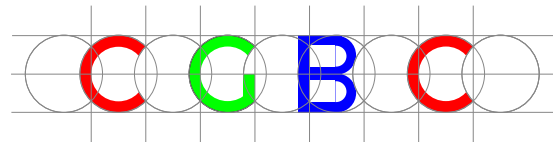
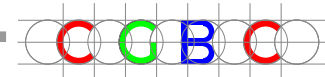


Implementing Data Structures Using TPIE

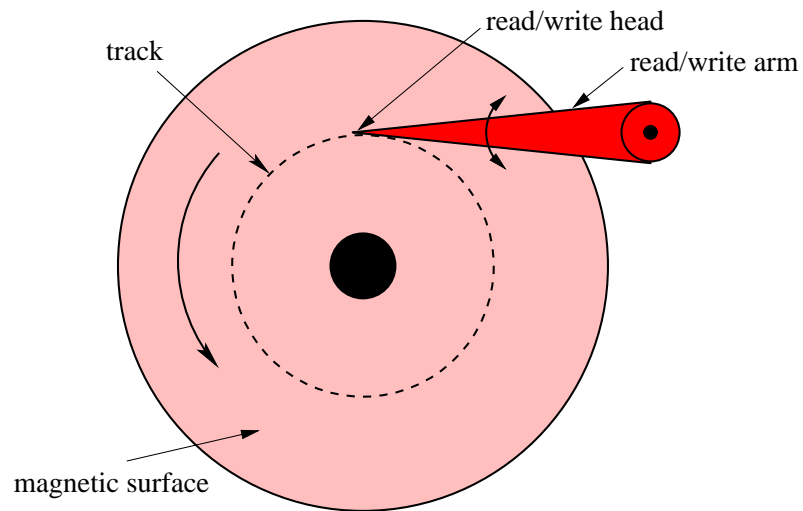
Lars Arge Octavian Procopiuc Jeffrey Scott Vitter



Center for Geometric and Biological Computing
Duke University

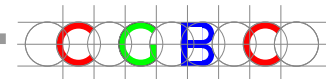


Disk Technology

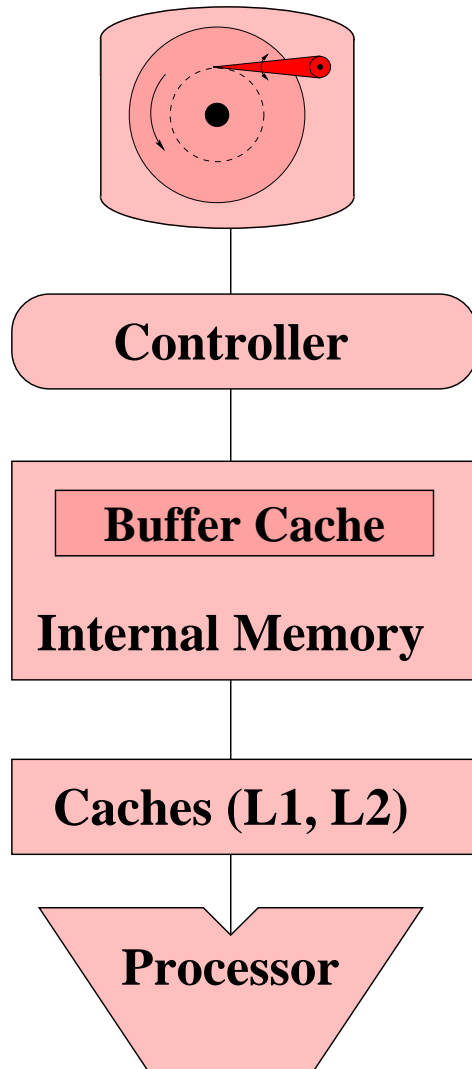


Characteristics:

- ▶ High latency
- ▶ Paged access
- ▶ Sequential access much faster than random access



I/O in a Real Computer



Other factors affecting I/O performance:

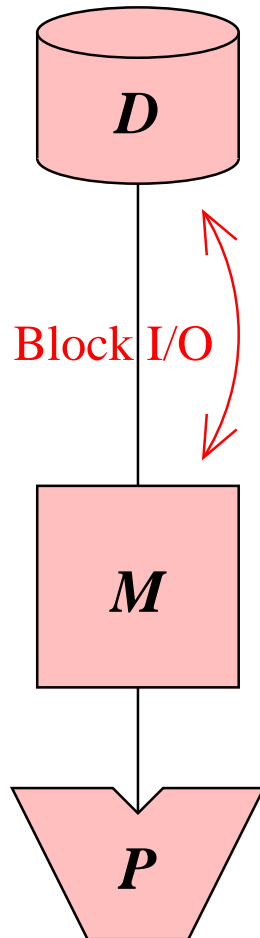
- ▶ Controller cache
- ▶ Operating system

I/O in UNIX-like operating systems:

- ▶ `read()/write()` system calls
(read-ahead, extra copy in buffer cache)
- ▶ `mmap()/munmap()` system calls
(no read-ahead, zero copy)

I/O in the Disk Model

[Aggarwal, Vitter 88]



Parameters:

- N = no. of input points
- M = no. of points that fit in memory
- B = no. of points per disk block

Cost measures:

- no. of I/Os performed
- total disk space used

Advantages: Few parameters, easy to handle.

Disadvantages:

- Unaware of caches
- Unable to differentiate between sequential and random I/Os.

What is TPIE?

TPIE: Transparent Parallel I/O Environment

- ▶ A programming environment for simplifying the implementation of external memory algorithms and data structures.
- ▶ A templated C++ library
(available from <http://www.cs.duke.edu/~tpie/>)

Other Environments

- ▶ LEDA-SM [Crauser, Mehlhorn 99]
Slightly different approach, optimized for random I/O patterns.
- ▶ GiST [Hellerstein, Naughton, Pfeffer 95]
Restricted to certain tree-based structures.

TPIE Goals

- ▶ Functionality goals:
 - Abstract away the details of I/O transfers
 - Manage data on external memory (disk)
 - Provide primitives to serve as building blocks

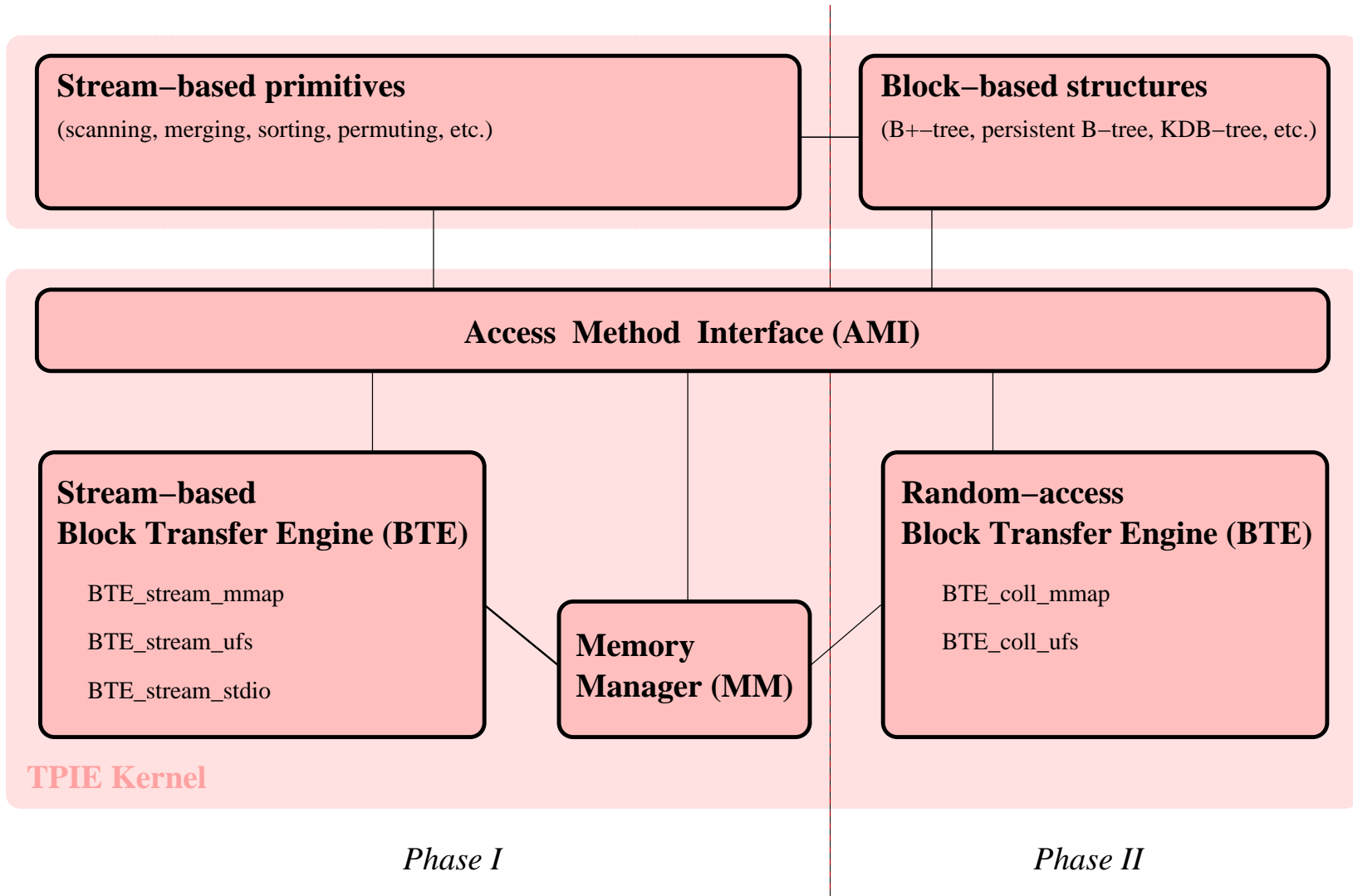
- ▶ Design goals:
 - Ease of use (simple and intuitive interface)
 - Portability (across UNIX platforms)
 - Efficiency (low CPU overhead)
 - Flexibility (modularization)

