Improving Performance of the PageRank Algorithm*

Malhar Jajoo[1], Baron Khan[1] and Nikolay Yotov[2]

***Abstract*— PageRank is a link-analysis algorithm at the heart of Google's search engine. This report aims to improve the performance of an open-source implementation of the algorithm, by parallelizing some of the computation. Various parallel libraries are explored and compared, and other methods for improving performance are also considered. The work done in this report can be found here: `https://github.com/BaronKhan/pagerank`.**

I. INTRODUCTION

PageRank is an algorithm, famously invented by Google founders Sergey Brin and Larry Page [1], that ranks web pages found as part of Google search engine results. It represents the World Wide Web (www) as a hyperlink structure with each webpage being represented by a node in the hyperlink graph and a link represented by an edge. It evaluates the importance of a node in the graph based on the quantity and quality of nodes that have edges to that node. An open-source C++ implementation of the algorithm is available online, created by Panos Louridas [2]. By default, this implementation of the algorithm is iterative and single-threaded. A brief explanation of the algorithm is provided in the section below.

A. Algorithm Overview

The PageRank algorithm assigns a rank value to each webpage (part of the search engine results) based on the incoming links from other webpages. Suppose a page P_j has l_j outgoing links. If one of those links is to page P_i, then P_j will pass on $1/l_j$ of its importance to P_i. The pagerank of P_i is then given by

$$I(P_i) = \Sigma_{Pj \in Bi}\, I(P_i)/l_j, \tag{1}$$

The above represents the pagerank of a single webpage/node, for easier representation, the equations for all nodes are represented using matrices and vectors.

$$I = HI, \tag{2}$$

where I is a vector of PageRank value of each node in the graph. H is a two dimensional matrix known as the Hyperlink Matrix and each column contains a scalar value based on number of outgoing links from the node corresponding to the column index. An entry at the i^{th} row and j^{th} column of H is given by

$$H_{ij} = \begin{cases} 1/l_j & if P_j \in B_i \\ 0 & otherwise \end{cases}$$

Hence the vector I is an eigenvector of the matrix H corresponding to the eigenvalue of one. I is a stationary vector and the H matrix is a column stochastic matrix (i.e the sum of each column entries is one).

Several methods are known for computing the eigenvector of a square matrix, however, the H matrix is very large (approximately 25 billion rows and columns, and the dimensions will keep increasing since the World Wide Web continues to expand). This necessitates the need for the Power Method. It can be described by the following recurrence relation:

$$I^{k+1} = HI^k, \tag{3}$$

where k refers to the loop iteration number.

Two key aspects can be noted:

- I^{k+1} eventually converges to stationary vector I. However, dangling nodes (no outgoing links) need to be handled and hence H is then replaced by a matrix $S = H + A$ where matrix H contains zeros in columns corresponding to dangling nodes and the matrix A accounts for this by assigning a fixed value of 1/number of outgoing links to each coloumn.
- The above recurrence relation implies that a software implementation of the Power method is inherently sequential containing loop iterations (with k being the iteration number) that depend on PageRank values from the previous iteration.

The convergence of the Power Method and more details is based on the following properties of the matrix S:

- primitive: $|\lambda_1| = 1$ and $|\lambda_2| < 1$.
- irreducible: all entries are positive.

The convergence proof is further explained diagrammatically in [3]. Also note that in practice a Google Matrix is used and is given by

$$G = \alpha S + \frac{(1-\alpha)}{n}1, \tag{4}$$

where "1" refers to a matrix of all ones, and the alpha parameter chosen in practice is approximately 0.85.

*This work is part of the assessment for the CO332 Advanced Computer Architecture course at Imperial College London.

[1]M. Jajoo and B. Khan are with the Department of Electronic Engineering and Department of Computing, Imperial College London, SW7 2AZ, UK `mj2514 at ic.ac.uk, bak14 at ic.ac.uk`

[2]N. Yotov is with the Department of Computing, Imperial College London, SW7 2AZ, UK `ny1215 at ic.ac.uk`

B. Research Objective

The objective of this report is to speed up the algorithm run time for a set of data on a specific machine. The main hypothesis for improving performance that will be tested is: **"A parallelized version of the PageRank algorithm leads to better performance compared to a sequential version"**.

