Implementing I/O-Efficient Data Structures Using TPIE

Lars Arge*, Octavian Procopiuc**, and Jeffrey Scott Vitter***

Center for Geometric and Biological Computing,
Department of Computer Science, Duke University, Durham, NC 27708, USA.
{large,tavi,jsv}@cs.duke.edu

Abstract. In recent years, many theoretically I/O-efficient algorithms and data structures have been developed. The TPIE project at Duke University was started to investigate the practical importance of these theoretical results. The goal of this ongoing project is to provide a *portable*, *extensible*, *flexible*, and *easy to use* C++ programming environment for *efficiently* implementing I/O-algorithms and data structures. The TPIE library has been developed in two phases. The first phase focused on supporting algorithms with a *sequential* I/O pattern, while the recently developed second phase has focused on supporting on-line I/O-efficient data structures, which exhibit a more *random* I/O pattern. This paper describes the design and implementation of the second phase of TPIE.

1 Introduction

In many modern massive dataset applications I/O-communication between fast internal memory and slow disks, rather than actual internal computation time, is the bottleneck in the computation. Examples of such applications can be found in a wide range of domains such as scientific computing, geographic information systems, computer graphics, and database systems. As a result, much attention has been focused on the development of I/O-efficient algorithms and data structures (see e.g. [4, 20]). While a lot of practical and often heuristic I/O-efficient algorithms and data structures in ad-hoc models have been developed in the database community, most theoretical work on I/O-efficiency in the algorithms community has been done in the Parallel Disk Model of Vitter and Shriver [21]. To investigate the practical viability of the theoretical work, the TPIE[1] project was started at Duke University. The goal of this ongoing project is to provide a *portable*, *extensible*, *flexible*, and *easy to use* programming environment for *efficiently* implementing algorithms and data structures developed for the Parallel Disk

* Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879, CAREER grant CCR–9984099, ITR grant EIA–0112849, and U.S.-Germany Cooperative Research Program grant INT–0129182.

** Supported in part by the National Science Foundation through ESS grant EIA–9870734 and RI grant EIA–9972879.

*** Supported in part by the National Science Foundation through research grants CCR–9877133 and EIA–9870734 and by the Army Research Office through MURI grant DAAH04–96–1–0013.

[1] TPIE: Transparent Parallel I/O Environment. Pronunciation: 'tE-'pI (like tea-pie)

Model. A project with similar goals, called LEDA-SM [13] (an extension of the LEDA library [15]), has also been conducted at the Max-Planck Institut in Saarbrucken.

TPIE is a templated C++ library[2] consisting of a kernel and a set of I/O-efficient algorithms and data structures implemented on top of it. The kernel is responsible for abstracting away the details of the transfers between disk and memory, managing data on disk and in main memory, and providing a unified programming interface appearing as the Parallel Disk Model. Each of these tasks is performed by a separate module inside the kernel, resulting in a highly extensible and portable system. The TPIE library has been developed in two phases. The first phase focused on supporting algorithms with a *sequential* I/O pattern, that is, algorithms using primitives such as scanning, sorting, merging, permuting, and distributing [19]. However, in recent years, a growing number of on-line I/O-efficient data structures have been developed, and the sequential framework is ill-equipped to handle the more *random* I/O patterns exhibited in these structures. Therefore a second phase of the TPIE development has focused on providing support for random access I/O patterns. Furthermore, just like a large number of batched algorithms were implemented in the first phase of the project, a large number of data structures have been implemented in the second phase, including B-trees [12], persistent B-trees [9], R-trees [10], CRB-trees [1], K-D-B-trees [18] and Bkd-trees [17]. The two parts of TPIE are highly integrated, allowing seamless implementation of algorithms and data structures that make use of both random and sequential access patterns.[3]

While the first part of TPIE has been described in a previous paper [19], this paper describes the design and implementation details of the second phase of the system. In Section 2 the Parallel Disk Model, as well as disk and operation technology that influenced the TPIE design choices are first described. Section 3 then describes the architecture of the TPIE kernel and its design goals. Section 4 presents a brief description of the data structures implemented using the random access framework, and finally Section 5 contains a case study on the implementation of the K-D-B-tree [18] along with some experimental results.

2 The I/O Model of Computation

In this section we discuss the physical disk technology motivating the Parallel Disk Model, as well as some operating system issues that influenced important TPIE design choices.

