TPIE

User Manual and Reference

Lars Arge Rakesh Barve David Hutchinson Octavian Procopiuc
Laura Toma Darren Erik Vengroff Rajiv Wickeremesinghe

DRAFT of November 21, 2003

TPIE User Manual and Reference
Edition 082902, for TPIE version 082902.

Contents

II Reference 45

Introduction

This manual describes TPIE, a Transparent Parallel I/O Environment, designed to assist programmers in writing high performance I/O-efficient programs for a variety of platforms.[1]

This manual, like the whole of the TPIE project, is work in progress. The authors are making it available in its current state in the hopes that it will be useful, but without any warranty whatsoever. Refer to the copyright page at the beginning of this manual for full details. Please send comments, bug reports, etc., to `tpie@cs.duke.edu`.

[1]: LA: Something about LEDA-SM somewhere?

Acknowledgements

The development of TPIE was supported in part by the National Science Foundation under grants CCR-9007851 and EIA-9870734 and by the U.S. Army Research Office under grants DAAL03-91-G-0035 and DAAH04-96-1-0013.

The authors would like to thank the following people for their contributions to the development of TPIE; Jeff Vitter, Jan Vahrenhold, Paul Natsev, Eddie Grove, Roberto Tamassia, Yi-Jen Chiang, Mike Goodrich, Jyh-Jong Tsay, Tom Cormen, Len Wisniewski, Liddy Shriver, David Kotz, and Owen Astrachan.

[2]: LA: There is some more text from Darren's ack. in here.

Part I

User Manual

Chapter 1

Overview

The data sets involved in some modern applications are too large to fit in the main memory of even the most powerful computers and must therefore reside on disk. Thus communication between internal and external memory, and not actual computation time, often becomes the bottleneck in the computation. This is due to the huge difference in access time of fast internal memory and slower external memory such as disks. While typical access time of main memory is measured in nanoseconds, a typical access time of a disk is on the order of milliseconds [20]. So roughly speaking there is a factor of a million difference in the access time of internal and external memory. A good example of an applications involving massive amounts of geometric data is NASA's Earth Observation System (EOS) [29, 42], which is expected to manipulate petabytes (thousands of terabytes, or millions of gigabytes) of data.

The goal of theoretical work in the area of *external memory (EM) algorithms* (also called *I/O algorithms* or *out-of-core algorithms*) is to eliminate or minimize the I/O bottleneck through better algorithm design.

In order to cope with the high cost of accessing data, efficient EM algorithms exploit locality in their design. They access a large *block* of B contiguous data elements at a time and perform the necessary algorithmic steps on the elements in the block while in the high-speed memory. The speedup can be considerable. A second effective strategy for EM algorithms is the use of multiple parallel disks; whenever an input/output operation is performed, D blocks are transferred in parallel between memory and each of the D disks (one block per disk).

The study of EM algorithm design was effectively started in the late eighties by Aggarwal and Vitter [6] and an important model for designing I/O algorithms called the Parallel Disk Model (PDM) was later proposed by Vitter and Shriver [56]. The PDM proposed that a good EM algorithm should transfer data between main memory and disk in a blocked manner, and should use all of the available disks concurrently. An optimal EM algorithm under this model minimizes the number of such blocked, parallel I/O operations it performs.

Subsequently, I/O algorithms for the PDM (mostly with a single disk and single processor) have been developed for many problem domains, including computational geometry [5, 38, 7, 13, 15, 4, 16, 41, 50, 51, 53, 3, 57, 2, 12, 13, 17, 37, 39, 14], graph algorithms [19, 7, 43, 1, 27, 8, 36, 45, 52], and string processing [34, 35, 11, 26].

The use of parallel disks has also received some theoretical attention [56, 46, 47, 30, 31]. There are more complicated models than the PDM, designed to address the I/O bottleneck in different ways. These include models that address the communication bottleneck between multiple layers in memory hierarchies [], and models incorporating parallel processors as well as parallel disks [22, 30, 31].

Implementations of these theoretical results are scarce. TPIE, *a Transparent Parallel I/O Environment*, is intended to bridge the gap between the theory and practice of parallel I/O systems. On one hand, TPIE attempts to provide usable implementations of (sometimes complex) theoretical algorithms, feeding back that experience to algorithm designers. On the other hand, TPIE also accommodates the use of heuristics from the practice of I/O algorithms in order to achieve maximum performance. Other EM implementation work includes benchmarking of certain geometric I/O algorithms by Chiang[18], experiments with FFT and related algorithms by Cormen et al. [25], implementation of the buffer tree [7] by Hutchinson et al. [40], and the LEDA-SM system for implementing data types by Crauser et al.[28]. Surveys of previous work in EM algorithm design and implementation can be found in [10, 9, 55]

The objectives of the TPIE project include the following:

- *Abstract away the details of how I/O is performed* so that programmers need only deal with a simple high level interface.
- *Provide a collection of I/O-optimal paradigms* for large scale computation that are efficient not only in theory, but also in practice.
- *Be flexible*, allowing programmers to specify the functional details of computation taking place within the supported paradigms. This will allow a wide variety of algorithms to be implemented within the system.
- *Be portable* across a variety hardware platforms.
- *Be extensible*, so that new features can be easily added later.

TPIE is implemented as a set of templated classes and functions in C++. It also includes a small library and a set of test and sample applications.

1.1 Hardware Platforms

TPIE has been tested on a variety of hardware platforms with a variety of UNIX operating systems. Combinations that have been tested include:

- Sun Sparc/Solaris 5.x
- DEC Alpha/Digital Unix 4.0
- DEC Alpha/FreeBSD 4.0
- Intel Pentium/FreeBSD 4.0
- Intel Pentium/Solaris 5.x

1.2 Future releases

The TPIE system is a research tool that is constantly evolving. The current release of TPIE (082902) includes the fundamental routines for solving fundamental *batched* problems such as sorting. These routines enable the programmer to write efficient and portable implementations of algorithms that makes use of fundamental *streaming* primitives [10, 54]. Relative to versions 0.8.02a and 0.9.01a, the current version of TPIE has been updated to improve performance and a number of bugs have been fixed. This manual has been updated to reflect these changes and several chapters have been expanded in order to allow the TPIE programmer to tune the system for best performance on a given platform. A list of the major changes can be found on the TPIE web page at `http://www.cs.duke.edu/TPIE/`. The TPIE project is work in progress – several major planned components have not yet been implemented and work is also being done to update and extend the current manual. Users of TPIE are encouraged to send bug reports, etc., to `tpie@cs.duke.edu`.

Extensions and/or improvements to TPIE are in progress. Projects include further performance improvements, support for parallel disks, addition of the distribution sweeping primitive [38], and addition of several application examples (examples of applications written using TPIE can be found in the papers listed on the TPIE home page). Another major project involves the addition of support for random access to blocks as opposed to the stream oriented access used in the current version of TPIE. This addition will facilitate the implementation of indexing structures (external data structures). Users interested in obtaining/testing preliminary versions of these extensions are encouraged to send a request to `tpie@cs.duke.edu`.

Chapter 2

Obtaining and Installing TPIE

2.1 Licensing

TPIE is available under the terms of the GNU General Public License, version 2. A copy of this license appears in Appendix D.

2.2 Where to get TPIE

The latest version of TPIE, 082902, is an alpha test version. It is available through the TPIE WWW Home Page at URL `http://www.cs.duke.edu/TPIE/`.

To obtain the TPIE source distribution, follow the pointers from the home page to the distribution itself, which consists of a gzipped tar file named `tpie_082902.tgz`. Your Web browser should be capable of downloading this file to your local machine.

2.3 Prerequisites

To uncompress and unarchive the distribution, you will need either the GNU `tar` utility, or `gzip` and a `tar` program. (the GNU version can decompress and untar at the same time with the 'z' option). The GNU `make` utility is also needed. This utility is usually located in `/usr/local/bin/make` (or is called `gmake`).

TPIE is heavily dependent on the compiler used, mainly because of the use of C++ templates. It currently requires the GNU C++ compiler, `gcc`, version 2.95 or later (it has also been successfully compiled with `gcc` version 2.7.2.1 on some systems). We are currently using `gcc`, version 2.95 for most development work on TPIE, and we expect that TPIE will also be compatible with future version of this compiler. TPIE has also been successfully compiled using `egcs`, version 2.91.66.

Information on how to obtain and install GNU software is available at URL
`http://www.gnu.org/software/software.html`.

2.4 Installation

Place `tpie_082902.tgz` in the directory in which TPIE is to be installed, `cd` into that directory, and execute the command

`tar xzf tpie_082902.tgz` (or `gunzip -c tpie_082902.tgz | tar xvf -`)

This will produce a directory `tpie_082902` with subdirectories `include`, `lib`, `lib/src`, `test`, and `doc`. Enter the directory `tpie_082902`. You must now configure TPIE for your particular system. To do this, use the command

Directory	Contents
`include`	The TPIE header files.
`lib`	The TPIE run-time library. This is relatively small, as most of the TPIE system remains in the form of templated header files.
`lib/src`	The source code for the TPIE run-time library.
`test`	A series of test applications designed to verify that TPIE is operating correctly. This directory also includes the code for the sample program discussed in Chapter 3, and the example applications described in Appendix B.
`doc`	Written documentation for TPIE, consisting of the document you are reading now, in DVI and Postscript(TM) formats.

Figure 2.1: Components of the TPIE distribution.

```
./configure
```

Certain configuration options can be specified to the `configure` script, but usually these will not be of interest the first time TPIE is installed. These options are described in Section 7.1.1.

The configuration program will take some time to examine the parameters of your system. Once it has done so, it will produce the various Makefiles and configuration files required to build TPIE on your system. When this is done, simply invoke your version of GNU `make`:

```
make all
```

to build the complete TPIE system. This will build the components of TPIE that must be tailored to your system. This includes: the TPIE run-time library `tpie_082902/lib/libtpie.a`, the test and sample programs in directory `tpie_082902/test`, and certain header files in `tpie_082902/include`. You should now have a complete TPIE system, consisting of the directories listed in Figure 2.1.

Chapter 3

A Taste of TPIE via a Sample Program

This chapter presents a quick look at TPIE via a simple TPIE program. A more detailed TPIE tutorial appears in Chapter 4.

One of the primary themes in TPIE is to allow a user to specify an I/O efficient computation via high-level coordination of data movement interspersed with appropriate internal memory computing, with the low level I/O details being transparent, or "under the hood". TPIE provides various classes of "management objects", (e.g. *scan management objects*, *merge management objects*, etc.) that allow the user to specify sophisticated data movement operations on *streams* of data in a simple and straightforward manner. These management object classes are built on top of a simple *stream interface* called `AMI_STREAM`. The tutorial in the next chapter explains how to specify and use such management object classes.

The sample program below uses simple stream operations[3] to generate a stream of random integers, scans this stream of integers and partitions them into several distinct streams. The manner in which I/O operations are handled by TPIE ensures that the program is I/O efficient.

The intent of this example is to illustrate the sort of things involved in TPIE programming; the typical include files, specifying how much memory the program should use, streaming operations, etc. The program is given in Section 3.1 and it is discussed in Section 3.2.

[3]: DH: we use only STREAM member functions in the sample program, but never describe them in the tutorial. We should either describe them or illustrate the power of TPIE using operation management objects (scanning, merging, sorting, etc.

3.1 Sample Program

The following sample program can be found in `tpie_082902/test/sample_pgm.cpp` after TPIE has been installed (see Section 2.4 of this manual for installation instructions).

```
#include <versions.h>
VERSION(sample_pgm_cpp,"$Id: sample_pgm.cpp,v 1.3 2002/06/26 23:38:11 tavi Exp $");

#include <iostream.h>
#include <stdlib.h>
#include <sys/time.h>
#include <limits.h> //for INT_MAX

//Include the file that sets application configuration: It sets what
//kind of BTE (Block Transfer Engine) to use and where applicable,
//what should be the size of the logical block (the logical block size
//is a user specified multiple of the physical block size) for a
//stream and so on;
#include "app_config.h"

//Include the file that will allow us to use AMI_STREAMs.
#include <ami.h>
```

```
//include "wall clock" timer that will allow us to time
#include <wall_timer.h>

//Include TPIE's internal memory sorting routines.
#include <quicksort.h>

//This program writes out an AMI_STREAM of random integers of
//user-specified length, and then, based on 7 partitioning elements
//chosen from that stream, partitions that stream into 8 buckets. Each
//of the buckets is implemented as an AMI stream and the program
//prints the size of each bucket at the end.

//The user needs to specify the length of the initial stream of
//integers and the size of the main memory that can be used.


void
main(int argc, char *argv[]) {

   //parse arguments
   if (argc < 3) {
      cout << "Input the number of integers to be generated\n";
      cout << " and the size of memory that can be used\n";
      exit(1);
   }
   int Gen_Stream_Length = atoi(argv[1]);
   long test_mm_size = atol(argv[2]);

   //Set the size of memory the application is allowed to use
   MM_manager.set_memory_limit(test_mm_size);

   //the source stream of ints
   AMI_STREAM<int> source;

   //the 8 bucket streams of ints
   AMI_STREAM<int> buckets[8];


   //************************************************************
   //generate the stream of randon integers
   AMI_err ae;
   int src_int;
   for (int i = 0; i < Gen_Stream_Length; i++) {

      //generate a random int
      src_int = random();

      //Now write out the integer into the AMI_STREAM source using
      //the AMI_STREAM member function write_item()
      if ((ae = source.write_item(src_int)) != AMI_ERROR_NO_ERROR) {
         cout << "AMI_ERROR " << ae << " during source.write_item()\n";
         exit(1);
      }

   }
   //print stream length
   cout << "source stream is of length " << source.stream_len() << endl;
```

```
//***********************************************************
//pick the first 7 integers in source stream as partitioning elements
//(pivots)

//Seek to the beginning of the AMI_STREAM source.
if ((ae = source.seek(0))!= AMI_ERROR_NO_ERROR) {
   cout << "AMI_ERROR " << ae << " during source.seek()\n";
   exit(1);
}

//read first 7 integers and fill in the partitioning array
int partitioning[8];
int *read_ptr;

for (int i = 0; i < 7; i++) {

   //Obtain a pointer to the next integer in AMI_STREAM source
   //using the member function read_item()
   if ((ae = source.read_item(&read_ptr)) != AMI_ERROR_NO_ERROR) {
      cout << "AMI_ERROR " << ae << " during source.read_item()\n";
      exit(1);
   }

   //Copy the current source integer into the partitioning element array.
   partitioning[i]= *read_ptr;
}
cout << "Loaded partitioning array\n";

//***********************************************************
//sort partitioning array

quick_sort_op((int *)partitioning,7);
cout << "sorted partitioning array\n";
partitioning[7] = INT_MAX;

//***********************************************************
//PARTITION INTS OF source INTO THE buckets USING partitioning ELEMENTS

struct timeval tp1, tp2;

//binary search variables.
int u,v,l,j;

//start timer
wall_timer wt;
wt.start();

//seek to the beginning of the AMI_STREAM source.
if ((ae = source.seek(0))!= AMI_ERROR_NO_ERROR) {
   cout << "AMI_ERROR " << ae << " during source.seek()\n";
   exit(1);
}

//scan source stream distributing the integers in the approriate
//buckets
for (int i = 0; i < Gen_Stream_Length; i++) {

   //Obtain a pointer to the next integer in AMI_STREAM source
   //using the member function read_item()
```

```
      if ((ae =   source.read_item(&read_ptr)) != AMI_ERROR_NO_ERROR) {
         cout << "AMI_ERROR " << ae << " during source.read_item()\n";
         exit(1);
      }
      v = *read_ptr;

      // using a binary search, find the stream index l to which v
      // should be assigned
      l = 0;
      u = 7;
      while (u >= l) {
         j = (l+u)>>1;
         if (v < partitioning[j]) {
            u = j-1;
         } else {
            l = j+1;
         }
      }

      // now write out the int into the AMI_STREAM buckets[l] using
      // the AMI_STREAM member function write_item().
      if ((ae = buckets[l].write_item(v)) != AMI_ERROR_NO_ERROR) {
         cout << "AMI_ERROR " << ae << " during buckets[" << l
              << "].write_item()\n";
         exit(1);
      }
   }

   //stop timer
   wt.stop();
   cout << "Time taken to partition is " << wt.seconds() << " seconds" << endl;

   //delete the file corresponding to the source stream when source
   //stream gets destructed (this is the default, so this call is not
   //needed)
   source.persist(PERSIST_DELETE);

   //let the file corresponding to buckets[i] persist on disk when the
   //buckets[i] stream gets destructed
   for (int i = 0; i < 8; i++) {
      buckets[i].persist(PERSIST_PERSISTENT);
      cout << "Length of bucket " << i << " is "
             << buckets[i].stream_len() << endl;
   }
}
```

3.2 Discussion of Sample Program

In this section we discuss the simple C++ sample program shown in the previous section. The file `app_config.h` is the TPIE configuration file. TPIE's `AMI_STREAM` stream I/O operations are carried out transparently by one of three possible *block transfer engines (BTEs)*. Briefly, the file `app_config.h` chooses a specific BTE, and the amount of internal memory used as buffer space for each `AMI_STREAM`. The `app_config.h` configuration file is further discussed in Section 7 which also contains a discussion of how to choose a BTE for a given platform. The file `<ami.h>` contains TPIE's templated classes and functions, while the file `<quicksort.h>` contains various quicksort polymorphs. Note that each `AMI_STREAM` corresponds to an underlying Unix file.

The program illustrates the use of the basic `AMI_STREAM` member functions `read_item()`, `write_item()`, seek() and persist(). Successful execution of these member functions is indicated by a return value of AMI_ERROR_NO_ERROR. The program distributes a randomly generated source stream of integers into eight bucket streams, and then displays the time taken by this operation and the size of each of the eight output buckets. The randomly generated stream is deleted upon completion of the program (source.persist(PERSIST_DELETE)), while the bucket streams are saved (made persistent with buckets[i].persist(PERSIST_PERSISTENT)) in the default scratch directory /var/tmp. The default location for the scratch files can be changed by setting the environment variable AMI_SINGLE_DEVICE appropriately (see Section 7.1.3).

TPIE can run with a user-specified amount of internal memory (although typically, about 4MB is required as a minimum for most simple applications) or it can run with virtual memory like an ordinary non-TPIE application. The former mode is invoked by calling MM_manager.ignore_memory_limit(), and the latter by calling MM_manager.enforce_memory_limit(). In the sample program, we call MM_manager.enforce_memory_limit(), which means that the program will abort if the allocated internal memory exceeds the specified amount. The call MM_manager.set_memory_limit(test_mm_ tells TPIE's internal memory manager MM_manager to prevent the program's internal memory usage from exceeding test_mm_size bytes (note that test_mm_size is the second input argument to our program). When MM_manager.enforce_memory_limit() is used, it is the responsibility of the user to inform MM_manager via set_memory_limit() of the desired memory limit. For example, one might set this value to the amount of physical main memory minus the main memory used by the operating system and other programs running on the machine.

