A multi-core CPU implementation of the classical Boson Sampling algorithm

Manan Vaswani

Level M
40cp project

Supervisor: Dr. Raphaël Clifford
Date: 6th May, 2019

Contents

1 Implementation

2 Critical Evaluation

We tested out the above implementation at each stage on the Blue Crystal supercomputer and recorded the timings. The relevant graphs were generated using the `matplotlib` library from Python.

2.1 Initial implementation

We first tested out the C++ code written by us without any optimisations made, compiled with both the Intel and gcc compilers. This has been plotted as a graph shown in figure 2. For each value of n from 2 to 30, 10 results were recorded, with their timing taken. The mean and standard deviation of the code runtimes for each of these values was calculated and is depicted in the graph. The error bars represent one

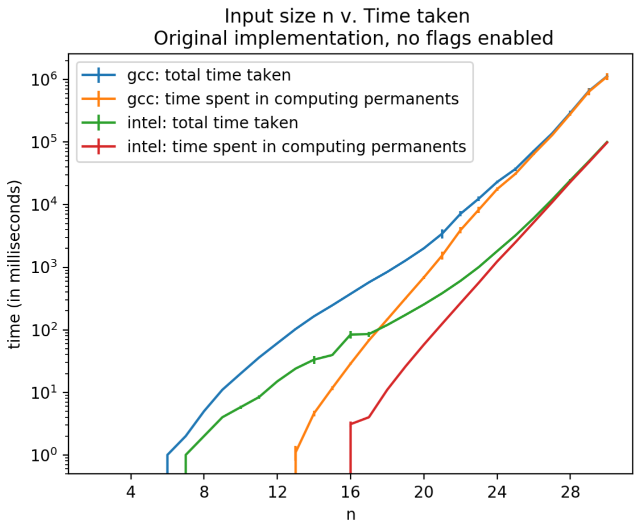

Figure 1: Original implementations

standard deviation in the data. Notice that the y-axis of the graph uses a log-scale, which is why the lines appear to be roughly linear. We expected the time taken to grow exponentially, and this was the observed result. These results are as expected, with the Intel compiled code being significantly faster than the gcc compiled code. The average time for the gcc compiled code to complete the algorithm was roughly 1100 seconds, whereas for the intel compiled code, it was 100 seconds. The reason that the Intel compiled code is over 10 times faster is that it is highly optimised by the compiler even without specifying any additional optimisation options. Additionally, we can see that the time taken for computation of permanents is consistently only a bit less than the total time taken, except for smaller values of n, in which case the amount of time taken to compute permanents was 0. This is because for small sizes of matrices, calculating the permanents requires only a few simple mathematical calculations. The overheads of running the rest of the algorithm appears to dominate for these small values of n.

It was also interesting to note that the timings to run the Intel compiled code were a lot more consistent than the timings for the gcc compiled code. The standard deviation for the gcc code with higher values of n was roughly 130, whereas for intel, it never exceeded 0.8, giving the coefficients of variation 11% and 0.8% respectively.

2.2 Testing the different compiler option/flags

After parallelising the code, in order to compare the different compiler options, the code was run on 16 cores with different options enabled each time. As before, 10 samples were taken for each value of n, and the values plotted in the graphs are the means