Magnetic Disks. The most common external memory storage device is the magnetic disk. A magnetic disk drive consists of one or more rotating platters with one read/write head per platter. Data are stored on the platter surface in concentric circles called tracks. To read or write data at a certain position on the platter, the read/write head must *seek*

[2] The latest TPIE release can be downloaded from `http://www.cs.duke.edu/TPIE/`

[3] The LEDA-SM library takes a slightly different approach, optimized for random access patterns. The algorithms and data structures implemented using LEDA-SM are somewhat complementary to the ones implemented in TPIE. The kernels of the two libraries are compatible, and as a result algorithms and structures implemented for one of the systems can easily be ported to the other.

to the correct track and then *wait* for the desired position on the track to pass by. Because mechanical movement is involved, the typical read or write time is on the order of milliseconds. By comparison, the typical transfer time of main memory is a few nanoseconds—a factor of 10^6 faster! Since the seek and wait time is much larger than the time needed to read a unit of data, magnetic disks transfer a large *block* of contiguous data items at a time. Accessing a block involves only one seek and wait, so the amortized cost per unit of data is much smaller than the cost of accessing a single unit.

Parallel Disk Model. The Parallel Disk Model (PDM) was introduced by Vitter and Shriver [21] (see also [3]) in order to more accurately model a two-level main memory-disk system with block transfers. PDM has become the standard theoretical model for designing and analyzing I/O-efficient algorithms. The model abstracts a computer as a three-component system: a processor, a fixed amount of main memory, and one or more independent disk drives. Data is transferred between disks and main memory in fixed-size *blocks* and one such transfer is called an *I/O operation* (or simply an *I/O*). The primary measures of algorithm performance in the model are the number of I/Os performed, the amount of disk space used, and the internal computation time. To be able to quantify these measures, N is normally used to denote the number of items in the problem instance, M the maximum number of items that fit in main memory, B the number of items per block, and D the number of independent disks. In this paper we only consider the one-disk model.

Random versus sequential I/O. Using the Parallel Disk Model, a plethora of theoretically I/O-efficient algorithms and data structures have been developed—refer to [4, 20] for surveys. Experimental studies (many using the TPIE system) have shown the model's accuracy at predicting the relative performance of algorithms—refer to the above mentioned surveys for references. However, they have also revealed the limits of the model, primarily its inability to distinguish between the complexities of sequential and random I/O patterns. Intuitively, accessing data sequentially is more efficient than accessing blocks in a random way on disk, since the first pattern leads to less seeks and waits than the latter. Furthermore, since studies have shown that the typical use of a disk file is to open it, read its entire contents sequentially, and close it, most operating systems are optimized for such sequential access.

UNIX I/O primitives. In the UNIX operating system, optimization for sequential access is implemented using a *buffer cache*. It consists of a portion of the main memory reserved for caching data blocks from disk. More specifically, when a user requests a data block from disk using the `read()` system call, the block is looked up in the buffer cache, and if not there, it is fetched from disk into the cache. From there, the block is copied into a user-space memory location. To optimize for sequential access a prefetching strategy is also implemented, such that blocks following a recently accessed block are loaded into the buffer cache while computation is performed. Similarly, when a block is written using a `write()` system call, the block is first copied to the buffer cache, and if necessary, a block from the buffer cache is written to disk.

When the I/O-pattern is random rather than sequential, the buffer cache is mostly useless and can actually have a detrimental effect. Not only are resources wasted on caching and prefetching, but the cache also incurs an extra copy of each data block.

Therefore most UNIX-based operating systems offer alternative I/O routines, called `mmap()` and `munmap()`, which avoid using the buffer cache. When the user requests a disk block using `mmap()`, the block is mapped directly in user-space memory.[4] The mapping is released when `munmap()` is called, allowing the block to be written back to disk. If properly implemented, these routines achieve a zero-copy I/O transfer, resulting in more efficient I/O than the `read()`/`write()` functions in applications that exhibit a random I/O access pattern. Another important difference between the two sets of functions is that in order to achieve zero-copy transfer, the `mmap()`/`munmap()` functions control which user-space location a block is mapped into. In the `read()`/`write()` case, where a copy is incurred anyway, it is the application that controls the placement of blocks in user-space. As described in the next section, all the above issues have influenced the design of the random access part of TPIE.

3 The TPIE Kernel

In this section we describe in some detail the architecture of the TPIE kernel and the main goals we tried to achieve when designing it. The kernel, as well as the rest of the TPIE library, are written in C++. We assume the reader is familiar with object-oriented and C++ terminology, like classes, templates, constructors and destructors.