The sample program can be compiled as follows: (recall that Section 2.3 discussed the version of GNU C++ required):

```
g++ sample_pgm.cpp -I../include/ -L../lib/ -ltpie -o sample_pgm
```

By way of example, the program can be run with 1000000 random integers and 5000000 bytes of main memory as follows:

```
sample_pgm 1000000 5000000
```

Chapter 4

Tutorial

4.1 Introduction

This tutorial is designed to introduce new users to the TPIE system. It introduces the fundamental paradigms of computation that TPIE supports, giving source code examples of each. The majority of the code presented in the tutorial is available in the test applications directory of the distribution, `tpie_082902/test/`.

For the sake of brevity, much of the code presented in this tutorial is incomplete, in the sense that necessary header files and macros are omitted. Details concerning how to write your own complete TPIE code is presented at the end of the tutorial in Section 4.13 (see also Sections 3 and 7.2.1[4])

TPIE is written in the C++ language, and this manual assumes that the reader is familiar with C++. If you would like to use TPIE but are not familiar with the C++ language, a number of good books are available. If you are familiar with C, [48] is a good place to start. A more basic, but very comprehensive book is [32], and [44] is an excellent source of information on intermediate and advanced C++. Finally, [33] is the definitive book on C++, though not necessarily the best place for new programmers to start.

[4]: LA: Maybe we should talk briefly about AMI, BTE, MM somewhere around here - we talk about main memory issues in merging and compiling sections.

Familiarity with the theoretical results on I/O-efficient algorithms is not necessary in order to use TPIE. However, this tutorial (and the rest of this manual) may be easier to follow with some general background information such as how a theoretically optimal external (merge) sort algorithm works. Good references are [54, 10, 6]. Some of the basic concepts required for understanding the discussion of I/O issues and external memory algorithms in this manual are outlined in Section 4.2.

4.2 Basic Concepts

Roughly speaking there is a factor of a million difference in the access time of internal and external memory. In order to cope with the high cost of accessing externally-stored data, efficient EM algorithms exploit locality in their design. They access a large *block* of B contiguous data elements at a time and perform the necessary algorithmic steps on the elements in the block while it is in the high-speed memory. The speedup can be considerable. A second effective strategy for EM algorithms is the use of multiple parallel disks; whenever an input/output operation is performed, D blocks are transferred in parallel between memory and each of the D disks (one block per disk).

[5]: Stuff there should maybe be something about: substreams, persistence, read-/write primitives, etc.

[6]: We start talking about memory constraints - there should be a general intro to blocks and stuff somewhere.

The performance of an EM algorithm on a given set of data is affected directly by how much internal memory is available for its use. We use M to denote the number of application data elements that fit into the internal memory available to the algorithm, and $m = M/B$ denotes the number of blocks that fit into the available internal memory. Such a block is more precisely called a *logical block* because it may be a different size (usually larger) than either the physical block size or the system block size. We will reserve the term *physical block* size to mean the block size used by a disk controller to communicate with a physical disk, and the *system block* size will be the size of block used within the operating system for I/O operations on disk devices. In EM algorithms we will assume that the logical block size is a multiple of the system block size. In TPIE, for instance, this factor is currently set to 32 if not changed by the user.

TPIE is implemented as a set of templated classes and functions in C++, and employs an object-oriented abstraction to EM computation. TPIE provides C++ templates of various optimal EM computation *patterns* or *paradigms*. Examples of such paradigms are the EM algorithms for merge sorting, distribution sweeping, time forward processing, etc. (see [55]). In a TPIE program, the application programmer provides application-specific details of the specific computation paradigm used, such as C++ object definitions of the application data records, and code for application-specific sub-computations at critical points in the computation pattern, but TPIE provides the application-independent parts of the pattern.

The definition of an application data element (or record) is provided by the user as a class definition. Such a class definition is typically used as a template parameter in a TPIE code fragment (e.g. a templated function). Other template parameters may be instantiated by choices the user makes between algorithm options (e.g. between sorting variants) or between operating system interfaces (e.g. the choice of BTE) for instance. These user selections allow a pattern replesented by a templated C++ code fragment to be instantiated as an actual piece of executable code, tailored to the data types required by the user's application.

Application-dependent sub-computations (e.g. a comparison function used to determine the order of two application data elements during a sort) are typically structured as methods of an *operation management object*. TPIE dictates the names and required functionality of each method of an operation management object, but the details of the computation performed by a specific method are application-specific and thus are the responsibility of the application programmer.

4.3 Streams

[7]: LA: Stuff there should maybe be something about: substreams, persistence, read/write primitives, ect. DH: See above.

In TPIE, a *stream* is an ordered collection of objects of a particular type, stored in external memory, and accessed in sequential order. Streams can be thought of as fundamental TPIE objects which map volatile, typed application data elements in internal memory to persistent, untyped data elements in external memory, and vice-versa. Streams are read and written like files in Unix and support a number of primitive file-like operations such as read(), write(), truncate(), etc. TPIE also supports the concept of a *substream*, which permits a contiguous subset of the elements in a stream to accessed sequentially. Multiple substreams can be created on streams and even on other substreams.

Various paradigms of external memory computation are supported on streams (and substreams) in TPIE, including scanning (see Section 4.5), merging (see Section 4.6), and sorting (see Section 4.8). TPIE reduces the programming effort required to perform an external sort, merge, etc., by providing the high level flow of control within each paradigm, and therefore structuring this part of the computation so that it will be I/O efficient. The programmer is left with the task of providing what amount to "event handlers", specifying the application-specific details of the computation. For instance in sorting, the programmer defines a stream of input data, a comparison function (the event handler for the task of comparing two application data elements), and an output stream for the results. In TPIE terminology, the collection of necessary event handlers for a particular EM computational paradigm is contained in an *operation management object*. Operation management objects differ according to which paradigm is used. See Section 4.5 for a discussion of scan management objects, Section 4.6 for a discussion of merge management objects, and Section 4.8 for a discussion of sort management objects.

Creating a stream of objects in TPIE is very much like creating any other object in C++. The only difference is that the data placed in the stream is stored in external memory (on disk). For example, to create an (empty) stream `stream0` capable of storing integers, we could use the following:

```
AMI_STREAM<int> stream0;
```

Alternatively, the following creates a pointer `p` to an empty stream of `applrec` objects:

```
AMI_STREAM<applrec> *p = new AMI_STREAM<applrec>;
```

The `AMI` prefix in `AMI_STREAM` stands for *Access Method Interface*. This layer of TPIE contains the services and functionality which a normal user of TPIE will require. `AMI_STREAM` is actually a compile-time macro that evaluates to the name of a particular implementation of streams, but for now it is safe to assume that it is simply a C++ class.

The AMI_STREAM constructor does not actually put anything into the stream; it simply creates the necessary data structures to keep track of the contents of the stream when data is actually put into it. One basic way of putting data into a stream is via AMI_scan(), which is described in Section 4.5.

4.4 Operation Management Objects

TPIE provides a structure for performing a number of basic operations, such as scanning a stream of items, merging streams of items, sorting a stream of items, etc. Much of the application-independent work in these operations is handled by TPIE. The TPIE user, however, must provide the code for the application-dependent aspects of these operations via a TPIE *operation management object*. An operation management object in TPIE is an object which contains member functions to control the critical, application-varying aspects of operations such as scanning, merging, and sorting. TPIE expects certain named methods of the object to be present, depending on the operation being performed.

The operation management object for scanning (called a "scan management object"), for instance, must provide methods initialize (for initializing any user-required data structures of the scan), and operate (for performing whatever steps are needed as each data item is encountered in the scan). See Section 4.5 below for more information.

In the example of merging, the rules for "combining" elements of the merge operation are necessarily application dependent. TPIE expects a "merge management object" to contain members initialize and operate, as well as several others. Detailed requirements for merge management objects are described in Section 4.6.

4.5 Scanning

The simplest paradigm available in TPIE is scanning. Scanning can be used to produce streams, examine the contents of streams, or transform streams.

4.5.1 Basic Scanning

One basic thing a scan can do is write a series of objects to a stream. In the following example, we create a stream of integers consisting of the first 10000 natural numbers.

```
class scan_count : AMI_scan_object {
private:
    int maximum;
    int nextint;
public:
    scan_count(int max) : maximum(max), ii(0) {};

    AMI_err initialize(void)
    {
        nextint = 0;
        return AMI_ERROR_NO_ERROR;
    };

    AMI_err operate(int *out1, AMI_SCAN_FLAG *sf)
    {
        *out1 = ++nextint;
        return (*sf = (nextint <= maximum)) ? AMI_SCAN_CONTINUE :
            AMI_SCAN_DONE;
    };
};
```

```
scan_count sc(10000);
AMI_STREAM<int> amis0;

void f()
{
    AMI_scan(&sc, &amis0);
}
```

The object sc is called a scan management object. It has two member functions, initialize() and operate(), which TPIE calls when asked to perform a scan. The first member function, initialize() is called at the beginning of the scan. TPIE expects that a call to this member function will cause the object to initialize any internal state it may maintain in preparation for performing a scan. The second member function, operate(), is called repeatedly during the scan to create objects to go into the output stream. operate() sets the flag *sf to indicate whether it generated output or not. TPIE will call operate() as long as operate() returns AMI_SCAN_CONTINUE. The normal way for operate() to signal that it is finished is to return the value AMI_SCAN_DONE.

AMI_scan behaves as the following pseudo-code:

```
AMI_err AMI_scan(scan_count &sc, AMI_STREAM<int> *pamis)
{
    int nextint;
    AMI_err ae;
    AMI_SCAN_FLAG sf;

    sc.initialize();
    while ((ae = sc.operate(&nextint, &sf)) == AMI_SCAN_CONTINUE) {
        if (sf) {
            Write nextint to *pamis;
        }
    }

    if (ae != AMI_SCAN_DONE) {
        Handle error conditions;
    }

    return AMI_ERROR_NO_ERROR;
}
```

Thus, after the function f() in the original example code returns, the stream amis0 will contain the integers from 1 to 10000 in increasing order.

One of the simplest things we can do with a stream of objects is scan it in order to transform it in some way. As an example, suppose we wanted to square every integer in the stream amis0. We could do so using the following code:

```
class scan_square : AMI_scan_object {
public:
    AMI_err initialize(void)
    {
        return AMI_ERROR_NO_ERROR;
    };

    AMI_err operate(const int &in, AMI_SCAN_FLAG *sfin,
                    int *out, AMI_SCAN_FLAG *sfout)
    {
        if (*sfout = *sfin) {
            *out = in * in;
            return AMI_SCAN_CONTINUE;
        } else {
            return AMI_SCAN_DONE;
        }
    };
};

scan_square ss;
AMI_STREAM<int> amis1;

void g()
{
    AMI_scan(&amis0, &ss, &amis1);
}
```

Notice that the call to `AMI_scan()` in `g()` differs from the one we used in `f()` in that it takes two stream pointers and a scan management object. By convention, the stream `amis0` is an input stream, because it appears before the scan management object `ss` in the argument list. By similar convention, `amis1` is an output stream. Because the call to `AMI_scan` has one input stream and one output stream, TPIE expects the `operate()` member function of `ss` to have one input argument (which is called `in` in the example above) and one output argument (called `out` in the example above). Note that the `operate()` member function of the class `square_scan` also takes two pointers to flags, one for input (`sfin`) and one for output (`sfout`). `*sfin` is set by TPIE to indicate that there is more input to be processed. `*sfout` is set by the scan management object to indicate when output is generated. A scan management object must contain an `operate()` member function that takes the appropriate types and number of arguments for the invocation of `AMI_scan()` that uses it, or else a compile-time error will be generated.

`AMI_scan` with one input stream and one output stream behaves as the following pseudo-code:

```
AMI_err AMI_scan(AMI_STREAM<int> *instream, scan_square &ss,
        AMI_STREAM<int> *outstream)
{
    int in, out;
    AMI_err ae;
    AMI_SCAN_FLAG sfin, sfout;

    sc.initialize();

    while (1) {
        {
             Read in from *instream;
             sfin = (read succeeded);
        }
        if ((ae = ss.operate(in, &sfin, &out, &sf)) ==
            AMI_SCAN_CONTINUE) {
            if (sfout) {
                Write out to *outstream;
            }
            if (ae == AMI_SCAN_DONE) {
                return AMI_ERROR_NO_ERROR;
            }
            if (ae != AMI_SCAN_CONTINUE) {
                Handle error conditions;
            }
        }
    }
}
```

`AMI_scan()` can operate on up to four input streams and four output streams. Here is an example that takes two input streams of values, `x` and `y`, and produces four output streams, one consisting of the running sum of the `x` values, one consisting of the running sum of the `y` values, one consisting of the running sum of the squares of the `x` values, and one consisting of the running sum of the squares of the `y` values.

```
class scan_sum : AMI_scan_object {
private:
    double sumx, sumx2, sumy, sumy2;
public:
    AMI_err initialize(void)
    {
        sumx = sumy = sumx2 = sumy2 = 0.0;
        return AMI_ERROR_NO_ERROR;
    };

    AMI_err operate(const double &x, const double &y,
                    AMI_SCAN_FLAG *sfin,
                    double *sx, double *sy,
                    double *sx2, double *sy2,
                    AMI_SCAN_FLAG *sfout)
    {
        if (sfout[0] = sfout[2] = sfin[0]) {
            *sx = (sumx += x);
            *sx2 = (sumx2 += x * x);
        }
        if (sfout[1] = sfout[3] = sfin[1]) {
            *sy = (sumx += y);
            *sy2 = (sumy2 += y * y);
        }
        return (sfin[0] || sfin[1]) ? AMI_SCAN_CONTINUE : AMI_SCAN_DONE;
    };
};

AMI_STREAM<double> xstream, ystream;

AMI_STREAM<double> sum_xstream, sum_ystream, sum_x2stream, sum_y2stream;

scan_sum ss;

void h()
{
    AMI_scan(&xstream, &ystream, &ss,
             &sum_xstream, &sum_ystream, &sum_x2stream, &sum_y2stream);
}
```

[8]: DH: should we summarize what polymorphs are available or give a pointer to the section that does?

4.5.2 ASCII Input/Output

TPIE provides a number of predefined scan management objects. Among the most useful are instances of the template classes `cxx_ostream_scan<T>` and `cxx_ostream_scan<T>`, which are used for reading ASCII data into streams and writing the contents of streams in ASCII respectively. This is done in conjunction with the `iostream` facilities provided in the standard C++ library. Any class `T` for which the operators `ostream &operator<<(ostream &s, T &t)` and `istream &operator>>(T &t)` are defined can be used with this mechanism.

As an example, suppose we have an ASCII file called `input_nums.txt` containing one integer per line, such as

```
17
289
4195835
3145727
.
.
.
```

The following code copies this file into a TPIE stream of integers, squares each one, and writes the results to the file `output_nums.txt`.

```
void f()
{
    ifstream in_ascii("input_nums.txt");
    ofstream out_ascii("output_nums.txt");
    cxx_istream_scan<int> in_scan(in_ascii);
    cxx_ostream_scan<int> out_scan(out_ascii);
    AMI_STREAM<int> in_ami, out_ami;
    scan_square ss;

    // Read them.
    AMI_scan(&in_scan, &in_ami);

    // Square them.
    AMI_scan(&in_ami, &ss, &out_scan);

    // Write them.
    AMI_scan(&out_ami, out_scan);

}
```

In order to read from an ASCII input file using the scan management object `in_scan`, `AMI_scan()` repeatedly calls `in_scan->operate()`, just as it would for any scan management object. Each time `in_scan->operate()` is called, it uses the `>>` operator to read a single integer from the input file. When the input file is exhausted, `in_scan->operate()` returns `AMI_SCAN_DONE`, and `AMI_scan()` returns to its caller. The behavior of `out_scan` is similar to that of `in_scan`, except that it writes to a file instead of reading from one.

4.5.3 Multi-Type Scanning

In all of the examples presented up to this point, scanning has been done on streams of objects that are all of the same type. `AMI_scan()` is not limited to such scans, however. In the following example, we have a scan management class that takes two streams of `double`s and returns a stream of complex numbers.

```
class complex {
public:
    complex(double real_part, imaginary_part);
    ...
};

class scan_build_complex : AMI_scan_object {
public:
    AMI_err initialize(void) {};
    AMI_err operate(const double &r, const double &i,
                    AMI_SCAN_FLAG *sfin,
                    complex *out, AMI_SCAN_FLAG *sfout)
    {
        if (*sfout = (sfin[0] || sfin[1])) {
            *out = complex((sfin[0] ? r : 0.0), (sfin[1] ? i : 0.0));
            return AMI_SCAN_CONTINUE;
        } else {
            return AMI_SCAN_DONE;
        }
    };
};
```

4.5.4 Out of Step Scanning

In all the examples up to this point, the `operate()` member function of a scan management object has been called exactly once for each object in the input stream(s). In this section, we discuss the concept of out of step scanning, which involves using a scan management object to reject certain inputs and ask that they be resubmitted in subsequent calls to the `operate()` member function.

Suppose we have two streams of integers, each of which is sorted in ascending order. We would like to merge the two streams into a single output stream consisting of all the integers in the two input streams, in sorted order. In order to do this with a scan, we must have the ability to look at the next integer from each stream, choose the smaller of the two and write it to the output stream, and then ask for the next number from the stream from which it was taken. There is a simple mechanism for doing this. The same flags that TPIE uses to tell the scan management object which inputs are available can also be used by the scan management object to indicate which inputs were used (i.e. "consumed") and which should be presented again.

Consider the following example of a scan management object class which performs this sort of binary merge:

```
class scan_binary_merge : AMI_scan_object {
public:
    AMI_err initialize(void) {};

    AMI_err operate(const int &in0, const int &in1, AMI_SCAN_FLAG *sfin,
                    int *out, AMI_SCAN_FLAG *sfout)
    {
        if (sfin[0] && sfin[1]) {
            if (in0 < in1) {
                sfin[1] = false;
                *out = in0;
            } else {
                sfin[0] = false;
                *out = in1;
            }
        } else if (!sfin[0]) {
            if (!sfin[1]) {
                *sfout = false;
                return AMI_SCAN_DONE;
            } else {
                *out = in1;
            }
        } else {
            *out = in0;
        }
        *sfout = 1;
        return AMI_SCAN_CONTINUE;
    }
};
```

In the operate method, we first check that both inputs are valid by looking at the flags pointed to by `sfin`. If both are valid, then we select the smaller of the inputs and copy it to the output. We then clear the other input flag to let TPIE know that we did not use that input, but we will need it later and it should be resubmitted on the next call to operate. The remainder of the function handles the cases when one of more of the input streams are empty.

