, The MIT Press, 2002.

[2] K. Zuse, *The Computer – My Life*. Springer-Verlag Berlin Heidelberg, 1993.

[3] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. Chapman & Hall/CRC Computational Science, CRC Press, 2010.

[4] InsideHPC, "What is high performance computing?" [Online] https://insidehpc.com/hpc-basic-training/what-is-hpc/ [Accessed - 07 October 2018]

[5] OpenMP A. R. Board, "OpenMP application program interface version 4.5," November 2015.

[6] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Survey*, vol. 26, pp. 345–420, December 1994.

[7] Intel Corporation, "Intel fortran compiler 18.0 developer guide and reference," 2018.

[8] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Softwares*, vol. 28, pp. 135–151, June 2002.

[9] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Softwares*, vol. 5, pp. 308–323, 1979.