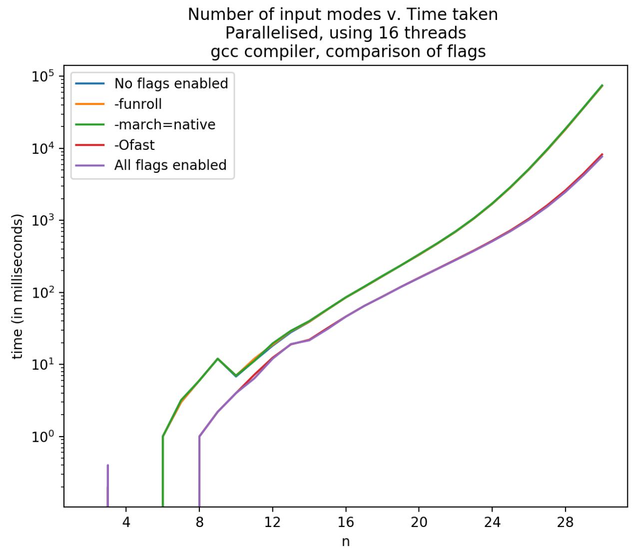

Figure 2: Comparing different gcc compiler options

2.2.1 gcc

We had four different compiler options to test with gcc, as explained in section 1.5.2. The `-funroll` option makes almost no difference, which we anticipated since it is known to not improve timings in every case. Specifically, in the case of our loop which was run an exponential number of times for permanent calculation, unrolling loops would potentially make the process of going from one iteration to the next faster, but a lot more memory would be used as a cost, which would slow it back down. What was more surprising was that the `-march=native` flag had only a minute effect (1 to 2 seconds for n=30) in speeding up the time. On the other hand, the `-Ofast` flag provided a major speed up, making the code over 10 times faster to run than with no compiler options enabled. The code was profiled again at this stage to understand how such a significant speed up was made. The results are shown in figure 4. The difference is that the top few hotspots are no longer dominated by the inbuilt function '__muldc3', and other specific operations related to multiplying complex numbers. The reason this has happened is that using `-Ofast` tells the compiler to ignore strict ISO standards, and instead of running the complex number multiplication function with a lot of checks, it simply performs the bare operations. While it does make the code susceptible to errors in edge cases (specifically involving operations where the real part of the number is set to infinity), we expect to never encounter these cases in our code, and can ensure that we don't by adding in some checks of our own.

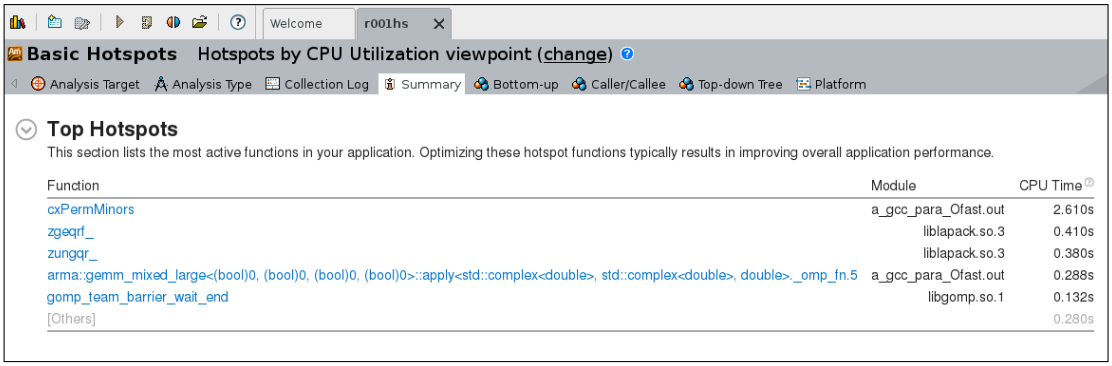

Figure 3: A summary of hotspots found in the code identified by the Intel Vtune Profiler after enabling the -Ofast flag

2.2.2 Intel compiler

The two chosen flags were tested on the Intel compiler and surprisingly, neither of the two made a big difference to the timing. This is shown in figure 5. Considering how much faster the Intel compiled code was even before any options were added, we presume that the Intel compiler available on Blue Crystal is maximally optimised for speed by default.

After enabling optimisation flags, the gcc and Intel compiled code have similar timings for $n = 30$, with the gcc compiled code occasionally being 1-2 seconds faster.

2.2.3 Multiple threads

One of the main results of our paper was to parallelise the algorithm, and the results obtained by doing so were as expected. The code was run after being compiled with both compilers and all listed flags enabled, with a different number of threads available each time for the sake of comparison. The results we obtained showed a speed up proportional to the number of threads used, shown in figures 6 and 7. In a few cases for values between $n = 8$ and $n = 12$, we can see that in figure 8 there are some inconsistencies, and using more threads might take a few milliseconds more than with less threads. This is because multithreading does have overheads with assigning values to threads as well as with creating threads.

Another set of timings were taken for running the Boson Sampling code with $n = 30$, and successively increasing the number of threads, shown in figure 8. The results show that the timing reduces in steps as the number of cores used is increased. One can also notice the trend that as the number of threads is doubled, the timing is roughly halved. A summary of the actual timings are also shown in table 4.