4.6 Merging

While AMI_scan() is limited to operate on up to four input and four output streams, theoretically efficient external memory algorithms often operate on eight or more streams, the exact number depending on the amount of internal memory available. An especially common operation is merging of a large number of input streams into one output stream.[1] An example of the this is external merge sorting. The scan_binary_merge scan management object presented in the previous section could be used recursively to implement a merge sorting algorithm. We could simply divide the input stream into sub-streams small enough to fit into main memory, read each sub-stream into memory and sort it, and then merge pairs of streams, then pairs of merged pairs of streams, and so on, until we had merged all the input back into one completely sorted stream. While this approach would correctly sort the input, it would not be nearly as efficient as possible on most machines. The reason is that we typically have enough main memory available to merge many more streams together at one time [6].

TPIE therefore provides the function AMI_merge() which, depending on the main memory available, merges a variable number of input streams into an output stream in a single scan of the input streams. As in the case of AMI_scan, the application-specific details of how the merge is performed are specified via an operation management object (in this case, a merge management object) with member functions initialize() and operate()[9].[2] However, often, as in the merge sort example, one wants to merge more streams than memory constraints allow in a single pass, and so the merge may have to be done in several recursive stages.[10] Since it can be cumbersome to compute precisely how many streams can be merged in one pass — one must keep track of the space needed for input blocks from each of the streams being merged, as well as the overhead of any data structures needed for the merge — TPIE provides a mechanism that does most of this work for us. The function AMI_partition_and_merge() divides an input stream into sub-streams just small enough to fit into main memory, operates on each in main memory, then merges them back into a single output stream, using intermediate streams if memory constraints dictate. As in the case of AMI_merge(), the functional details of AMI_partition_and_merge() are specified via a merge management object. In fact the merge management object for AMI_merge() is a special case of the one for AMI_partition_and_merge(). The following example illustrates the use of AMI_partition_and_merge():

[9]: LA: The AMI_merge stuff should be rewritten with some examples

[10]: LA: Here we start talking about memory constraints - there should be a general intro to blocks and stuff somewhere.

```
class my_merger : AMI_merge_base {
public:
    AMI_err initialize(arity_t arity, const T * const *in,
                       AMI_merge_flag *taken_flags,
                       int &taken_index);
    AMI_err operate(const T * const *in, AMI_merge_flag *taken_flags,
                    int &taken_index, T *out);
    AMI_err main_mem_operate(T* mm_stream, size_t len);
    size_t space_usage_overhead(void);
    size_t space_usage_per_stream(void);
};

AMI_STREAM<T> instream, outstream;

void f()
{
    my_merger mm;
    AMI_partition_and_merge(&instream, &outstream, &mm);
}
```

The member functions of the merge management object mm are as follows:

[1] Note that "merge" here means the process of reading the content of a number of input streams in some interleaved order producing an output stream. Merging a number of sorted input streams into a sorted output stream is a special (but common) case of merging.

[2] ami_merge also has three specialized polymorphs for merging according to a total order on the data elements. These specialized polymorphs do not use a merge management object. Refer to Section 5.6.

`initialize()` Tells the object how many streams TPIE has chosen to (`arity`) and what the first item from each stream is (`in`). The variables `taken_flags` and `taken_index` provide two mechanisms for the merge manager to tell TPIE what objects it took from the input streams. These are discussed in more detail in the context of a merge sorting example in Section 4.6.1.

`operate()` Just as in scanning, this member function is called repeatedly to process input objects.

`main_mem_operate()` Called by TPIE to operate on an array of data in main memory.

`space_usage_overhead()` Called by TPIE prior to initialization to assess how much main memory this object will use.

`space_usage_per_item()` Called by TPIE prior to initialization to assess how much main memory may be used per input stream. Merge management objects are allowed to use main memory space linear in the number of input streams.

The following pseudo-code describes the operation of `AMI_partition_and_merge()`. Note that for simplicity of presentation, boundary conditions are not covered.

```
AMI_err AMI_partition_and_merge(instream, outstream, mm)
{
    max_ss = max # of items that can fit in main memory;
    Partition instream into num_substreams substreams of size max_ss;

    Foreach substream[i] {
        Read substream[i] into main memory;
        mm->main_mem_operate(substream[i]);
        Write substream[i];
    }

    Call mm->space_usage_overhead() and mm->space_usage_per_stream;

    Compute merge_arity; // Maximum # of streams we can merge.

    while (num_substreams > 1) {
        for (i = 0; i < num_substreams; i += merge_arity) {
            Merge substream[i] .. substream[i+merge_arity-1];
        }
        num_substreams /= merge_arity;
        max_ss *= merge_arity;
    }

    Write single remaining substream to outstream;

    return AMI_ERROR_NO_ERROR;
}
```

4.6.1 Implementing Mergesort: An Extended Example

In the following we give an example of the implementation and use of a merge management object for merge sorting integers. We use merge sorting as a non-trivial example to illustrate the interfaces and mechanisms involved in using a merge management object. However, the reader should refer to Section 4.8 on sorting for a more straightforward and efficient way to sort with TPIE.

First, we declare the class:

[11]: LA: We should probably change AMI_merge_base to AMI_merge_object at some point

```
class MergeMgr : public AMI_merge_base<int> {
private:
    arity_t input_arity;
    pqueue  *pq;
public:
    MergeMgr (void);
    virtual ~MergeMgr  (void);
    AMI_err initialize (arity_t arity, const int * const *in,
                        AMI_merge_flag *taken_flags, int &taken_index);
    AMI_err operate    (const int * const *in, AMI_merge_flag *taken_flags,
                        int &taken_index, int *out);
    AMI_err main_mem_operate      (int* mm_stream, size_t len);
    size_t  space_usage_overhead  (void);
    size_t  space_usage_per_stream(void);
};
```

In addition to the standard class members for a merge management object, we have the following:

`input_arity` The number of input streams the merge management object must handle.

`pq` A priority queue into which items will be placed.

`MergeMgr()` A constructor.

`~MergeMgr()` A destructor.

Construction and destruction are fairly straightforward. At construction time, we have no priority queue because we do not yet know how big the priority queue should be. `pq` will be set up when `initialize` is called. The destructor checks whether `pq` is valid, and deletes it if it is. The constructor and destructor are implemented as follows:

```
MergeMgr::MergeMgr(void)
{
    pq = NULL;
}

MergeMgr::~MergeMgr(void)
{
    if (pq != NULL) {
        delete pq;
    }
}
```

When `AMI_partition_and_merge()` is invoked it calls the member functions `space_usage_overhead()` and `space_usage_per_stream()` of the merge management object (`MergeMgr` in this case). These return the number of bytes of main memory that the merge management object will allocate when initialized. In our example, the return value from `space_usage_overhead()` indicates that space will needed for a priority queue, and the return value from `space_usage_per_stream()` indicates that space will be needed for an object of type `int` and one of type `arity` associated with each stream.

```
size_t MergeMgr::space_usage_overhead(void)
{
    return sizeof(pqueue<arity_t,int>);
}


size_t MergeMgr::space_usage_per_stream(void)
{
    return sizeof(arity_t) + sizeof(int);
}
```

As an early step in its processing, `AMI_partition_and_merge()` will divide the input stream into "memoryloads" (which fit into main memory). It then calls the member function `main_mem_operate()` of the merge management object to perform application specific processing on these memoryloads. Since we are sorting in this example, we simply sort each memoryload via quicksort. The sorted memoryloads are then stored on the disks as substreams.

```
AMI_err MergeMgr::main_mem_operate(int* mm_stream, size_t len)
{
    qsort(mm_stream, len, sizeof(int), c_int_cmp);
    return AMI_ERROR_NO_ERROR;
}
```

Having sorted all of the initial substreams, `AMI_partition_and_merge()` begins to merge them. Before merging a set of substreams, the merge management object's member function `initialize()` is called to inform the merge management object of the number of streams it should be prepared to handle at the merge step. The object is also provided with the first object from each of the streams to be merged. In our example the `initialize()` member function is as follows:

```
AMI_err MergeMgr::initialize(arity_t arity, const int * const *in,
                                    AMI_merge_flag *taken_flags,
                                    int &taken_index)
{
    input_arity = arity;

    if (pq != NULL) {
        delete pq;
    }

    // Construct a priority queue that can hold arity items.
    pq = new pqueue_heap_op(arity);

    for (arity_t ii = arity; ii--; ) {
        if (in[ii] != NULL) {
            taken_flags[ii] = 1;
            pq->insert(ii,*in[ii]);
        } else {
            taken_flags[ii] = 0;
        }
    }

    taken_index = -1;
    return AMI_MERGE_READ_MULTIPLE;
}
```

Note the use of the return value `AMI_MERGE_READ_MULTIPLE`. This indicates that the flags in `*taken_flags` are set to indicate which of the inputs were used (and should not be presented again). This is very similar to the use of input flags to indicate which inputs were used by a scan management object as described in Section 4.5.4. The reason that we have a special return value to indicate when these flags are used is to increase performance. In order for `AMI_scan()` to determine which inputs were taken, it must examine all the flags. In a many way merge, this might be time consuming. In cases where only one item is taken, its index can be returned in `taken_index` in order to save the time that would be spent scanning the flags. This technique is illustrated in our `operate()` member function, below.

```
AMI_err MergeMgr::operate(const int * const *in,
                                  AMI_merge_flag *taken_flags,
                                  int &taken_index,
                                  int *out)
{
    // If the queue is empty, we are done.  There should be no more
    // inputs.
    if (!pq->num_elts()) {
        return AMI_MERGE_DONE;
    } else {
        arity_t min_source;
        int min_t;

        pq->extract_min(min_source,min_t);
        *out = min_t;
        if (in[min_source] != NULL) {
            pq->insert(min_source,*in[min_source]);
            taken_index = min_source;
        } else {
            taken_index = -1;
        }
        return AMI_MERGE_OUTPUT;
    }
}
```

[12]: LA: Something here?

4.7 Distribution

<TO BE WRITTEN>

[13]: LA: We should look at the kb_sort stuff and get it cleaned-up/done

4.8 Sorting

4.8.1 Comparison Sorting

Sorting is a common primitive operation in many algorithms. It can be performed in a variety of ways. The two most basic approaches for external memory sorting are based on merging (See Section 4.6), and distribution (See Section 4.7). TPIE currently provides several efficient sorting functions based on merging. In the future a number of other sorting algorithms may be implemented and it is the intention that when calling `AMI_sort()`, TPIE should automatically select the best algorithm for the given hardware platform.

4.8.2 Merge Sorting

While the `AMI_merge` example in Section 4.6 was not intended as an illustration of how to sort in TPIE, it contains the main ideas of how merge sorting should be done in order to achieve I/O optimality, and how it is done internally by TPIE's merge sort services.

Merge sort consists of two phases: the run formation phase and the merging phase. During the *run formation phase*, the N input elements are input M (one *memory-load*) at a time; each memory-load is sorted and written to the disks as a "run". In the *merge phase*, the sorted runs are merged together approximately M/B at a time (where M is the internal memory size and B is the block size) in a round-robin manner until a single sorted run remains. Typically, a heap or similar data structure is used during the merge phase to select the next record to be output from the set of records presented by the sorted runs being merged.

Currently, TPIE offers three merge sorting variants. The user must decide which variant is most appropriate for their circumstances. All accomplish the same goal, but the performance can vary depending on the situation. They differ mainly in the way they perform the merge phase of merge sort, specifically how they maintain their heap data structure used in the merge phase. The three variants are as follows:

- `AMI_sort`: keeps the (entire) first record of each sorted run (each is a stream) in a heap. This approach is most suitable when the record consists entirely of the record key.
- `AMI_ptr_sort`: keeps a pointer to the first record of each stream in the heap. This approach works best when records are very long and the key field(s) take up a large percentage of the record.
- `AMI_key_sort`: keeps the key field(s) and a pointer to the first record of each stream in the heap. This approach works best when the key field(s) are small in comparison to the record size.

Any of these variants will accomplish the task of sorting an input stream in an I/O efficient way, but there can be noticeable differences in processing time between the variants. As an example, `AMI_key_sort` appears to be more cache-efficient than the others in many cases, and therefore often uses less processor time, despite extra data movement relative to `AMI_ptr_sort`.

In addition to the three variants discussed above, there are multiple choices within each variant regarding how the actual comparison operations are to be performed. These choices are described in detail for `AMI_sort`, below.

AMI_sort()

`AMI_sort()` has three polymorphs, described below. We will refer to these as the (1) comparison operator, (2) comparison function, and (3) comparison object versions of `AMI_sort`. The comparison operator version tends to be the fastest and most straightforward to use. The comparison object version is comparable in speed (maybe slightly slower), but somewhat more flexible, as it can support multiple, different sorts on the same keys. The comparison function version is slightly easier to use than the comparison object version, but typically it is measureably slower.

Comparison operator version: This version works on streams of objects for which the operator `<` is defined. For example, the following code would sort a stream `instream` of `int` objects, creating the sorted stream `outstream`.

```
AMI_STREAM<int> instream;
AMI_STREAM<int> outstream;

void f()
{
    AMI_sort(&instream, &outstream);
}
```

Comparison function version: This version uses an explicit function to determine the relative order of two objects in the input stream. This is useful in cases where we may want to sort a stream of objects in several different ways. For example, the following code sorts a stream of complex numbers in two ways, by their real parts and by their imaginary parts.

```
class complex {
public:
    complex(double real_part, imaginary_part);
    double re(void);
    double im(void);
    ...
};

int compare_re(const complex &c1, const complex &c2)
{
    return (c1.re() < c2.re()) ? -1 :
           ((c1.re() > c2.re()) ? 1 : 0);
}

int compare_im(const complex &c1, const complex &c2)
{
    return (c1.im() < c2.im()) ? -1 :
           ((c1.im() > c2.im()) ? 1 : 0);
}

AMI_STREAM<complex> instream;
AMI_STREAM<complex> outstream_re;
AMI_STREAM<complex> outstream_im;

void f()
{
    AMI_sort(&instream, &outstream_re, compare_re);
    AMI_sort(&instream, &outstream_im, compare_im);
}
```

Comparison object version: This version of AMI_sort() is similar to the comparison function version, except that the comparison function is now a method of a user-defined comparison class. This polymorph of AMI_sort expects an object as its third argument. This object must have a public member function named compare, whose calling sequence and functionality is the same as for a comparison function (as described above). The complex number example might be solved using the comparison class version of AMI_sort as follows:[14]

[14]: LA: Check that this is correct.

```
class compare_re_class {
public:
    int compare ( const complex &c1, const complex &c2 ) {
        return (c1.re() < c2.re()) ? -1 :
               ((c1.re() > c2.re()) ? 1 : 0);
    };
};

class compare_im_class {
public:
    int compare ( const complex &c1, const complex &c2 ) {
        return (c1.im() < c2.im()) ? -1 :
               ((c1.im() > c2.im()) ? 1 : 0);
    };
};

AMI_STREAM<complex> instream;
AMI_STREAM<complex> outstream_re;
AMI_STREAM<complex> outstream_im;

compare_re_class compare_re
compare_im_class compare_im

void f()
{
    AMI_sort(&instream, &outstream_re, &compare_re);
    AMI_sort(&instream, &outstream_im, &compare_im);
}
```

AMI_ptr_sort()

The AMI_ptr_sort variant of merge sort in TPIE keeps only a pointer to each record in the heap used to perform merging of runs. Similar to AMI_sort above, it offers comparison operator, comparison function, and comparison class polymorphs. The syntax is identical to that illustrated in the AMI_sort examples; simply replace AMI_sort by AMI_ptr_sort.

AMI_key_sort()

The AMI_key_sort variant of TPIE merge sort keeps the key field(s) plus a pointer to the corresponding record in an internal heap during the merging phase of merge sort. It requires a sort management object with member functions compare and copy. The usage of AMI_key_sort() is illustrated by the following example:

Consider the class rectangle below, meant to describe axis-parallel rectangles,

```
class rectangle{
double northEast_x;
double northEast_y;
double southWest_x;
double southWest_y;
char   long_text_description[200];
}
```

and suppose that we want to sort a stream of rectangles in descending order according to their southWest_y coordinate. The user-written sort management object smo below contains a member function copy for copying the desired key field from a record (whose address will be provided by TPIE) to a location in the heap (determined by TPIE).

```
// Here is the definition of the sort management class
class SortManager {
private:
    int result;
public:
   inline int compare (const double & k1, const double & k2) {
      return ((k1 < k2)? -1 : (k1 > k2) ? 1 : 0);
   }
   inline void copy (double *key, const rectangle &record) {
      *key = record.southwest_y;
   }
};

// create a sort management object
SortManager <rectangle,double> smo;
```

Assuming that the size of each double is 8 bytes, we can sort the stream as follows:

```
AMI_STREAM<rectangle> instream;
AMI_STREAM<rectangle> outstream;
double dummyKey;

void f()
{
    AMI_key_sort(&instream, &outstream, dummyKey, &smo );
}
```

The third argument of AMI_key_sort() is a a dummy argument having the same type as the key field, and the fourth argument is the sort management object[15].

[15]: LA: Maybe we should add something about this being a C++ requirement?

4.8.3 Key Bucket Sorting

<TO BE WRITTEN>

4.9 Permutation

4.9.1 General Permutation

Permutation is a basic building block in many I/O algorithms. Routing a general permutation in the I/O model is asymptotically as complex as sorting [6], though for some important classes of permutations, such as BMMC permutations (See Section 4.9.2) faster algorithms are possible [21]. In this section, we discuss `AMI_general_permute()`, which routes arbitrary permutations, but always takes as long as sorting, regardless of whether the particular permutation can be done more quickly or not.

General permutations are routed using the function `AMI_general_permute()`. Like other AMI functions, `AMI_general_permute()` relies on an operation management object to determine its precise behavior. Unlike functions covered up to now, however, the type of the operation management object need not depend on the type of object in the stream being permuted.

A general permutation management object must provide two member functions, `initialize()` and `destination()`. `initialize()` is called to inform the general permutation object of the length of the stream to be permuted. `destination()` is then called repeatedly to determine the destination for each object in the stream based on it's initial position.

Here is an example of using general permutation to reverse the order of the items in a stream.

```
class reverse_order : public AMI_gen_perm_object {
private:
    off_t total_size;
public:
    AMI_error initialize(off_t ts) {
        total_size = ts;
        return AMI_ERROR_NO_ERROR;
    };
    off_t destination(off_t source) {
        return total_size - 1 - source;
    };
};

AMI_STREAM<int> amis0, amis1;

void f()
{
    reverse_order ro;

    AMI_general_permute(&amis0, &amis1, (AMI_gen_perm_object *)&ro);
}
```

4.9.2 Bit Permutation

[16]: LA: Do we really want this in the tutorial?