TPIE Development

TPIE has been developed in 2 phases:

- ▶ **I.** Primitives for solving **batched problems** [Vengroff, Vitter 95].
Ex: scanning, sorting, permuting, merging.
Underlying abstraction: data stream.
- ▶ **II.** Primitives for solving **online problems**.
Ex: external memory data structures.
Underlying abstractions: disk block, block collection.

TPIE Architecture

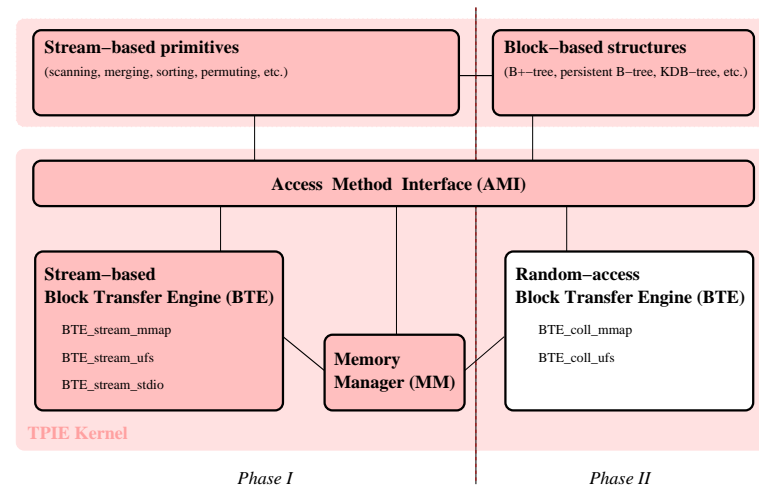


The TPIE Kernel: BTE

Random-Access

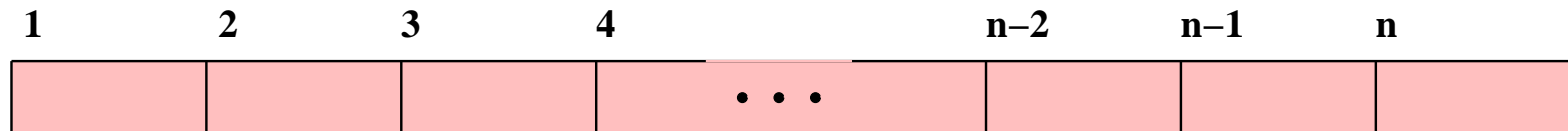
Block Transfer Engine

- ▶ Responsible for delivering and organizing disk blocks.
- ▶ Implements the functionality of a **block collection**.
- ▶ Operations supported:
 - **Read** a block from disk into memory
 - **Write** a block from memory to disk
 - **Create** a new block in collection
 - **Delete** a block from collection