3.1 Overview

As mentioned in the introduction, the TPIE library has been built in two phases. The first phase was initially developed for algorithms based on sequential scanning, like sorting, permuting, merging, and distributing. In these algorithms, the computation can be viewed as a continuous process in which data is fed in *streams* from an outside source and streams of results are written behind. This stream-based view of I/O computation is not utilizing the full power of the parallel disk model, but it provides a layer of abstraction that frees the developer from worrying about details like managing disk blocks and scheduling I/Os.

The TPIE kernel designed in this first phase consists of three modules: the *Stream-based Block Transfer Engine (BTE)*, responsible for packaging data into blocks and performing I/O transfers, the *Memory Manager (MM)*, responsible for managing main memory resources, and the *Application Method Interface (AMI)*, which provides the public interface and various tools. The BTE implements a stream using a UNIX file and its functionality, like reading, writing, or creating a block, is implemented using UNIX I/O calls. Since the performance of these calls is paramount to the performance of the entire application, different BTE implementations are provided, using different UNIX I/O calls (see Figure 1): `BTE_stream_mmap` uses `mmap()` and `munmap()`, `BTE_stream_ufs` uses the `read()` and `write()` system calls, and `BTE_stream_stdio` uses `fread()` and `fwrite()`. Other implementations can easily be added without affecting other parts of the kernel. The Memory Manager (MM)

[4] Some systems use the buffer cache to implement `mmap()`, mapping pages from the buffer cache into user-space memory.

module maintains a pool of main memory of given size M and insures that this size is not exceeded. When either a TPIE library component or the application makes a memory allocation request, the Memory Manager reserves the amount requested and decreases a global counter keeping track of the available memory. An error is returned if no more main memory is available. Finally, the AMI provides a high-level interface to the stream functionality provided by the BTE, as well as various tools, including templated functions for scanning, merging, and sorting streams. In addition, the AMI provides tools for testing and benchmarking: a tool for logging errors and debugging messages and a mechanism for reporting statistics.

As mentioned, the stream-based view of I/O provided by these modules is ill-equipped for implementing on-line I/O-efficient data structures. The second phase of TPIE provides support for implementing these structures by using the full power of the disk model. Maintaining the design framework presented above, the new functionality is implemented using a new module, the Random-access BTE, as well as a new set of AMI tools. Figure 1 depicts the interactions between the various components of the TPIE kernel. The rest of this section describes the implementation of the Random-access BTE module and the new AMI tools.

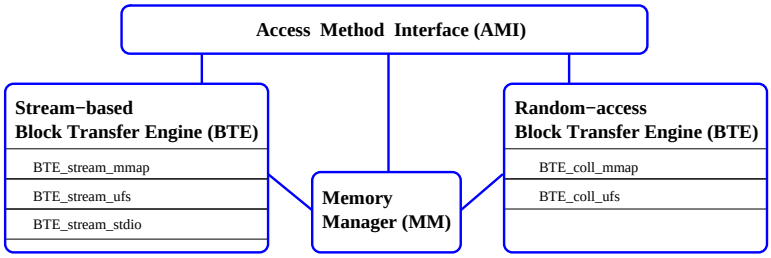


Fig. 1. The structure of the TPIE kernel.

3.2 The Random Access Block Transfer Engine (BTE).

The Random-access BTE implements the functionality of a *block collection*, which is a set of fixed-size blocks. A block can be viewed as being in one of two states: on disk or in memory. To change the state of a block, the block collection should support two operations: *read*, which loads a block from disk to memory, and *write*, which stores an in-memory block to disk. In addition, the block collection should be able to *create* a new block and *delete* an existing block. In order to support these operations, a unique *block ID* is assigned to each block in a block collection. When requesting a new block using the create operation, the collection returns a new block ID, which can then be used to read, write, or delete the block.