2.2.4 Final result

Using all the optimisations we have made, we ran the code on Blue Crystal, using 28 cores, and with all mentioned compiler-flags enabled for optimisation. The results are

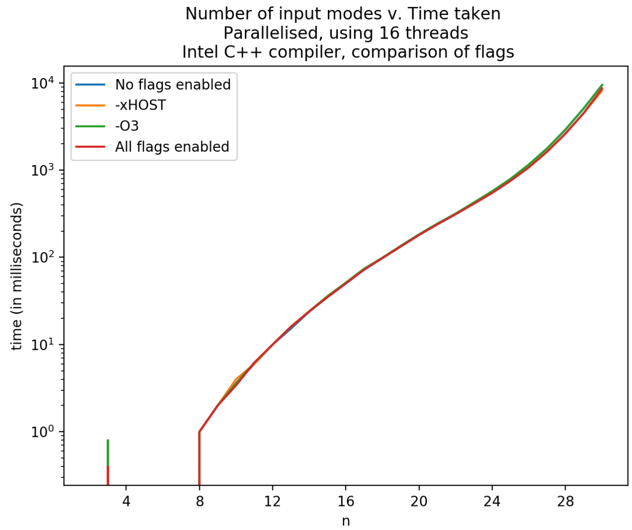

Figure 4: Comparing different Intel compiler options

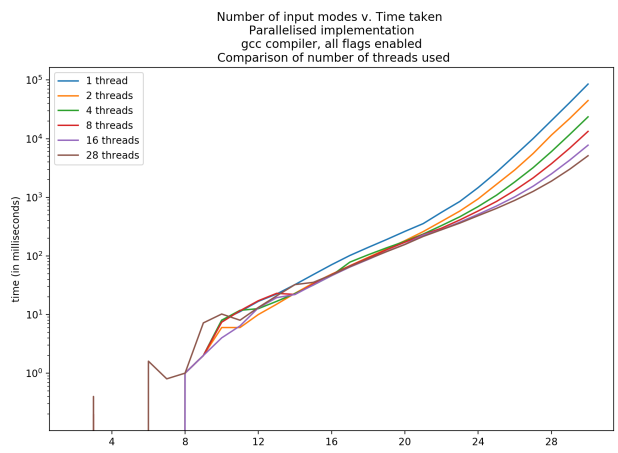

Figure 5: Comparing different number of threads used with gcc compiled code

Table 1: Timings to run code (in seconds), as number of threads is changed, for $n = 30$

No. of cores	**Time for gcc**	**Time for Intel**
1	85	88
2	44	46
4	23	24
8	13	14
16	8	8
28	5	5