BTE Block Collection

- ▶ Organized as a linear array of blocks



- ▶ The position in this array is used as **block ID**
 - ▶ **Write**: Uses per-block **dirty flag**. If not set, in-memory data is simply discarded.
 - ▶ To insure fast deletes, we use a stack of block IDs
-
- | | | | |
|---|---|-----|--|
| 4 | 2 | ••• | |
|---|---|-----|--|
- ▶ **Delete**: Pushes block ID onto stack.
 - ▶ **Create**: Pops block ID from stack and returns it (if empty, creates new blocks at the end of array).

BTE Block Collection (cont'd)

- ▶ Implementation uses two UNIX files: one for the array of blocks and one for the stack.
- ▶ Multiple implementations provided:
 - `BTE_coll_mmap`: uses `mmap()` and `munmap()` system functions.
 - `BTE_coll_ufs`: uses `read()` and `write()` system functions.
- ▶ Other implementations can be easily provided.

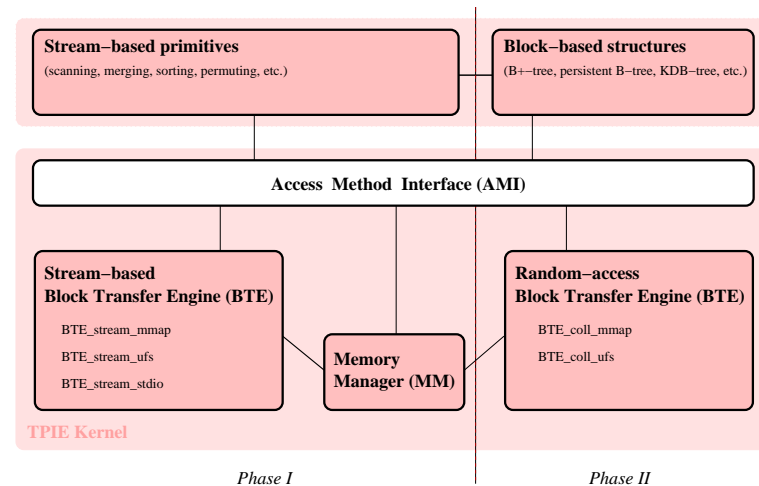
The TPIE Kernel: AMI

Application Method Interface

- ▶ `AMI_collection`: a class for manipulating block collections.
- ▶ `AMI_block<E, I>`: a class for manipulating typed blocks.
- ▶ The format of a typed block:

	Links					Elements				Info
--	--------------	--	--	--	--	-----------------	--	--	--	-------------

 - **Links**: Pointers to other blocks (using block IDs).
 - **Elements**: Keys for discriminating among the links.
 - **Info**: block-wide information.
- ▶ The Info and Element types are given by the template parameters.



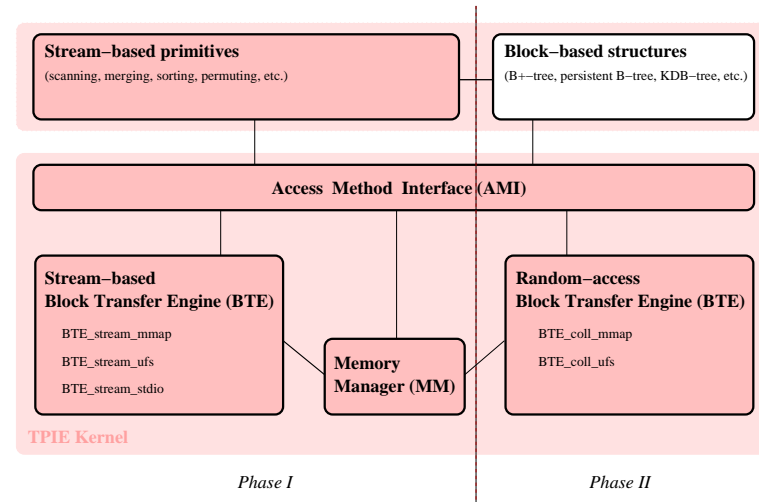
AMI (cont'd)

- ▶ **Reading** or **creating** a block is done by **constructing** an `AMI_block` instance.
- ▶ **Writing** or **deleting** a block is done by **destructing** an `AMI_block` instance.
- ▶ Exact behavior is determined by persistence flag
 - *Persist*: the block is written back to the collection;
 - *Delete*: the block is deleted from the collection.