In our implementation, a block collection is organized as a linear array of blocks stored in a single UNIX file. A block from a collection is uniquely determined by its index in this array—thus this index is used as the block ID. When performing a read or write operation we start by computing the offset of the block in the file using the block

ID. This offset is then used to seek in the file and transfer the requested block. Furthermore, the write operation uses a *dirty flag* to avoid writing blocks that have not been modified since they were read. This per-block dirty flag should be set by the user-level application whenever the contents of the in-memory block are modified. Unlike read and write, the create and delete operations modify the size of the block collection. To implement these operations, we employ a stack storing the IDs of the blocks previously deleted from the collection. When a block is deleted, its ID is simply pushed onto this stack. When a new block is requested by the create procedure, the stack is first checked, and if it's not empty, the top ID is popped and returned; if the stack is empty, the block ID corresponding to a new block at the end of the file is returned. The use of the stack avoids costly reorganization of the collection during each delete operation. However, it brings up a number of issues that need to be addressed. First, the stack has to reside on disk and has to be carried along with the file storing the blocks. In other words, a collection consists of two files: one containing data blocks and one containing block IDs. The second issue concerns space overhead. When multiple create and delete operations are performed on a collection, the number of blocks stored in the data blocks file can be much larger than the number of blocks in the collection. To eliminate this space overhead, the collection can be reorganized. However, such a reorganization would change the IDs of some blocks in the collection and therefore it cannot be performed by the BTE, which has no knowledge of the contents of the blocks: If a block contains IDs of other blocks, then the contents of that block would need to be updated as well. A reorganization procedure, if needed, should therefore be implemented on the application level.

We decided to implement the block collection on top of the UNIX file system and not, e.g., on the raw disk, to obtain portability and ease of use. Using raw disk I/O would involve creating a separate disk partition dedicated to TPIE data files, a non-trivial process that usually requires administrator privileges. Some operating systems offer user-space raw I/O, but this mechanism is not standardized and is offered only by a few operating systems (such as Solaris and Linux). On the other hand, storing a block collection in a file allows the use of existing file utilities to copy and archive collections.

A BTE block collection is implemented as a C++ class that provides a standard interface to the AMI module. The interface consists of a constructor for opening a collection, a destructor for closing the collection, the four block-handling routines described above, as well as other methods for reporting the size of the collection, error handling, etc. Except for the four block-handling routines, all methods are implemented in a base class. We implemented two BTE collections as extensions to this base class: `BTE_coll_mmap` and `BTE_coll_ufs`. As the names suggest, `BTE_coll_mmap` uses `mmap()` and `munmap()` to perform I/O transfers, while `BTE_coll_ufs` uses the `read()` and `write()` system calls. The implementation of choice for most systems is `BTE_coll_mmap` since, as mentioned in Section 2, the `mmap()/munmap()` functions are more suited for the random I/O pattern exhibited by online algorithms. We implemented `BTE_coll_ufs` to compare its performance with `BTE_coll_mmap` and to account for some systems where `mmap()` and `munmap()` are very slow.

3.3 The Access Method Interface (AMI).

The AMI tools needed to provide the random access I/O functionality consist of a front-end to the BTE block collection and a typed view of a disk block. In the BTE, we viewed the disk block as a fixed-size sequence of "raw" bytes. However, when implementing external memory data structures, disk blocks often have a well-defined internal structure. For example, a disk block storing an internal node of a B^+-tree contains the following: an array of b pointers to child nodes, an array of $b-1$ keys, and a few extra bytes for storing the value of b. Therefore, the AMI contains a templated C++ class called `AMI_block<E,I>`. The contents of a block are partitioned into three fields: an array of zero or more links to other blocks (i.e., block IDs), an array of zero or more elements of type `E` (given as a template parameter), and an *info* field of type `I` (also given as a template parameter). Each link is of type `AMI_bid` and represents the ID of another block in the same collection. This way the structure of a block is uniquely determined by three parameters: the types `E` and `I` and the number of links. Easy access to elements and links is provided by simple array operators. For example, the ith element of a block b is referenced by `b.el[i]`, and the jth link is referenced by `b.lk[j]`.

The `AMI_block<E,I>` class is more than just a structuring mechanism for the contents of a block. It represents the in-memory image of a block. To this end, constructing an `AMI_block<E,I>` object loads the block in memory, and destroying the object unloads it from memory. The most general form of the constructor is as follows.

```
AMI_block<E,I>(AMI_collection* c, size_t links, AMI_bid bid=0)
```

When a non-zero block ID is passed to the constructor, the block with that ID is loaded from the given collection. When the block ID is zero, a new block is created in the collection. When deleting an `AMI_block<E,I>` object, the block is deleted from the collection or written back to disk, depending on a *persistence flag*. By default, a block is *persistent*, meaning that it is kept in the collection after the in-memory image has been destroyed. The persistence flag can be changed for individual blocks.

