2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-3577-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPSW52791.2021.00050

PIGO: A Parallel Graph Input/Output Library

Kasimir Gabert
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia
kasimir@gatech.edu

Ümit V. Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia
umit@gatech.edu

***Abstract*—Graph and sparse matrix systems are highly tuned, able to run complex graph analytics in fractions of seconds on billion-edge graphs. For both developers and researchers, the focus has been on computational kernels and not end-to-end runtime. Despite the significant improvements that modern hardware and operating systems have made towards input and output, these can still become application bottlenecks. Unfortunately, on high-performance shared-memory graph systems running billion-scale graphs, reading the graph from file systems easily takes over $2000\times$ longer than running the computational kernel. This slowdown causes both a disconnect for end users and a loss of productivity for researchers and developers.**

We close the gap by providing a simple to use, small, header-only, and dependency-free C++11 library that brings I/O improvements to graph and matrix systems. Using our library, we improve the end-to-end performance for state-of-the-art systems significantly—in many cases by over $40\times$.

I. Introduction

There is significant work developing highly optimized sparse linear algebra and graph systems, including competitions for graph kernels [1] and sparse machine learning [2], programming models and corresponding high-performance libraries [3]–[6], and full graph databases [7]. These all focus on providing high-performance kernel runtimes for a variety of datasets. In this paper, we are not providing yet another computational graph library or a new graph programming model. Instead, we address an important—yet largely overlooked—aspect of *using* and *developing* sparse graph and matrix systems: the input and output (I/O). Our focus is on shared-memory multi-core servers. We show that graph I/O is frequently the single slowest factor in the end-to-end performance of otherwise fast computations.

In many cases implementations have stuck with sequential I/O, presumably under a longstanding belief that parallel I/O is not achievable without a dedicated parallel I/O system. In the literature, graph and matrix I/O times are rarely reported and, hence, not highly optimized. While it is the case that SATA serializes disk access [8], implementing only sequential I/O misses three major and common opportunities for parallelism. First, a hardware Redundant Array of Inexpensive Disks (RAID) controller can read from multiple drives over separate SATA connections in parallel [9]. Second, Non-Volatile Memory (NVM) is now widely deployed [10] and provides parallelism through both SSDs and the NVM express (NVMe) interface to motherboards [8]. Third, file systems include tuned and effective caches, supporting parallel reads and writes [11]. In these cases, reading and writing in parallel can provide a significant end-to-end improvement for applications.

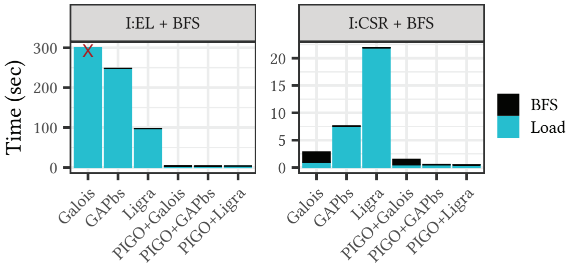

Fig. 1. Edge list (I:EL) and binary (I:CSR) input (I:) times and BFS runtimes for com-Friendster, a social network graph with 65 M vertices and 3.6 B edges. Note the differing scales between I:EL and I:CSR. Using PIGO significantly improves end-to-end runtimes for all systems, bringing them much closer to kernel runtimes. Galois did not finish I:EL within several thousand seconds.

Our goal is to *remove the burden of building efficient I/O*. We target two types of users: the researcher, who is developing a single graph kernel and does not care about the production readiness of the code; and the developer, who is building a graph library for production level end-user use.

Importantly we need to be *useful* for researchers who want to test out a kernel idea quickly without buying into a large, complex graph system, complete with a steep learning curve and powerful, yet complex programming models. We do not want researchers to have to stick with a particular graph format, spend energy converting between them, or designing ad-hoc binary formats with potentially subtle errors. At the same time, we seek to provide *best-in-class performance*.

To achieve our goal, we provide PIGO[1], a small, C++11 header-only library. PIGO takes in a filename and returns the graph loaded into memory. Compared against optimized and parallel graph loading in state-of-the-art libraries or simple, ad-hoc loading, we show that using PIGO can quickly increase both productivity and end-to-end performance. In Figure 1, we show three leading graph libraries [3]–[5] running breadth-first search (BFS) on a large graph. When running without PIGO, it takes over one hundred seconds to load from an ASCII edge list file and 0.01 seconds to run. With PIGO, the loading time is reduced to 3.5 seconds (from an edge list) or 0.5 seconds (from a binary compressed sparse row).

[1] Available at https://github.com/GT-TDAlab/PIGO.

978-1-6654-3577-2/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPSW52791.2021.00050

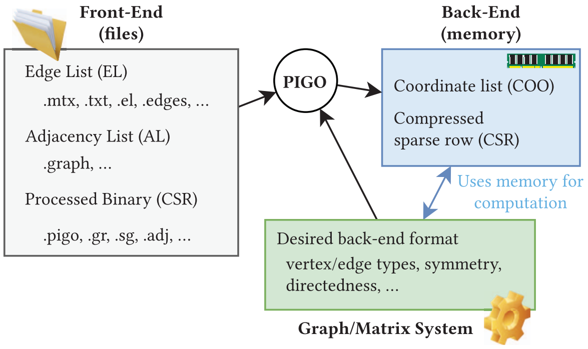

Fig. 2. As an I/O library, PIGO takes a desired back-end configuration from the computation system and transforms the front-end appropriately.

II. PIGO I/O Library

A. Requirements

PIGO is built to satisfy the following requirements.

Requirement 1: Fast enough to make effective use of modern hardware and operating systems' parallel I/O performance.

Requirement 2: Support for common front-end and back-end graph formats, removing the need for slow and ad-hoc preprocessing, including weights, symmetry, and directedness.

Requirement 3: Easy and useful integration with both graph analysis systems and one-off graph and matrix programs.

We address these requirements through our design. First, we read in parallel and we decode bytes directly into integers, floats, comments, or spaces. Our library was developed with performance as a goal and written carefully as such. For example, we limit unnecessary runtime code through template parameters which avoids branch mispredictions.

As a library focused solely on I/O, we are able to address Requirement 2 by ensuring we can quickly add new formats. Front-end formats are graphs that are stored on disk, downloaded, and used in graph pipelines. The two main formats are *edge lists* (`EL`) and *adjacency lists* (`AL`), both of which are ASCII encoded. On the back-end, we currently support two structures. The first is a *coordinate list* (COO), which stores non-zeros with their explicit row and column coordinates, and the second a *compressed sparse row* (CSR), which stores non-zero elements in a single contiguous block of memory and a separate offsets block that stores the beginning of rows. At the moment we support several basic preprocessing steps such as removing self loops and symmetrizing the graph. Current and future work includes more robust preprocessing and support for additional formats. Figure 2 shows PIGO's high-level design meeting this requirement.

Finally, we build PIGO to be a C++11 header-only library, allowing it to be easily integrated into projects and systems. The programming interface is designed to be simple to learn and quick to use, as shown in Section II-C. For further usability, we have an experimental port which wraps PIGO into a shared object, making it potentially available to applications in C, Rust, Python, and more.

```
COO<Label, Ordinal, LabelStorage, Flags...>
     COO { filename };
void COO.save(filename);
LabelStorage COO.x();   // Get row labels
LabelStorage COO.y();   // Get col labels

CSR<Label, Ordinal, LabelStorage,
    OrdinalStorage, ...> CSR { filename };
void CSR.save(filename);
LabelStorage CSR.endpoints();
OrdinalStorage CSR.offsets();

Graph { filename } : CSR;
EdgeIt Graph.neighbors(Label v);

Matrix { filename } : CSR;
RowIt Matrix.row(Label r);
```

Fig. 3. The high-level API for PIGO.

B. Overview

PIGO reads and writes *in parallel* from a variety of ASCII files along with custom binary files. Reading binary files is straightforward: the size information for data can be quickly encoded and read from the file header or metadata, and the transfer itself consists of parallel and independent `memcpy` calls. ASCII files, however, are the standard file format for both large and small sparse matrices [12] and graphs [13].

PIGO handles reading and writing ASCII files with two passes. We focus on reading throughout this paper, however writing follows a similar strategy of first counting and then copying. First, PIGO loads the input file into memory via `mmap`[2]. In the first pass, the structure is read out in parallel. That is, the number of spaces, newlines, and integers are counted—while ignoring comment lines and end-of-line comments. After this, memory is allocated and a prefix sum is performed so each thread knows its position in the back-end memory to write to. The second parallel pass then iterates over the file again, parses the integers and copies out the data.

C. Application Programming Interface

PIGO is used by declaring a *back-end* format and providing a filename as input. PIGO then loads the file in parallel and converts it appropriately into the requested back-end structure. Parameters, such as the data types to use inside the matrix and preprocessing flags, such as whether to symmetrize the matrix, are given as template parameters.

In Figure 3 we show the main concept of our API. Our complete API is documented in our repository.

`Label` contains the type of the row or column (or vertex) labels. In many cases, it can be a 32-bit unsigned integer. The `Ordinal` type contains the type that will count (hence ordinal). If the number of edges is large, this may need to be 64-bit. The `Storage` types indicate how

[2] `mmap` is a POSIX-compliant call to place the file contents into memory, making the whole file available as a `char*`.

```
#include "pigo.hpp"
#include <iostream>
int main(int argc, char** argv) {
    pigo::Graph g { argv[1] };
    for (auto n : g.neighbors(123))
        std::cout << n << std::endl;
    return 0;
}
```

Fig. 4. An example program using PIGO with default template values.

```
template <class vertex>
graph<vertex> readGraphFromFile(char* fname
    , bool, bool) {
    pigo::Graph<uintE, uintT,
            uintE*, uintT*> g {fname};
    long n = g.n();
    long m = g.m();
    uintT* offsets = g.offsets();
    uintE* edges = g.endpoints();
    // Continue with remaining Ligra code
    ...
}
```

Fig. 5. A high-level replacement `readGraphFromFile` function for Ligra. This will cause Ligra to read with PIGO, resulting in PIGO+Ligra.

PIGO should allocate the memory. PIGO supports raw pointers (`T*`), vectors (`std::vector<T>`), and shared pointers (`std::shared_ptr<T>`). Finally, the `Flags` are used to indicate various preprocessing steps, for example to symmetrize the file or to remove self loops.

D. Example Programs

Here we show two example uses of PIGO. The first is a simple standalone program that a researcher might write to develop some kernel. To install PIGO, only a single `pigo.hpp` file is needed. The complete example can be seen in Figure 4.

For the next example, we show how to extend Ligra [3] to PIGO+Ligra, allowing it to take advantage of significantly improved *binary and ASCII* loading. The function `readGraphFromFile` is replaced with the code in Figure 5. As PIGO takes care of reading the file, and can handle preprocessing steps, all that needs to occur beyond the PIGO call is building Ligra's vertex objects.

E. Algorithm Details

There are two main problems reading ASCII files. The first is if we evenly partition the data into chunks, the partitions may not line up on clean integer boundaries. The second is that the destination to write to in memory is not known apriori. We solve the first problem by adjusting the start and end boundaries locally for each thread. Concretely, each thread finds either the next newline or the next integral character and sets that as the thread data boundary. For P threads, this can result in up to P additional reads of bytes, however for any reasonable file the number of bytes overlapping between segments is a small constant. Solving the next problem is done via the two passes described in Section II-B. Note that these passes are done in parallel. We parallelize with OpenMP.

```
Input: memory mapped file $F$
$N[1,\ldots,\text{num threads}] \leftarrow \langle 0,0 \rangle$
for chunk $c \in F$ do in parallel
    $t \leftarrow$ thread ID
    $N[t] \leftarrow \langle$ integer count , newline count $\rangle$
allocate offsets, endpoints
prefixSum$(N)$
for chunk $c \in F$ do in parallel
    $\langle e, v \rangle \leftarrow N[t]$
    foreach integer in $c$ do
        endpoints$[e] \leftarrow$ data
        if passed newline then
            offsets$[v] \leftarrow e$
            $v \leftarrow v + 1$
        $e \leftarrow e + 1$
```

Algorithm 1: The core idea of the `AL` reading.

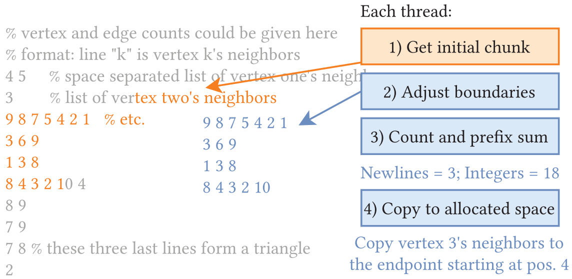
Each thread:
% vertex and edge counts could be given here
% format: line "k" is vertex k's neighbors
4 5 % space separated list of vertex one's neigh
3 % list of vertex two's neighbors
9 8 7 5 4 2 1 % etc.
3 6 9
1 3 8
8 4 3 2 10 4
8 9
7 9
7 8 % these three last lines form a triangle
2
9 8 7 5 4 2 1
3 6 9
1 3 8
8 4 3 2 10
1) Get initial chunk
2) Adjust boundaries
3) Count and prefix sum
Newlines = 3; Integers = 18
4) Copy to allocated space
Copy vertex 3's neighbors to the endpoint starting at pos. 4

Fig. 6. An example reading an `AL` in PIGO.

To make this more concrete, we present the core idea for reading `AL` in Algorithm 1. A visual example is shown in Figure 6. `EL` reading is similar but more simple, as N only needs to contain newlines. Binary reading and writing simply reads or writes in parallel at byte boundaries.

III. Experiments and Results

In this section we present experiments and results. We want to demonstrate that PIGO can read ASCII and binary files significantly faster than known graph systems, providing significant end-to-end improvements.

As exemplar graph systems, we use Ligra [3], GAPbs [4], and Galois [5], chosen as leading systems. We are not aware of any systems with a larger focus on I/O than them. We compiled with GCC 9.1.0 and ran on a machine with 1TB of RAM, an Intel DC P3700 2TB NVMe SSD connected with PCIe 3.0, and two 18-core Intel Xeon E5-2695 v4 CPUs at 2.10 GHz. We ran Ubuntu 16.04 with ext4. Our datasets are from the Network Repository [13] and SuiteSparse [12].

In Figure 7 we show the potential parallel improvement for reading a large binary file starting from an empty, or *cold* cache. A *warm* cache is common when running multiple experiments or using a freshly downloaded file. With NVMe, we show even a cold cache has parallel potential: reading

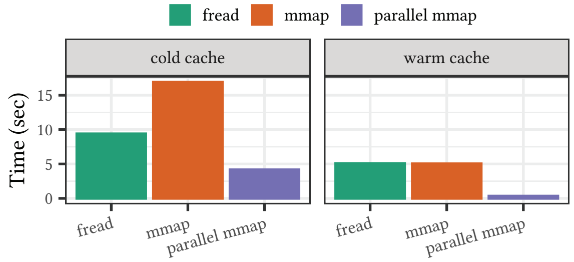

Fig. 7. A microbenchmark showing the worst-cast (cold cache) read times of 10GB of random data with a parallel `mmap` and sequential strategies.

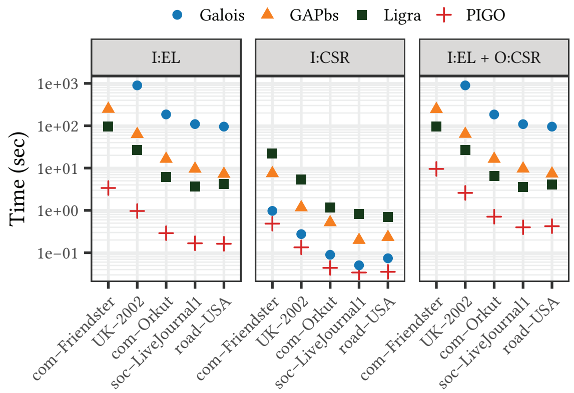

Fig. 8. I/O times for tested graphs with GAPbs's I/O, Ligra's I/O, and PIGO. Graphs are arranged in size, with com-Friendster 31 GB on disk and road-USA 940 MB. The last column is reading, converting to CSR, and then writing.

in parallel is around twice as fast as reading sequentially. However, with warm caches, we show the memory of the system becomes a factor. Here, reading in parallel is 15× faster and gets close to the system memory bandwidth.

We stress there is a significant room for I/O improvement beyond simply reading bytes in parallel, as evidenced by PIGO's overall performance gains. In Figure 8 we show the loading times for graphs both from `EL` and from binary formats. Galois uses `mmap` for binary files and Ligra can either use `mmap` or `fread`, with subsequent parallel preprocessing. Galois has the second best binary reading yet the slowest ASCII reading. PIGO remains faster in all cases. For both Ligra and GAPbs, loading from `EL` and converting to CSR with PIGO is faster than loading binary files without PIGO.

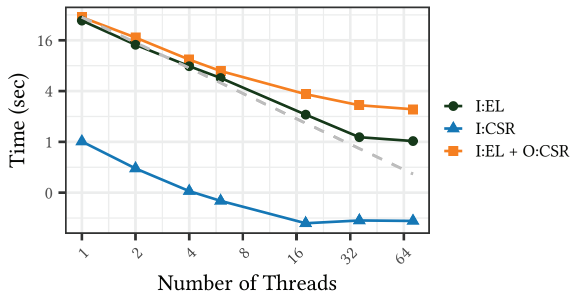

Fig. 9. A scalability study showing the impact of increasing threads on UK-2002. The NUMA boundary is reached at 18 cores, after which scalability only continues to increase for ASCII processing and COO to CSR conversions.

We show the scalability in Figure 9. Up to the NUMA boundary scalability increases with all approaches. Overall, we have shown that PIGO provides significant performance improvements.

IV. Conclusion

We tackle the long-standing belief that parallel I/O is not fruitful for loading sparse matrix and graph files. While there may be limited parallel improvements to cold cache raw binary reads over SATA, we show there is much to be gained with RAID controllers, NVMe SSDs, or a warm cache. We introduce a simple to use, header-only C++ library that enables both highly-tuned graph systems and small, one-off graph kernels to take advantage of parallel I/O. Our library is open source and it brings over $40\times$ end-to-end performance improvements to state-of-the-art graph and sparse matrix systems.

References

[1] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song *et al.*, "Static graph challenge: Subgraph isomorphism," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–6.

[2] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[3] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[4] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[5] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 456–471.

[6] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang, "Lagraph: A community effort to collect graph algorithms built on top of the graphblas," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2019, pp. 276–284.

[7] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," *arXiv preprint arXiv:1910.09017*, 2019.

[8] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "Mqsim: A framework for enabling realistic studies of modern multi-queue ssd devices," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 49–66.

[9] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, 1988, pp. 109–116.

[10] M. Jung, "OpenExpress: Fully hardware automated open research framework for future fast NVMe devices," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 649–656.

[11] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[12] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom, "The SuiteSparse matrix collection website interface," *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019.

[13] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com