A Comparative Study of Parallel Algorithms for the Girth Problem

Michael J. Dinneen **Masoud Khosravani** **Kuai Wei**

Department of Computer Science,
University of Auckland,
Private Bag 92019, Auckland, New Zealand
mjd,masoud,kuai@cs.auckland.ac.nz

Abstract

In this paper we introduce efficient parallel algorithms for finding the girth in a graph or digraph, where girth is the length of a shortest cycle. We empirically compare our algorithms by using two common APIs for parallel programming in C++, which are OpenMP for multiple CPUs and CUDA for multi-core GPUs. We conclude that both hardware platforms and programming models have their benefits.

1 Introduction

Graphs are models widely used in science and engineering, and graph algorithms are the basic blocks of many algorithmic solutions to real world problems. In this paper we study the problem of efficiently finding the girth in a graph or digraph on today's common workstations or servers, which often have several processing units (CPUs and GPUs). Modern graph applications require us to find fast algorithms capable of processing large volume of data. In such cases even a low-order polynomial time algorithm may not be able to accomplish a computational task in an acceptable time limit on a single CPU. As a common solution, one can deploy a large number of processors to do the task concurrently. We will discuss how to design and implement parallel girth algorithms and will present actual timing results for classes of hard test graphs.

1.1 Background on parallel programming

Designing parallel PRAM algorithms for graph problems has been the topic of a lot of research; see [3, 18, 19]. Several popular textbooks on parallel computing, such as [9], now address commonly-used shared memory parallel models like Pthreads (POSIX Thread API) and OpenMP (the standard directive-based parallel [17]). In addition to utilizing multiple CPU processors, recently there are more interests in the research community to explore the power of Graphics Processing Units (GPU) for solving graph problems. GPUs are high performance many-core processor devices that were originally designed to handle computation in image processing. General-Purpose computation on Graphics Processing Units (GPGPU) is the technique to use GPUs for solving a wider range of problems.

Among the first concrete results, Harish and Narayanin [10] introduced some parallel GPU algorithms for various graph problems. In [14], Katz and Kider presented an algorithm using GPUs for solving the all-pairs shortest path problem. Checking graph connectivity was the topic of the paper [20] by Soman, Kishore and Narayanan. As a final example, Leist and Playne [11] gave a GPU parallel algorithm for graph component labeling.

Despite the fact that designing and implementing parallel algorithms have been a major research topic, there are a few results on comparative studies of different APIs and architectures. Comparing CUDA and OpenMP for implementing various parallel girth algorithms is another focus of this paper. With respect to restrictions imposed by the architecture of GPUs, designing and implementing efficient parallel GPU algorithms for irregular data types is a challenging task. When one tries to implement a parallel algorithm for irregular data types on GPUs, there is a large gap between the theoretical and the actual results. Since GPGPU follows the Single Instruction Multiple Data (SIMD) paradigm, as an alternative benchmark, we use the OpenMP API standard, which supports multi-CPU shared-memory parallel programming. It supports C/C++, and Fortran programming languages on many architectures and operating systems. We note that CUDA is (currently) restricted to only NVIDIA graphic cards. However, OpenCL may also be easily used as an implementation choice for many other platforms (e.g. ATI Radeon GPUs) and usually with very little (if any) performance loss [6].

1.2 The girth problem

Girth is defined to be the length of a shortest cycle in a graph if one exists. Generating random graphs with large girth has applications in modelling and testing software systems and coding theory. Producing Tanner graphs with large girths is a main step in construction a Low-Density Parity-Check (LDPC) code; see [2, 12, 15]. A Tanner graph is a bipartite graph whose adjacency matrix is the parity-check matrix of a binary code.

Girth and diameter of a graph are related parameters. The diameter of a biconnected graph with girth

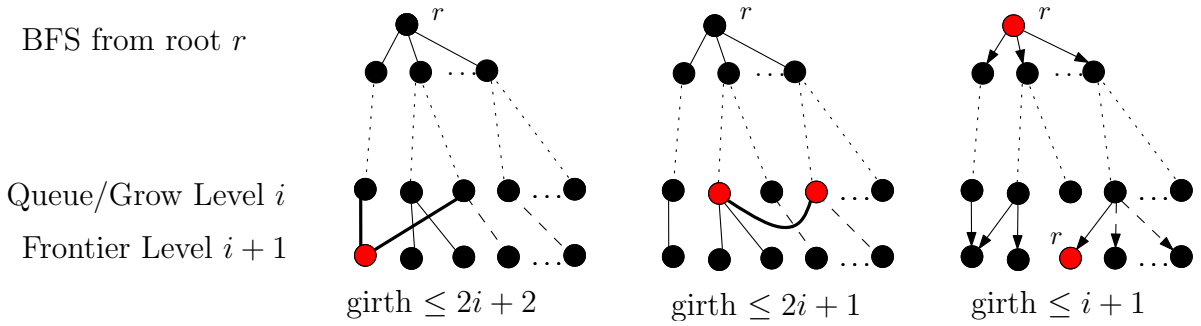

Figure 1: The process of detecting a short cycle via BFS.