Despite the algorithm being inherently sequential, as described above, the matrix-vector multiplications involving the stationary vector can be parallelized, and element-wise operations on vectors can be done in parallel. This strategy is further justified by the availability of hardware that supports parallel execution, as described in Section II below.

Several parallelization methods are explored in this report, with their resulting performances compared, and the best one chosen and further improved upon. We explore Intel C++ parallel programming libraries, Threaded Building Blocks (TBB) and OpenMP, as well as a GPU implementation of the algorithm using OpenCL. We also explore other methods for improving performance, such as algorithmic restructuring, code optimisation techniques and using different compilers for high performance, such as the Intel C++ Compiler.

II. Hardware Description

The machine that the PageRank algorithm will be optimised for is the *edge04* workstation located in the Department of Computing at Imperial College London. This is a HP EliteDesk 800 G2 Tower Desktop PC model [4] running a 64-bit Ubuntu operating system, and it features an Intel Core i7-6700 processor, with a base clock rate of 3.40GHz that can increase to 4.00GHz using Intel's Turbo Boost Technology, depending on the workload [5]. The processor consists of 4 CPU cores, and a total of 8 threads [6]. It also supports SIMD extensions and hyper-threading for simultaneous multi-threading, improving parallel computations.

In terms of memory, the processor features an 8MB *SmartCache*. This is a proprietary cache designed by Intel that allows multiple CPU cores to access the same level 2 cache. This shared cache makes sharing memory between CPU cores a lot faster, and also means that a single-core process can utilise the entire cache while the other CPU cores are inactive [7].

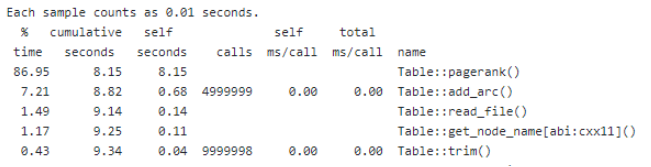
The hardware also features an NVIDIA GeForce GT 730 graphics card (with GDDR5 memory) with a base clock rate of 902MHz [8], as well as a secondary integrated Intel HD Graphics 530 card [9].

III. Program Analysis

The *pagerank_test* program measures the time for the *pagerank()* member function of the *Table* class to execute, and reports it as the PageRank time. This program can be run with different datasets containing different graph sizes. Below is a table and graph showing the average PageRank times of different datasets on the *edge04* machine for the default code provided, with 5 samples used for each dataset:

Test Dataset Name	**No. of Vertices in Graph**	**Average Execution Time / s**	**Smallest Execution Time / s**	**Largest Execution Time / s**
small	50,000	0.0262	0.0261	0.0268
medium	500,000	0.625	0.616	0.647
large	5,000,000	8.12	8.07	8.16
enormous	8,000,000	17.7	17.6	17.8
ginormous	19,000,000	32.4	32.0	32.9