Bit permuting is a permutation technique in which the destination address of a given item is computed by manipulating the bits of its source address. The particular class of bit permutations that TPIE supports is the set of bit matrix multiply complement (BMMC) permutations. These permutations are defined on sets of objects whose size is a power of 2.

Suppose we have an input consisting of $N = 2^n$ objects. A BMMC permutation on the input is defined by a nonsingular $n \times n$ bit matrix A and an n element column vector c of bits. Source and destination addresses are interpreted as column vectors of bits, with the low order bit of the address at the top. The destination address x' corresponding to a given source address x is computed as

$$x' = Ax + c$$

[17]: DH: we should spell out what GF(2) is.

where addition and multiplication of matrix elements is done over GF(2)[17]. For a detailed description of BMMC permutations, see [23].

[18]: LA: Is it really implemented?

Routing BMMC permutations in TPIE is done using the `AMI_BMMC_permute()` entry point[18], which takes an input stream, and output stream, and a pointer to a bit permutation management object. In the following example, we route a permutation that simply reverses the order of the source address bits to produce the destination address.

First, we construct the matrices the permutation will use.

```
bit_matrix A(n,n);
bit_matrix c(n,1);

{
    unsigned int ii,jj;

    for (ii = n; ii--; ) {
        c[ii][0] = 0;
        for (jj = n; jj--; ) {
            A[n-1-ii][jj] = (ii == jj);
        }
    }
}
```

Now we simply construct a permutation management object from the matrices and perform the permutation.

```
AMI_bit_perm_object bpo(A,c);

ae = AMI_BMMC_permute(&amis0, &amis1, (AMI_bit_perm_object *)&bpo);
```

4.10 Distribution Sweeping

[19]: LA: Get sweeping under distribution in the index

<TO BE WRITTEN>

4.11 Matrix Operations

[20]: LA: Read through this and check code.

In addition to streams, which are linearly ordered collections of objects, the AMI provides a mechanism for storing large matrices in external memory. Matrices are a subclass of streams, and can thus be used with any of the stream operations discussed above. When a matrix is treated as a stream its elements appear in row major order. In addition to stream operations, matrices support three simple arithmetic operations, addition, subtraction, and multiplication.

It is assumed that the class `T` of the elements in a matrix forms a quasiring with the operators `+` and `*`. Furthermore, the object `T((int)0)` is assumed to be an identity for `+`. At the moment, it is not assumed that the operator `-` is an inverse of `+`, and therefore no reduced complexity matrix multiplication algorithms analogous to Strassen's algorithm are used.

TPIE provides support for both dense and sparse matrices.

4.11.1 Dense Matrix Operations

Dense matrices are implemented by the templated class `AMI_matrix`, which is a subclass of `AMI_STREAM`. Dense matrices can be initialized or "filled" using `AMI_scan()`, though typically they are filled using the function `AMI_matrix_fill()`. `AMI_matrix_fill()` wuses a scan management object whose member function `element` is given the row and column of each element of the matrix and must return the value to be inserted at that position of the matrix. In the following example, we create a 1000 by 1000 upper triangular matrix of ones and zeroes:

```
template<class T>
class fill_upper_tri : public AMI_matrix_filler<T> {
    AMI_err initialize(unsigned int rows, unsigned int cols)
    {
        return AMI_ERROR_NO_ERROR;
    };
    T element(unsigned int row, unsigned int col)
    {
        return (row <= col) ? T(1) : T(0);
    };
};

AMI_matrix m(1000, 1000);

void f()
{
    fill_upper_tri<double> fut;

    AMI_matrix_fill(&em, (AMI_matrix_filler<T> *)&fut);
}
```

Arithmetic on dense matrices is performed in a straightforward way using the functions `AMI_matrix_add()`, `AMI_matrix_subtract()`, and `AMI_matrix_multiply()`, as is the following example:

```
AMI_matrix m0(1000, 500), m1(500, 2000), m2(1000, 2000);
AMI_matrix m3(1000, 500), m4(1000, 500);

void f()
{
    // Add m3 to m4 and put the result in m0.
    AMI_matrix_add(em3, em4, em0);

    // Multiply m0 by em1 to get m2.
    AMI_matrix_mult(em0, em1, em2);

    // Subtract m4 from m3 and put the result in m0.
    AMI_matrix_subtract(em3, em4, em0);
}
```

4.11.2 Sparse Matrix Operations

<TO BE WRITTEN>

4.11.3 Elementwise Arithmetic

The functions `AMI_matrix_add()` and `AMI_matrix_subtract()` defined in Section 4.11.1 perform elementwise arithmetic on matrices. At times, we might also wish to perform elementwise multiplication or division, or perform a scalar arithmetic operation on all elements of a matrix. TPIE provides mechanisms for doing this not only on matrices, but on arbitrary streams, so long as they are of objects for which the appropriate arithmetic operators (i.e. `+`, `-`, `*`, `/`) are defined.

Elementwise arithmetic is done with scan management objects of the classes `AMI_scan_add`, `AMI_scan_sub`, `AMI_scan_mult` and `AMI_scan_div`. Here is an example that performs elementwise division on the elements of two streams.

```
#include <ami_stream_arith.h>

void foo()
{
    AMI_STREAM<int> amis0;
    AMI_STREAM<int> amis1;
    AMI_STREAM<int> amis2;

    AMI_scan_div<int> sd;

    // Divide each element of amis0 by the corresponding element of
    // amis1 and put the result in amis2.
    AMI_scan(&amis0, &amis1, &sd, &amis2);
}
```

4.12 External Stack

[21]: LA: put stack, external in index

<TO BE WRITTEN>

4.13 Compiling and Executing a TPIE Program

The fragments of code presented in this tutorial are designed for instructive purposes but they are incomplete. In order to successfully compile, link, and run a complete TPIE application, some additional code and configuration is needed. The configuration and compilation of a TPIE program is discussed below. The recommended way for a novice TPIE programmer to learn how to write a complete TPIE application is to go through the sample program of Chapter 3 or to look at the source code provided in the `test` directory.

The main steps involved in setting up and running a TPIE program are as follows[22]:

[22]: DH: This mixes apples and oranges. Should rewrite with more focus, e.g compiling a TPIE Hello World program?

1. Various behaviors of TPIE at run time can be controlled by compile-time variables. These variables are defined in the file `app_config.h` which is included at the beginning of a TPIE program before including any TPIE headers. The test application code distributed with TPIE contains such a file (`/test/app_config.h`). See Section 7 for a discussion of these options and of how they should be set on a given hardware platform for best performance.

2. TPIE's templated classes and functions are included by including the header file `ami.h` from the `include` directory. Normally, this directory is indicated via the `-I` argument to the compiler.

3. The following statements are used to indicate that the TPIE memory manager `MM_manager` should restrict the internal memory that it uses (to the value of `mm_size` in this case).

```
   MM_manager.enforce_memory_limit ();
   MM_manager.set_memory_limit (mm_size);
```

Calling the `MM_manager` member function `enforce_memory_limit ()` indicates that the program should abort if the allocated internal memory exceeds a specified amount[23]. This amount is set by the member function `set_memory_limit (mm_size)`. Normally, this amount is the amount of physical main memory minus the main memory used by the operating system and other programs running on the machine. If `MM_manager.ignore_memory_limit ()` is called, the application will run with virtual memory like an ordinary non-TPIE application.

4. If a TPIE program file `foo.cpp` exists in the TPIE base directory it can be compiled with the following command:

```
g++ foo.cpp -Iinclude/ -Llib/ -ltpie -o foo
```

Users interested in setting up a `Makefile` for the compiling task can look at a sample `Makefile` in the `test` subdirectory.

[23]: LA: Something about how TPIE counts memory?

Part II

Reference

Chapter 5

TPIE Programmer's Reference

5.1 Registration-based Memory Manager

5.1.1 Files

`#include <mm_register.h>`
Note that there is no need to include this file when using the AMI entry points, since it is included by all AMI header files.

5.1.2 Class Declaration

```
class MM_register;
```

5.1.3 Global Variables

`MM_register` **`MM_manager`**`;`
This is the only instance of the `MM_register` class that should exist in a program.

5.1.4 Description

The TPIE memory manager `MM_manager`, the only instance of class `MM_register`, traps memory allocation and deallocation requests in order to monitor and enforce memory usage limits. The actual memory allocation requests are done using the standard C++ operators `new` and `delete`, which have been replaced with in-house versions that interact with the memory manager.

5.1.5 Public Member Functions

`MM_err` **`enforce_memory_limit`**`();`
Instruct TPIE to abort computation when the memory limit is exceeded.

`MM_err` **`ignore_memory_limit`**`();`
Instruct TPIE to ignore the memory limit set using `set_memory_limit`.

`size_t` **`memory_available`**`();`
Return the number of bytes of memory which can be allocated before the user-specified limit is reached.

`size_t` **`memory_limit`**`();`

Return the memory limit as set by the last call to method `set_memory_limit`.

`size_t` **`memory_used`**`();`

Return the number of bytes of memory currently allocated.

`MM_err` **`set_memory_limit`**`(size_t size);`

Set the application's memory limit. The memory limit is set to `size` bytes. If the specified mamory limit is greater than or equal to the amount of memory already allocated, `set_memory_limit` returns `MM_ERROR_NO_ERROR`, otherwise it returns `MM_ERROR_EXCESSIVE_ALLOCATION`. By default, successive calls to operator `new` will cause the program to abort if the resulting memory usage would exceed `size` bytes. This behaviour can be controlled explicitly by the use of methods `enforce_memory_limit`, `warn_memory_limit` and `ignore_memory_limit`.

`int` **`space_overhead`**`();`

TPIE imposes a small space overhead on each memory allocation request received by operator `new`. This involves increasing each allocation request by a fixed number of bytes. The precise size of this increase is machine dependent, but typically 8 bytes. Method `space_overhead` returns the size of this increase.

`MM_err` **`warn_memory_limit`**`();`

Instruct TPIE to issue a warning when the memory limit is exceeded.

5.2 Streams

5.2.1 Files

```
#include <ami_stream.h>
```

5.2.2 Class Declaration

`template<class T> class` **`AMI_STREAM`**`;`

5.2.3 Description

An `AMI_STREAM<T>` object stores an ordered collection of objects of type `T` on external memory.

The stream type of an `AMI_STREAM` indicates what operations are permitted on the stream. An `AMI_STREAM<T>` object can have one of four different types:

- `AMI_READ_STREAM`: Input operations on the stream are permitted, but output is not permitted.
- `AMI_WRITE_STREAM`: Output operations are permitted, but input operations are not permitted.
- `AMI_APPEND_STREAM`: Output is appended to the end of the stream. Input operations are not permitted. This is similar to `AMI_WRITE_STREAM` except that if the stream is constructed on a file containing an existing stream, objects written to the stream will be appended at the end of the stream.-24-
- `AMI_READ_WRITE_STREAM`: Both input and output operations are permitted.

[24]: LA: It this true? DH: seems ok

5.2.4 Constructors, Destructor and Related Functions

AMI_STREAM();

A new stream of type AMI_READ_WRITE_STREAM is constructed on a file with a randomly generated name.

AMI_STREAM(const char *path_name);

A stream of type AMI_READ_WRITE_STREAM is constructed on the file whose path name is given. If the file does not already exist, a new stream is constructed on a newly created file with the specified file name. If the file already exists, it is checked if it contains a valid stream, and if so, the new stream is constructed on this file. If the file does not contain a valid stream, the status flag is set to AMI_STREAM_STATUS_INVALID.

AMI_STREAM(const char *path_name, AMI_stream_type st);

A stream of type st is constructed on the file whose pathname is given.

AMI_STREAM(BTE_STREAM<T> &bs);

A stream is constructed from an existing BTE_STREAM (see Section 6.2). This constructor will not normally be used by a TPIE application programmer. The new AMI_STREAM gets the same type as the BTE_STREAM.

~**AMI_STREAM**();

Destructor. Free the memory buffer and close the file. IF the persistence flag is PERSIST_DELETE, also remove the file.

AMI_err **new_substream**(AMI_stream_type st, off_t sub_begin, off_t sub_end, AMI_base_stream<T> **sub_stream);

A substream is an AMI stream that is part of another AMI stream. More precisely, a substream B of a stream A is defined as a contiguous range of objects from the ordered collection of objects that make up the stream A. If desired, one can construct substreams of substreams of substreams *ad infinitum.* Since a substream is a stream in its own right, many of the stream member functions can be applied to a substream. A substream can be created via the pseudo-constructor[1] new_substream(). Here, st specifies the type of the substream, and the offsets sub_begin and sub_end define the positions in the original stream A where the new substream B will begin and end. Upon completion, *sub_stream points to the newly created substream.

5.2.5 Public Member Functions

bool **operator!**() const;

Return true if the status of the stream is not AMI_STREAM_STATUS_VALID, false otherwise. See also is_valid() and status().

off_t **chunk_size**() const;

Return the maximum number of items (of type T) that can be stored in one block.

static const tpie_stats_stream& **gstats**();

Return an object containing the statistics of all streams opened by the application (global statistics). See also stats().

bool **is_valid**() const;

Return true if the status of the stream is AMI_STREAM_STATUS_VALID, false otherwise. See also status().

AMI_err **main_memory_usage**(size_t *usage, MM_stream_usage usage_type);

This function is used for obtaining the amount of main memory used by an `AMI_STREAM<T>` object (in bytes). `usage_type` is one of the following:

`MM_STREAM_USAGE_CURRENT` Total amount of memory currently used by the stream.

`MM_STREAM_USAGE_MAXIMUM` Max amount of memory that will ever be used by the stream.

`MM_STREAM_USAGE_OVERHEAD` The amount of memory used by the object itself, without the data buffer.

`MM_STREAM_USAGE_BUFFER` The amount of memory used by the data buffer.

`MM_STREAM_USAGE_SUBSTREAM` The additional amount of memory that will be used by each substream created.

`AMI_err` **`name`**`(char **stream_name);`
Store the path to the UNIX file name holding the stream, in newly allocated memory.

`void` **`persist`**`(persistence p)`
Set the persistence flag to `p`, which can have one of two values: `PERSIST_DELETE` and `PERSIST_PERSISTENT`.

`AMI_err` **`read_array`**`(T *mm_array, off_t *len);`
Read `*len` items from the current position of the stream into the array `mm_array`. The "current item" pointer is increased accordingly.

`AMI_err` **`read_item`**`(T **elt);`
Read the current item from the stream and advance the "current item" pointer to the next item. The item read is pointed to by `*elt`. If no error has occured, return `AMI_ERROR_NO_ERROR`. If the "current item" pointer is beyond the last item in the stream, return `AMI_ERROR_END_OF_STREAM`.

`AMI_err` **`seek`**`(off_t off);`
Move the current position to `off`.

`const tpie_stats_stream&` **`stats`**`() const;`
Return an object containing the statistics of this stream. The types of statistics computed for a collection are tabulated below. See also `gstats()`.

BLOCK_READ	Number of block reads
BLOCK_WRITE	Number of block writes
ITEM_READ	Number of item reads
ITEM_WRITE	Number of item writes
ITEM_SEEK	Number of item seek operations
STREAM_OPEN	Number of stream open operations
STREAM_CLOSE	Number of stream close operations
STREAM_CREATE	Number of stream create operations
STREAM_DELETE	Number of stream delete operations
SUBSTREAM_CREATE	Number of substream create operations
SUBSTREAM_DELETE	Number of substream delete operations

`AMI_stream_status` **`status`**`() const;`
Return the status of the stream instance. The result is either `AMI_STREAM_STATUS_VALID` or `AMI_STREAM_STATUS_INVALID`. The only operation that can leave the stream invalid is the constructor (if that happens, the log file contains more information). No items should be read from or written to an invalid stream.

`off_t` **`stream_len`**`(void);`
Return the number of items stored in the stream.

AMI_err **truncate**(off_t off);

Resize the stream to off items. If off is less than the number of objects in the stream, truncate() truncates the stream to off objects. If off is more than the number of objects in the stream, truncate() extends the stream to the specified number of objects. In either case, the "current item" pointer will be moved to the new end of the stream.

AMI_err **write_array**(const T *mm_array, off_t len);

Write len items from array mm_array to the stream, starting in the current position. The "current item" pointer is increased accordingly.

AMI_err **write_item**(const T &elt);

Write elt to the stream in the current position. Advance the "current item" pointer to the next item. If no error has occured, return AMI_ERROR_NO_ERROR.

5.3 Scanning

5.3.1 Files

```
#include <ami_scan.h>
```

5.3.2 Function Declaration

```
template<class T1, class T2, ..., class ST, class U1, class U2, ...> AMI_err
AMI_scan(AMI_STREAM<T1> *in1, AMI_STREAM<T2> *in2, ..., ST *smo, AMI_STREAM<U1> *out1,
AMI_STREAM<U2> *out2, ...);
```

5.3.3 Description

AMI_scan() reads zero, one or multiple input streams (up to four), each potentially of a different type, and writes zero, one or multiple output streams (up to four), each potentially of a different type. smo is a pointer to a *scan management object* of user-defined class ST, as described below.

5.3.4 Scan Management Objects

A scan management object class must inherit from AMI_scan_object:

```
template<class T1, class T2,..., class U1, class U2,...>
class ST: public AMI_scan_object;
```

In addition, it must provide two member functions for AMI_scan() to call: initialize() and operate().

```
AMI_err initialize(void);
```

Initializes a scan management object to prepare it for a scan. This member function is called once by each call to AMI_scan() in order to initialize the scan management object before any data processing takes place. This function should return AMI_ERROR_NO_ERROR if successful, or an appropriate error otherwise. See Section C.1 for a list of error codes.

Most of the work of a scan is typically done in the scan management object's operate() member function:

```
AMI_err operate(const T1 &in1, const T2 &in2,...,  AMI_SCAN_FLAG *sfin,
                U1 *out1, U2 *out2,..., AMI_SCAN_FLAG *sfout);
```

One or more input objects or one or more output parameters must be specified. These must correspond in number and type to the streams passed to the polymorph of `AMI_scan()` with which this scan management object is to be used.

If present, the inputs `*in1, ...` are application data items of type `T1`, and `sfin` points to an array of flags, one for each input. On entry to `operate()`, flags that are set (non-zero) indicate that the corresponding inputs contain data. If on exit from `operate()`, the input flags are left untouched, `AMI_scan()` assumes that the corresponding inputs were processed. If one or more input flags are cleared (set to zero) then `AMI_scan()` assumes that the corresponding inputs were not processed and should be presented again on the next call to `operate()`. This permits out of step scanning, as illustrated in Section 4.5.4.