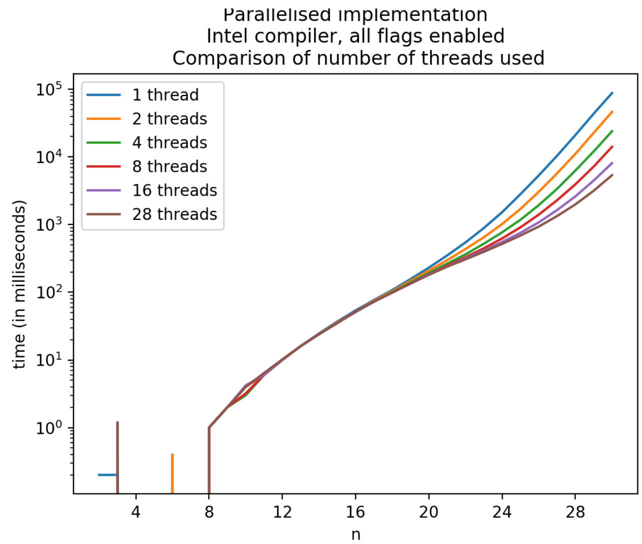

Figure 6: Comparing different number of threads used with intel compiled code

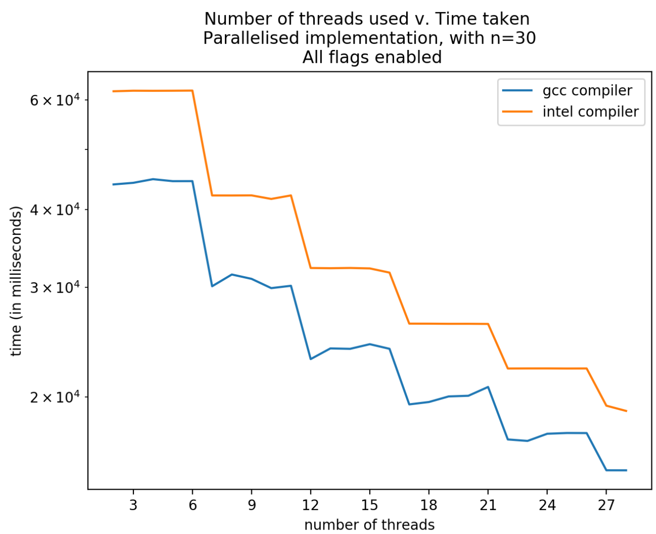

Figure 7: Comparing different number of threads used

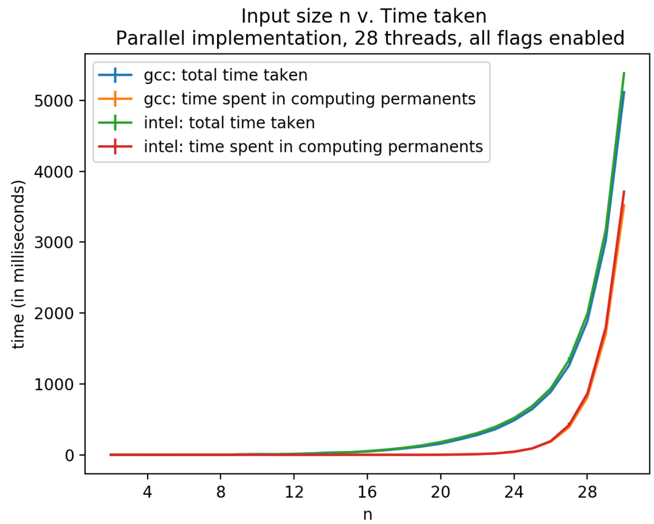

Figure 8: Comparing final results

shown in 9. The gcc compiled code gives marginally faster results as compared to the Intel compiled code, and the times taken to simulate sampling bosons for $n = 30$ are 5.1 seconds and 5.3 seconds respectively.

3 Conclusion

The final results produced by us showed a speed up of around 200 times with the gcc compiler, and 20 times for the Intel compiler, as compared to our initial measurments. We also ran a few tests for $n = 35$, and it took roughly 320 seconds to run. While these numbers may not seem impressive compared to the benchmark on the Tianhe-2 supercomputer [20], our implementation uses a maximum of 28 cores, whereas the benchmark for $n = 50$ computed in 600 minutes was made using 312?,000 cores. We project that in order to break that benchmark, we would need access to a supercomputer with only $\approx 10,000$ cores.

3.0.1 Further Works

References

[1] Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An Introduction to Quantum Computing*. Oxford University Press, Inc., New York, NY, USA, 2007.

[2] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *arXiv e-prints*, pages quant–ph/9508027, Aug 1995.

[3] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[4] Christos H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. John Wiley and Sons Ltd., Chichester, UK, 2003.

[5] Aram W. Harrow and Ashley Montanaro. Quantum computational supremacy. *Nature*, 549(7671):203–209, Sep 2017.

[6] Thomas Monz, Daniel Nigg, Esteban A. Martinez, Matthias F. Brandl, Philipp Schindler, Richard Rines, Shannon X. Wang, Isaac L. Chuang, and Rainer Blatt. Realization of a scalable shor algorithm. *Science*, 351(6277):1068–1070, 2016.

[7] Juha J. Vartiainen, Antti O. Niskanen, Mikio Nakahara, and Martti M. Salomaa. Implementing Shor's algorithm on Josephson charge qubits. *Phys. Rev. A*, 70(1):012319, Jul 2004.

[8] Sergio Boixo, Sergei V. Isakov, Vadim N. Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J. Bremner, John M. Martinis, and Hartmut Neven. Characterizing Quantum Supremacy in Near-Term Devices. *arXiv e-prints*, page arXiv:1608.00263, Jul 2016.

[9] Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, Erik W. Draeger, Eric T. Holland, and Robert Wisnieff. Breaking the 49-Qubit Barrier in the Simulation of Quantum Circuits. *arXiv e-prints*, page arXiv:1710.05867, Oct 2017.