The next step was to analyse the bottlenecks in the program. Using the GNU *grof* performance analysis tool, the execution times and a count of all the main function calls were obtained when running the program with the *large-test* target in the Makefile, which loads a graph with 5 million vertices.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 86.95      8.15     8.15                             Table::pagerank()
  7.21      8.82     0.68  4999999     0.00     0.00  Table::add_arc()
  1.49      9.14     0.14                             Table::read_file()
  1.17      9.25     0.11                             Table::get_node_name[abi:cxx11]()
  0.43      9.34     0.04  9999998     0.00     0.00  Table::trim()
```

Fig. 1: The profile of the program using GNU grof.

The analysis shows that the `pagerank()` function takes up the majority of the execution, and performs no function calls, so all computation happens inside the function. The `pagerank()` function itself consists of a while loop with several for loops which are executed on each iteration of the while loop. The first for loop calculates the sum of the current iteration's pagerank vector elements (and the sum for dangling nodes), the second for loop normalises the elements of the pagerank vector, and finally, there is a nested for loop for calculating the current iteration of the Google Matrix equation using the Power method [3].

IV. Parallelization Strategy and Issues

After program analysis, parallelization opportunities were identified in the code. These fall under two main categories that are explained below.

A. Parallelizing Independent Loop Iterations

Loops can have iterations that either depend on results from a previous iteration due to a data dependence, or can be independent of other iterations. This property of loops allows executing them in parallel. Often, a simple re-structuring of the code may also convert dependent loop iterations into independent iterations. A simple example of such code restructuring can be to replace an accumulator variable with a vector sized as large as the number of loop iterations (if known in advance) and store each *parallelized* iteration's result into the vector. After completion of the parallelized iterations, the vector elements can be summed to obtain the accumulated value. This strategy improves performance when an accumulator variable is the only data dependence between loop iterations of computationally intensive code.

B. Parallel Reduce

Although there are several for loops in the default implementation, the iterations are not independent as they are accumulating values into an accumulator variable. This variable is shared across iterations. If each iteration ran in parallel, they would each access the same variable and try to update it, possibly overwriting another iteration's update. The C++ standard library provides a std::atomic<>template type for integers and floats (IEEE 754 single or double precision) but does not provide arithmetic operations for the latter. Therefore, for-loops containing a floating point accumulator cannot be parallelized without rewriting the loop structure. To avoid this, we resort to using a reduction.

A reduction is where a sequence of operations over the same associative[1] operator (e.g. (((a + b) + c) + d)) can be rearranged and calculated in a tournament style (e.g. (a + b) + (c + d)), where each row of the tournament can be computed in parallel.

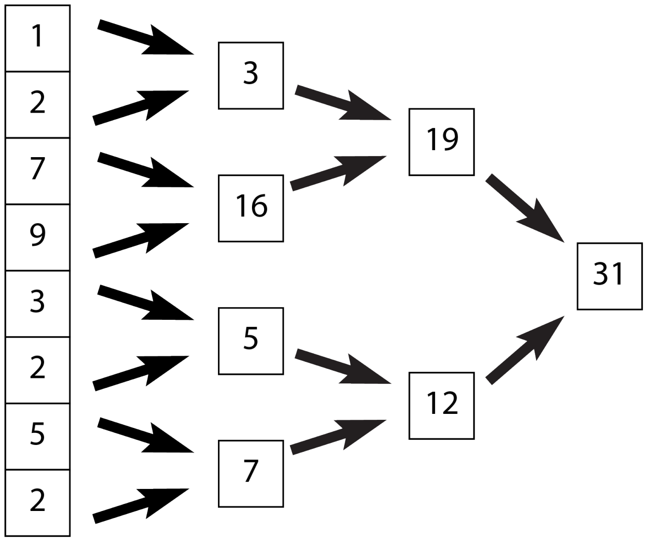


Fig. 2: A diagram of a reduction over the plus (+) operator. Each column of operations can be computed in parallel. Image taken from [10], [11].

V. Timing Issues

Another side issue discovered while testbenching the new parallel versions of the algorithms was the timing. Initially, the `clock()` system call was being used to get the start and end times to calculate the time taken, but according to its man page [12], the function returns an approximation of processor time used by the program. This includes the sum of the processor times spent in each CPU, so if multiple CPUs are used in a parallel manner during that time, the time calculated will be (approximately) the wall clock time multiplied by the number of CPUs. Therefore, the `gettimeofday()` function was used instead to get the wall time.

[1]The operator does not need to be commutative.

VI. Verification Strategy

Gathering performance data is conducted using a Bash script which runs a user-defined number of iterations on selected test cases, both of which must be specified. For the data in this report, the number of iterations is five and all available test suites are used. Then, the running time for each run is extracted to a .tempdata file and finally a Python script is automatically called to compute the mean and standard deviation for each type of test.

The mean was selected as a simple measurement unit due to fluctuations in value of the run time on each separate execution. To keep the data in line, standard deviation is also computed to give a statistical reassurance that the results of the runs tend to the mean.