If present, the outputs `*out1, ...` are application data items of type `U1`, and `sfout` points to an array of flags, one for each output. On exit from `operate()`, the outputs should contain any objects to be written to the output streams, and the output flags must be set to indicate to `AMI_scan()` which outputs are valid and should be written to the output streams.

The return value of `operate()` will normally be one of the following:

- `AMI_SCAN_CONTINUE`: indicates that the function should be called again with any "taken" inputs replaced by the next objects from their respective streams

- `AMI_SCAN_DONE`: indicates that the scan is complete and no more input needs to be processed.

Note that `operate()` is permitted to return `AMI_SCAN_CONTINUE` even when the input flags indicate that there is no more input to be processed. This is useful if the scan management object maintains some internal state that must be written out after all input has been processed.

Examples of the use of scan management objects are given in Section 4.5 as well as in the test applications that appear in the TPIE distribution.

5.4 Scanning from a C++ stream

5.4.1 Files

```
#include <ami_scan_utils.h>
```

5.4.2 Class Declaration

```
template<class T> class cxx_istream_scan;
```

5.4.3 Description

A scan management class template for reading the contents of an ordinary C++ input stream into a TPIE stream. It works with streams of any type for which a `>>` operator is defined for C++ stream input.

5.4.4 Constructor

`cxx_istream_scan``(istream *instr = &cin);`

Create a scan management object for scanning the contents of C++ stream `*instr`. The actual scanning is done using `AMI_scan` with no input streams and one output stream.

5.5 Scanning into a C++ stream

5.5.1 Files

```
#include <ami_scan_utils.h>
```

5.5.2 Class Declaration

```
template<class T> class cxx_ostream_scan;
```

5.5.3 Description

A scan management class template for writing the contents of a TPIE stream into an ordinary C++ output stream. It works with streams of any type for which a `<<` operator is defined for C++ stream output.

5.5.4 Constructor

```
cxx_ostream_scan(istream *outstr = &cout);
```

Create a scan management object for scanning into C++ stream `*outstr`. The actual scanning is done using `AMI_scan` with one input stream and no output streams.

5.6 Stream Merging

5.6.1 Files

```
#include <ami_merge.h>
```

5.6.2 Function Declarations

```
template<class T>
AMI_err AMI_merge(AMI_STREAM<T> **instreams, arity_t arity, AMI_STREAM<T> *outstream);
```

```
template<class T>
AMI_err AMI_merge(AMI_STREAM<T> **instreams, arity_t arity, AMI_STREAM<T> * outstream,
int (*cmp)(const T&, const T&));
```

```
template<class T>
AMI_err AMI_merge(AMI_STREAM<T> **instreams, arity_t arity, AMI_STREAM<T> * outstream,
CmpObj *co);
```

```
template<class T, class KEY>
AMI_err AMI_merge(AMI_STREAM<T> **instreams, arity_t arity, AMI_STREAM<T> *outstream,
int keyoff, KEY dummy);
```

5.6.3 Description

TPIE provides several merge entry points for merging sorted streams to produce a single, interleaved output stream. `AMI_merge` has four polymorphs, described below. We will refer to these as the (1) comparison operator, (2) comparison function, (3) comparison class and (4) key-based versions of `AMI_merge`. The comparison operator version tends to be the fastest and most straightforward to use. The comparison class version is comparable in speed (maybe slightly slower), but somewhat more flexible, as it can support multiple, different merges on the same keys. The comparison function version is slightly easier to use than the comparison class version, but typically it is measureably slower.

5.7 Generalized Stream Merging

5.7.1 Files

```
#include <ami_merge.h>
```

5.7.2 Function Declaration

```
template<class T, class MergeMgr>
AMI_err AMI_generalized_merge(AMI_STREAM<T> **instreams, arity_t arity, AMI_STREAM<T>
*outstream, MergeMgr *mo);
```

5.7.3 Description

TPIE entry point `AMI_generalized_merge()` allows an arbitrary number of streams to be merged into one stream in one pass, subject to the available main memory. TPIE will attempt to read the first block of each stream into the internal memory, and will update the contents of these buffers as the merge progresses. At least one block buffer is also required for the output stream from the merge. The function takes four arguments: `instreams` is an array of pointers to the input streams, all of which are of type `AMI_STREAM<T>`, `arity` is the number of input streams, `outstream` is the output stream, of type `AMI_STREAM<T>`, and `mo` points to a merge management object that controls the merge (merge management objects are described below).

If the merge cannot be completed in one pass due to insufficient memory, the function fails and it returns `AMI_ERROR_INSUFFICIENT_MAIN_MEMORY`. Otherwise, it returns `AMI_ERROR_NO_ERROR`.

5.7.4 Merge Management Objects

A merge management object class must inherit from `AMI_generalized_merge_base`:

```
template<class T>
class MergeMgr: public AMI_generalized_merge_base;
```

In addition, a merge management object must provide `initialize()` and `operate()` member functions, whose purposes are analogous to their namesakes for scan management objects.

The user's `initialize()` member function is called by the merge function once so that application-specific data structures (if any) can be initialized.

```
AMI_err initialize(arity_t arity, const T * const *in,
                   AMI_merge_flag *taken_flags,
                   int &taken_index);
```

where

- `arity` is the number of input streams in the merge,

- `in` is a pointer to an array of pointers to input objects, each of which is the first objects appearing in one of the input streams,
- `taken_flags` an array of flags indicating which of the inputs are present (i.e. which of the input streams is not empty), and a pointer to an output object.

The typical behavior of `initialize()` is to place all the input objects into a data structure and then return `AMI_MERGE_READ_MULTIPLE` to indicate that it used (and is now finished with) all of the inputs which were indicated to be valid by `taken_flags`. `initialize` need not process all inputs; it can turn off any flags in `taken_flags` corresponding to inputs that should be presented to `operate()`. Alternatively, it can set `taken_index` to the index of a single input it processed and return `AMI_MERGE_CONTINUE`.

When performing a merge, TPIE relies on the application programmer to provide code to determine the order of any two application data elements, and certain other application-specific processing. By convention, TPIE expects these decisions to be made by the `operate()` function:

```
AMI_err operate(const T * const *in, AMI_merge_flag *taken_flags,
                int &taken_index, T *out);
```

The `operate()` member function is called repeatedly to process input objects. Typically, `operate()` will choose a single input object to process, and set `taken_index` to the index of the pointer to that object in the input array. This object is then typically added to a dynamic data structure maintained by the merge management object. If output is generated, for example by removing an object from the dynamic data structure, `operate()` should return `AMI_MERGE_OUTPUT`, otherwise, it returns either `AMI_MERGE_CONTINUE` to indicate that more input should be presented, or `AMI_MERGE_DONE` to indicate that the merge has completed.

Alternatively, `operate()` can clear the elements of `taken_flags` that correspond to inputs it does not currently wish to process, and then return `AMI_MERGE_READ_MULTIPLE`. This is generally undesirable because, if only one input is taken, it is far slower than using `taken_index` to indicate which input was taken. The merge management object must clear all other flags, and then TPIE must test all the flags to see which inputs were or were not processed.

5.8 Stream Partitioning and Merging

5.8.1 Files

```
#include <ami_merge.h>
```

5.8.2 Function Declaration

```
template<class T>
AMI_err AMI_partition_and_merge(AMI_STREAM<T> **instreams, arity_t arity, AMI_STREAM<T>
*outstream);
```

```
template<class T>
AMI_err AMI_partition_and_merge(AMI_STREAM<T> **instreams, arity_t arity, AMI_STREAM<T>
*outstream, int (*cmp)(const T&, const T&));
```

```
template<class T, class KEY>
AMI_err AMI_partition_and_merge(AMI_STREAM<T> **instreams, arity_t arity, AMI_STREAM<T>
*outstream, int keyoffset, KEY dummy);
```

5.8.3 Description

Each of these functions partitions a stream into substreams small enough to fit in main memory, sorts them in main memory, and then merges them together, possibly in several passes if low memory conditions dictate. The difference between the three polymorphs is the comparison method: in the first polymorph, comparison is done using the comparison operator of class `T`; in the second polymorph, comparison is done using the comparison function `cmp`; in the third polymorph, comparison is done using a key of type `KEY`, extracted from objects of type `T` at byte offset `keyoffset`.

In order to complete the merge successfully, these functions need sufficient memory for a binary merge. If not enough memory is available, the function fails and it returns `AMI_ERROR_INSUFFICIENT_MAIN_MEMORY`. Otherwise, it returns `AMI_ERROR_NO_ERROR`.

5.9 Generalized Stream Partitioning and Merging

5.9.1 Files

```
#include <ami_merge.h>
```

5.9.2 Function Declaration

```
template<class T, class MergeMgr>
AMI_err AMI_generalized_partition_and_merge(AMI_STREAM<T> *instream, AMI_STREAM<T>
*outstream, MergeMgr *mo);
```

5.9.3 Description

This function partitions a stream into substreams small enough to fit in main memory, operates on each in main memory, and then merges them together, possibly in several passes if low memory conditions dictate. This function takes three arguments: `instream` points to the input stream, `outstream` points to the output stream, and `mo` points to a merge management object that controls the merge. This function takes care of all the details of determining how much main memory is available, how big the initial substreams can be, how many streams can be merged at a time, and how many levels of merging must take place.

In order to complete the merge successfully, the function needs sufficient memory for a binary merge. If not enough memory is available, the function fails and it returns `AMI_ERROR_INSUFFICIENT_MAIN_MEMORY`. Otherwise, it returns `AMI_ERROR_NO_ERROR`.

5.9.4 Merge Management Objects

The `AMI_partition_and_generalized_merge()` entry point requires a merge management object similar to the one described in Section 5.7.4. The following three additional member functions must also be provided.

-
  ```
  AMI_err main_mem_operate(T* mm_stream, size_t len);
  ```

  where

  - `mm_stream` is a pointer to an array of objects that have been read into main memory,
  - `len` is the number of objects in the array.

  This function is called by `AMI_partition_and_merge()` when a substream of the data is small enough to fit into main memory, and the (application-specific) processing of this subset of the data can therefore be completed in internal memory.

-
  ```
  size_t space_usage_per_stream(void);
  ```

  This function should return the amount of main memory that the merge management object will need per per input stream. Merge management objects are allowed to maintain data structures whose size is linear in the number of input streams being processed.

-
  ```
  size_t space_usage_overhead(void);
  ```

  This function should return an upper bound on the number of bytes of main memory-26- the merge management object will allocate in addition to the portion that is linear in the number of streams.

[25]: LA: Why do we have it then? DH: for completeness

5.10 Merge Sorting

5.10.1 Files

```
#include <ami_sort.h>
```

5.10.2 Function Declarations

```
template<class T>
AMI_err AMI_sort(AMI_STREAM<T> *instream, AMI_STREAM<T> *outstream);
```

```
template<class T>
AMI_err AMI_sort(AMI_STREAM<T> *instream, AMI_STREAM<T> *outstream, int (*cmp)(const
T&, const T&));
```

```
template<class T, class CMPR>
AMI_err AMI_sort(AMI_STREAM<T> *instream, AMI_STREAM<T> *outstream, CMPR *cmp);
```

```
template<class T>
AMI_err AMI_ptr_sort(AMI_STREAM<T> *instream, AMI_STREAM<T> *outstream);
```

```
template<class T>
AMI_err AMI_ptr_sort(AMI_STREAM<T> *instream, AMI_STREAM<T> *outstream, int (*cmp)(const
T&, const T&));
```

```
template<class T, class CMPR>
AMI_err AMI_ptr_sort(AMI_STREAM<T> *instream, AMI_STREAM<T> *outstream, CMPR *cmp);
```

```
template<class T, class KEY, class CMPR>
AMI_err AMI_key_sort(AMI_STREAM<T> *instream, AMI_STREAM<T> *outstream, KEY dummykey,
CMPR *cmp) ;
```

5.10.3 Description

TPIE offers several entry points for sorting which use merging as their underlying paradigm. Please see Section 4.8.2 for more details of this approach.

Currently, TPIE offers three merge sorting variants. The user must decide which variant is most appropriate for their circumstances. All accomplish the same goal, but the performance can vary depending on the situation. They differ mainly in the way they perform the merge phase of merge sort, specifically how they maintain their heap data structure used in the merge phase. The three variants are as follows:

- `AMI_sort`: keeps the (entire) first record of each sorted run (each is a stream) in a heap. This approach is most suitable when the record consists entirely of the record key.
- `AMI_ptr_sort`: keeps a pointer to the first record of each stream in the heap. This approach works best when records are very long and the key field(s) take up a large percentage of the record.
- `AMI_key_sort`: keeps the key field(s) and a pointer to the first record of each stream in the heap. This approach works best when the key field(s) are small in comparison to the record size.

Any of these variants will accomplish the task of sorting an input stream in an I/O efficient way, but there can be noticeable differences in processing time between the variants. As an example, `AMI_key_sort` appears to be more cache-efficient than the others in many cases, and therefore often uses less processor time, despite extra data movement relative to `AMI_ptr_sort`.

In addition to the three variants discussed above, there are multiple choices within each variant regarding how the actual comparison operations are to be performed. These choices are described in detail for `AMI_sort`, below.

AMI_sort()

`AMI_sort()` has three polymorphs, described below. We will refer to these as the (1) comparison operator, (2) comparison function, and (3) comparison class versions of `AMI_sort`. The comparison operator version tends to be the fastest and most straightforward to use. The comparison class version is comparable in speed (maybe slightly slower), but somewhat more flexible, as it can support multiple, different sorts on the same keys. The comparison function version is slightly easier to use than the comparison object version, but typically it is measureably slower. Please refer to Section 4.8.2 for examples of the use of these versions of `AMI_sort()`.

Comparison operator version. This version works on streams of objects for which the operator `<` is defined.

Comparison function version. This version uses an explicit function to determine the relative order of two objects in the input stream. The function takes two arguments of type `T` and returns -1, 0, or 1, if the first object is less than, equal or greater than the second object in the desired order. This is useful in cases where we may want to sort a stream of objects in several different ways.

Comparison class version. This version of `AMI_sort()` is similar to the comparison function version, except that the comparison function is now a method of a user-defined comparison object. This object must have a public member function named `compare`, having the following prototype:

```
inline int compare (const KEY & k1, const KEY & k2);
```

The user-written `compare` function computes the order of the two user-defined keys `k1` and `k2`, and returns -1, 0, or $+1$ to indicate that $k1 < k2$, $k1 == k2$, or $k1 > k2$ respectively. It will be called by the internals of `AMI_key_sort` to determine the relative order of records during the sort.

AMI_ptr_sort()

The `AMI_ptr_sort` variant of merge sort in TPIE keeps only a pointer to each record in the heap used to perform merging of runs. Similar to `AMI_sort` above, it offers comparison operator, comparison function, and comparison class polymorphs. The syntax is identical to that illustrated in the `AMI_sort` examples; simply replace `AMI_sort` by `AMI_ptr_sort`.

AMI_key_sort()

The `AMI_key_sort` variant of TPIE merge sort keeps the key field(s) plus a pointer to the corresponding record in an internal heap during the merging phase of merge sort. It requires a sort management object with member functions `compare` and `copy`. The `dummyKey` argument of `AMI_key_sort()` is a a dummy argument having the same type as the user key, and `*smo` is the sort management object, having user-defined `compare` and `copy` member functions as described below.

The `compare` member function has the following prototype:

```
inline int compare (const KEY & k1, const KEY & k2);
```

The user-written `compare` function computes the order of the two user-defined keys `k1` and `k2`, and returns −1, 0, or +1 to indicate that $k1 < k2$, $k1 == k2$, or $k1 > k2$ respectively. It will be called by the internals of `AMI_key_sort` to determine the relative order of records during the sort.

The `copy` member function has the following prototype:

```
inline void copy (KEY *key, const T &record);
```

The user-written `copy` function constructs the user-defined key `*key` from the contents of the user-defined record `record`. It will be called by the internals of `AMI_key_sort` to make copies of record keys as necessary during the sort.

5.11 Internal Memory Sorting

5.11.1 Files

```
#include <quicksort.h>
```

5.11.2 Function Declarations

```
template<class T>
void quick_sort_op(T *data, size_t len);

template<class T>
void quick_sort_cmp(T *data, size_t len, int (*cmp)(const T&, const T&));

template<class T>
void quick_sort_obj(T *data, size_t len, CMPR *cmp);
```

5.11.3 Description

These are internal memory in-place sorting routines that implement the quicksort algorithm (randomized). These routines are used by the external memory sorting routines (see Section 5.10) on streams that are small enough to fit in memory. The three polymorphs use different comparison methods: `quick_sort_op` uses the comparison operator <, `quick_sort_cmp` uses the comparison function `cmp`, and `quick_sort_obj` uses a comparison object of type `CMPR`.

5.12 Stacks

5.12.1 Files

```
#include <ami_stack.h>
```

5.12.2 Class Declaration

```
template<class T> class AMI_stack;
```

5.12.3 Description

External stacks are implemented through the templated class `AMI_stack<T>`, which is a subclass of `AMI_STREAM<T>`. As a consequence, it inherits all public members of `AMI_STREAM<T>`, including its constructors. See Section 5.2.

5.12.4 Public Member Functions

`AMI_err` **`push`**`(const &T t);`
: Insert a copy of the object `t` to the top of the stack, increasing its length by one.

`AMI_err` **`pop`**`(T **ppt);`
: Remove the top object from the stack, decreasing its length by one and returning the address of a pointer to the popped object in `ppt`.

5.13 Blocks

5.13.1 Files

```
#include <ami_block.h>
```

5.13.2 Class Declaration

`template<class E, class I> class` **`AMI_block`**`;`
: The types `E` and `I` should have a default constructor, a copy constructor and an assignment operator. The size returned by `sizeof(E)` and `sizeof(I)` should be the total size of the items copied by the copy constructor/assignment operator.

5.13.3 Description

An instance of class `AMI_block<E,I>` is a typed view of a logical block, which is the unit amount of data transfered between external storage and main memory.

The `AMI_block` class serves a dual purpose: (a) to provide an interface for seamless transfer of blocks between disk and main memory, and (b) to provide a structured access to the contents of the block. The first purpose is achieved through internal mechanisms, transparent to the user. When creating an instance of class `AMI_block`, the constructor is responsible for making the contents of the block avilable in main memory. When the object is deleted, the destructor is responsible for writing back the data, if necessary, and freeing the memory. Consequently, during the life of an `AMI_block` object, the contents of the block is available in main memory. The second purpose is achieved by partitioning the contents of the block into three fields:

Links: an array of pointers to other blocks, represented as block identifiers, of type `AMI_bid`;

Elements: an array of elements of parameter type `E`;

Info: an info field of parameter type `I`, used to store a constant amount of administrative data;

The number of elements and links that can be stored is set during construction: the number of links is passed to the constructor, and the number of elements is computed using the following formula:

$$\text{number\_of\_elements} = \left\lfloor \frac{\text{block\_size} - (\texttt{sizeof}(I) + \texttt{sizeof(AMI\_bid)} * \text{number\_of\_links})}{\texttt{sizeof}(E)} \right\rfloor$$

5.13.4 Constructors and Destructor

`AMI_block``(AMI_COLLECTION *pcoll, unsigned int l, AMI_bid bid);`

Read the block with id `bid` from block collection `*pcoll` in newly allocated memory and format it using the template types and the maximum number of links `l`. Persistency is set to `PERSIST_PERSISTENT`.

