Design Considerations for a Software Library Supporting Algorithms for Massive Data Sets (WP1, D6)

Peter Sanders*

May 29, 2001

1 Introduction

The status of the project is that we have completed the single-disk external memory library LEDA-SM [3] as planned (Section 2. Furthermore, our algorithm design work has made progress with respect to supporting parallel disks. [11, 9, 10, 4].

On the software side our next main goal is to develop essential components of a new library (working name AlPHA **Al**gorithms for **P**arallel **H**ard disk **A**rrays) with a focus on high performance operation in particular with respect to supporting parallel disks.

In the following we outline general design issues for AlPHA. We have no detailed design yet for two reasons. On the one hand, Andreas Crauser, the author of LEDA-SM will soon leave our group and we are still looking for an new researcher working full time on the project. On the other hand, we have made contacts with other groups working on external memory libraries (e.g., TPIE by Duke University, Durham NC) and see the possibility for a cooperation that leads to a widely accepted system. In this situation, pushing a detailed design from our side would be counterproductive.

Figure 1 outlines the layer structure of AlPHA. Subsequent sections discuss the layers one by one. Section 3 explains why different more or less portable options for the operating system platform make a thin compatibility layer desirable that shields the rest of the system from operating system dependencies. Our main focus of software development will first be a low level abstraction of external memory that nevertheless already allows to transparently integrate nontrivial disk scheduling algorithms. Sections 4 and 5 give details. A lot of higher level reusable software could be incorporated in a way compatible to the C++ standard template library as explained in Section 6. But the functionality of existing libraries should also be available to application programs by integrating them into the system. Section 7 summarizes the main design issues and outlines challenges for work farther in the future.

*Max-Planck-Institute for Computer Science, Im Stadtwald, 66123 Saarbrücken, Germany, `sanders@mpi-sb.mpg.de`.

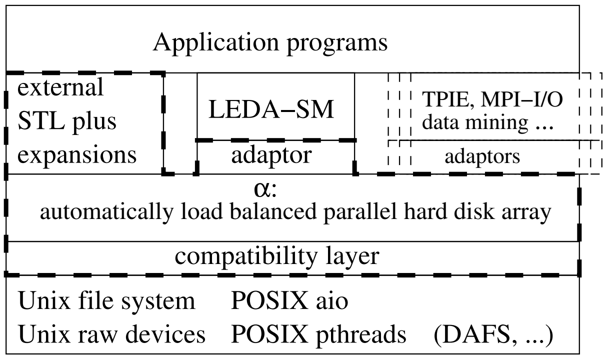

Figure 1: Layers of AlPHA (surrounded by fat dashed line) and other components for processing large volumes of data.

2 Experiences with LEDA-SM

We have completed the single disk external memory library LEDA-SM (Library of Efficient Data types and Algorithms - Secondary Memory) [3] that can serve as the basis for our further work. LEDA-SM consists of two main parts.

The lower level part is a block processing engine that provides us with the external memory abstraction of the model of Vitter and Shriver [15] including a memory manager. This engine works on either Unix file systems or Unix raw devices. Since there are functions in Unix that work with both sources of data, this flexibility comes at relatively low cost. Although such a block engine is a rather simple device, it has important advantages compared to the traditional file systems. It allows comfortable use of the high bandwidth offered by raw devices. Furthermore, previous systems like TPIE [14] open one file for each data stream. This leads to problems with the maximum allowed number of open files in algorithms such as sorting or buffer trees [1] that work on huge number of data streams concurrently. LEDA-SM supports parallel disks to the limited (since suboptimal) extend that one can use a file system that stripes its blocks over several disks.

The higher level of LEDA-SM implements data structures and algorithms using similar abstractions as the internal memory library LEDA [8]. Besides simple data structure like stacks, queues, and arrays with different caching strategies there are more sophisticated data structures like priority queues, B-trees, buffer trees, and and suffix arrays. There are also algorithms for sorting, matrix arithmetics and some simple graph algorithms. To our best knowledge, no previous library [14, 2] implements such a wide spectrum of sophisticated algorithms in one consistent framework.

3 Interfacing with the Operation System

On the one hand, AlPHA should be able to work on a wide range of systems. On the other hand, it should be able to take advantage of emerging standards for high performance I/O. The following decisions are likely to achieve this goal:

- We will provide a thin compatibility layer that shields the rest of the library from the rest of the hardware.

- We will restrict the first implementations to Unix but may make some effort to allow a later port to Windows.

- Basic Unix is not flexible enough to allow independent parallel disk access within a user thread. But there are two mechanisms in the POSIX standard that allow it — threads and asynchronous I/O. Either mechanism alone already allows sufficient functionality for a useful implementation. It is likely that eventually a good implementation uses both mechanisms together.

- In the basic implementation, we will adopt the approach from LEDA-SM to use only functions that are available both for the file system and for raw devices. Our support for raw devices may be more efficient however since asynchronous I/O and multithreading can dissolve some of the disadvantages of raw devices.

- High-end systems for external memory computing are increasingly composed of networks of servers and storage devices. DAFS[1] seems to be an emerging standard for supporting such systems. Hence, we consider to design the system in such a way that a migration to DAFS is possible.