VII. Threaded Building Blocks

Intel Threaded Building Blocks (TBB) is a parallel programming library intended to improve performance for program execution on multicore machines [13].

It provides simple and effective parallel (program) constructs and concurrent data structures that are used to parallelize execution of independent iterations of loops. As seen below, this aligns well with the parallelization opportunities identified in Section IV and hence this library was chosen.

A. Implementation

The TBB parallel_for and TBB parallel_reduce are utilized to parallelize independent loop iterations and floating point reduce operations respectively. A brief explanation is given below on each construct. For details of the code, refer to the **tbb_opencl** branch of the Github repository [2].

1) TBB Parallel for: The parallel for loops are implemented using the template construct *tbb::parallel_for()*. It can be invoked on parallelizable for loops with a fixed number of iterations. A grain size can also be manually specified however the automatic partitioner used by TBB provided the best results [14]. Different configurations of this construct were explored and have been presented in Section VII-B below.

2) TBB Parallel Reduce: The parallel reduce is implemented using the construct *tbb::parallel_reduce()* [15]. It utilizes a C++ Functor class with a split constructor to split the expression tree (as seen in Figure 2). The split is done by TBB and depends on the thread availability on the processor. After the results from each split is calculated, it is combined using a Join constructor. Note here that the operator does not need to be commutative and the join constructor can be specified to combine left operand (operator) right operand. This is explained in more detail in [16].

B. Results

As seen in Figure 3 and Tables I, II and III, TBB with the outer loop parallelized and the inner loop sequential results in the best performance. TBB with the inner loop parallelized and the outer loop sequential results in the worst performance.

[2]`https://github.com/BaronKhan/pagerank`

Test Dataset	**No. of Vertices in Graph**	**Average Execution Time / s**	**Standard Deviation from mean**
small	50,000	0.00957	0.001
medium	500,000	0.2406	0.0041
large	5,000,000	3.707	0.0449
enormous	8,000,000	6.760	0.108
ginormous	19,000,000	15.341	0.192

TABLE I

Test Dataset	**No. of Vertices in Graph**	**Average Execution Time / s**	**Standard Deviation from mean**
small	50,000	0.0826	0.001
medium	500,000	0.987	0.003
large	5,000,000	12.26	0.062
enormous	8,000,000	23.479	0.751
ginormous	19,000,000	49.393	0.388

TABLE II

Test Dataset	**No. of Vertices in Graph**	**Average Execution Time / s**	**Standard Deviation from mean**
small	50,000	0.339	0.0146
medium	500,000	4.7373	0.0260
large	5,000,000	60.960	0.331
enormous	8,000,000	108.99	0.733
ginormous	19,000,000	240.4854	1.314

TABLE III

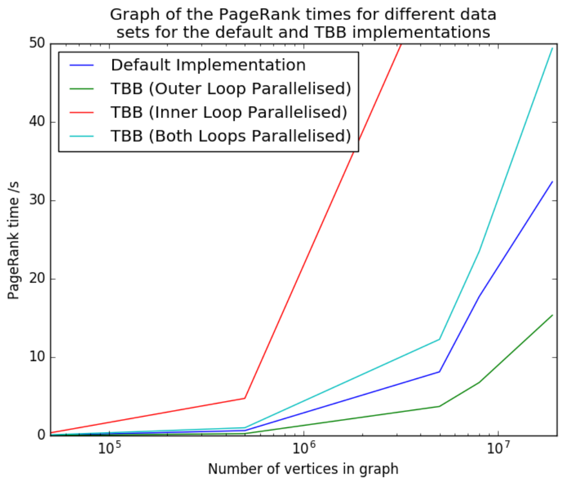
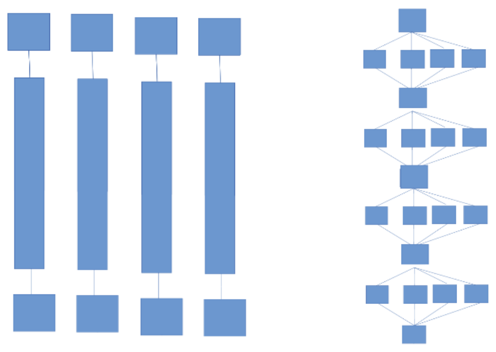


Fig. 3: Comparing results for TBB experiment. TBB with **only** the outer for loop parallelized has best performance.

This is diagrammatically explained in Figure 4. It can be seen that in the case when the inner loop is parallelized and the outer loop is sequential, there is a setup and teardown of threads in each iteration of the outer loop. This overhead is responsible for increasing execution runtime for this case. When both outer and inner loops are parallelized, it is difficult to provide a quantitative justification and hence empirical results (taking the average over 5 runs) are relied upon.



(a) Outer loop parallelized, inner loop sequential.

(b) Inner loop parallelized, outer loop sequential.

Fig. 4: Diagrammatic form of Only outer vs Only inner loop parallelization using tbb::parallel_for(). The boxes represent threads and each link signifies start up or destruction of a thread.

VIII. OpenMP

OpenMP is another library used to make multithreading programming easy [17], and an implementation is explored[3]. It works with machines that use a shared memory architecture (as the chosen *edge04* machine does, with its *SmartCache* technology) and is backwards compatible with compilers that do not support it (they simply ignore the related lines of code) [18]. OpenMP allows the user to set the number of threads they would like to use using `omp_set_num_threads(int)`.

A. OpenMP Implementation

OpenMP lets the user set pragmas before pieces of code that should be executed in parallel. If the compiler is compatible and the code is compiled with the *fopenmp* flag, then the code following the pragmas will be executed in parallel. For instance, putting `#pragma parallel for` just before a for loop will distribute its iterations across the OpenMP threads and execute them in parallel.

Since the for loops in `Table::pagerank()` usually have an accumulator variable that is shared across iterations, a reduction needs to be performed. In the first for loop within the while loop, the following pragma is used:

```
#pragma omp parallel for reduction(+:dangling_pr,sum_pr)
for (size_t k = 0; k < pr.size(); k++) {
    double cpr = pr[k];
    sum_pr += cpr;
    if (num_outgoing[k] == 0) {
        dangling_pr += cpr;
    }
```

Here, a parallel reduction is performed over the dangling_ptr and sum_ptr variables using the + operator. In the outer loop of the nested for loop, we have:

[3]see branch openmp

```
#pragma omp parallel for reduction(+:diff)
for (i = 0; i < num_rows; i++) {
    /* The corresponding element of the H m
```

The inner loop of the nested for loop could not be parallelized as it is not in a canonical form of a standard incrementing for loop (it uses an iterator instead), though this may have given more speedup.

OpenMP also has support to explicitly force the processor to use SIMD Vectorization, where multiple operations can be performed with a single instruction. In the second for loop within the while loop of `Table::pagerank()` which performs normalisation, there is only one statement in the for loop, so parallelizing it would be unnecessary overhead. Instead we have:

```
#pragma omp simd
for (i = 0; i < pr.size(); i++) {
    old_pr[i] = pr[i] / sum_pr;
```

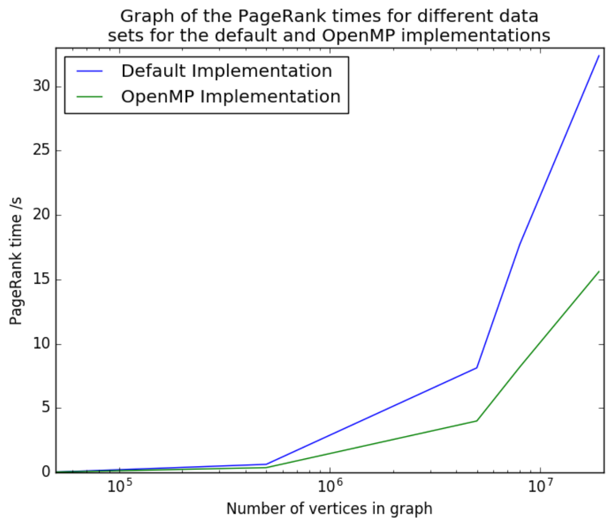
This forces the compiler to use a SIMD instruction on the pagerank arrays.

B. *OpenMP Results*

Below is a table showing the average PageRank times for different test suites with 5 samples each, using OpenMP, as well as the standard deviation:

Test Suite	**No. of Vertices in Graph**	**Average PageRank Time /s**	**Standard Deviation**
small	50,000	0.0299	0.00536
medium	500,000	0.368	0.0190
large	5,000,000	4.00	0.205
enormous	8,000,000	8.19	0.482
ginormous	19,000,000	15.6	0.193

Below is a graph comparing the PageRank times for the default implementation and the OpenMP implementation:



The graph shows that the OpenMP implementation of the algorithm, while showing no signs of speedup for graphs with less than 100000 vertices (and in some cases being slower than the default version), it becomes almost twice as fast for higher numbers of vertices. Ideally, considering that 4 CPUs are being used concurrently, the speedup should be closer to a factor of 4, but given the overhead in calling the OpenMP constructs, a speedup by a factor of 2 is reasonable.

IX. GPU IMPLEMENTATION: OPENCL

The Open Computing Language (OpenCL) is a heterogenous computing framework developed and maintained by the Khronos Group. It defines a C++ [19] and C99 [20] based language for the host (CPU) API and device (GPU) API. The main execution workhorse is the *OpenCL NDRange Kernel*. NDRange refers to the iteration space of the OpenCL kernel and this can be configured using the Host API. The following section only explores details required to setup the OpenCL kernel and does not explain the boilerplate Host API code required to setup the OpenCL kernel. The reader may wish to look at the implementation in the **tbb_opencl** branch of the Github repository.

A. *Implementation*

This section explains the process required to convert the default PageRank implementation in standard C++ to the OpenCL C API for the kernel. The PageRank algorithm is implemented as an OpenCL NDRange Kernel. These steps are briefly explained below:

- Changing input representation
  - In order to invoke the OpenCL kernel, the default data structures need to be converted to primitive data structures acceptable by OpenCL [21]. The default data structure for representing Matrix H is a C++ vector of vectors[4] This vector is then flattened out into a one dimensional array. Since the rows of the H matrix are of uneven lengths, two additional arrays are also utilized to track the start index and length of each row in order to find the index in the flattened array. These data structures can be seen in Figure 5 and a corresponding implementation exists in the GitHub repository branch [5]
  - It can also be noted that changing the input representation is not part of the timed code and hence does not add any overhead to the execution runtime.
- Passing parameters to the Kernel
  - Scalar arguments are passed using the `cl::Kernel::setArg()` function.
  - For non-scalar arguments (pointers) that reside in CPU (host) memory, fixed width GPU buffers are allocated. For fixed non-scalar arguments, the respective data is then copied over from CPU memory to GPU memory.
  - For both scalar and non-scalar, fixed arguments are copied over to the kernel before the outermost while loop in the code. However, arguments

[4]this may exist in the default implementation in order to efficiently represent sparse matrix H.

[5]see branch tbb_opencl

whose value changes between iterations need to be set again using `cl::Kernel::setArg()` (non-scalar arguments are also copied using `clEnqueueWriteBuffer()`), once in every iteration of the outer while loop.

- Finally, the kernel is invoked on a one-dimensional NDRange (parallelized over pagerank array).

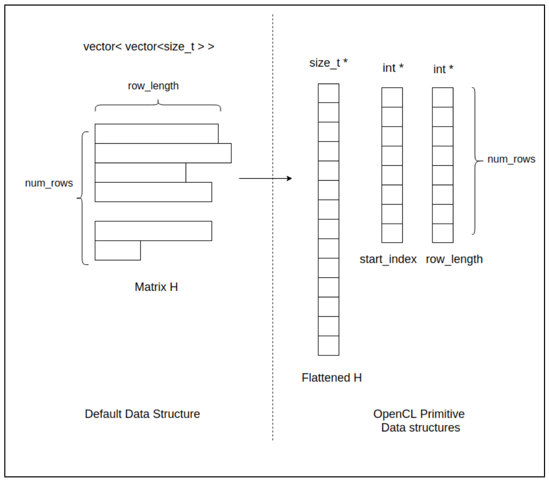


Fig. 5: Figure showing conversion of default data structures to a primitive representation.

B. Results

Test Suite	No. of Vertices in Graph	Average PageRank Time /s	Standard Deviation
small	50,000	0.2577	0.0411
medium	500,000	0.637	0.0204
large	5,000,000	4.965	0.0216
enormous	8,000,000	8.583	0.0028
ginormous	19,000,000	18.930	0.0781

The GPU implementation reduces program execution runtime across all datasets except the small dataset. This is because the **overhead** of setting up the GPU kernel using the Host API (before the while loop), and reading and writing current and previous pagerank buffers in every iteration of the loop, outweighs the performance benefits of using the GPU. The GPU seems to perform worse than the OpenMP and TBB libraries used earlier. This could be due to various factors like utilizing global memory for variables, or passing extra arrays to GPU to keep track of the index in the flattened array as shown in Figure 6.

X. Compiler optimisations

Part of the hypothesis is a comparison between the G++ GNU Compiler and the Intel C++ Compiler. Results for the initial configuration with the `-O3` flag show that both compilers on this level of optimisation perform at a parity for the all types of solutions. Following the Intel compiler optimisation guide [22] results in the addition of extra Intel compiler flags: `-xCORE-AVX2 -ipo -no-prec-div`. Each flag meaning as follows:

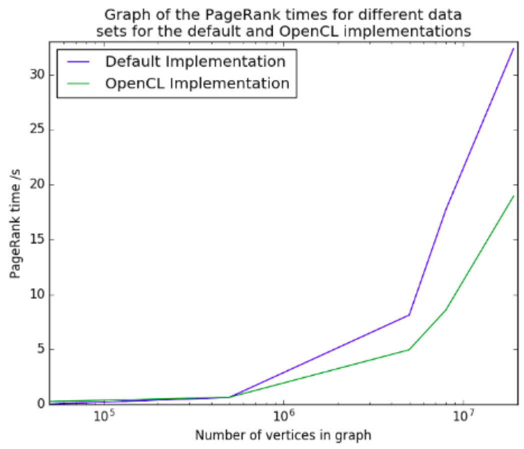


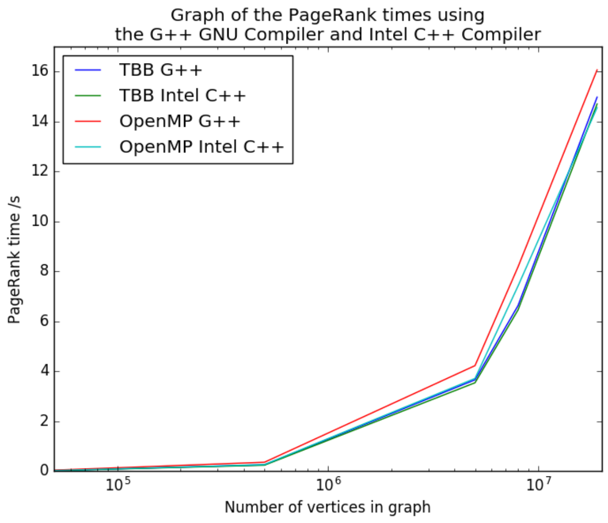
Fig. 6: Graph showing speedup obtained (compared to the default implementation) using GPU OpenCL implementation.

- `-xCORE-AVX2` - a machine specific optimisation using instruction set extension AVX2.
- `-ipo` - interprocedural optimisation.
- `-no-prec-div` - not applying precision improvements of FP division.

Improvements in performance of about 2-5% are noticeable for TBB test suites. For OpenMP a significant boost of 25-50% for small tests and 10% for larger tests suites. For small and medium test suites, it can be argued whether an improvement exists, as the expected improvement is too small and is overshadowed by fluctuations. For TBB implementations, the most noticeable effect uses the flag `-ipo` whereas for the OpenMP implementation `-xCORE-AVX2` is used. TBB (outer-loop only) with extra Intel flags outperforms its OpenMP counterpart on all but the ginormous tests. On the ginormous test suite, TBB has quite a small standard deviation (about 0.10) which keeps a consistent performance in the range of 14.65-14.75 seconds. On the other hand, the OpenMP solution has 13.90s best single-run performance, but most tests fall into the range 14.40-14.90 seconds range which results in a higher standard deviation(about 0.50). These results where achieved through multiple runs of the tests to verify the reliability of the results.

Test suite	TBB G++	TBB Intel C++ w/ flags	OpenMP G++	OpenMP Intel C++ w/ flags
small	0.0118388	0.0089150 (25%)	0.0315828	0.0139374 (50%)
medium	0.244468	0.2311328 (5%)	0.3486278	0.2531194 (25%)
large	3.651052	3.53266 (4%)	4.218648	3.705996 (13%)
enormous	6.6198979	6.4470059 (3%)	8.171786	7.405516 (10%)
ginormous	14.973419	14.6999 (2%)	16.06594	14.55392 (10%)

TABLE IV: Average time per 5 executions for TBB outer-loop and OpenMP implementations using G++ and Intel Compiler with flags (percentage improvements)



XI. Final Configuration

The final optimum configuration found is the Thread Building Blocks solution with the Intel C++ compiler and the following flags: `-O3 --xCORE-AVX2 -ipo -no-prec-div`. TBB was chosen over OpenMP as both have quite close performance but TBB is more consistent with its results.

XII. Conclusion

In conclusion, the TBB outer loop performs best in a one-step optimisation by only changing the algorithm from sequential to parallel. Further optimisations using a compiler that utilizes CPU-specific features manages to further improve the TBB implementation. However, from this optimisation step the OpenMP implementation benefits significantly and in terms of speed, it is matches for big datasets, if not exceeding the speed of the TBB counterpart. Finally, the OpenCL implementation has achieved significant reduction in speed but it does not outperform the CPU based parallelism.

Acknowledgment

- Baron Khan implemented the OpenMP solution and performed program profiling and analysis.