`AMI_block``(AMI_COLLECTION *pcoll, unsigned int l);`

Create a new block in collection `*pcoll`, allocate memory for it, and format it using the template types and the maximum number of links `l`. Persistency is set to `PERSIST_PERSISTENT`. The id of the block can be inquired using the access member function `bid()`.

∼**`AMI_block`**`();`

Destructor. If persistency is `PERSIST_DELETE`, remove the block from the collection. If it is `PERSIST_PERSISTENT`, write the block to the collection. Deallocate the memory.

5.13.5 Public Member Objects

`b_vector<E>` **`el`**`;`

Access to the elements is done through this object, using the public methods of the `b_vector` class (described below).

`b_vector<AMI_bid>` **`lk`**`;`

Access to the links is done through this object, using the public methods of the `b_vector` class (described below).

5.13.6 Public Member Functions

`AMI_block<E,I>&` **`operator=`**`(AMI_block<E,I>& B);`

Copy block `B` into the current block, if both blocks are associated with the same collection. Returns a reference to this block.

`bool` **`operator!`**`() const;`

Return `true` if the block's status is not `AMI_BLOCK_STATUS_VALID`. See also `is_valid()` and `status()`.

`AMI_bid` **`bid`**`() const;`

Return the block id.

`size_t` **`block_size`**`() const;`

Return the size of this block in bytes.

`char&` **`dirty`**`();`

Return a reference to the dirty bit. The dirty bit is used to optimize writing in some implementations of the block collection class. It should be set to 1 whenever the block data is modified. See the implementation details for more.

`char` **`dirty`**`() const;`

Return the value of the dirty bit.

static size_t **el_capacity**(size_t bsz, size_t l);

Return the capacity of the el vector of a block with size bsz and number of links l.

I ***info**();

Return a pointer to the info element.

const I* **info**() const;

Return a const pointer to the info element.

bool **is_valid**() const;

Return true if the block's status is AMI_BLOCK_STATUS_VALID. See also status().

void **persist**(persistence p);

Set the persistency flag to p. The possible values for p are PERSIST_PERSISTENT and PERSIST_DELETE.

persistence **persist**() const;

Return the value of the persistency flag.

AMI_block_status **status**() const;

Return the status of the block. The result is either AMI_BLOCK_STATUS_VALID or AMI_BLOCK_STATUS_INVALID. The status of an AMI_block instance is set during construction. The methods of an invalid block can give erroneous results or fail.

AMI_err **sync**();

Synchronize the in-memory image of the block with the one stored in external storage.

5.13.7 The b_vector class

The b_vector class stores an array of objects of a templated type T. It has a fixed maximum size, or capacity, which is set during construction (since instances of this class are created only by the AMI_block class, the constructors are not part of the public interface). The items stored can be accessed through the array operator.

Class Declaration

template<class T> class **b_vector**;

The type T should have a default constructor, as well as copy constructor and assignment operator.

Member Functions

T& **operator[]**(size_t i);

Return a reference to the *i*th item.

const T& **operator[]**(size_t i) const;

Return a const reference to the *i*th item.

size_t **capacity**() const;

Return the capacity (i.e., maximum number of T elements) of this b_vector.

size_t **copy**(size_t start, size_t length, b_vector<T>& src, size_t src_start = 0);

Copy length items from the src vector, starting with item src_start, to this vector, starting with item start. Return the number of items copied. Source can be *this.

size_t **copy**(size_t start, size_t length, const T* src);

Copy length items from the array src to this vector, starting in position start. Return the number of items copied.

`void` **`insert`**`(const T& t, size_t pos);`
: Insert item `t` in position `pos`; all items from position `pos` onward are shifted one position higher; the last item is lost.

`void` **`erase`**`(size_t pos);`
: Erase the item in position `pos` and shift all items from position `pos+1` onward one position lower; the last item becomes identical with the next to last item.

5.14 Block Collections

5.14.1 Files

```
#include <ami_coll.h>
```

5.14.2 Class declaration

```
class AMI_COLLECTION;
```

5.14.3 Description

A block collection is a set of fixed size blocks. Each block inside the collection is identified by a block ID, of type `AMI_bid`.

5.14.4 Constructors and Destructor

`AMI_COLLECTION``(size_t lbf = 1);`
: Create a *new* collection with access type `AMI_WRITE_COLLECTION` using temporary file names. The files are created in a directory given by the `AMI_SINGLE_DEVICE` environment variable (or `‘‘/var/tmp/’’` if that variable is not set). The `lbf` (logical block factor) parameter determines the size of the blocks stored (the block size is `lbf` times the operating system page size). The persistency of the collection is set to `PERSIST_DELETE`.

`AMI_COLLECTION``(char *base_file_name, AMI_collection_type t = AMI_READ_WRITE_COLLECTION, size_t lbf = 1);`
: Create a new or open an existing collection using `base_file_name` to find the necessary files. The access type is set to `t`. It has one of the following values:

  `AMI_READ_COLLECTION` Open an existing collection read-only;

  `AMI_WRITE_COLLECTION` If the files specified by `base_file_name` exist, open a collection using those files for reading and writing. If the files do not exist, create a new collection with read and write acces;

  The `lbf` (logical block factor) parameter determines the size of the blocks stored (the block size is `lbf` times the operating system page size). The persistency of the collection is set to `PERSIST_PERSISTENT`.

∼**`AMI_COLLECTION`**`();`
: Destructor. Closes all files. If persistency is set to `PERSIST_DELETE`, it also removes the files. There should be no blocks in memory. If the destructor detects in-memory blocks, it issues a warning in the TPIE log file (if logging is turned on). The memory held by those blocks is lost to this program.

5.14.5 Member Functions

`bool` **`operator!`**`() const;`

Return `true` if the status of the collection is not `AMI_COLLECTION_STATUS_VALID`, `false` otherwise. See also `is_valid()` and `status()`.

`size_t` **`block_factor`**`() const;`

Return the logical block factor. The block size is obtained by multiplying the operating system page size by this value.

`size_t` **`block_size`**`() const;`

Return the size of a block stored in this collection, in bytes (all blocks in a collection have the same size).

`static const tpie_stats_collection&` **`gstats`**`();`

Return an object containing the statistics of all collections opened by the application (global statistics). See also `stats()`.

`bool` **`is_valid`**`() const;`

Return `true` if the status of the collection is `AMI_COLLECTION_STATUS_VALID`, `false` otherwise. See also `status()`.

`void` **`persist`**`(persistence p);`

Set the persistency flag to `p`. The possible values for `p` are `PERSIST_PERSISTENT` and `PERSIST_DELETE`.

`persistence` **`persist`**`() const;`

Return the value of the persistency flag.

`size_t` **`size`**`() const;`

Return the number of blocks in the collection.

`const tpie_stats_collection&` **`stats`**`() const;`

Return an object containing the statistics of this collection. The types of statistics computed for a collection are tabulated below. See also `gstats()`.

`BLOCK_GET`	Number of block reads
`BLOCK_PUT`	Number of block writes
`BLOCK_NEW`	Number of block creates
`BLOCK_DELETE`	Number of block deletes
`BLOCK_SYNC`	Number of block sync operations
`COLLECTION_OPEN`	Number of collection open operations
`COLLECTION_CLOSE`	Number of collection close operations
`COLLECTION_CREATE`	Number of collection create operations
`COLLECTION_DELETE`	Number of collection delete operations

`AMI_collection_status` **`status`**`() const;`

Return the status of the collection. The result is either `AMI_COLLECTION_STATUS_VALID` or `AMI_COLLECTION_STATUS_INVALID`. The only operation that can leave the collection invalid is the constructor (if that happens, the log file contains more information). No blocks should be read from or written to an invalid collection.

`void *`**`user_data`**`();`

Return a pointer to a 512-byte array stored in the header of the collection. This can be used by the application to store initialization information (e.g., the id of the block containing the root of a B-tree).

5.15 B+-tree

5.15.1 Files

```
#include <ami_btree.h>
```

5.15.2 Class Declaration

```
template<class Key, class Value, class Compare, class KeyOfValue>
class AMI_btree;
```

5.15.3 Description

The `AMI_btree<Key, Value, Compare, KeyOfValue>` class implements the behavior of a dynamic B+-tree or (a, b)-tree storing fixed-size data items. All data elements (of type `Value`) are stored in the leaves of the tree, with internal nodes containing keys (of type `Key`) and links to other nodes. The keys are ordered using the `Compare` function object, which should define a strict weak ordering (as in the STL sorting algorithms). Keys are extracted from the `Value` data elements using the `KeyOfValue` function object.

5.15.4 Constructors and Destructor

`AMI_btree(const AMI_btree_params &params = btree_params_default);`

Construct an empty AMI_btree using temporary files. The tree is stored in a directory given by the `AMI_SINGLE_DEVICE` environment variable (or `"/var/tmp/"` if that variable is not set). The persistency flag is set to `PERSIST_DELETE`. The `params` object contains the user-definable parameters (see Appendix for an explanation of the `AMI_btree_params` class and the default values).

`AMI_btree(const char *bfn, BTE_collection_type t = BTE_WRITE_COLLECTION, const AMI_btree_params &params = btree_params_default);`

Construct a B-tree using the files given by `bfn` (base file name). The files created/used by a Btree instance are outlined in the following table.

bfn`.l.blk`	Contains the leaves block collection.
bfn`.l.stk`	Contains the free blocks stack for the leaves block collection.
bfn`.n.blk`	Contains the nodes block collection.
bfn`.n.stk`	Contains the free blocks stack for the nodes block collection.

The persistency flag is set to `PERSIST_PERSISTENT`. The `params` object contains the user-definable parameters (see Appendix for an explanation of the `AMI_btree_params` class and the default values).

`~AMI_btree();`

Destructor. Either remove or close the supporting files, depending on the persistency flag (see method `persist()`).

5.15.5 Member functions

`bool erase(const Key& k);`

Delete the element with key `k` from the tree. Return true if succeded, false otherwise (key not found).

`bool` **`find`**`(const Key& k, Value& v);`

Find an element based on its key. If found, store it in `v` and return true.

`size_t` **`height`**`() const;`

Return the height of the tree, including the leaf level. A value of 0 represents an empty tree.

`bool` **`insert`**`(const Value& v);`

Insert an element `v` into the tree. Return true if the insertion succeded, false otherwise (duplicate key).

`bool` **`is_valid`**`() const;`

Return `true` if the status of the tree is `AMI_BTREE_STATUS_VALID`, `false` otherwise. See also `status()`.

`AMI_err` **`load`**`(AMI_STREAM<Value>* is, float lf = 0.7, float nf = 0.5)`

Bulk load from the stream `is` of elements. Leaves are filled to `lf`×capacity, and nodes are filled to `nf`×capacity.

`AMI_err` **`load`**`(AMI_btree<Key, Value, Compare, KeyOfValue>* bt, float leaf_fill = .7, float node_fill = .5);`

Bulk load from another B-tree. This is a means of reoganizing a B-tree after a lot of updates. A newly loaded structure may use less space and may answer range queries faster.

`AMI_err` **`load_sorted`**`(AMI_STREAM<Value>* is, float lf = 0.7, float nf = 0.5);`

Same as `load()` above, but bypasses the expensive sorting step, by assuming that the stream `is` is sorted.

`const AMI_btree_params&` **`params`**`() const;`

Return a const reference to the `AMI_btree_params` object used by the B-tree. This object contains the true values of all parameters (unlike the object passed to the constructor, which may contain 0-valued parameters to indicate default behavior; see Section 5.15.6 below).

`void` **`persist`**`(persistence p);`

Set the persistency flag to `p`. The persistency flag dictates the behavior of the destructor of this AMI_btree object. If `p` is `PERSIST_DELETE`, all files associated with the tree will be removed, and all the elements stored in the tree will be lost after the destruction of this AMI_btree object. If `p` is `PERSIST_PERSISTENT`, all files associated with the tree will be closed during the destruction of this AMI_btree object, and all the information needed to reopen this tree will be saved.

`bool` **`pred`**`(const Key& k, Value& v);`

Find the highest element stored in the tree whose key is lower than `k`. If such an element exists, return `true` and store the result in `v`. Otherwise, return `false`.

`void` **`range_query`**`(const Key& k1, const Key& k2, AMI_STREAM<Value>* os);`

Find all elements within the range given by keys `k1` and `k2` and write them to stream `os`.

`size_t` **`size`**`() const;`

Return the number of elements stored in the leaves of this tree.

`AMI_err` **`sort`**`(AMI_STREAM<Value>* is, AMI_STREAM<Value>* &os);`

As a convenience, this function sorts the stream `is` and stores the result in `os`. If the value of `os` passed to the function is `NULL`, a new stream is created and `os` points to it.

`AMI_btree_status` **`status`**`() const;`

Return the status of the collection. The result is either `AMI_BTREE_STATUS_VALID` or `AMI_BTREE_STATUS_INVALID`. The only operation that can leave the tree invalid is the constructor (if that happens, the log file contains more information).

`bool` **`succ`**`(const Key& k, Value& v);`

Find the lowest element stored in the tree whose key is higher than `k`. If such an element exists, return `true` and store the result in `v`. Otherwise, return `false`.

`AMI_err` **`unload`**`(AMI_STREAM<Value>* s);`

Write all elements stored in this tree to the given stream, in sorted order. No changes are performed on the tree.

5.15.6 The `AMI_btree_params` Class

The `AMI_btree_params` class encapsulates all user-definable B-tree parameters. These parameters dictate the layout of the tree and its behavior under insertions and deletions. An instance of the class created using the default constructor gives default values to all parameters. Each paramter can then be changed independently.

Class Declaration

`class` **`AMI_btree_params`**`;`

Constructor

`AMI_btree_params``()`

Initialize a `Btree_params` object with default values. The default values are given in the following table.

Parameter	*Value*
`leaf_size_min`	0
`node_size_min`	0
`leaf_size_max`	0
`node_size_max`	0
`leaf_block_factor`	1
`node_block_factor`	1
`leaf_cache_size`	5
`node_cache_size`	10

Public Member Objects

`size_t` **`leaf_size_min`**

Minimum number of elements in a leaf. A value of 0 tells the class to use the default B+-tree behavior. This parameter is a guideline. To improve performance, some leaves may have fewer elements.

`size_t` **`node_size_min`**

Minimum number of keys in an internal node. A value of 0 tells the class to use the default B+-tree behavior. As above, this parameter is a guideline.

`size_t` **`leaf_size_max`**

Maximum number of elements in a leaf. A value of 0 tells the class to fill a leaf to capacity. This value is strictly enforced.

`size_t` **`node_size_max`**

Maximum number of keys in an internal node. A value of 0 tells the class to fill a node to capacity. This value is strictly enforced.

`size_t` **`leaf_block_factor`**

The size (in bytes) of a leaf block is `leaf_block_factor`× `os_block_size`, where `os_block_size` is the operating-system specific page size.

`size_t` **`node_block_factor`**
The size (in bytes) of an internal node block is `node_block_factor`× `os_block_size`.

`size_t` **`leaf_cache_size`**
The size (in number of leaf blocks) of the leaf block cache. The cache implements an LRU replacement policy.

`size_t` **`node_cache_size`**
The size (in number of node blocks) of the node block cache. The cache implements an LRU replacement policy.

5.16 Cache Manager

5.16.1 Files

```
#include <ami_cache.h>
```

5.16.2 Class Declaration

```
template<class T, class W> class AMI_CACHE_MANAGER;
```

5.16.3 Description

5.16.4 Constructors and Destructor

`AMI_CACHE_MANAGER``(size_t capacity);`
Construct a fully-associative cache manager with the given capacity.

`AMI_CACHE_MANAGER``(size_t capacity, size_t assoc);`
Construct a cache manager with the given capacity and associativity.

∼**`AMI_CACHE_MANAGER`**`();`
Destructor. Write out all items still in the cache.

5.16.5 Member Functions

`bool` **`read`**`(size_t k, T & item);`
Read an item from the cache based on key `k` and store it in `item`. If found, the item is removed from the cache. Return true if the key was found.

`bool` **`write`**`(size_t k, const T & item);`
Write an item in the cache based on the given key `k`. If the cache was full, the least recently used item is writen out using the `W` function object, and it is removed from the cache.

`bool` **`erase`**`(size_t k);`
Erase an item from the cache based on the given key `k`. Return true if the key was found.

Chapter 6

The Implementation of TPIE

6.1 The Structure of TPIE

TPIE has three main components, the Access Method Interface (AMI), a Block Transfer Engine (BTE) component, and a Memory Manager (MM) component. Various Block Transfer Engines (BTEs) can be chosen for handling disk block transfers, perhaps more than one in a single application. The MM component provides low level memory management services such as allocating, deallocating, and accounting of internal memory. The AMI works on top of the Memory Manager and one or more BTEs to provide a uniform interface for application programs. Applications that use this interface are portable across hardware platforms, since they never have to deal with the underlying details of how I/O is performed on a particular machine. This chapter describes the design decisions, algorithms, and implementation decisions that were used to build the MM, BTE and AMI components of TPIE. The Reference Section of this manual contains a description of the AMI and Memory Manager entry points that an application programmer might normally use. Typically, an application programmer will not request services from a BTE directly. For this reason, the BTE services are not presented in the Reference Section of this manual, but are presented here for those readers who wish to understand the implementation details of TPIE.

The MM manages random access memory on behalf of TPIE. Currently, TPIE is distributed with an MM designed for a single processor, or multiprocessor system with a single global address space. This MM is relatively simple; its task is to allocate and manage the physical memory used by the BTE component.

The AMI is an interface layer between the BTE and user level processes. It implements fundamental access methods, such as scanning, permutation routing, merging, and distribution. It also provides a consistent, object-oriented interface to application programs. The key to keeping the AMI simple and flexible is the fact that its user accessible functions serve more as templates for computation than as actual problem solving functions. The details of how a computation proceeds within the template is up to the application programmer, who is responsible for providing the functions that the template applies to data.