$2d$ is at least d. The degree-diameter problem is the well-known problem of finding the largest possible graph with a given degree and diameter [5], and each best-known case usually has a large girth [8]. A large graph with bounded degree and diameter is a good model for an interconnection network topology that has some restrictions on the number of connections between hubs or routers and its maximum communication time between any two nodes.

There is an $O(mn)$ sequential algorithm for finding the girth of a graph G, where n is the order and m is the size of G (see [4]). One basically repeats a Breadth-First Search (BFS) algorithm from each node of a graph while tracking the cycles that are encountered. By imposing some restrictions on the input graph or relaxing the exactness of the solution, one can find a faster solution for the girth problem. For example, Itai and Rodeh [13] presented an $O(n^2)$ algorithm that finds a cycle which may have one edge more than the minimum. When a graph is restricted to be planar or has bounded genus, there is a linear time algorithm for the girth problem; see Djidjev [7]. Recently, Lingas and Lundell [16] presented a new approximation algorithm for the girth problem.

To our knowledge, this is the first study on parallel CPU and GPU implementations of the girth problem. One of our parallel algorithm uses parallel BFS, while the other algorithm is based on adjacency matrix multiplication. For each case we tailor our implementations to fit the hardware constraints (e.g. number and speed of processors; memory size and latency) of the selected platform.

1.3 Organization of the paper

The structure of this paper is as follows. In the next section, we introduce two parallel algorithms for computing the girth of a graph. Then in Section 3, we explain how those algorithms have been implemented by CUDA and OpenMP APIs. This section also describes a couple of potential optimizations. In Section 4, we describe the methods of generating four different sets of test graph data, specifically designed to strain our girth algorithms. A discussion on the results of testing our algorithms is the topic of Section 5. At the end of the paper we summarize our results and suggest topics for further study.

2 Two Parallel Algorithms

In this section we explain formally our parallel algorithms, including sample pseudo-code, for computing the girth.

Our first algorithm uses a slightly-modified parallel implementation of BFS, starting from each node. The algorithm (running in parallel from all roots) stops when the length of the first cycle is found. The approaches of detecting the girth in undirected or directed graphs are different.

1. For an undirected graph, a cycle is detected when a node in the frontier of the BFS has two parents already visited, or if it finds two nodes at the same distance (level) that are joined by an edge.

2. For a directed graph, a cycle containing the root is detected when the root node first appears in a frontier level of the BFS.

Figure 1 shows how an upper-bound of a smallest cycle is obtained via BFS. For undirected graphs we need to finish the current frontier for the two-parent case (left subfigure; even-length cycle); however, we can terminate immediately the search for the cross-edge case (middle subfigure; odd-length cycle). For directed graphs we can terminate the search when the root is first revisited (right subfigure; first cycle). The smallest upper-bound found over all BFSs is the actual girth of the graph and inter-process synchronization is needed to stop all parallel BFSs whenever the first cycle is found. See Algorithm 1.

Algorithm 1: Parallel girth algorithm via BFS.

```
Input: A Graph G = (V, E)
Output: The girth of G
girth = |V| + 1;
foreach node v ∈ G in parallel do
    Run BFS algorithm rooted at v;
    Let c be the length of first circuit detected;
    girth = min(girth, c);
```

Our second algorithm is based on doing repeated matrix multiplications of the adjacency matrix of a digraph. Let M be the adjacency matrix of a digraph G. It is well-known from graph theory, that the value of each entry $a_{i,j}$ of M^k represents the number of walks of length k from nodes i to j. Specially, the value on the diagonal entry $a_{i,i}$ shows the number of

directed circuits (closed walks) that start and end at i. This is easily adapted for our directed girth algorithm (Algorithm 2).

Algorithm 2: Girth via matrix multiplication.

Input: A Directed Graph $G = (V, E)$ as Adjacency Matrix M
Output: The *girth* of G
$M^0 = I$;
$M^1 = M$;
$i = 1$;
while $Trace(M^i) = 0$ **do**
  Compute in parallel (binary matrix multiplication): $M^{i+1} = M^i \times M$;
  $i = i + 1$;
$girth = i$;

To use this approach for undirected graphs, we need to adapt the aforementioned property of the powers of adjacency matrices to detect the smallest undirected cycle. First we need to ignore the circuits of length 2 (e.g. any edge (u, v) implies a circuit (u, v, u)). Secondly, note that any other smallest circuit of length at least 3 that we detected is, in fact, a cycle corresponding to the girth. Furthermore, we are only interested in knowing that the number of walks between i and j, $i \neq j$, is at least 2, so a possible optimization technique is to restrict to Boolean entries instead of integer entries.

Let the entry $b^k_{i,j}$ of matrix N^k denote the number of walks between i and j that do not traverse the same edge consecutively; clearly $b^k_{i,j} \leq a^k_{i,j}$ where $a^k_{i,j}$ is the entry of M^k. We can calculate N^k from matrices N^{k-1}, N^{k-2}, and $N^1 = M^1$, using a simple recurrence (modified vector products with respect to N^{k-2} where a row of N^{k-1} times a column of M yield an entry of N^k).

$$b^k_{i,j} = \bigvee_{\forall s \,:\, a^1_{s,j}=1} b^{k-1}_{i,s} \wedge \overline{b^{k-2}_{i,s}}$$

We parallelize by data partitioning the output rows of the matrix N^k; rows assigned to the available processors. The undirected graph version of Algorithm 2 also uses the two-path idea as illustrated in Figure 1, where we stop computing when k is half of the actual girth. The process is to first test if the following odd-length cycle condition is met:

$$\exists\{r, u, v\} : a^1_{u,v} \wedge b^k_{r,u} \wedge b^k_{r,v}$$

Then (if the previous condition is not met) test if the following even-length cycle condition is satisfied:

$$\exists\{r, u, v, w\} : a^1_{u,w} \wedge a^1_{v,w} \wedge b^k_{r,u} \wedge b^k_{r,v}$$

Note that all the values of the existential variables are distinct and both these conditions may be tested during the generation using the recurrence for N^{k+1}. Thus, the seemingly extra intra-level detection time of $O(n^3)$ is not required.

We end this section by mentioning the expected running times of our two algorithms for *sparse* graphs—those graphs with $m = O(n)$ edges. Sparse graphs are usually those graphs with large girth and will be prominent in the test cases for our algorithms. Also note that sparse $n \times n$ matrix multiplication can be done in time $O(n^2)$ if one uses the appropriate data representation [1, 6].

```
int undirectedGirth(const Graph &G)
{
 int n = G.order();
 int level[n];
 int smallest = n+1;  // value for infinity

 for (int r=0; r<n-2; r++) // minimum is 3-cycle
 {
   fill(level,level+n,-1); // unseen flags as -1
   level[r]=0;

   queue<int> toGrow;  // sequential FIFO queue
   toGrow.push(r);

   while ( !toGrow.empty() )
   {
     int grow = toGrow.front(); toGrow.pop();

     // try next r if this BFS is too deep
     if ( level[grow]*2+1 >= smallest ) break;

     const vector<int> nbrs = G.neighbors(grow);
     for (int i=0; i<nbrs.size(); i++)
     {
        int u = nbrs[i];
        if ( u < r ) continue;  // optimization

        if ( level[u] < 0 )
        {
          level[u]=level[grow]+1; // now seen
          toGrow.push(u);
        }
        else if ( level[u]==level[grow] )
        {
          if ( level[u]*2+1<smallest )
               smallest=level[u]*2+1;
          break; // try next r
        }
        else if ( level[u]==level[grow]+1 )
        {
          if ( level[u]*2 < smallest )
               smallest = level[u]*2;
        }
     }
   } // while BFS queue not empty
 }
 return smallest;
}
```

Figure 2: Sequential C++ girth function.

Theorem 1. *Given a sparse input graph G, Algorithm 1 runs on a machine with p processors in time $O(n^2/p)$.*

Theorem 2. *Given a sparse input graph G, Algorithm 2 runs on a machine with p processors in time $O(gn^2/p)$, where g is the girth of G.*

Note that the girth g is usually much smaller than the order n. Thus, both algorithms may be more practical than the other for different types of input cases.

3 Parallel Implementations

As noted in Section 2, BFS and adjacency matrix multiplication are the building blocks of our parallel

girth algorithms. Each program is a modified version of one of those basic algorithms. The program names that we introduce in this section correspond to the column headings of our final timing results of Tables 1 and 2 (at the end of the paper). Note the suffix `_p` on a program name denotes a "preprocessed" version, which is explained at the end of this section.

3.1 Cuda BFS program

In our programs `cuda_BFS` and `cudaBFS_p` we assign one CUDA *block* of threads (also known as *workgroup* in OpenCL) to each node. Each block is responsible to run one BFS in parallel using inter-block shared memory in addition to the GPU global memory. Furthermore, different blocks will run in parallel. Since our device memory size is limited, for large graphs we have to divide the nodes into separate groups.

The following proposition gives an optimization to save memory when searching for the girth in sparse undirected graphs.

Proposition 3. *When implementing the BFS-based algorithm for computing the undirected girth, it only needs to remember the nodes visited in the last three levels.*

Proof. Suppose we are exploring neighbors at level $i \geq 1$, where the root node is at level 0. By definition, the frontier level $i+1$ should only contain nodes not placed at any level $0 \leq j \leq i$, as depicted in Figure 1. Furthermore, a node x at level i can not have a neighbor y at distance $j < i-1$ from the root since that would imply x should be at level $j+1 < i$. Thus, we only need to remember nodes at levels $i-1$, i, and $i+1$ when doing the BFS search. □

Our CUDA programs implement Algorithm 1 by using three arrays to represent the last three levels of the BFS search tree: one for parent (of the grow) level, one for the grow level, and one for the frontier level. The algorithm initializes the arrays with the root node as the only item in the parent level and the neighbors of the root as the grow level. The frontier level is processed by finding the neighbors of the elements of the grow level that are not already in either the parent or grow levels.

All threads are intra-block synchronized at the end of each level. At this time, we change the roles of the arrays by doing pointer exchanges to save time by not having to copy the grow level to the new parent level and the frontier level to the new grow level. The old parent level is emptied and becomes the target for the new frontier. As soon as the first cycle is detected by one thread, its length is compared (and atomic exchanged) with the length of the shortest-known cycle that is stored in shared memory.

We also save the value of the shortest cycle from shared memory to global memory (inter-block communication) for determining the minimization of all BFS searches. This allows for early termination of deep BFS trees and reduces the overall running time of the implementation.

3.2 Cuda matrix-based algorithms

As explained earlier in Section 2, the programs `cudaMAT` and `cudaMAT_p` use the powers of the adjacency matrix for finding a smallest cycle originating from any node. We have only one thread being assigned sole ownership in writing the values of the i-th row of the output matrix of paths of length k. Since, for large graphs, we have more rows than the total number of available threads. Thus, we assign a contiguous group of rows to a particular thread. In our implementation, we have to do block synchronization (unlike our BFS implementation). This is done by repeated kernel launches, as illustrated in the following CUDA snippet.

```
for (int dist=1; dist <= order/2; dist++)
{
  Girth_Kernel<<<NUM_BLOCKS, NUM_THREADS>>>
  (graph, girth_D, DistMAT, DistLens, dist);

  cudaMemcpy(&girth_H, girth_D, sizeof(int),
             cudaMemcpyDeviceToHost);
  if (girth_H <= order) break;
}
```

3.3 OpenMP implementations

Our OpenMP implementations (`ompBFS`, `openBFS_p`, `openMAT` and `openMAT_p`) of the algorithms follow the same logic as explained for CUDA implementations. These were developed by adding `#pragma omp` directives to our sequential C++ code (e.g. add one above the first C++ `for` loop of Figure 2).

3.4 Final implementation remarks

In some of the implementations (denoted with a suffix `_p`), we first apply a preprocessing procedure to eliminate all nodes that are not clearly involved in any cycle of the graph. In other words, we iteratively delete all nodes of degree at most one. For the digraph input cases, we iteratively delete all sinks and sources. Note that the preprocessing times are included in the reported computational elapsed times.

The speed-up of this procedure is the result of reducing the order of the input graph. If the graph is not reduced significantly, then the preprocessing procedure may increase the running time. To explicitly display the impact of this procedure, we use it for all our parallel implementations.

The algorithms for finding girth in directed graphs are implemented by applying proper changes to the algorithms for the undirected ones. In `d_cudaBFS` and `d_ompBFS` we discover the directed cycles as soon as a back edge to the root is detected (see Figure 1).

In `d_cudaMAT` and `d_ompMAT` we use the parallel version of adjacency matrix multiplication as explained in Algorithm 2.

While processing graphs, we ignore those nodes that have larger index than the current root. This helps shrink the search space with no change in the correctness of the algorithm. Consider a cycle C of shortest length. If we start a BFS at the node with smallest index of C we will detect this cycle since no nodes of C are ignored. This pruning method is applied for both BFS and adjacency matrix multiplication approaches.

4 Generating Girth Test Data

The standard random graph generators are not desirable because they either produce graphs with no cycles or very small girth (e.g. dense graphs). To test the performance and correctness of our algorithms, we produced several classes of graphs of large order and large girth.

We have four test suites for undirected graphs:

big cycles We construct random trees in which each node is replaced by a big cycle. Then we choose one node from each cycle that represent two neighboring nodes and connect them by an edge.

Cayley graphs Let S be a set of generator for a finite group $(H, \cdot)$. The nodes of a Cayley graph is the set H and $S \subseteq H$ is used to define edges. We connect a node h to a node h' if there is an element $s \in S$ such that $h' = h \cdot s$. The graph is undirected if S is closed under inverses (e.g. $s \in S$ implies $s^{-1} \in S$). We used the semi-direct product procedure given in [5] to generate large sparse graphs.

cycle graphs These graphs are generated by connecting a sequence of large cycles on a path and adding a few extra edges randomly. These extra edges may span the length of the connected cycles or may be a chord of one cycle.

sparse graphs These graphs are produced by taking random trees, generated by using Prüfer codes, and then randomly connecting pairs of nodes.

We also have three test suites for directed graphs:

directed big cycles These are generated using the same procedure as `big cycles` but with each cycle being a directed cycle.

cycle digraphs Each of these digraphs was created by generating a union of large random directed cycles.

sparse digraphs These digraphs are created by first generating a rooted random tree (all arcs directed from parent to children). Then several random directed edges are added from a descendant node to an ancestor to form directed cycles.

Each test suite[1] consists of eight subsets of [di]graphs, indexed from 0 to 7. Each subset, labeled by i, consists of 25 [di]graphs with the number of nodes ranging between $2^i \cdot 1000$ and $2^{i+1} \cdot 1000$. So the overall range of our test graphs varies from graphs with 1000 nodes up to graphs with 256000 nodes.

5 Comparative Study

We implemented our parallel algorithms using C++ (gcc 4.4) with the two APIs: CUDA 3.2 and OpenMP 3.0. To run our CUDA programs, we used an Nvidia Tesla C2050 series (Fermi class) graphics card. The C2050 has Nvidia compute capability 2.0 and consists of 14 multiprocessors (MPs). Each MP has 32 cores and 3Gb cache (global memory). Each of the 448 cores operates at 1.15 GHz frequency. For our graphics card, each block (of threads) supports up to 1024 threads. For running our OpenMP programs, we used two hyper-threaded quad-core 2.5 GHz Intel CPUs which provides at least 8 and up to 16 independent Pthreads. Due to the hardware available to us, we are restricted to using a smaller number of OpenMP threads compared to what our GPU device has. However, one benefit of using OpenMP over CUDA is that the memory available is larger (48Gb vs 3Gb DRAM) and much faster (data transfer rate).

We provide in Table 1 and Table 2 a summary of our programs. These tables contain the average running times for each of the 25 graphs per subset of a test suite, then the average of all 200 graphs in each test suite, and finally the overall average running times. These times are wall clock times in seconds. For the CUDA implementation we do not include the I/O time for loading the graphs into device memory. We also do not include any disk I/O time for any program.

To have a better evaluation of our algorithms, we also use two sequential algorithms for computing the girth of a graph. One of them is the Sage's (Mathematics Software[2]) algorithm for finding the girth of undirected graphs [21]. Our other sequential program, `girthseq`, for undirected graphs use the BFS algorithm that was presented earlier (and listed in Figure 2). We also have a similar C++ implementation, `d_girthseq`, for directed graphs.

In general, as expected, the overall average performance (the last row in the two tables) of the sequential algorithms (`girthsage` and `girthseq` in Table 1 and `d_girthseq` in Table 2) are much slower than our parallel implementations.

Our two parallel algorithms running on OpenMP (`ompBFS` and `ompMAT`) perform about the same, which is about eight times faster than `girthseq`. On the other hand, our two parallel implementations running on the GPU (`cudaBFS` and `cudaMAT`) have performances that vary for undirected graphs and directed graphs. Our `cudaBFS` has the best overall performance for undirected graphs (18.6 times speed-up), and `cudaMAT` has the best overall performance for directed graphs (31.5 times speed-up).

Even though CUDA programs have the best overall performance in both undirected and directed graphs, the OpenMP still have advantages for solving small graphs. More specifically, OpenMP programs always outperform on the smaller graphs (subset 0) in the four test suites of undirected graphs and (subsets 0–2) in the three test suites of directed graphs. When the graph orders increase, the CUDA programs show their advantages.

For both CUDA and OpenMP, we find that the pre-processed versions are faster for the graphs in `sparse_graphs`, but increases the computation time for all other test classes. This is expected because only the class of `sparse_graphs` contains many nodes that are not on any cycle. We note that for digraphs, we could not gain better performance with our chosen

[1] These test suites are available by request.

[2] Note we selected this open-source platform for our base-line benchmark since it seems to out-perform our commercial software such as Mathematica 7.

preprocessing implementations.

6 Conclusions and Future Work

In conclusion, both OpenMP and CUDA based parallel programs improve the computation time of detecting the girth in undirected and directed graphs for our extensive test data. For small graphs/digraphs OpenMP seems to be faster (can't exploit multiple threads) than larger graphs. Both algorithm design approaches and both implementation APIs are valuable.

We note that the amount of human effort for CUDA is clearly expensive—we are waiting for higher-level programming tools (like OpenMP but for GPUs).

For the future we would like to try C# Parallel Task Library, CUDA 4.0 Thrust Library and new C++ Patterns Library (PPL). Also, we want to try other parallel hardware and possible different graph test cases for the girth problem. Also, we would like to consider performing performance evaluations on emerging, virtualized computing models (cloud resources) such as Amazon EC2 or Google AppEngine.

Acknowledgements

The authors would like to thank the University of Auckland for support in an FRDF grant 9843/3626216 for providing the necessary hardware and a Faculty of Science PhD research stipend for the second author.

References

[1] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39:85–97, March 1996.

[2] Shashi Kiram Chilappagari, Dung Viet Nguyen, Bane Vasić, and Michael W. Marcellin. Girth of the Tanner graph and error correction capability of LDPC codes. In *Communication, Control, and Computing, 2008 46th Annual Allerton Conference on*, pages 1238–1245, September 2008.

[3] Francis Y. Chin, John Lam, and I-Ngo Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25:659–665, September 1982.

[4] Michael J. Dinneen, Georgy Gimel'farb, and Mark C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages, 2nd Edition*. Pearson (Education New Zealand), 2009. ISBN 978-1-4425-1206-1 (pages 264).

[5] Michael J. Dinneen and Paul R. Hafner. New results for the degree/diameter problem. *Networks*, 24:359–367, October 1994.

[6] Michael J. Dinneen, Masoud Khosravani, and Andrew Probert. Using OpenCL for implementing simple parallel graph algorithms. In Hamid R. Arabnia, editor, *Proceedings of the 17th annual conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11), part of WORLDCOMP'11*, pages 1–6, Las Vegas, Nevada, July 18–21 2011. CSREA Press.

[7] Hristo Djidjev. A faster algorithm for computing the girth of planar and bounded genus graphs. *ACM Transactions on Algorithms*, 7(1):3, 2010.

[8] Geoffrey Exoo and Robert Jajcay. On the girth of voltage graph lifts. *European Journal of Combinatorics*, 32:554–562, May 2011.

[9] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, 2 edition, January 2003.

[10] Pawan Harish and P.J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor Prasanna, editors, *High Performance Computing HiPC 2007*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer Berlin / Heidelberg, 2007.

[11] Kenneth A. Hawick, Arno Leist, and Daniel P. Playne. Parallel graph component labelling with GPUs and CUDA. *Parallel Computing*, 36(12):655 – 678, 2010.

[12] Xiao-Yu Hu, Evangelos Eleftheriou, and Dieter-Michael Arnold. Regular and irregular progressive edge-growth Tanner graphs. *IEEE Transactions on Information Theory*, 51(1):386–398, 2005.

[13] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM J. Computing*, 7(4):413–423, 1978.

[14] Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH / EUROGRAPHICS Symposium on Graphics Hardware*, GH'08, pages 47–55. Eurographics Association, 2008.

[15] Sunghwan Kim, Jong-Seon No, Habong Chung, and Dong-Joon Shin. Quasi-cyclic low-density parity-check codes with girth larger than 12. *IEEE Transactions on Information Theory*, 53(8):2885–2891, 2007.

[16] Andrzej Lingas and Eva-Marta Lundell. Efficient approximation algorithms for shortest cycles in undirected graphs. *Information Processing Letters*, 109(10):493–498, 2009.

[17] OpenMP. The OpenMP API specification for parallel programming, site visited 2011. `http://openmp.org`.

[18] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Survey*, 16:319–348, September 1984.

[19] V. Nageshwara Rao and Vipin Kumar. Parallel depth first search. Part I. Implementation. *International Journal of Parallel Programming*, 16:479–499, 1987.

[20] Jyothish Soman, Kothapalli Kishore, and P.J. Narayanan. A fast GPU algorithm for graph connectivity. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.

[21] William Stein. *Sage: Open Source Mathematical Software (Version 4.6)*. The Sage Group, October 2010. `http://www.sagemath.org`.

Table 1: Timings in seconds of girth algorithms on undirected graphs.

	Subset	girth_sage	girthseq	cudaBFS	cudaBFS_p	cudaMAT	cudaMAT_p	ompBFS	ompBFS_p	ompMAT	ompMAT_p
big cycles	0	0.0212	0.0018	0.0006	0.0007	0.0048	0.0043	0.0003	0.0003	0.0012	0.0023
	1	0.1312	0.0089	0.0017	0.0019	0.0193	0.0141	0.0010	0.0010	0.0057	0.0044
	2	0.5884	0.0373	0.0061	0.0065	0.0488	0.0387	0.0036	0.0036	0.0219	0.0196
	3	2.2220	0.1516	0.0218	0.0251	0.1202	0.0895	0.0147	0.0144	0.0770	0.0738
	4	11.6868	0.7531	0.1144	0.1203	0.3557	0.2623	0.0710	0.0710	0.3720	0.3727
	5	49.4324	3.2966	0.4731	0.4977	0.9750	0.7130	0.3387	0.3593	1.6402	1.6643
	6	174.0984	12.9000	1.6726	1.7588	3.5142	2.4929	1.2805	1.3231	5.4710	5.7810
	7	504.0968	38.1329	4.8217	5.0730	9.8500	6.9969	3.7729	3.8581	17.4768	18.8116
	Average	92.7847	6.9103	0.8890	0.9355	1.8610	1.3265	0.6853	0.7039	3.1332	3.3412
Cayley graphs	0	0.0596	0.0038	0.0036	0.0015	0.0352	0.0343	0.0006	0.0007	0.0031	0.0023
	1	0.1724	0.0133	0.0033	0.0036	0.1237	0.1040	0.0019	0.0020	0.0104	0.0084
	2	0.5064	0.0430	0.0109	0.0115	0.6081	0.5148	0.0060	0.0062	0.0529	0.0561
	3	1.1616	0.1668	0.0263	0.0292	1.1158	0.9550	0.0223	0.0218	0.1280	0.1220
	4	4.8736	0.6849	0.1209	0.1221	4.2231	3.6082	0.1106	0.1518	0.5886	0.6021
	5	9.2036	2.4132	0.2133	0.2172	5.0933	4.3397	0.3369	0.3764	0.9630	0.9246
	6	7.1400	4.9858	0.1338	0.1407	2.2464	1.9095	0.6583	0.7053	0.4896	0.4715
	7	12.7224	19.7854	0.1740	0.1836	1.4997	1.1844	2.6333	2.6962	0.6287	0.5478
	Average	4.4800	3.5120	0.0858	0.0887	1.8682	1.5812	0.4712	0.4951	0.3580	0.3418
cycle graphs	0	0.0212	0.0016	0.0007	0.0009	0.0027	0.0026	0.0003	0.0029	0.0012	0.0003
	1	0.0588	0.0066	0.0012	0.0035	0.0027	0.0026	0.0008	0.0046	0.0022	0.0049
	2	0.2236	0.0316	0.0034	0.0080	0.0039	0.0039	0.0031	0.0079	0.0056	0.0290
	3	0.5428	0.1294	0.0068	0.0095	0.0071	0.0050	0.0132	0.0186	0.0200	0.0183
	4	1.1220	0.4681	0.0133	0.0142	0.0056	0.0060	0.0494	0.0577	0.0366	0.0307
	5	2.0728	1.9861	0.0240	0.0255	0.0077	0.0084	0.2429	0.3891	0.1281	0.2356
	6	4.7428	7.9299	0.0533	0.0562	0.0161	0.0182	1.0435	1.2081	0.1983	0.2992
	7	7.7152	31.1896	0.0878	0.0905	0.0272	0.0314	4.2280	4.3701	0.3352	0.4226
	Average	2.0624	5.2179	0.0238	0.0260	0.0091	0.0098	0.6977	0.7574	0.0909	0.1301
sparse graphs	0	0.0096	0.0011	0.0015	0.0008	0.0175	0.0018	0.0003	0.0004	0.0009	0.0003
	1	0.0520	0.0064	0.0024	0.0010	0.0565	0.0041	0.0011	0.0011	0.0029	0.0010
	2	0.2084	0.0321	0.0027	0.0014	0.0571	0.0079	0.0048	0.0036	0.0102	0.0028
	3	0.6536	0.1291	0.0076	0.0025	0.2135	0.0132	0.0196	0.0124	0.0404	0.0066
	4	1.9240	0.5460	0.0207	0.0051	0.7572	0.0249	0.0848	0.0526	0.1384	0.0195
	5	5.6632	2.0746	0.0663	0.0102	2.3316	0.0435	0.2965	0.3819	0.5345	0.1590
	6	10.5944	6.4405	0.1040	0.0201	2.8499	0.0694	0.9124	0.9687	0.9201	0.2733
	7	34.1212	23.4932	0.4045	0.0382	17.3415	0.1912	3.4374	3.0170	3.4024	0.4003
	Average	6.6533	4.0904	0.0762	0.0099	2.9531	0.0445	0.5946	0.5547	0.6312	0.1078
AVERAGE		26.4951	4.9326	0.2687	0.2650	1.6728	0.7405	0.6122	0.6277	1.0533	0.9802

Table 2: Timings in seconds of girth algorithms on directed graphs.

	Subset	d_girthseq	d_cudaBFS	d_cudaMAT	d_ompBFS	d_ompBFS_p	d_ompMAT
directed big cycles	0	0.0016	0.0268	0.0068	0.0003	0.0005	0.0008
	1	0.0082	0.4085	0.0121	0.0008	0.0012	0.0051
	2	0.0372	0.5955	0.0274	0.0033	0.0042	0.0247
	3	0.1682	0.9255	0.0641	0.0143	0.0163	0.1168
	4	0.7007	0.2859	0.1419	0.0616	0.0662	0.4696
	5	2.7614	0.6990	0.2931	0.2471	0.2596	1.6903
	6	9.8746	1.5924	0.6623	0.9421	1.0185	5.5603
	7	41.9673	5.7967	2.1259	4.0266	4.2226	25.0513
	Average	6.9399	1.2913	0.4167	0.6620	0.6986	4.1149
cycle digraphs	0	0.0024	0.0086	0.0638	0.0004	0.0017	0.0061
	1	0.0076	0.0075	0.0934	0.0010	0.0030	0.0040
	2	0.0252	0.0025	0.0135	0.0029	0.0059	0.0015
	3	0.1074	0.0016	0.0030	0.0122	0.0158	0.0016
	4	0.4542	0.0030	0.0053	0.0505	0.0560	0.0033
	5	1.5451	0.0035	0.0025	0.1654	0.1769	0.0054
	6	7.4248	0.0083	0.0074	0.9728	1.0582	0.0465
	7	26.8069	0.0105	0.0064	3.5560	3.7332	0.1559
	Average	4.5467	0.0057	0.0244	0.5952	0.6313	0.0280
sparse digraphs	0	0.0013	0.5340	0.0025	0.0003	0.0008	0.0003
	1	0.0062	0.1696	0.0042	0.0009	0.0018	0.0007
	2	0.0287	0.0389	0.0084	0.0034	0.0054	0.0018
	3	0.1181	0.0193	0.0083	0.0136	0.0508	0.0038
	4	0.4792	0.0139	0.0164	0.0548	0.0630	0.0116
	5	1.7474	0.1223	0.0111	0.1933	0.2058	0.0859
	6	6.6283	0.0343	0.1097	0.8764	1.0621	0.2391
	7	19.8751	0.1582	0.1472	2.6206	2.8045	0.3118
	Average	3.6105	0.1363	0.0385	0.4704	0.5243	0.0819
AVERAGE		5.0324	0.4778	0.1599	0.5759	0.6181	1.4083