As its name suggests, the `AMI_collection` type that appears in the above constructor represents the AMI interface to a block collection. The functionality of the `AMI_collection` class is minimal, since the block operations are handled by the `AMI_block<E,I>` class. The main operations are open and close, and are performed by the constructor and destructor. An instance of type `AMI_collection` is constructed by providing a file name, an access type, and a block size.

```
AMI_collection(char* fn, AMI_collection_type ct, size_t bl_sz)
```

This constructor either opens an existing collection or creates and opens a new one. The destructor closes the collection and, if so instructed, deletes the collection from disk. The exact behavior is again determined by a *persistence flag*, which can be set before calling the destructor.

3.4 Design Goals

This subsection summarizes the main goals we had in mind when designing the TPIE kernel and the methods we used to achieve these goals.

Ease of use. The usability of the kernel relies on its intuitive and powerful interface. We started from the parallel disk model and built the Random-access BTE module to simulate it. Multiple BTE implementations exist and they are easily interchangeable, allowing an application to use alternative low-level I/O routines. The user interface, provided within the AMI, consists of a typed view of a block—the `AMI_block<E,I>` class—and a front-end to the block collection—the `AMI_collection` class. The design of these two classes provides an easy to understand, yet powerful application interface. In addition, the `AMI_block<E,I>` class provides structure to the contents of the block in order to facilitate the implementation of external memory data structure.

Flexibility. As illustrated in Figure 1, the TPIE kernel is composed of four modules with a well-defined interface. Each of the modules has at least a default implementation, but alternative implementations can be provided. The best candidates for alternative implementations are the two BTE modules, since they allow the use of different I/O mechanisms. The Stream-based BTE has three implementations, using different system calls for performing the I/O in stream operations. The Random-access BTE has two implementations, which use different low-level system calls to perform the I/O in block collection operations.

Efficiency. In order to obtain a fast library, we paid close attention to *optimizing disk access*, *minimizing CPU operations*, and *avoiding unnecessary in-memory data movement*. To optimize disk access, we used a per-block dirty flag that indicates whether the block needs to be written back or not. To minimize CPU operations, we used templated classes with no virtual functions; because of their inherently dynamic nature, virtual functions are not typically inlined by C++ compilers and have a relatively high function call overhead. Finally, to avoid in-memory copying, we used the `mmap()/munmap()` I/O system calls; they typically transfer blocks of data directly between disk and user-space memory.

Portability. Being able to easily port the TPIE kernel on various platforms was one of our main goals. As discussed in Section 3.1, the default methods used for performing the disk I/O are those provided by the UNIX-based operating systems and were chosen for maximum portability. Alternative methods can be added, but the existing implementations insure that the library works on all UNIX-based platforms.

4 Data Structures

As mentioned in the introduction, there are various external memory data structures implemented using the TPIE kernel. They are all part of the extended TPIE library. In this section, we briefly survey these data structures.

B-tree. The B-tree [12] is the classical external memory data structure for online searching. In TPIE we implemented the more general (a, b)-tree [14], supporting insertion, deletion, point query, range query, and bulk loading.[5] All these operations are encapsulated in a templated C++ class. The template parameters allow the user to choose the

[5] Bulk loading is a term used in the database literature to refer to constructing an index from a given data set from scratch.

type of data items to be indexed, the key type, and the key comparison function. A full description of the (a, b)-tree implementation will be given in the full version of this paper.

Persistent B-tree. The persistent B-tree [9] is a generalization of the B-tree that records all changes to the initial structure over a series of updates, allowing queries to be answered not only on the current structure, but on any of the previous ones as well. The persistent B-tree can be used to answer 3-sided range queries and vertical ray shooting queries on segments in $\mathbb{R}^2$. More details on the implementation can be found in [5].

R-tree. The R-tree and its variants are widely used indexing data structures for spatial data. The TPIE implementation uses the insertion heuristics proposed by Beckmann et al. [10] (their variant is called the R*-tree) and various bulk loading procedures. More details are given in [6, 7].

Logarithmic method. The logarithmic method [16] is a generic dynamization method. Given a static index with certain properties, it produces a dynamic structure consisting of a set of smaller static indexes of geometrically increasing sizes. We implemented the external memory versions of this method, as proposed by Arge and Vahrenhold [8] and Agarwal et al. [2]. More details on the implementation can be found in [17].

K-D-B-tree. The K-D-B-tree [18] combines the properties of the kd-tree [11] and the B-tree to handle multidimensional points in an external memory setting. Our implementation supports insertion, deletion, point query, range query and bulk loading. More details on the implementation can be found in [17].

Bkd-tree. The Bkd-tree [17] is a data structure for indexing multidimensional points. It uses the kd-tree [11] and the logarithmic method to provide good worst-case guarantees for the update and query operations. More details can be found in [17].

5 Case Study: Implementing the K-D-B-tree

We conclude this paper with some details of the K-D-B-tree implementation in order to illustrate how to implement a data structure using TPIE. We chose the K-D-B-tree because it is a relatively simple yet typical example of a tree-based structure implementation.

The K-D-B-tree is a data structure for indexing multidimensional points that attempts to combine the query performance of the kd-tree with the update performance of the B-tree. More precisely, a K-D-B-tree is a multi-way tree with all leaves on the same level. In two dimensions, each internal node v corresponds to a rectangular region r and the children of v define a disjoint partition of r obtained by recursively splitting r using axis-parallel lines (similar to the kd-tree [11] partitioning scheme). The points are stored in the leaves of the tree, and each leaf or internal node is stored in one disk block.

The implementation of the K-D-B-tree is parameterized on the type c used for the point coordinates and on the dimension of the space d.

```
template<class c, size_t d> class Kdbtree;
```

The K-D-B-tree is stored in two block collections: one for the (internal) nodes, and one for the leaves. Using two collections to store the K-D-B-tree allows us to choose the block size of nodes and that of leaves *independently*; it also allows us to have the nodes clustered on disk, for improved performance.

By the flexible design of the `AMI_block` class, we can simply extend it and use the appropriate template parameters in order to provide the required structure for nodes and leaves.

```
template<class c, size_t d>
class Kdbtree_node: AMI_block<box<c, d>, kdbtree_node_info>;
template<class c, size_t d>
class Kdbtree_leaf: AMI_block<point<c, d>, kdbtree_leaf_info>;
```

In other words, a `Kdbtree_node<c,d>` object consists of an array of d-dimensional boxes of type `box<c,d>`, an array of links pointing to the children of the node, and an info element of type `kdbtree_node_info`. The info element stores the actual fanout of the node (which is equal to the number of boxes stored), the weight of the node (i.e., the number of points stored in the subtree rooted at that node), and the splitting dimension (a parameter used by the insertion procedure, as described in [18]). The maximum fanout of a node is computed (by the `AMI_block` class) using the size of the `box<c,d>` class and the size of the block, which is a parameter of the nodes block collection. A `Kdbtree_leaf<c,d>` object consists of an array of d-dimensional points of type `point<c,d>`, no links, and an info element of type `kdbtree_leaf_info` storing the number of points, a pointer to another leaf (for threading the leaves), and the splitting dimension.

As already mentioned, the operations supported by this implementation of the K-D-B-tree are insertion, deletion, point query, window query, and bulk loading. It has been shown that batched algorithms for bulk loading can be much faster than using repeated insertions [6]. For the K-D-B-tree, we implemented two different bulk loading algorithms, as described in [17]. Both algorithms start by sorting the input points and then proceed to build the tree level by level, in a top down manner. The implementation of these algorithms shows the seamless integration between the stream-handling AMI tools and the block handling AMI tools: The initial sorting is done by the built-in AMI sort function, and the actual building is done by scanning the sorted streams and producing blocks representing nodes and leaves of the K-D-B-tree.

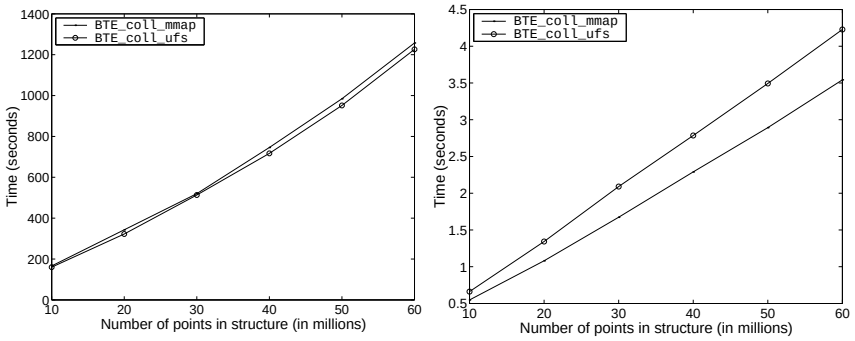
The update operations (insertion and deletion) are implemented by closely following the ideas from [18]. The query operations are performed as in the kd-tree [11]. Figure 2 shows the implementation of the simple point query procedure. Starting from the root, the procedure traverses the path to the leaf that might contain the query point. The traversal is done by iteratively fetching a node using its block ID (line 7), finding the child node containing the query point (line 8), and releasing the node (line 10). When the child node is a leaf, that leaf is fetched (line 12), its contents are searched for the query point (line 13), and then the leaf is released (line 14). These pairings of fetch and release calls are typical examples of how applications use the TPIE kernel to perform I/O. Intuitively, `fetch_node` reads a node from disk and `release_node` writes it back. The point query procedure is oblivious to how the I/O is performed or whether any I/O was performed at all. Indeed, the fetch and release functions employ

```
 1  bool find(point_t& p) {
 2    bool ans;  size_t i;
 3    Kdbtree_node<c,d>* bn;
 4    region_t<c,d> r;
 5    kdb_item_t<c,d> ki(r, header_.root_bid, header_.root_type);
 6    while (ki.type != BLOCK_LEAF) {
 7      bn = fetch_node(ki.bid);
 8      i = bn->find(p);
 9      ki = bn->el[i];
10      release_node(bn);
11    }
12    Kdbtree_leaf<c,d>* bl = fetch_leaf(ki.bid);
13    ans = (bl->find(p) < bl->size());
14    release_leaf(bl);
15    return ans;
16  }
```

Fig. 2. Implementation of the point query procedure.

a *cache manager* to improve I/O performance. By using application-level caching (instead of fixed, kernel-level caching) we allow the application developer to choose the most appropriate caching algorithm. A few caching algorithms are already provided in TPIE, and more can be easily added by extending the cache manager base class.

Experiments. Using the K-D-B-tree implementation, we performed experiments to show how the choice of I/O system calls affects performance. We bulk loaded and performed range queries on K-D-B-trees of various sizes.[6] The data sets consisted of uniformly distributed points in a squared-shaped region. The graph in Figure 3(a) shows the running times of bulk loading, while the graph in Figure 3(b) shows the running time of one range query, averaged over 10 similar-size queries. Each experiment was performed



(a) (b)

Fig. 3. (a) Performance of K-D-B-tree bulk loading (b) Performance of a range query (averaged over 10 queries, each returning 1% of the points in the structure)

using the two existing Random-access BTE implementations: `BTE_coll_mmap` and `BTE_coll_ufs`. As expected, the running time of the bulk loading procedure—a highly sequential process—is not affected by the choice of Random-access BTE. On the other hand, the performance of a range query is affected significantly by this choice:

[6] All experiments were performed on a dedicated Pentium III/500MHz computer running FreeBSD 4.4, with 128MB of main memory and an IBM Ultrastar 36LZX SCSI disk.

Using the ufs-based Random-access BTE results in higher running times. This validates our analysis from Section 2 and confirms that `BTE_coll_mmap` is the implementation of choice for the Random-access BTE.

References

1. P. K. Agarwal, L. Arge, and S. Govindarajan. CRB-tree: An optimal indexing scheme for 2d aggregate queries. Manuscript, 2002.
2. P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. 28th Intl. Colloq. Automata, Languages and Programming (ICALP)*, 2001.
3. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
4. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.
5. L. Arge, A. Danner, and S.-M. Teh. I/O-efficient point location using persistent B-trees. Manuscript, 2002.
6. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1):104–128, 2002.
7. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter. A unified approach for indexed and non-indexed spatial joins. In *Proc. Conference on Extending Database Technology*, pages 413–429, 1999.
8. L. A. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. In *Proc. ACM Symp. Computational Geometry*, 2000.
9. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.
10. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 322–331, 1990.
11. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.
12. D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11:121–137, 1979.
13. A. Crauser and K. Mehlhorn. LEDA-SM: Extending LEDA to secondary memory. In *Proc. Workshop on Algorithm Engineering*, 1999.
14. S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.
15. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, UK, 2000.
16. M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes Comput. Sci.* Springer-Verlag, Heidelberg, West Germany, 1983.
17. O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. Manuscript, 2002.
18. J. T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 10–18, 1981.
19. D. E. Vengroff and J. S. Vitter. Supporting I/O-efficient scientific computation in TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, pages 74–77, 1995.
20. J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.
21. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.