[10] E. Knill, R. Laflamme, and G. J. Milburn. A scheme for efficient quantum computation with linear optics. *Nature*, 409(6816):46–52, 2001.

[11] Scott Aaronson and Alex Arkhipov. The computational complexity of linear optics. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 333–342, New York, NY, USA, 2011. ACM.

[12] Margherita Barile and Eric W Weisstein. Galton board. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GaltonBoard.html.

[13] Seinosuke Toda. Pp is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20:865–877, 10 1991.

[14] E. R. Caianiello. On quantum field theory — i: explicit solution of dyson's equation in electrodynamics without use of feynman graphs. *Il Nuovo Cimento (1943-1954)*, 10(12):1634–1652, Dec 1953.

[15] Wojciech Roga and Masahiro Takeoka. Classical simulation of boson sampling with sparse output. *arXiv e-prints*, page arXiv:1904.05494, Apr 2019.

[16] P. H. Lundow and K. Markström. Efficient computation of permanents, with applications to boson sampling and random matrices. *arXiv e-prints*, page arXiv:1904.06229, Apr 2019.

[17] Saleh Rahimi-Keshari, Timothy C. Ralph, and Carlton M. Caves. Sufficient Conditions for Efficient Classical Simulation of Quantum Optics. *Physical Review X*, 6(2):021039, Apr 2016.

[18] W. Larry Hastings. Monte carlo sampling methods using markov chains and their applications. 1970.

[19] Alex Neville, Chris Sparrow, Raphaël Clifford, Eric Johnston, Patrick M. Birchall, Ashley Montanaro, and Anthony Laing. Classical boson sampling algorithms with superior performance to near-term experiments. *Nature Physics*, 13:1153 EP –, 10 2017.

[20] Junjie Wu, Yong Liu, Baida Zhang, Xianmin Jin, Yang Wang, Huiquan Wang, and Xuejun Yang. A benchmark test of boson sampling on Tianhe-2 supercomputer. *National Science Review*, 5(5):715–720, 07 2018.

[21] Peter Clifford and Raphaël Clifford. The classical complexity of boson sampling. *CoRR*, abs/1706.01260, 2017.

[22] Marvin Marcus and Henryk Minc. Permanents. *The American Mathematical Monthly*, vol. 72:577–591, 1965.

[23] Herbert John Ryser. *Combinatorial Mathematics*. Mathematical Association of America, 1963.

[24] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189 – 201, 1979.

[25] Albert Nijenhuis and Herbert S. Wilf. 23 - the permanent function (perman). In *Combinatorial Algorithms (Second Edition)*, pages 217 – 225. Academic Press, second edition edition, 1978.

[26] K Balasubramanian. *Combinatorics and diagonals of matrices*. PhD thesis, Indian Statistical Institute, Calcutta, Dec 1980.

[27] Eric Bax. *Finite-difference algorithms for counting problems*. PhD thesis, California Institute of Technology, February 1998.

[28] Eric Bax and Joel Franklin. A finite-difference sieve to count paths and cycles by length. *Information Processing Letters*, 60(4):171 – 176, 1996.

[29] David G. Glynn. The permanent of a square matrix. *European Journal of Combinatorics*, 31(7):1887 – 1891, 2010.

[30] William Feller. *An Introduction to Probability Theory and its Applications*, volume 1. Wiley, 1968.

[31] A. J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10(8):127–128, April 1974.

[32] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969.

[33] Peter Clifford and Raphaël Clifford. *BosonSampling: Classical Boson Sampling*, 2017. R package version 0.1.1.

[34] Dirk Eddelbuettel, Romain Francois, JJ Allaire, Kevin Ushey, Qiang Kou, Nathan Russell, Douglas Bates, and John Chambers. *RCpp*, 2017. R package version 0.12.12.

[35] Conrad Sanderson. Armadillo c++ linear algebra library, June 2016.

[36] Maris Ozols. How to generate a random unitary matrix. Technical report, 2009.

[37] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[38] Intel Software. Intel vtune amplifier 2018. Software available at https://software.intel.com/en-us/vtune, 2018.

[39] God-bolt.

[40] muldc3.c - implement __muldc3.

[41] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*. GNU Press.

[42]

[43]