6.2 The Block Transfer Engine (BTE)

The BTE component is intended to bridge the gap between the I/O hardware and the rest of the system. It is the layer that is ultimately responsible for moving blocks of data from physical disk devices to main memory and back. It works alongside the traditional buffer cache in a UNIX system. Unlike the buffer cache, which must support concurrent access to files from multiple address spaces, the BTE is specifically designed to support high throughput processing of data from secondary memory through a single, user level address space. In order to efficiently support the merging, distribution, and scanning paradigms, several BTEs support stream-oriented input and output of data blocks. To further improve performance, some BTE implementations move data from disk directly into user space rather than using a kernel-level buffer cache. This saves both main memory space and copying time. In the future, TPIE will also provide a BTE implementation with support for random-access to disk blocks. Such functionality is useful when implementing external data structures (indexes). Currently, all TPIE BTEs are designed for processing streams on a single disk. Planned BTE implementations will also support streams stored on multiple disks.

[27]: LA: This whole section needs an overhaul - discuss implementation!

Streams in TPIE are implemented as sequentially accessed files, and each BTE offers public member functions that are analogous to, and implemented via corresponding functions available in the underlying *file access method*. The main difference is that TPIE maintains a typed view of streams, where user-defined data elements (i.e. objects) are accessed in a stream, rather than the untyped view of the data offered by the corresponding file access primitives.[28]

[28]: LA: Some intro about streams, files, ect - like in AMI reference. Also something about; block access, read ahead, double buffering, logical blocks, file pointer, stream types, persistence, ect.?

Version 082902 of TPIE supports the following three BTE stream implementations:

1. `BTE_stream_stdio`, which uses the UNIX `stdio` library as its file access method;
2. `BTE_stream_ufs`, which performs I/O via UNIX `read()/write()` system calls;
3. `BTE_stream_mmap`, which uses the UNIX `mmap()/munmap()` calls to perform I/O.

The choice of BTE in a given application is controlled by compile-time variables in the `app_config.h` file. The `BTE_stream_stdio` implementation is selected by defining the variable `BTE_STREAM_IMP_STDIO`, the `BTE_stream_ufs` implementation by defining `BTE_STREAM_IMP_UFS`, and the memory mapped implementation by defining `BTE_STREAM_IMP_MMAP`. For example, the following code (taken from `app_config.h` selects the `BTE_stream_ufs` Block Transfer Engine (and does not select the others):

```
//#define BTE_STREAM_IMP_MMAP
//#define BTE_STREAM_IMP_STDIO
#define BTE_STREAM_IMP_UFS
```

Multiple implementations are allowed to coexist, with some restrictions, e.g. declarations of streams must use explicit subclasses of `AMI_stream_base` to specify what type of streams they are.[29][30] The best choice of BTE for a given application is both application and system dependent. Section 7.2.1 discuss how to choose an appropriate BTE (also refer to the more detailed descriptions of the BTE's in the next three subsections). If none of the available BTEs is selected, `BTE_STREAM_IMP_UFS`[31] is defined by default and a warning is generated at compile time.

[29]: LA: This needs to be extended somewhere.

[30]: DH: I am skeptical that it works with multiple impl., but I don't see why we need this feature with single disks. Multiple impls. in an app may be possible with separate compiles? We may need more flexibility with multi disk implementations however.

[31]: DH: We should change the default to BTE_IMP_UFS

The first block in a TPIE stream, referred to as the *header block*, contains contextual information for the stream. Currently, the format of the header block is BTE-dependent. Please refer to the descriptions of the individual BTEs in Sections 6.2.2,**??**, and 6.2.4 for more details.

6.2.1 BTE Common Functionality

All BTE stream implementations inherit from class `BTE_stream_base` (in file `bte_stream_base.h`) and must support the member functions listed below.

Since the details of each access method (ie `BTE_stream_ufs`, `BTE_stream_stdio` or `BTE_stream_mmap` are different, the BTEs generally have different member function implementations, including constructors (construction involves opening the appropriate stream file, etc.). The `BTE_stream_ufs`, `BTE_stream_stdio`, and `BTE_stream_mmap` implementations are defined in the respective files `include/bte_stream_ufs.h`, `include/bte_stream_stdio.h`, and `include/bte_stream_mmap.h`.[32] Functionally, the member functions for each BTE are similar.

[32]: LA: Some note about construction being BTE dependent - why are they? DH: DONE

The common BTE member functions[1] and their usage are outlined below:

Stream Constructors

```
    BTE_stream_mmap  ( const char *dev_path, BTE_stream_type st );
    BTE_stream_stdio ( const char *dev_path, BTE_stream_type st );
    BTE_stream_ufs   ( const char *dev_path, BTE_stream_type st );
```

[1] These functions can be defined as `virtual` to assist in debugging a new BTE if the compile-time variable `BTE_VIRTUAL_BASE` is defined to be non-zero (see also file `app_config.h`). For best run-time performance, the default value for `BTE_VIRTUAL_BASE` is zero.

These constructors each create a stream which is backed by the file whose path is pointed to by `dev_path`. The stream can be opened in one of several modes, depending on the value of `st`:

- `BTE_READ_STREAM`: the stream can be read, but not written. The underlying file must have been created previously in this case.

- `BTE_WRITE_STREAM` or `BTE_WRITEONLY_STREAM`: the stream can be written, but not read. This will allow a new stream to be created or overwrite a previously-created one.

- `BTE_APPEND_STREAM`: the stream is opened, and positioned to the end of stream so that new data can be appended. This is valid only if the stream was previously created.

new_substream

```
BTE_err new_substream(BTE_stream_type st, off_t sub_begin, off_t
                      sub_end, BTE_stream_base<T> **sub_stream);
```

Constructs a substream of the current stream. A substream is a logical contiguous subset of a stream, i.e. from the implementation point of view, it is a contiguous subset of the data elements in a file. The parameters of `new_substream()` are as follows:

- `sub_begin` and `sub_end` are object offsets into the current stream, indicating the limits of the new substream.

- `st` is the stream type of the new substream. Please refer to the discussion of substream constructors above for a list of the valid (sub)stream types.

- `sub_stream` will be given the address of a pointer to the new substream.

While it might seem logical to explicitly define a substream in TPIE as inheriting (in the C++ sense) from a stream, this would imply a need for both to have a constructor, and hence for the stream constructor to be `virtual`. This is a problem in C++, and so the "pseudo-constructor" approach described here is used instead.[33] Because `new_substream()` is not a constructor, but rather a function each particular implementation of which calls an appropriate constructor, it can be a pure virtual function in the stream base class, which forces it to be defined for all actual stream implementations.[34]

[33]: LA: Extend like in AMI ref. Stream type?

[34]: LA: I don't understand this paragraph. DH: reworked. Clearer?

read_item

```
BTE_err read_item(T **elt);
```

Returns the address of a pointer to the next element in the stream in `elt`. Since the application data elements are blocked, this often does not require a physical I/O operation. If the current block has been exhausted, the first element of the next block will be accessed. In some cases the BTE will try to prefetch the next block and it may already be in memory. (See descriptions of the individual BTEs for more details about prefetching or read-ahead.)

write_item

```
BTE_err write_item(const T &elt);
```

Writes the data element pointed to by `elt` to the current position in the stream. Streams are normally written in sequential order, and so the current position is usually directly after the previous element written. The current position (which we will refer to as the value of the *file pointer*) can be modified via the `seek` command below.[35]

[35]: LA: Extend! DH:DONE

seek

```
BTE_err seek(off_t offset);
```

Seeks to the object offset `offset` in the stream. The BTE `seek` member function is similar to and utilizes the `seek` function supported by the underlying file I/O system being used. The difference is that TPIE performs a seek to the requested application data element in the stream rather than to a byte offset within a file of untyped data.

truncate

```
BTE_err truncate(off_t offset);
```

Truncates/extends the stream to the specified number of application data elements. The file pointer[36] will be moved to the end of the stream. This is analogous to the `truncate` function of the underlying file access method. Note: `truncate` is not supported for substreams.

[36]: LA: ? DH: defined above

main_memory_usage

```
BTE_err main_memory_usage(size_t *usage,
                          MM_stream_usage usage_type);
```

Queries how much memory is used by the stream for `usage_type` purposes. On return, `usage` points to the value used. The valid values for `usage_type` are as follows[37]:

[37]: LA: ? DH: DONE

- `MM_STREAM_USAGE_OVERHEAD`: the size in bytes of the stream object plus, if this is not a substream, the stream header block.
- `MM_STREAM_USAGE_BUFFER`: the number of bytes consumed by block buffers (blocks of elements in main memory).
- `MM_STREAM_USAGE_CURRENT`: `MM_STREAM_USAGE_OVERHEAD` plus the number of bytes currently consumed by block buffers for this stream.
- `MM_STREAM_USAGE_MAXIMUM`: `MM_STREAM_USAGE_OVERHEAD` plus the maximum number of bytes that will be consumed by block buffers for this stream.
- `MM_STREAM_USAGE_SUBSTREAM`: The same as `MM_STREAM_USAGE_OVERHEAD` but assumes that this is a substream.

get_status

```
BTE_stream_status get_status(void);
```

Returns the status of the stream as one of the following values[38]:

[38]: LA: Is this really used anywhere? DH: Yes, appears in the code.

- `BTE_STREAM_STATUS_NO_STATUS`: No status information exists for the stream.
- `BTE_STREAM_STATUS_INVALID`: An error was encountered while manipulating the stream and the stream can no longer be used.
- `BTE_STREAM_STATUS_END_OF_STREAM`: The end of stream was encountered.

The `get_status` member function is one of the few that is implemented in `bte_stream_base.h` and its implementation is therefore common to all BTEs. The status is stored in the private variable `status` of a BTE object.

stream_len

```
off_t stream_len(void);
```

Returns the current number of application data elements in the stream.

name

```
BTE_err name(char **stream_name);
```

Returns the path name of the file backing the stream. The name will be stored in newly allocated space.
read_only

```
    int read_only(void);
```

Returns true if the stream mode is BTE_READ_STREAM. The file mode is set when creating a stream. See BTE constructors, above for details.-39-

[39]: LA:? DH: DONE

available_streams

```
    int available_streams(void);
```

Returns the number of additional streams that can be activated (i.e. created) before an operating system limit on the number of open files would be exceeded. This reflects only the operating system limit on number of open files. The TPIE memory requirements for active streams are not reflected in this limit.-40-

[40]: LA: Extend DH:DONE

chunk_size

```
    off_t chunk_size(void);
```

Returns the number of application data elements that fit in a logical block of the current stream.-41-

[41]: LA: ?

persist

```
    void persist(persistence);
```

Sets the persistence attribute of the current stream to the value of `persistence`. This may be one of the following values:

- `PERSIST_DELETE`: Delete the stream from the disk when the stream destructor is called.
- `PERSIST_PERSISTENT`: Do not delete the stream from the disk when the stream destructor is called.

By default, all streams are deleted when their destructors are called.

6.2.2 BTE *stdio*

The *stdio* BTE is implemented by the class `BTE_stream_stdio`, defined in file `/include/bte_stdio.h`. `BTE_stream_stdio` streams are stored as ordinary UNIX files which are manipulated via the *stdio* file access method, i.e. via the standard `C` I/O library. The read/write primitives of `BTE_stream_stdio` streams are implemented using the system calls `fread` and `fwrite`. The underlying operating system blocking and prefetching assure that stream accesses are done in blocks and so both blocking and prefetching are therefore automatic and invisible to the TPIE developer. Note that this means that a (OS) kernel call is incurred every time a stream object is accessed, and that every object passes through kernel level buffer space on its way to user space.-42-43-

[42]: DH: this seems incorrect to me.

[43]: LA: Extend. Also something about no logical block size concept.

The `BTE_stream_stdio` class inherits from `BTE_stream_base` (in file `/include/bte_stream_base.h`):

```
class BTE_stream_stdio : public BTE_stream_base {
  private:
     FILE  *file;
     BTE_stdio_header      header;
     ...
}
```

`BTE_stream_stdio` defines the public member functions defined for `BTE_stream_base` (please see Section 6.2.1 for a list of these member functions and their semantics).-44- In addition to these, `BTE_stream_stdio` defines its own constructors:

[44]: LA: Extend - discuss implementation!

```
     BTE_stream_stdio(const char *dev_path, const BTE_stream_type st);
     BTE_stream_stdio(const BTE_stream_type st);
     BTE_stream_stdio(const BTE_stream_stdio<T> &s);
```

The first block of a TPIE *stdio* stream is a header block with the following structure:

```
typedef struct BTE_stdio_header_v1 {
    unsigned int magic_number;  // Set to BTE_STDIO_HEADER_MAGIC_NUMBER
    unsigned int version;       // Should be 1 for current version.
    unsigned int length;        // # of bytes in this structure.
    unsigned int block_length;  // # of bytes in a block.
    size_t item_size;           // The size of each item in the stream.
} BTE_stdio_header;
```

6.2.3 BTE *mmap*

The *mmap* BTE is implemented by the class BTE_stream_mmap, defined in file /include/bte_stream_stdio.h. BTE_stream_mmap streams are stored as ordinary UNIX files. The TPIE BTE_stream_mmap implementation uses the Unix mmap and munmap calls to perform I/O. The UNIX mmap call allows a part of a file to be associated with a corresponding section of internal memory. When the memory is accessed, the corresponding portion of the file is copied directly into that memory buffer. The BTE_stream_mmap primitives explicitly maintain the currently accessed block of the file mapped into memory. When an object outside the current block boundaries is requested, the current block is unmapped and a new one is mapped from the source file.

The BTE_stream_mmap is not limited to mapping in blocks of size equal to the physical block size of the OS (typically 8K bytes). Often improved performance can be obtained by mapping blocks of much larger size (for example 256KB[2]). This is typically due to (track) buffering and prefetching performed in the disk controller. The (logical) block size used by BTE_stream_mmap can be set using the macro BTE_STREAM_MMAP_BLOCK_FACTOR in the app_config.h file. However, choosing a large (logical) block size limits the amount of main memory available for an application program and thus the (logical) block size should be chosen very carefully in order to obtain maximal performance.

Unlike in BTE_stream_stdio, where a function call is incurred every time a stream object is accessed, the BTE_stream_mmap only incurs such a call on every (logical) block. This can lead to improved performance, especially on systems with a relatively slow CPU compared to the disk. However, while prefetching of disk blocks is implicitly done by the operating system in BTE_stream_stdio, BTE_stream_mmap has to implement its own prefetching scheme. (Note on the other hand that unlike in BTE_stream_stdio objects does not pass through the kernel level buffer space on its way to user space). BTE_stream_mmap prefetching can be turned on by defining the macro BTE_MMAP_READ_AHEAD in the app_config.h file. If this compile-time variable is defined, BTE_stream_mmap optimize for sequential read speed by reading (mapping) blocks into main memory before the data they contain is actually needed. Two methods of read-ahead is provided:

- If the USE_LIBAIO macro is undefined (and BTE_MMAP_READ_AHEAD is set), read ahead is done using mmap calls.
- If the USE_LIBAIO macro is defined (and BTE_MMAP_READ_AHEAD is set), read ahead is done using the asynchronous I/O library. This feature requires the asynchronous I/O library libaio.

[45]: LA: Discuss the two and how they differ.

By default BTE_MMAP_READ_AHEAD is defined, USE_LIBAIO is not.[45]

BTE_stream_mmap class inherits from BTE_stream_base:

```
class BTE_stream_mmap : public BTE_stream_base {
  private:
     // descriptor of the mapped file.
     int fd;
     // A pointer to the mapped in header block for the stream.
     mmap_stream_header *header;
     ...
}
```

[2] We use the notation KB to mean kilobytes.

`BTE_stream_mmap` defines the public member functions defined for `BTE_stream_base` (please see Section 6.2.1 for a list of these member functions and their semantics). In addition to these it defines its own constructors:[46][47]

[46]: LA: Extend - discuss implementation.

[47]: LA: Why special substream constructor? DH:Previously discussed.

```
BTE_stream_mmap(const char *dev_path, BTE_stream_type st);
BTE_stream_mmap(BTE_stream_type st);
BTE_stream_mmap(BTE_stream_mmap<T> &s);

// A substream constructor.
BTE_stream_mmap(BTE_stream_mmap *super_stream,
                BTE_stream_type st,
                off_t sub_begin, off_t sub_end);
```

The `BTE_stream_mmap` header structure is as follows:

```
struct mmap_stream_header {
  public:
    unsigned int magic_number;  // Set to MMAP_HEADER_MAGIC_NUMBER
    unsigned int version;       // Should be 1 for current version.
    unsigned int length;        // # of bytes in this structure.
    off_t item_logical_eof;     // The number of items in the stream.
    size_t item_size;           // The size of each item in the stream.
    size_t block_size;          // The size of a physical block on the device
                                // where this stream resides.
    unsigned int items_per_block;
};
```

6.2.4 BTE *ufs*

The *ufs* BTE is implemented by the class `BTE_stream_ufs`, defined in file `/include/bte_stdio.h`. `BTE_stream_ufs` streams are stored as ordinary UNIX files. `BTE_stream_ufs` streams use `read()/write()` calls to implement their I/O.

Like in the case of `BTE_stream_mmap` the (logical) block size can be controlled using the macro `BTE_STREAM_UFS_BLOCK_FACTOR`. `BTE_stream_ufs` also has the same advantage as `BTE_stream_mmap` over `BTE_stream_stdio` of only incurring one kernel call per block. However, unlike `BTE_stream_mmap` (but like `BTE_stream_stdio`), prefetching is done implicitly by the filesystem underlying TPIE (and objects have to pass through kernel level buffer space) In `BTE_stream_ufs` streams, when the asynchronous I/O library `libaio` is available, there is a provision to do (user-level) prefetching within `BTE_stream_ufs` streams but we do not recommend its use on account of the implicit filesystem prefetching.

`BTE_stream_ufs` class inherits from `BTE_stream_base`:

```
class BTE_stream_ufs : public BTE_stream_base {
  private:
     // descriptor of the mapped file.
     int fd;
     // A pointer to the mapped in header block for
     the stream.  mmap_stream_header *header; ...  }
```

`BTE_stream_ufs` defines the public member functions defined for `BTE_stream_base` (please see Section 6.2.1 for a list of these member functions and their semantics). In addition to these it defines its own constructors:[48]

[48]: LA: Why a special substream constructer?

```
BTE_stream_ufs(const char *dev_path, BTE_stream_type st);
BTE_stream_ufs(BTE_stream_type st);
BTE_stream_ufs(BTE_stream_ufs<T> &s);

// A substream constructor.
```

```
  BTE_stream_ufs(BTE_stream_ufs *super_stream,
                 BTE_stream_type st,
                 off_t sub_begin, off_t sub_end);
```

The BTE_stream_ufs header structure is as follows:

```
struct ufs_stream_header {
  public:
    unsigned int magic_number;  // Set to UFS_HEADER_MAGIC_NUMBER
    unsigned int version;       // Should be 1 for current version.
    unsigned int length;        // # of bytes in this structure.
    off_t item_logical_eof;     // The number of items in the stream.
    size_t item_size;           // The size of each item in the stream.
    size_t block_size;          // The size of a physical block on the device
                                // where this stream resides.
    unsigned int items_per_block;
};
```

[49]: LA: Discuss implementation!

