External Memory Pipelining Made Easy With TPIE

Lars Arge[1], Mathias Rav[1], Svend C. Svendsen[1]
MADALGO, Dept. of Computer Science
Aarhus University
Aarhus, Denmark
{large,rav,svendcs}@madalgo.au.dk

Jakob Truelsen
SCALGO
Aarhus, Denmark
jakob@scalgo.com

***Abstract*—When handling large datasets that exceed the capacity of the main memory, movement of data between main memory and external memory (disk), rather than actual (CPU) computation time, is often the bottleneck in the computation. Since data is moved between disk and main memory in large contiguous blocks, this has led to the development of a large number of I/O-efficient algorithms that minimize the number of such block movements. However, actually implementing these algorithms can be somewhat of a challenge since operating systems do not give complete control over movement of blocks and management of main memory.**

TPIE is one of two major libraries that have been developed to support I/O-efficient algorithm implementations. It relies heavily on the fact that most I/O-efficient algorithms are naturally composed of components that stream through one or more lists of data items, while producing one or more such output lists, or components that sort such lists. Thus TPIE provides an interface where list stream processing and sorting can be implemented in a simple and modular way without having to worry about memory management or block movement. However, if care is not taken, such streaming-based implementations can lead to practically inefficient algorithms since lists of data items are typically written to (and read from) disk between components.

In this paper we present a major extension of the TPIE library that includes a pipelining framework that allows for practically efficient streaming-based implementations while minimizing I/O-overhead between streaming components. The framework pipelines streaming components to avoid I/Os between components, that is, it processes several components simultaneously while passing output from one component directly to the input of the next component in main memory. TPIE automatically determines which components to pipeline and performs the required main memory management, and the extension also includes support for parallelization of internal memory computation and progress tracking across an entire application. Thus TPIE supports efficient streaming-based implementations of I/O-efficient algorithms in a simple, modular and maintainable way. The extended library has already been used to evaluate I/O-efficient algorithms in the research literature, and is heavily used in I/O-efficient commercial terrain processing applications by the Danish startup SCALGO.

***Keywords*-I/O-efficient algorithms; C++; software framework.**

1: Supported in part by the Danish National Research Foundation grant DNRF84 and Innovation Fond Denmark. MADALGO is the Center for Massive Data Algorithmics, a center of the Danish National Research Foundation.

I. Introduction

When handling large datasets that exceed the capacity of the main memory, movement of data between main memory and external memory (disk), rather than actual (CPU) computation time, is often the bottleneck in the computation. The reason for this is that disk access is orders of magnitude slower than internal memory access. Thus, since data is moved between disk and main memory in large contiguous blocks, it is often more important to design algorithms that minimize block movement than computation time when handling massive data. This has led to the development of a large number of *I/O-efficient algorithms* in the I/O-model by Aggarwal and Vitter [1]. In this model, the computer is equipped with a two-level memory hierarchy consisting of an internal memory capable of holding M data items, and an external memory of conceptually unlimited size. All computation has to happen on data in internal memory, and data is transferred between internal and external memory in blocks of B consecutive data items. Such a transfer is called an *I/O-operation* or *I/O*, and the cost of an algorithm is the number of I/Os it performs. The number of I/Os required to read or write N items from disk is $\mathrm{Scan}(N) = \lceil N/B \rceil$, while the number of I/Os required to sort N items is $\Theta(\mathrm{Sort}(N)) = \Theta((N/B)\log_{M/B}(N/B))$ [1].