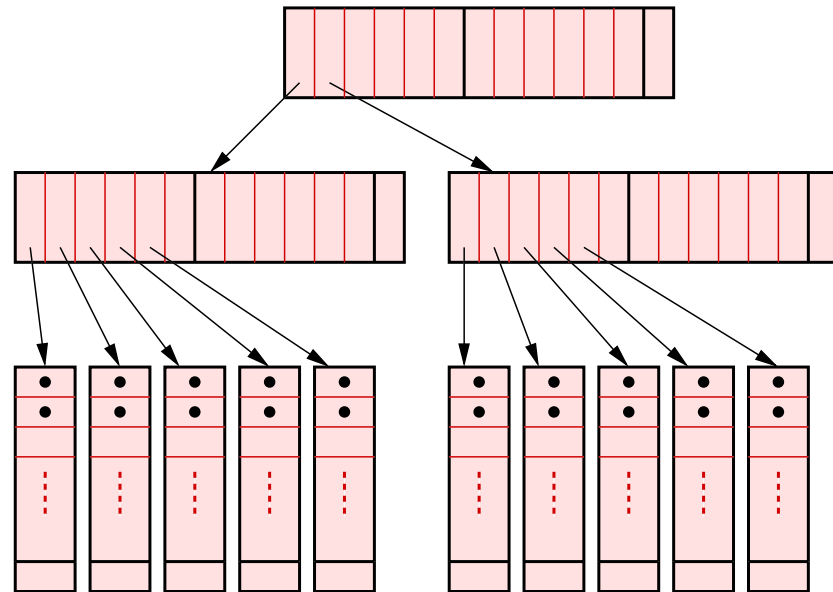
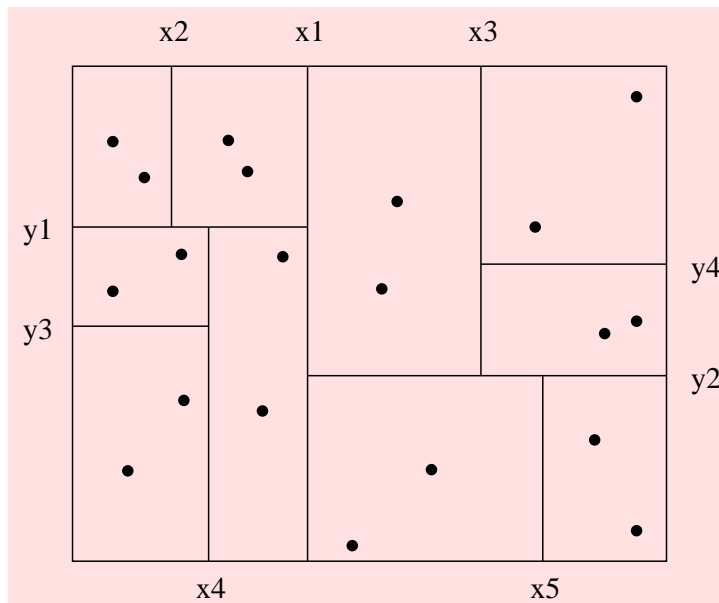
The Extended TPIE Library

- ▶ Techniques for caching blocks in main memory.
- ▶ I/O-efficient data structures.
- ▶ Examples:

- (a, b) -tree (includes B^+ -tree as a special case)
- Persistent B-tree [Arge, Danner, Teh 02]
- Bkd-tree, K-D-B-tree, Logarithmic method [Procopiuc, Arge, Agarwal, Vitter 02]
- R-tree and variants [Arge, Procopiuc, Ramaswamy, Suel, Vahrenhold, Vitter 99]
- EPS-tree [Arge, Procopiuc, Samoladas, Vitter 02]
- CRB-tree [Arge, Agarwal, Govindarajan 02]



Case study: the K-D-B-tree



The K-D-B-tree contains 2 types of blocks:

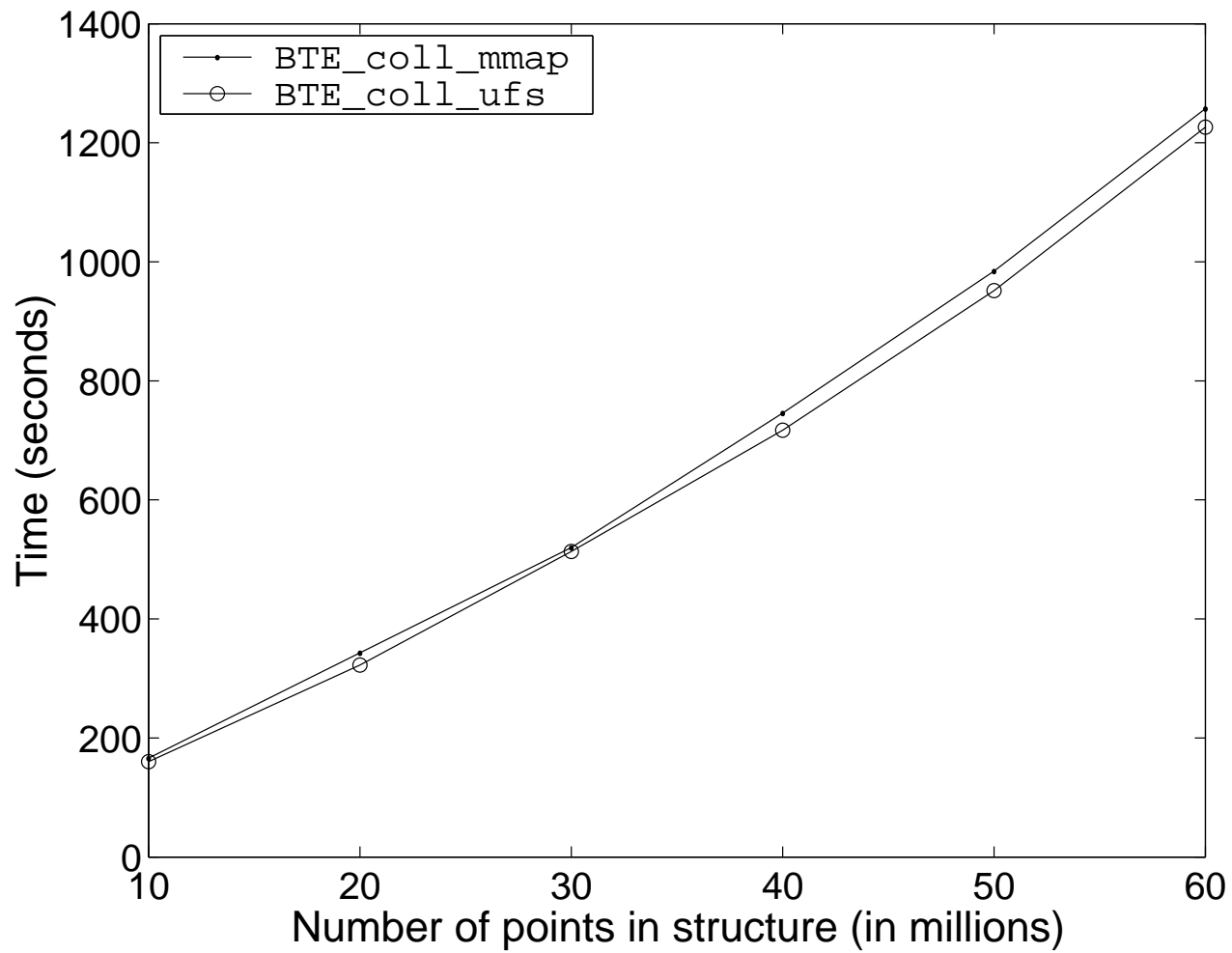
▶ Leaf node

- Links: 0
- Elements: B points
- Info: one integer

▶ Internal node

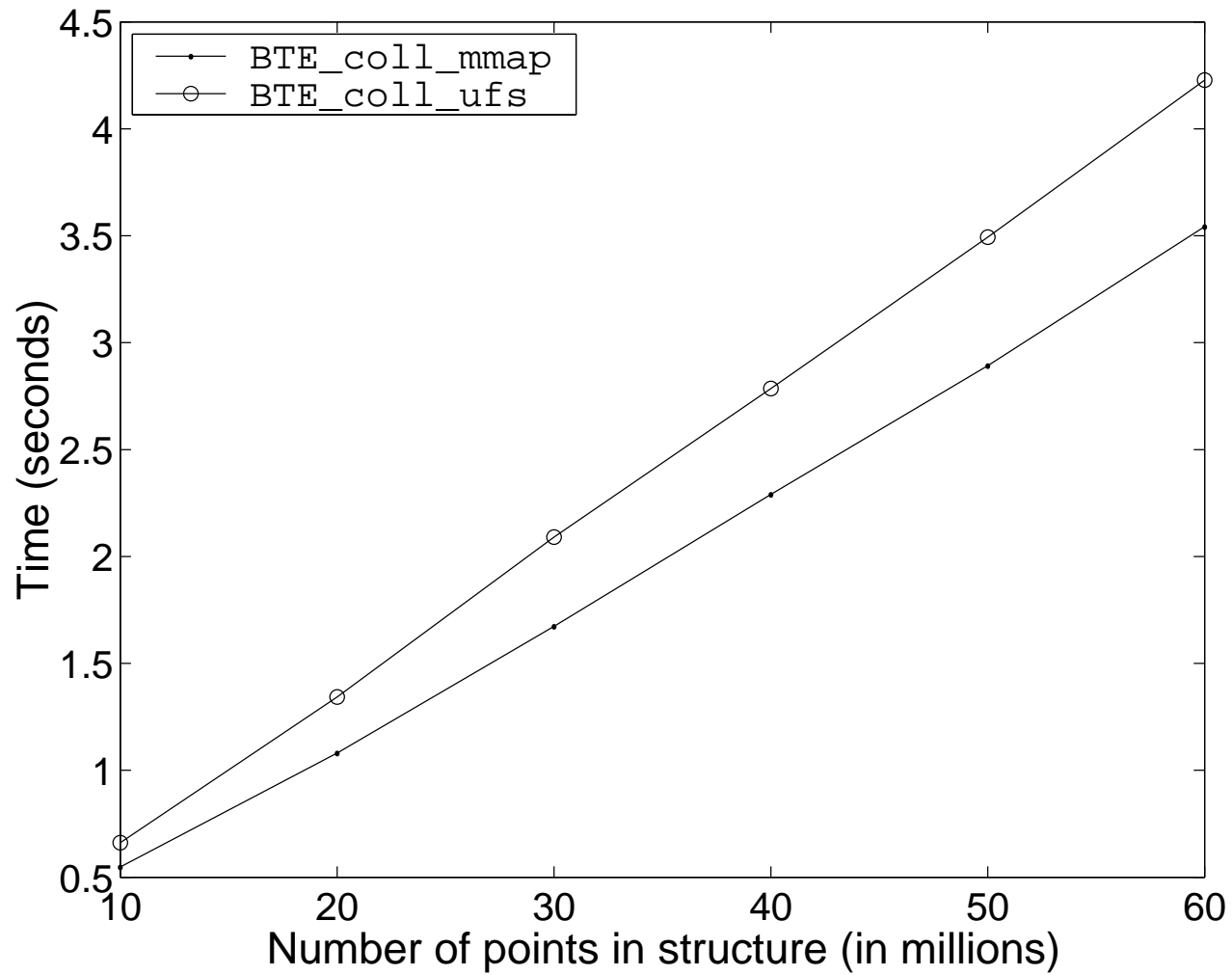
- Links: $\approx B/2$
- Elements: $\approx B/2$ rectangles
- Info: one integer

K-D-B-tree Experiments



Bulk loading performance.

K-D-B-tree Experiments



Query performance.

Conclusions

- ▶ We have developed the second phase of TPIE, adding support for I/O-efficient data structures.
- ▶ The library is portable, flexible and easy to use.
- ▶ A large number of data structures have been implemented.
- ▶ For the latest release, visit
<http://www.cs.duke.edu/~tpie/>

Future Work

- ▶ Provide more BTE implementations.
Ex. direct I/O (system dependent), network I/O.
- ▶ Implement more block-based data structures.
Ex. extendible hashing, buffer-tree.