4 Support For Parallel Disks

AlPHA should support a number of useful algorithms that have been developed recently and support a shared memory view on distributed hardware containing multiple independent disks: An external memory segment consists of a homogeneous virtual address space (64 bit). An arbitrary number of concurrent accesses to arbitrary pieces of the memory can be made as long as the pieces have starting addresses and lengths that are a multiple of some minimal blocks size (e.g., 512 byte for the disk sector, or 8192 for a virtual memory page size). Physically, this address space is distributed over all the disks using random placement of data and (optionally) redundant storage. The accesses can be efficiently supported using scheduling algorithms that exploit redundancy [11, 9] or lookahead [5, 4]. A prototype library may support only a single segment. Later versions could also offer multiple dynamically growing segments with different allocation strategies for each segment.

Additionally, there is a cache that avoids many disk accesses. We plan to support scheduling algorithms for integrated prefetching and caching [7, 6] and user configured replacement strategies.

5 Support for Parallelism

AlPHA will be thread safe to the extent possible so that parallel applications on shared memory systems can be implemented. The compatibility layer working with the standard Unix file system or raw devices will use threads or asynchronous I/O for implementing

[1] `http://www.dafscollaborative.org/`

parallel disk access. Later versions may implement parallel shared memory algorithms for basic tasks such as sorting. Thanks to the shared memory abstraction, this can be done transparently without changes of outside interfaces.

Support for distributed memory parallel machines is a more difficult issue. Later versions could provide different levels of support:

- No special support. Every processor (or SMP node) runs a local incarnation of the library on its own local disks. Other libraries such as MPI [12] are used to organize processor interaction.

- The parallel disk support implements private and shared segments that are distributed over all the disks of the system. The application processes remain responsible for coordinating accesses to shared segments of this memory.

- Distributed segments with locking of data and mechanisms to coordinate work between processors. This would replicate functionality of other libraries such as MPI but has the advantage to provide one seamless programming environment for parallel processing with large volumes of data.

6 Compatibility with the STL

The C++ standard template library provides simple algorithms and data structures and an elegent and efficient interface to the file system and internal memory data structures. Of particular importance are *streams* and *iterators*. Streams provide uniform access to files and sequential data structures. By providing a stream interface for AlPHA, applications can freely switch between internal representations, files, and our high performance implementation depending on the problem sizes and machines used. Similar considerations hold for iterators that add random access and hence allow a wider range of applications.

Functionality that can be supported over the interface of the STL are arrays, stacks, queues, priority queues, search trees, hash tables, sorting, median finding, random permutations, strings.

Some data structures will be augmented with additional functions that make more efficient external memory implementations possible. Usually this involves bulk updates like inserting many elements into a search tree, Or performing a large batch of array accesses. (The gather and scatter operations known from vector computers).

7 Conclusions

The current design of AlPHA already anticipates the coexistence of an STL-like library and LEDA-SM on top of the basic support functionality for parallel disks. Similarly, other libraries might be added, for example, for data mining, relational data base like functionality, or geometry or particular applications, e.g., from bioinformatics.

Our main focus for the near future is on the following goals that have not been completely met by previous systems:

- Effective support for parallel disks. This is important because disks are cheap and allow an external memory throughput comparable to the internal memory bandwidth. Such balanced systems are challenging for external memory libraries and algorithms since both external memory access and internal work should be efficient.

- For the balanced systems mentioned above it is also important to support overlapping of internal work and I/O.

- Interfaces to the hardware that can be adapted to emerging technologies such as DAFS.

- Interfaces to applications that are consistent with industry standards such as STL [13].

- Support for external data structures persisting over several program runs using file names. This can be implemented by storing the low volume data that organizes external memory data into ordinary files.

A challenge for the future is support for multiple processors.

References

[1] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *4th WADS*, number 955 in LNCS, pages 334–345. Springer, 1995.

[2] Alex Colvin and Thomas H. Cormen. ViC*: A Compiler for Virtual-Memory C*. Technical Report PCS-TR97-323, Dartmouth College, Computer Science, Hanover, NH, November 1997.

[3] Andreas Crauser. External memory algorithms and data structures in theory and practice. Technical Report ALCOMFT-TR-01-18, MPI-Informatik, 2001. PhD Thesis, Universität des Saarlandes.

[4] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with applications to integrated caching and prefetching and to external sorting. In *9th European Symposium on Algorithms (ESA)*, LNCS. Springer, 2001. to appear, also report ALCOMFT-TR-01-79.

[5] M. Kallahalla and P. J.Varman. Optimal read-once parallel disk scheduling. In *IOPADS*, pages 68–77, 1999.

[6] M. Kallahalla and P.J. Varman. Optimal prefetching and caching for parallel I/O systems. In *ACM Symposium on Parallel Architectures and Algorithms*, 2001. To appear.

[7] Tracy Kimbrel and Anna R. Karlin. Near-optimal parallel prefetching and caching. *SIAM Journal on Computing*, 29(4):1051–1082, 2000.

[8] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[9] P. Sanders. Asynchronous scheduling of redundant disk array. In *12th ACM Symposium on Parallel Algorithms and Architectures*, pages 89–98, 2000. also report ALCOMFT-TR-01-99.

[10] P. Sanders. Reconciling simplicity and realism in parallel disk models. In *12th ACM-SIAM Symposium on Discrete Algorithms*, pages 67–76, Washington DC, 2001. also report ALCOMFT-TR-01-82.

[11] P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *11th ACM-SIAM Symposium on Discrete Algorithms*, pages 849–858, 2000.

[12] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI – the Complete Reference*. MIT Press, 1996.

[13] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 3rd edition, 1998.

[14] D. E. Vengroff. *TPIE User Manual and Reference*. Duke University, 1995. `http://www.cs.duke.edu/~dev/tpie_home_page.html`.

[15] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two level memories. *Algorithmica*, 12(2–3):110–147, 1994.