While many I/O-efficient algorithms have been developed in the I/O-model of computation, actually implementing these algorithms can be somewhat of a challenge since operating systems do not give complete control over movement of blocks and management of main memory. However, two major libraries TPIE [13] and STXXL [10] have been developed to support I/O-efficient algorithm implementations. It turns out that most I/O-efficient algorithms are naturally composed of components that stream through one or more lists of data items, while producing one or more such output lists, or components that sort such lists. TPIE in particular uses this to provide an interface where list stream processing and sorting can be implemented in a simple and modular way, without having to worry about memory management or block movement. However, if care is not taken, such a streaming-based implementation can lead to practically inefficient algorithms since lists of data items are typically written to (and read from) disk between components. In implementations consisting of many small (but I/O-efficient) components, the I/Os incurred when writing and reading such lists can easily comprise more than half of the total number of I/Os. While this may not be a problem when considering asymptotic theoretical performance, it is unacceptable in practice when the total execution time is measured in hours or days.

In this paper we present a major extension of the TPIE library that includes a pipelining framework that allows for practically efficient streaming-based implementations while minimizing I/O-overhead between streaming components. The framework pipelines streaming components to avoid I/Os between components, that is, it processes several components simultaneously while passing output from one component directly to the input of the next component in main memory. TPIE automatically determines which components to pipeline and performs the required main memory management, and the extension also includes support for parallelization of internal memory computation and progress tracking across an entire application. Thus TPIE supports efficient streaming-based implementations of I/O-efficient algorithms, and TPIE applications are naturally implemented as reusable components, thereby reducing programming time and code duplication.

A. *Previous Work*

As mentioned, two major software libraries support I/O-efficient algorithm implementations for big data analysis, namely TPIE [13] and STXXL [10]. They are both C++ software libraries, and as opposed to many of the frameworks that have emerged for supporting big data analysis in the last decade, such as e.g. MapReduce [9], Spark [15], and Flink [3], they mainly support single-host implementations. One reason for this is that the libraries, in particular TPIE, are designed to support implementations on standard commodity hardware. Another reason is that no efficient distributed algorithms are known for many of the problems for which I/O-efficient algorithms have been studied and implemented; we refer to surveys [14] and descriptions of implementations (e.g. [2], [4], [6], [11], [12]) for references. Thus in this paper we also focus on single-host implementations. However, is should be mentioned that in the context of distributed programming, pipelining has recently been studied with the Thrill framework [8].

Although both are libraries for implementation of I/O-efficient algorithms, the overall philosophies of TPIE and STXXL are somewhat different. The philosophy of TPIE (the Templated Portable I/O Environment) is to provide a high-level interface that allows for easy translation of abstract I/O-efficient algorithm descriptions into code that is portable across computational platforms and not unnecessarily complex. Thus building on the fact that most I/O-efficient algorithms are composed of streaming components, TPIE provides a generic stream interface that hides how blocked I/O is performed and instead provides methods for processing one data item at a time. TPIE also provides internal memory management, where memory allocations are automatically counted towards an application-wide memory limit, and where an application can at any point determine the currently available main memory. Thus applications do not have to explicitly keep track of available memory, which often simplifies implementations considerably. For example, in the TPIE built-in streaming-based implementation of the I/O-optimal $\mathcal{O}(\mathrm{Sort}(N))$ external multi-way merge-sort, the number of sorted streams that can be merged I/O-efficiently (without being swapped out by the operating system) depends on the available main memory, where care has to be taken to ensure that the memory used to hold blocks of items for each used stream is counted towards the amount of available memory; the TPIE memory management allows for determining the number of streams to merge without explicitly keeping track of available memory and memory used for blocked I/O. Overall, TPIE is designed to remove focus from the tedious details of creating I/O-efficient applications and allows for implementations that are efficient on all hardware platforms with minimal configuration.

The philosophy of STXXL (Standard Template library for XXL data sets) on the other hand is to achieve maximum I/O-throughput by reducing I/O-overhead as much as possible, e.g. by exposing the characteristics of the hardware to the application programmer. Thus, to avoid any overhead induced by the operating system, STXXL allows the user to configure separate disks for use with applications outside of the file system of the operating system. In fact, STXXL project programmers recommend that a separate disk is set aside for STXXL applications. STXXL also explicitly supports parallel disks. Like TPIE, STXXL supports streaming-based implementations and includes various basic streaming components such as sorting, but unlike TPIE it actually contains support for pipelining of streaming components. However, STXXL expects the application programmer to explicitly define which components to pipeline and explicitly manage main memory. Thus, the programmer e.g. has to specify how much memory each streaming component in a pipelined application should use. A separate (not officially released) branch of STXXL contains support for utilizing multi-core processors for the internal-memory work of pipelined applications [7]. Overall, STXXL is designed such that an application can be tailored to the available hardware, and with the proper configuration an STXXL application can achieve close to full utilization of the available I/O bandwidth.

B. *Our Results*

In this paper we present a major extension of the TPIE library that includes a pipelining framework that allows for practically efficient streaming-based implementations while minimizing I/O-overhead between streaming components. The extension also includes support for progress tracking across an entire application, and for parallelization of internal memory computation.

Like STXXL, the TPIE pipelining framework saves I/Os by passing intermediate results between streaming components directly in main memory. However, TPIE pipelining is the first framework to provide automatic pipeline and memory management, and thus combining the best of the TPIE and STXXL streaming philosophies. The framework is component-centric in that the memory requirement of each streaming component is specified locally by the component developer. The automatic pipeline and memory management then means that at runtime TPIE will automatically determine which components to pipeline, and distribute memory among multiple components of a large application in a way that automatically uses all the main memory available to the application. Thus, unlike in STXXL, a TPIE programmer e.g. does not have to consider how the memory use of the individual components has to be adjusted when they are combined into an application. While such adjustments along with adjustments of the grouping of components into pipelines can be done manually for small projects, it can be very cumbersome for large-scale professional software projects involving many programmers, where modification of a component to use more memory can very easily lead to memory over-usage problems (if the memory use of other components are not adjusted accordingly). Thus the TPIE component-centric approach simplifies the application development process, promotes modularity and supports maintainability. The extended library has already been used to evaluate I/O-efficient algorithms in the research literature (e.g. [4], [6]) and is heavily used in I/O-efficient commercial terrain processing applications by the Danish startup SCALGO[1]. The extension is integrated into the official TPIE project that is available on GitHub as free and open-source software[2].

II. An Example Problem

In this section we present an example of a typical sub-problem in an I/O-efficient application. We show that the problem benefits from a pipelined implementation; by implementing every sub-problem in a bigger data processing application using pipelining, more than half of the I/Os can be saved.

A. The Raster Transformation Problem

In geographic information systems (GIS), a terrain is often represented as a raster of heights with each cell indicating the height of the terrain in a certain point. Since the Earth is spherical and a raster is flat, it is not possible to map the entire surface of the Earth continuously to a raster. However, if only a particular region, country or continent needs to be represented, it is possible to project the chosen region to a plane in a way that roughly maintains the geodesic distances and areas. When several rasters must be processed together they must be in the same projection.

We call the problem of transforming a raster from one projection to another the *raster transformation problem*. Essentially, the problem consists of projecting each cell of the raster from the source projection plane to the unit sphere, and from the unit sphere to the target projection plane. These two steps can be represented by a function $f : \mathbb{Z}^2 \to \mathbb{Z}^2$ that maps each cell of the *target* raster projection to the corresponding cell of the *source* raster projection. Thus, in the raster transformation problem we are given an input raster A of size $W \times H$ (that is a W by H matrix of numbers) stored in row-major order, and we want to produce an output raster B of size $W' \times H'$ in row-major order, such that the value of a cell (x, y) in B is copied from the value of a cell $(x', y') = f(x, y)$ in A. Below we for convenience let $N = WH = W'H'$ be the number of cells in both the input and output raster.

The raster transformation problem can easily be solved in optimal $\mathcal{O}(N)$ time simply by for each cell (x, y) in B reading the corresponding input value at $f(x, y)$ in A. However, this solution might be very I/O-inefficient. For example, if f represents matrix transposition where $f(x, y) = (y, x)$, then each access to A requires a new block to be read (assuming $W, H \geq \frac{M}{B}$), and thus the solution performs $\Theta(N)$ I/Os in the worst case. For matrix transposition in particular, only $\Theta(\mathrm{Sort}(N))$ I/Os are required [1]. In fact, in general the raster transformation problem can be solved in $\mathcal{O}(\mathrm{Sort}(N))$ I/Os using a simple five step streaming algorithm (refer to Figure 1a): First a stream S_1 is constructed containing for each cell (x, y) of B an item consisting of a pair $(f(x, y), (x, y))$. Next S_1 is sorted such that $f(x, y)$ appear in the same row-major order used to store A. In the third

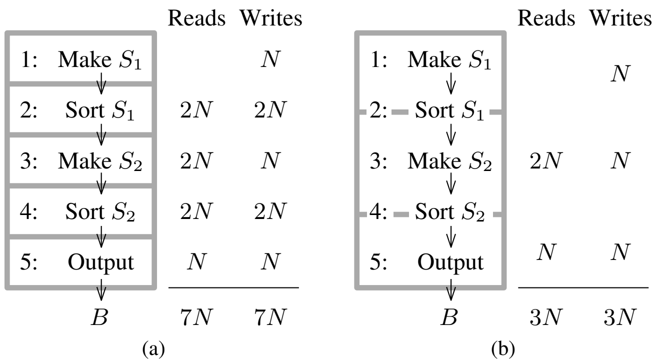


Figure 1. Raster transformation algorithm. (a) The algorithm without pipelining, requiring $7N$ reads and writes (assuming use of merge-sort using just one merge step). (b) The algorithm with pipelining, requiring just $3N$ reads and writes.

[1]SCALGO: Scalable Algorithmics. https://scalgo.com

[2]TPIE: Templated Portable I/O Environment. http://madalgo.au.dk/tpie

step, A and S_1 are then scanned simultaneously to construct a stream S_2 containing an item (x, y, v) for each pair $(f(x, y), (x, y))$ in S_1 where v is the value of A at position $f(x, y)$. Then S_2 is sorted into the row-major order used to store B. In the fifth and final step, S_2 is scanned and for each entry (x, y, v) the value v is output to $B[x, y]$. Since the algorithm performs a constant number of scanning and sorting steps it uses $\mathcal{O}(\mathrm{Sort}(N))$ I/Os and can easily be implemented using the streaming support of either TPIE or STXXL. Refer to GitHub[3] for a TPIE code example.

As discussed in the introduction, streaming-based implementations of even simple I/O-efficient algorithms, as the raster transformation algorithm above, might not be practically efficient because items are written to disk between steps. By applying pipelining to the above algorithm (and assuming use of merge-sort using just one merge step) it is possible to save $4N$ reads and $4N$ writes, which is over half of the I/Os in practice. Although this does not change the asymptotic I/O-complexity of the algorithm, it translates into a running time reduction from 39 hours to 17 hours if we assume an input size N of 1 TB and a disk read/write speed of 100 MB/s. The pipelining process conceptually transforms the five-step algorithm into a three-phase algorithm as indicated in Figure 1b, where e.g. the second phase consists of the merging part of the step two sorting, step three, and the run formation part of the step four sorting. Refer to the full paper for details [5].

B. STXXL Implementation

When implementing the raster transformation algorithm with pipelining using the STXXL streaming layer, the five steps of the algorithm are implemented individually as is natural from a software engineering point of view; we call each such individual part of a pipeline a *component*. However, since STXXL does not handle memory management, the implementation that combines the components then has to identify the three phases of the algorithm explicitly and compute how much memory is allocated to each of the components of a phase. The interested reader can refer to GitHub[4] for STXXL code that implements this, that is, the code that implements the five step algorithm in three phases. The code illustrates how three phases are explicitly identified and memory allocated. For example, in phase two the memory available for the two sorting components (merging of step two and run formation of step four) is computed by setting aside a buffer of size B of the available main memory for reading the input, and then share the remaining memory between the two sorting components.

Apart from the complexity that the need for phase identification and memory allocation adds to pipelined STXXL code, there are also some C++ syntax issues that add to the code complexity. More precisely, the C++ syntax used is quite verbose, since for technical reasons names of the components often need to be repeated. The reason is that STXXL combines pipelining components using a C++ feature known as *template instantiation* that allows for the compiler to inline function calls between different components of the pipeline. For performance reasons, this is necessary when many small components are pipelined. However, the template instantiation syntax is not well-suited for use in large pipelined applications.

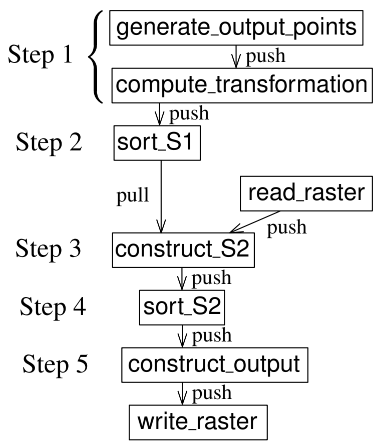
C. TPIE Implementation Using Pipelining

As in the case of pipelined STXXL, in the implementation of the five step raster transformation algorithm using the extended TPIE library with pipelining, the components of the pipeline are implemented individually. However unlike in the STXXL implementation, the combination of the components becomes very simple, since TPIE is component-centric and automatically identifies phases and performs memory management. To illustrate this, a diagram showing the eight components used to implement the five steps is given in Figure 2a along with the pipelining code in Figure 2b. Note how the code in Figure 2b lines 3-12 naturally corresponds to the pipeline in Figure 2a. Note also that the reading and writing of rasters are handled by two special components to separate the handling of specific raster formats from the algorithm, and how the two sorting components are implemented using two different built-in TPIE sorting components defined in lines 3 and 4 on Figure 2b. The reason two different sorter implementations are used (and that the pipeline is defined in two statements defining `p1` and `p2`, respectively) is that the output from the component sorting S_1 has to be read by the component constructing S_2 simultaneously with the output from the component reading the input raster A. Thus, the component constructing S_2 has to control when data is received from the sorting component, which is done through *pull-based streaming*. This functionality is implemented with a TPIE so-called *passive sorter* with an input and an output part defined in line 3. On the other hand, the component sorting S_2 is a more traditional pipelined component that uses *push-based streaming*, where input data is received from preceding component (in this case the component constructing S_2) when ready, and output data in turn pushed to the subsequent component. It is defined with an ordinary TPIE sorter in line 4.

Memory management. As mentioned, TPIE automatically manages memory and divides available memory among components in a pipeline. Thus in the pipeline definition in Figure 2b there is no code at all dealing with memory distribution. Often many components use only a small amount of static memory, whereas components such as sorting require dynamically allocated memory depending on the amount of available memory. In the latter case the component has to specify its minimum and maximum memory requirements

[3]https://github.com/Mortal/pipelining/blob/8996e5c87d/tpie_imperative/transform_paper.cpp#L60-L104

[4]https://github.com/Mortal/pipelining/blob/8996e5c8/stxxl/transform.cpp



(a)

```
1   void transform(raster_input & A, raster_output & B,
2                  tpie::progress_indicator_base & pi) {
3       auto sort_S1 = tpie::pipelining::passive_sorter<projected_point>();
4       auto sort_S2 = tpie::pipelining::sort(point::yorder());
5       tpie::pipelining::pipeline p1 = generate_output_points()
6           | tpie::pipelining::parallel(compute_transformation())
7           | sort_S1.input();
8       tpie::pipelining::pipeline p2 = read_raster(A)
9           | construct_S2(sort_S1.output())
10          | sort_S2
11          | construct_output()
12          | write_raster(B);
13      p1.forward("inputsize", A.dimensions());
14      p1.forward("outputsize", B.dimensions());
15      uint64_t n = A.cell_count() + B.cell_count();
16      p1(n, pi, TPIE_FSI);  // Execute the pipeline
17  }
```

(b)

Figure 2. Raster transformation algorithm. (a) Pipeline illustrated as components. (b) TPIE code implementing the pipeline.

in its implementation.

Metadata. Often many components in a pipeline need some sort of metadata about the items that are being streamed between components. In the example, certain components need to use the dimensions of the input and output rasters. While such metadata can of course be passed as parameters to the individual components in the pipeline definition, doing so makes the definition needlessly cluttered. Instead, TPIE provides a general facility for passing metadata between pipeline components. Thus, in Figure 2b lines 13-14, the pipeline definition uses forward() to pass the dimensions of the input and output rasters to the components that need them. The individual components can then obtain the metadata using fetch().

Progress reporting. In the example, the TPIE support for progress reporting (e.g. as a progress bar) is also used. As with memory requirements, the code required to supply progress information is not part of the code in Figure 2b where the pipeline is defined, but rather part of the implementation of the individual component. When executing the pipeline in line 16 of Figure 2b the argument pi is a reference to a progress indicator object that tells TPIE how to display progress. To provide accurate progress estimations, TPIE actually uses statistical information about progress of previous runs of the code. To store information about runs, the problem's instance size n, computed in line 15, as well as a symbol TPIE_FSI used to identify the application being executed, are also passed to the TPIE framework when executing the pipeline in line 16.

Parallelism. The example takes advantage of TPIE support for multi-core CPU parallelism in two ways. First, in the TPIE built-in implementation of multi-way merge-sort, the initial run-formation phase is automatically parallelized. Second, by wrapping the component compute_transformation in the directive tpie::pipelining::parallel(...) in line 6, TPIE automatically distributes this part of the computation among all available CPU cores.

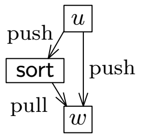
III. TPIE Pipelining

In this section we briefly describe the TPIE pipelining framework and how it is implemented. We refer the reader to the full version of this paper for details [5].

As described in Section II-C, a TPIE pipeline consists of a number of components that push data to or pull data from other components. We distinguish between two types of components, namely *regular components* that produce output as the input is processed, and *blocking components* that have to process all input before producing any output. Blocking components consist of two sub-components, namely an input and an output sub-component, where the input sub-component processes all input before the output sub-component is invoked to produce the output; the input sub-component might store intermediate results on disk. Merge-sorting is an example of a blocking component; the input sub-component naturally corresponds to the initial run formation step and the output sub-component to the merging step. Blocking components introduce the need for *pipeline phases* that are executed independently, since the input and output sub-components cannot be executed simultaneously. In the raster transformation algorithm in Section II, the three phases are exactly needed due to the two blocking sorting components.

A pipeline can conveniently be represented by a directed acyclic *flow graph*, with *regular nodes* corresponding to regular components and *input nodes* and *output nodes* corresponding to the sub-components of blocking components. Regular nodes are connected with other regular nodes and input and output nodes by edges directed along the streaming direction and labeled as *push edges* or *pull edges* in a natural way; note that a node cannot have both an outgoing push and an outgoing pull edge. Each input node is also connected with a directed *blocking edge* to the corresponding output node. To be able to automatically determine the phases of a pipeline, TPIE requires that the flow graph corresponding to the pipeline has two particular properties: First, if all blocking edges are removed, then no input and output nodes corresponding to the same blocking component should be in the same connected component. Second, if all push and pull edges are contracted, then the graph should be acyclic. The first property means that the connected components directly identify the pipeline phases that need to be executed independently. When each such connected component is contracted, each directed edge (u, v) in the resulting graph indicates that the phase corresponding to u needs to be executed before the phase corresponding to v. Thus the second property ensures that there exists a valid (topological) order in which to execute the components. When these two properties are fulfilled, TPIE builds the flow graph and computes connected components to identify phases, and then contracts the components and topologically sorts the graph to find the order in which to execute the phases. In each phase, TPIE assigns main memory to the individual components according to the given minimum and maximum requirements.

While the second flow graph property above has to be fulfilled for any pipelined program to be valid, the first property only has to be fulfilled if we require that all but blocking components can be pipelined, that is, that streaming items are only written to disk by blocking components. For example, consider the small pipeline shown on the right, where component u pushes to both a sorter and to a component w, which in turn pulls output from the sorter. In this case, u, w and the input and output nodes of the sorter are all in the same connected component in the phase graph without blocking edges. However, it is not possible to pipeline all the components in one phase, since the output from the sorter used in w is not available at the same time as the output from u also used in w. To remedy this problem, and make the flow graph fulfill the second property, a simple blocking component that delays the stream of items from u to w, by writing them temporarily to disk, can be inserted between u and w in the pipeline (such that the example has two phases). To support this, TPIE not only contains a built-in sorting blocking component, but also blocking components that delay and reverse a stream. Each of these components come in an *active* and a *passive* variant. In the active variant both the input and output sub-components use push-based streaming, and in the passive variant the input sub-component is push-based and the output sub-component pull-based.



References

[1] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 1988.

[2] Deepak Ajwani, Roman Dementiev, Ulrich Meyer, and Vitaly Osipov. Breadth first search on massive graphs. In *9th DIMACS Impl. Challenge Workshop: Shortest Paths*, 2006.

[3] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6), 2014.

[4] Lars Arge, Gerth Stølting Brodal, Jakob Truelsen, and Constantinos Tsirogiannis. An optimal and practical cache-oblivious algorithm for computing multiresolution rasters. In *Proc. 21st European Symp. Alg.*, 2013.

[5] Lars Arge, Mathias Rav, Svend C. Svendsen, and Jakob Truelsen. External Memory Pipelining Made Easy With TPIE. *ArXiv e-prints*, October 2017. `arXiv:1710.10091`.

[6] Lars Arge, Jakob Truelsen, and Jungwoo Yang. Simplifying massive planar subdivisions. In *Proc. 16th Workshop on Alg. Eng. Exp.* SIAM, 2014.

[7] Andreas Beckmann, Roman Dementiev, and Johannes Singler. Building a parallel pipelined external memory algorithm library. *Proc. 2009 Intl. Parallel and Distributed Processing Symp.*, 2009.

[8] Timo Bingmann, Michael Axtmann, Emanuel Jöbstl, Sebastian Lamm, Huyen Chau Nguyen, Alexander Noe, Sebastian Schlag, Matthias Stumpp, Tobias Sturm, and Peter Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. In *2016 IEEE Intl. Conf. Big Data*. IEEE, 2016.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Comm. ACM*, 51(1), 2008.

[10] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: standard template library for XXL data sets. *Softw., Pract. Exper.*, 38(6), 2008.

[11] Roman Dementiev, Peter Sanders, Dominik Schultes, and Jop Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *Exploring New Frontiers of Theoretical Informatics*. Springer, 2004.

[12] Ulrich Meyer and Vitaly Osipov. Design and implementation of a practical I/O-efficient shortest paths algorithm. In *Proc. Meeting on Alg. Eng. Exp.* Society for Industrial and Applied Mathematics, 2009.

[13] Thomas Mølhave. Using TPIE for processing massive data sets in C++. *SIGSPATIAL Special*, 4(2), 2012.

[14] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CSUR)*, 33(2), 2001.

[15] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10), 2010.