- Malhar Jajoo implemented TBB and OpenCL solutions.
- Nikolay Yotov implemented testing scripts and worked on the Intel compiler optimisation.

References

[1] C. Adams, “What Is PageRank?” June 2016. [Online]. Available: https://www.bruceclay.com/blog/what-is-pagerank/

[2] P. Louridas, “pagerank,” 2015, January 2. [Online]. Available: https://github.com/louridas/pagerank

[3] D. Austin, “How Google Finds Your Needle in the Web’s Haystack,” *AMS*. [Online]. Available: http://www.ams.org/publicoutreach/feature-column/fcarc-pagerank

[4] HP, “HP EliteDesk 800 G2 Tower PC,” p. 2, 2013. [Online]. Available: http://www.hp.com/sbso/hpinfo/newsroom/HPElitePC2015/AMSHPEliteDesk800G2TowerPCDatasheet.pdf

[5] Intel, “Intel Turbo Boost Technology 2.0.” [Online]. Available: https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

[6] Intel, “Intel Core i7-6700 Processor.” [Online]. Available: https://ark.intel.com/products/88196/Intel-Core-i7-6700-Processor-8M-Cache-up-to-4_00-GHz

[7] J. Doweck, “Inside Intel Core Microarchitecture and Smart Memory Access,” Intel, Tech. Rep. [Online]. Available: http://www.iuma.ulpgc.es/~nunez/procesadoresILP/Intel64-Core-smartmemoryaccess-sma.pdf

[8] Nvidia, “GeForce GT 730 — Specifications.” [Online]. Available: https://www.geforce.com/hardware/desktop-gpus/geforce-gt-730/specifications

[9] K. Hinum, “Intel HD Graphics 530,” Oct 2016. [Online]. Available: https://www.notebookcheck.net/Intel-HD-Graphics-530.148358.0.html

[10] R. V. Meter, “Computer Architecture 2012 Fall.” [Online]. Available: http://web.sfc.keio.ac.jp/~rdv/keio/sfc/teaching/architecture/computer-architecture-2012/lec03-fastest.html

[11] Intel Developer Zone, “parallel_reduce Template Function.” [Online]. Available: https://software.intel.com/en-us/node/506154

[12] Linux man page, “clock(3): determine processor time.” [Online]. Available: https://linux.die.net/man/3/clock

[13] Intel, “Intel Threading Building Blocks (Intel TBB).” [Online]. Available: https://software.intel.com/en-us/intel-tbb/?cid=sem43700010399115721

[14] Intel. (2016) Tbb automatic paritioner. [Online]. Available: https://software.intel.com/en-us/node/506062

[15] ——. (2016) Tbb parallel reduce. [Online]. Available: https://software.intel.com/en-us/node/506153

[16] ——. (2016) Tbb parallel reduce explanation. [Online]. Available: https://software.intel.com/en-us/node/506063

[17] OpenMP, “Home - OpenMP,” 2016. [Online]. Available: http://www.openmp.org/

[18] Darthmouth College, “What is OpenMP,” 2009, March 23. [Online]. Available: https://www.dartmouth.edu/~rc/classes/intro_openmp/print_pages.shtml

[19] Intel. (2016) Opencl c++ specification. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2-cplusplus.pdf

[20] Intel. (2016) OpenCL C specification. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-openclc.pdf

[21] The Khronos Group Inc, “Scalar Data Types.” [Online]. Available: https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/scalarDataTypes.html

[22] Y. Wang, “Step by Step Performance Optimization with Intel C++ Compiler,” Oct 2014. [Online]. Available: https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler