Using TPIE for Processing Massive Data Sets in C++

Thomas Mølhave*
Department of Computer Science
Duke University
Durham, NC
thomasm@cs.duke.edu

1 Introduction

The adaptation of I/O-efficient algorithms in commercial and research applications can be facilitated by well-designed software libraries. The Templated Portable I/O Environment (TPIE) [2] for C++ is one such library based on the I/O-model of Agarwal and Vitter [3]. TPIE contains a number of powerful algorithms and data structures, enabling the user to quickly develop software that scales to very large data sets. Figure 1 illustrates the power of I/O-efficient algorithms in general and TPIE in particular, in this case using an external memory sorting algorithm and priority queue. As the data size grows close to the 6GiB of main memory of the computer, the sorting algorithm from the C++ Standard Template Library (STL), `std::sort`, slows down dramatically. Beyond that point using `std::sort` is infeasible as running times extend into days and weeks even for date sizes only slightly larger than the main memory. STL's `std::priority_queue` behaves in the same way. The sorting algorithm and the priority queue from TPIE are well behaved, even as the size of the input data grows to terabytes. The STXXL [8] and LEDA-SM [5] libraries have goals and features similar to those of TPIE. STXXL aims to be very close to C++'s Standard Template Library (STL) but also offers pipe-lining and some usage of multiple cores. LEDA-SM is an extension to the Library of Efficient Data Types and Algorithms (LEDA) and consists of a number of I/O-efficient data structures and algorithms. Unfortunately, the LEDA-SM project is not active according to a statement on the project's website. On a slightly different level the cluster-friendly FG [4] library provides a framework for pipe-line structured programs that also scale to large data sets. A significantly reworked version 2.0 has been announced on the project website. Moving further into the distributed computing paradigm, the MapReduce [7] and Hadoop [1] frameworks are very popular for implementing algorithms on clusters with large numbers of computing nodes, but that is outside the scope of this article. We refer to [10] and the references therein for a more extensive survey of I/O-efficient algorithms and software libraries.

In this article we focus on the TPIE library, but many of the ideas carry over to other libraries such as STXXL. TPIE has a long history; it was started in 1994 at Duke University in North Carolina when it was known as the *Transparent Parallel I/O Environment* [9]. TPIE is currently hosted by the Center for Massive Data Algorithmics (MADALGO) at Aarhus University. From the beginning the goals of TPIE have been to hide the details of how I/Os are performed, and to provide the user with a set of standard tools and paradigms that can be used to effectively implement algorithms for large data sets.

*This work is supported by NSF under grants CCF-06 -35000, CCF-09-40671, and CCF-1012254, by ARO grants W911NF-07-1-0376 and W911NF-08-1-0452, and by U.S. Army ERDC-TEC contract W9132V-11-C-0003

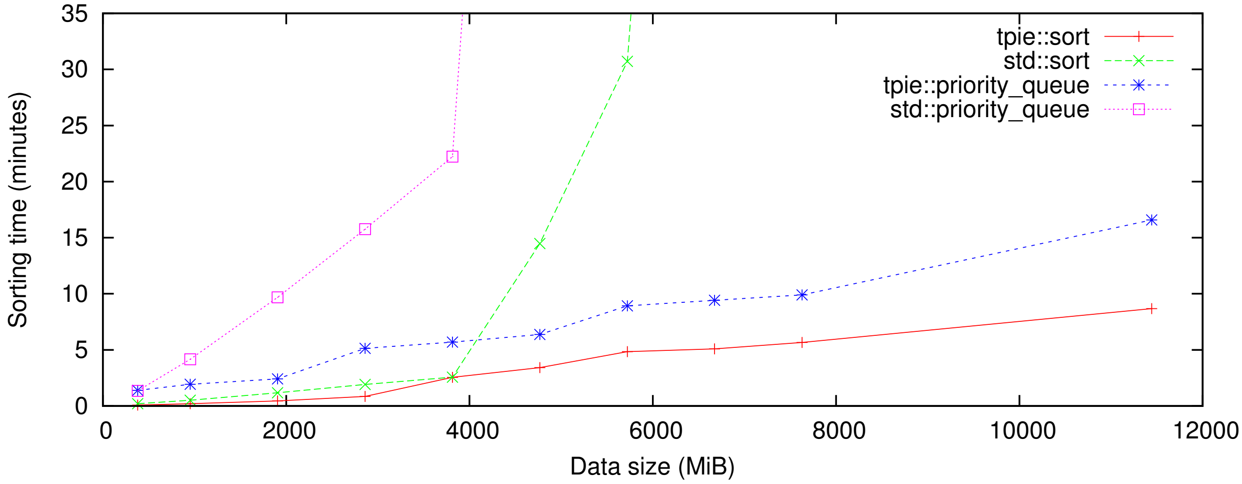

Figure 1: Performance of 32-bit integer sorting using `std::sort` and `std::priority_queue` from the Standard Template Library and TPIE's `tpie::sort` and `tpie::priority_queue`. The machine used had a 64 bit Intel Core i7 CPU and 6GiB of main memory. When the input data reached 4GiB the `std::priority_queue` version started hitting the disk due to the OS paging algorithm and it had not terminated after running for 6 hours. The same thing happened for `std::sort` when the data size neared 6GiB.

TPIE is used in many implementations of I/O-efficient algorithms and plays an important part in several large software projects such as TerraS̃TREAM [6]. TPIE is also currently used in commercial software packages on both Windows- and Linux-based systems.

2 Brief TPIE Overview

In this section we give a short overview of the fundamentals of TPIE and more information can be found in the online documentation [2].

Streams The fundamental primitive in TPIE is the `tpie::file_stream` class which models a sequence of data stored on disk that can be scanned sequentially. Algorithms developed with TPIE frequently store and manipulate streams as their basic unit of storage. An active `tpie::file_stream` uses $\Theta(B)$ bytes of memory and the rest of the contents of the stream is stored on disk. As a working example, assume we are writing a program that receives data in the form of a point cloud from $\mathbb{R}^3$ represented by C++ class `point`. A `tpie::file_stream` can be populated with these points for further processing using the following code:

```
tpie::file_stream<point> s;
s.open();
while (has_more_points()) {
  const point& p = get_next_point();
  s.write(p);
}
```

The stream `s` now contains all the input points (assuming `has_more_points` and `get_next_point` have been correctly implemented). If we want to scan those points in lexicographical order we need to sort them. This is done easily using the TPIE sort function.

```
tpie::sort(s); //sort s according to point::operator<
```

This assumes that `operator<` has been overloaded for the point class, but if that is not the case, or another order is desired, a custom comparator can be passed to the sorting function. The sorting function is based on the standard I/O-efficient merge sort [3] and uses $O(M)$ bytes of memory, and it can use multiple CPU cores for the internal sorting steps. As mentioned before, Figure 1 shows the performance of `tpie::sort` versus `std::sort`. As expected `tpie::sort` is superior when the data size grows large, but we also notice that it, in this case, is faster on smaller data sets as well. This comes from the fact that `tpie::sort` uses all the available CPU cores whereas `std::sort` runs only on a single core by default.

Data Structures TPIE contains implementations of I/O-efficient stacks, queues and priority queues. Their interfaces match the corresponding structures from the STL. For instance, an I/O-efficient priority queue, stack and queue can be defined in the following way:

```
tpie::stack<point> stck;
tpie::queue<point>q;
tpie::priority_queue<point> pq;
```

TPIE's priority queue behaves very well when the data size grows large, as can be seen in Figure 1. Here we compare `tpie::priority_queue` against `std::priority_queue` for the (somewhat contrived) problem of using a priority queue for sorting. Note that the priority queue is an example of a data structure that requires $\Omega(M)$ memory for efficient worst-case performance. This requirement is fulfilled using the TPIE memory accounting system system, described next.

Memory Accounting Since TPIE models the two-level I/O model it needs to know the memory size M and the block size B. When TPIE is initialized, the user can supply the value of M to the system — the value of B is set by default. TPIE contains its own memory system that ensures no more than M bytes of memory can be used. This implies, among other things, that the sorting algorithm will query the memory system to get the current amount of available memory and restrict itself accordingly. Similarly, the priority queue can be given a value $k \leq M$ and will never use more than k bytes of memory regardless of how many elements it contains at any given time. For instance, if the user needs to use two priority queues simultaneously each of them should be initialized by a value $k < M/2$. The first versions of TPIE globally overloaded `operator new` in order to keep track of the system-wide memory usage of the application, but this is impractical for large software projects where the overhead of keeping track of various strings and other trivial bookkeeping is high. Furthermore this approach is near impossible when third party dynamically loaded external libraries with their own memory systems are used. In order for the memory system to work in heterogeneous applications TPIE needs to be told explicitly about the memory use of data structures that are not part of TPIE. This includes STL data structures (e.g.. `std::vector` and `std::set`) as well as arrays and other heap-allocated objects. Memory accounting for STL data structures can be achieved by using the `tpie_allocator` type. For example, a vector of type `T` can be declared in the following way:

```
std::vector<T, tpie::allocator<T> > vec;
```

This works for all data structures that use the `std::allocator` paradigm. For standard heap-allocated objects, TPIE supplies `tpie_new`, which can be used as a drop-in replacement of the standard `new` operator. Finally, instead of trying to duplicate `delete[]` for arrays, TPIE encourages the use of `tpie::array` instead. These arrays behave much like standard arrays, but allocate memory using the TPIE memory system. Memory allocated using the standard C++ allocators will work as usual, but will not count towards the limit of M maintained by TPIE.

Scratch Space The data stored in a TPIE stream (or other TPIE data structures) is stored in a temporary file on disk. By default TPIE uses the standard system path (e.g. `/var/tmp` on Ubuntu), but this can be configured by the user. It is a good idea to direct TPIE to use a fast disk since the speed of this disk (or disks in case of RAID arrays) will be the bottleneck for most TPIE programs.

3 Conclusion & Future Plans

TPIE is still under active development and has a thriving, growing community of users and developers. The TPIE website [2] contains more information including full API documentation, and in-depth download and getting-started instructions.

A number of ideas are currently on the drawing board. One of the most important of these is pipe-lining support, which is also supported by STXXL. The idea is to be able to separate the algorithms into modules that are implemented independently, which will allow TPIE to better schedule I/Os. For instance, if several subsequent scans of the same `file_stream` are performed, these can be coalesced into one scan and combined with the pre-computing phase of any subsequent sorting operation. Pipe-lining does not change the asymptotic performance of the algorithm, but can significantly reduce the constants involved. Another significant feature is support for additional data structures, such as B-trees and similar tree-based data-structures.

References

[1] Apache Hadoop. `http://hadoop.apache.org`.

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

[3] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[4] T. H. Cormen and E. R. Davidson. FG: A framework generator for hiding latency in parallel programs running on clusters. In *Proc. Parallel and Distributed Computing Systems*, pages 137–144, 2004.

[5] A. Crauser and K. Mehlhorn. LEDA-SM: Extending LEDA to secondary memory. In *Algorithm Engineering*, pages 228–242. 1999.

[6] A. Danner, T. Mølhave, K. Yi, P. Agarwal, L. Arge, and H. Mitasova. TerraStream: From elevation data to watershed hierarchies. In *Proc. ACM International Symposium on Advances in Geographic Information Systems*, pages 28:1–28:8, 2007.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[8] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2008.

[9] D. E. Vengroff. A transparent parallel I/O environment. In *In Proc. DAGS Symposium on Parallel Computation*, pages 117–134, 1994.

[10] J. S. Vitter. *Algorithms and Data Structures for External Memory.* Now Publishers Inc., 2008.