6.3 The Memory Manager (MM)

The Memory Manager components of TPIE provide services related to the management of internal memory:

- allocation and deallocation of (internal) memory as requested by the new and delete operators,
- accounting of memory usage (when required),
- enforcing of the user-specified internal memory usage limit (when required),
- logging of memory allocation requests (when required).

In version 082902of TPIE, the memory manager MM_manager, is built from the source files mm_register.cpp, mm_base.cpp, mm_register.h, and mm_base.h. The memory manager traps memory allocation and deallocation requests in order to monitor and enforce memory usage limits. It provides a number of user-callable functions and services, which are documented in Chapter 5.

<TO BE WRITTEN>

[50]: LA: There is some text in here that I commented out.

[51]: (Need some comments on the current simple MM that we have and some OS issues that come up in attempting to make it more robust.)

6.4 The Access Method Interface (AMI)

The AMI-level entry points provided by TPIE are documented in the Reference section of this manual (see Section **??**), and a number of examples of their use are given in the Tutorial section (see Section 4). In this section we examine the TPIE source files which compose the AMI and provide a brief discussion of their purpose and relationship to to each other. We also discuss the algorithmic decisions that were made in constructing the various TPIE services such as creation, scanning, merging, sorting, and permutation of streams, and the services of the block collection class. The presentation of this section is organized by TPIE service (creation of streams, scanning of stream, etc.) and within each service the relevant source files are itemized (alphabetically) and discussed. The index of this manual provides a more direct way to find the documentation for a particular source file.

6.4.1 Using Multiple BTE Implementations

For instance, if a single implementation is needed, it is sufficient for the application code to create a stream of int as follows:

```
AMI_STREAM<int> aStream;
```

However, if different implementations, say a ufs and an mmap stream are desired, the code would be similar to the following:

```
    BTE_stream_mmap   firstStream( const char *dev_path, BTE_stream_type st );
    BTE_stream_stdio secondStream( const char *dev_path, BTE_stream_type st );
```

6.4.2 General Considerations

The following TPIE files are fundamental and therefore involved in every TPIE program, no matter which TPIE services are accessed[52]:

[52]: DH: There may be a better place for some of the stuff in this intro

- include/ami.h: This file should be included (at compile time) in every TPIE application program that uses the AMI-level interface. It in turn inputs the definitions for the AMI-level services of TPIE. The files input by include/ami.h are itemized below.
- test/app_config.h: This file contains TPIE flags and settings that can be customized to an individual application. The options available in this file are described in detail in Section 7.1.2.
- include/config.h: This file contains flags and indicators that describe the machine and operating system on which TPIE is currently running. It is generated automatically by the TPIE installation process and is not intended to be modified.
- lib/libtpie.a: This is the TPIE load library. It contains code for the memory manager MM_manager, TPIE logging, BTE statistics, and a small number of other services. While most TPIE code is in the form of templates, which generate code at compile-time, application programs must also link with lib/libtpie.a. This is described further in Section 4.13.

File: include/ami.h

This C++ include file inputs the various include files required for compilation of an application program with TPIE. The relevant portion of include/ami.h is included below:

```
// Get the base class, enums, etc...
#include <ami_base.h>

// Get the device description class
#include <ami_device.h>

// Get an implementation definition
#include <ami_imps.h>

// Get templates for ami_scan().
#include <ami_scan.h>

// Get templates for ami_merge().
#include <ami_merge.h>

// Get templates for ami_sort().
#include <ami_sort.h>

// Get templates for general permutation.
#include <ami_gen_perm.h>

// Get templates for bit permuting.
#include <ami_bit_permute.h>
```

Each of these files is discussed in the sections which follow.

6.4.3 Creation of Streams

AMI stream objects are created in a TPIE program via the AMI_STREAM keyword. AMI_STREAM is a macro that resolves to a class template invocation appropriate to the declared target I/O architecture. This involves the declared AMI implementation (see Section **??**), BTE implementation (see Section 6.2) and whether one or multiple disks are used. In the current version of TPIE an AMI_STREAM is stored in a standard UNIX file on a single disk. For most applications, an AMI_STREAM will have type AMI_stream_single, which is the TPIE base class for a stream backed by a single file (normally on a single disk). The string AMI_STREAM is #define'd to be the string AMI_stream_single in the header file ami_imps.h. This arrangement is intended to permit alternative implementations of AMI_STREAM if necessary in the future.

The TPIE code associated with the creation and manipulation of stream objects is contained mainly within the following files:

- include/ami_base.h:

  1. defines the AMI_err codes returned by the AMI-level services (these and their meanings are listed in Appendix C.1.
  2. defines codes for the AMI stream types (AMI_READ_STREAM, AMI_WRITE_STREAM, AMI_APPEND_STREAM, AMI_READ_WRITE_STREAM.
  3. defines the member functions of AMI_stream_base as virtual if compile time variable AMI_VIRTUAL_BASE is defined.

- include/ami_single.h: This include file defines the class AMI_stream_single, which is the base class for all AMI-level streams backed by a single Unix file.

  1. the template parameter <T> represents the type of the application data element in the stream.
  2. At construction time, an AMI_stream_single<T> is mapped onto a BTE_STREAM<T> and the AMI stream type is mapped as follows:
     - AMI_READ_STREAM is mapped to BTE_READ_STREAM
     - AMI_APPEND_STREAM is mapped to BTE_APPEND_STREAM
     - AMI_WRITE_STREAM and AMI_READ_WRITE_STREAM are mapped to BTE_WRITE_STREAM
  3. The AMI-level services for streams are implemented by corresponding BTE-level services. The AMI member functions described in Sections 5.2 (see examples of use of some of these in Section 3.1) are implemented by calls to the BTE-level functions documented in Section 6.2. An AMI stream has little internal context information as a result. Two exceptions are the following:
     - r_only: this flag is one (TRUE) if the stream is only readable, and not writeable, corresponding to the status AMI_READ_STREAM. Maintaining this flag allows the AMI stream member functions to catch erroneous attempt by application-level code to write to a stream whose underlying file was opened for reading.
     - destruct_bte: this flag is one (TRUE) if the destructor of the underlying BTE stream should be called by the AMI-level destructor (the normal situation). If the AMI-level stream was constructed from a pre-existing BTE stream via the constructor

       ```
       AMI_stream_single(BTE_STREAM<T> *bs);
       ```

       then a destruct_bte value of zero is used to prevent the destructor from deleting the BTE stream (owned by another AMI stream).

6.4.4 Scanning

The file include/ami_scan.h defines the scanning functionality provided by TPIE. This file is mechanically generated by make when TPIE is compiled and built. It is essentially a concatenation of the files include/ami_scan.h.head, include/ami_scan_mac.h, and include/ami_scan.h.tail – for more details, see the file include/Makefile. It is not intended that include/ami_scan.h be modified manually.

[53]: LA: Talk about streams ect (built on top of BTE_STREAM.) DH: Done

[54]: LA: Remember to include something about r_only flag somewhere. DH: Done

[55]: LA: There is a lot of text in here I have commented out

6.4.5 Merging

<TO BE EXTENDED>

`ami_merge_base.h`

Generalized Merging: using Merge Management Objects

<TO BE EXTENDED>

`ami_merge.h`[56]

[56]: LA: Remember something about ami_merge really called ami_single_merge.

Specialized Merging: without Merge Management Objects

<TO BE EXTENDED>

The `AMI_merge()` polymorphs (defined in file `ami_optimized_merge.h`) work without a merge management object and perform standard (total order/comparison based) merging. The elimination of the merge management object object results in one less level of function calls and thus in improved efficiency over a similar comparison based merging based on a merge management object. The merge is controlled by a priority queue (defined in file `include/mergeheap.h`) which exploits the fact that `delete-min` and `insert` are always performed together. This improves efficiency. In `AMI_key_merge()` the heap stores the key and not the entire object. When the object size is large compared to the key size, this often leads to further performance improvement.[57]

[57]: LA: Extend this stuff so it becomes readable

Partition and Merge

<TO BE EXTENDED>

`AMI_partition_and_merge()` repeatedly merges together the maximum possible number of sub-streams using `AMI_merge()`. An important point is that the substreams input to `AMI_merge()` during the execution of `AMI_partition_and_merge()` all originate from the same underlying parent `AMI_STREAM`; thus filesystem accesses are inherently non-sequential. Throughout the execution of `AMI_partition_and_merge()`, there are only two active `AMI_STREAM`s at any time: One which stores the substreams being merged at that stage and one which stores the substreams output by that stage.[58]

[58]: LA: Extend this stuff so that it becomes readable!

6.4.6 Comparison Sorting

The TPIE source files involved in merge sorting include the following:

1. `include/ami_sort.h`: This file simply includes several sorting-related files in the compilation:

   - `include/ami_sort_single.h`
   - `include/ami_optimized_sort.h`
   - `include/ami_sort_single_dh.h`

   These files are discussed briefly below.

2. `include/ami_sort_single.h`: This file contains a number of templates for outdated (version 1) sort routines:

   -
     ```
     AMI_err AMI_sort_V1 ( AMI_STREAM<T> *instream,
                           AMI_STREAM<T> *outstream );
     ```

```
AMI_err AMI_sort_V1 ( AMI_STREAM<T> *instream,
                      AMI_STREAM<T> *outstream,
                      int (*cmp)(CONST T&, CONST T&) );
```

-
  ```
  AMI_err AMI_sort_V1 ( AMI_STREAM<T> *instream,
                        AMI_STREAM<T> *outstream,
                        CMPR *cmp)
  ```

Except for their name (AMI_sort_V1) these routines have the same syntax as the version 2 sort routines AMI_sort (see Section 5.10 for details). The version 1 routines are generally slower than the version 2 routines. They are included in TPIE for historical comparison purposes. The current versions (version 2) of these routines can be found in include/ami_sort_single_dh.h).

The version 1 merge sort polymorphs of AMI_sort_V1() are implemented using AMI_partition_and_merge() and a merge management object. The merge_sort_manager class, a base class for merge management objects needed in AMI_sort_V1(), is also defined in file include/ami_sort_single.h. The member functions main_mem_operate(), initialize(), and operate() of the merge management object provide the sorting-specific details required for merge sorting:

- main_mem_operate() is simply an in-memory sorting algorithm based on quicksort (in file quicksort.h).
- Member function operate() selects the next output element during the merging of runs using a standard priority queue (in file pqueue_heap.h).
- Member function initialize() just initializes this priority queue.-59-

[59]: LA: extend/rewrite?

3. include/ami_optimized_sort.h: Contains beta test versions of some of the sorting methods now accessed via file include/apm_dh.h (see below for details).
4. include/apm_dh.h: Described in detail below.
5. include/ami_sort_single_dh.h: Described in detail below.

File: include/ami_sort_single_dh.h

This file contains the templates for the TPIE sort routines, as described in Section 5.10. The AMI_sort and AMI_ptr_sort templates are:

```
AMI_err AMI_sort     ( AMI_STREAM<T> *, AMI_STREAM<T> *, sort_manager & );
AMI_err AMI_ptr_sort ( AMI_STREAM<T> *, AMI_STREAM<T> *, sort_manager & );
```

The AMI_key_sort template is:

```
AMI_err AMI_key_sort ( AMI_STREAM<T> *, AMI_STREAM<T> *, KEY, sort_manager & );
```

Please refer to Section 5.10 for details of these templates. In all cases the first two arguments are input and output streams. The sort_manager object is customized for the particular sorting options implied by the template that is used. All of these templates invoke AMI_partition_and_merge_dh (see file include/apm_dh.h below for details) and pass their customized sort management object to indicate the specific details or options required.

Also in this file are the following (templated) class definitions:

- class sort_manager is a base class for sort management objects. It is inherited by the following sort management classes, which are also defined in include/ami_sort_single_dh.h:
- sort_manager_op: used by the operator variant of the AMI_sort and AMI_ptr_sort templates.
- sort_manager_obj: used by the comparison object variant of the AMI_sort and AMI_ptr_sort templates.-60-

[60]: DH: "comparison object" should be called "comparison management object"

Sorting Template	Comparison Variant	Class of Sort Mgmt Object	Class of Merge Mgmt Object
AMI_sort	operator	sort_manager_op	merge_heap_dh_op
AMI_sort	object	sort_manager_obj	merge_heap_dh_obj
AMI_sort	function	sort_manager_cmp	merge_heap_dh_cmp
AMI_ptr_sort	operator	sort_manager_op	merge_heap_pdh_op
AMI_ptr_sort	object	sort_manager_obj	merge_heap_pdh_obj
AMI_ptr_sort	function	sort_manager_cmp	merge_heap_pdh_cmp
AMI_key_sort		sort_manager_kobj	merge_heap_dh_kobj

Figure 6.1: Customization of Arguments to AMI_partition_and_merge_dh.

- sort_manager_cmp: used by the comparison function variant of the AMI_sort and AMI_ptr_sort templates.
- sort_manager_kcmp: used by the AMI_key_sort template (which uses a comparison object).

All sort_manager objects contain the following member functions:

- member function sort_fits_in_memory, which returns TRUE if there is sufficient internal memory to sort the input stream without external merging.
- member function main_mem_operate_init, which initializes any internal memory data structures needed for internal sorting (e.g. an array of pointers in the case of AMI_ptr_sort and an array of keys in the case of AMI_key_sort).
- member function main_mem_operate, which performs an internal sort of the appropriate type (e.g. quicksort of records, quicksort of pointers to records, or quicksort of keys)
- member function main_mem_operate_cleanup, which cleans up after main_mem_operate (e.g. frees the data structures used after an internal memory sort).
- member function single_merge, which merges a set of input streams into an output stream. This resolves to a call to AMI_single_merge_dh in all cases. However the merge heap object used in that call is customized according to which sort template was invoked by the application programmer.

All sort_manager objects also contain a *merge heap* object, which is custom-chosen by the particular sort template selected by the application programmer. The merge heap classes are defined in file include/mergeheap_dh.h. The following table shows the class of sort management object and merge heap object used for each sorting template and variant:

File: include/apm_dh.h

This file contains code for the following templates:

-
  ```
  template < class T, class M>
  AMI_err AMI_single_merge_dh ( AMI_STREAM <T> **instreams,
                                arity_t arity,
                                AMI_STREAM <T> *outstream.
                                M MergeHeap );
  ```

  Function: Merge an array *instreams of arity input streams using the services of merge heap object Mergeheap, to create the output stream *outstream.

-
  ```
  template < class T, class M>
  AMI_err AMI_partition_and_merge_dh ( AMI_STREAM <T> *instream,
                                       AMI_STREAM <T> *outstream.
                                       M mgmt_obj );
  ```

Function: Cut the input stream `*instream` into pieces that fit into memory, `operate` on each of them (e.g.sort them) internally, and recursively merge them as memory permits until only a single output stream `*outstream` remains.

Template `AMI_partition_and_merge_dh` is invoked directly by the user-callable sort templates. Its sort management object `mgmt_obj` contains the following:

- a merge heap object, selected by the particular sorting template invoked by the application programmer, and instantiated from one of the classes defined in file `include/mergeheap_dh.h`.
- member function `sort_fits_in_memory` returns TRUE if there is sufficient internal memory to sort the input stream without external merging.
- member function `main_mem_operate_init` initializes any internal memory data structures needed for internal sorting (e.g. an array of pointers in the case of `AMI_ptr_sort` and an array of keys in the case of `AMI_key_sort`).
- member function `main_mem_operate` performs an internal sort of the appropriate type (e.g. quicksort of records, quicksort of pointers to records, or quicksort of keys)
- member function `main_mem_operate_cleanup` cleans up after `main_mem_operate` (e.g. frees the data structures used after an internal memory sort).
- member function `single_merge` merges a set of input streams into an output stream. This resolves to a call to `AMI_single_merge_dh` in all cases. However the merge heap object used in that call is customized according to which sort template was invoked by the application programmer.

The merge sorting algorithm used by `AMI_partition_and_merge_dh` is outlined below:

1. If `sort_fits_in_memory`

   (a.) `main_mem_operate_init`

   (b.) `main_mem_operate`

   (c.) `main_mem_operate_cleanup`

   else

   2. Partition the input into memoryloads, sort each one using

      (a.) `main_mem_operate_init`
      (b.) `main_mem_operate`
      (c.) `main_mem_operate_cleanup`

      and write them out into separate streams.

      3. If sorting can be completed by a single pass of external merging, then do so using

         (a.) `single_merge`

      4. Recursively merge as many streams as memory constraints allow, using

         (a.) `single_merge`

         until only a single output stream remains.

One important difference between the `AMI_sort_V1()` and the `AMI_sort()`, `AMI_ptr_sort()`, and `AMI_key_sort()` implementations is in the way they store the (sub-) streams to be merged after the initial in-memory sorting is performed. The number, say R, of streams (files) being merged together at any time is the maximum possible for the amount of memory available. In contrast to the `AMI_sort_V1()` implementations which only have two streams (files) active at any time with the R substreams stored 'inside' them, the newer implementations have R files active. While the R sub-streams sent for merging by the `AMI_sort()` function all have the same parent stream (file), the R sub-streams sent for merging by the new implementations all reside in different parent streams (files). Thus, each stream (file) involved is accessed in a sequential manner. This can greatly improve performance, apparently because the operating system can perform read-ahead more effectively on the individual streams (files)-61-.

[61]: LA: This section really needs to be rewritten. DH: Ok now?

File: `include/mergeheap_dh.h`

This file defines the following classes:

- `merge_heap_dh_op`
- `merge_heap_dh_cmp`
- `merge_heap_dh_obj`
- `merge_heap_pdh_op`
- `merge_heap_pdh_cmp`
- `merge_heap_pdh_obj`
- `merge_heap_dh_kobj`

Each of the above classes implements the following member functions:

- `allocate`: Allocates space for the heap.
- `insert`: Inserts (copies) an element into the heap array.
- `deallocate`: Deallocates the space used by the heap.
- `initialize`: Makes the heap array into a heap by repeatedly calling internal member function `Heapify`.[3]
- `get_min_run_id`: Returns the number of the run which has the smallest element in the heap.
- `delete_min_and_insert`: Replaces the smallest item in the heap with a new element and re-heapifies the heap.
- `sizeofheap`: Returns the number of elements in the heap.

[62]: LA: Knuth forecasting stuff?

6.4.7 Distribution

<TO BE WRITTEN>

6.4.8 Key Bucket Sorting

<TO BE WRITTEN>

6.4.9 General Permuting

<TO BE WRITTEN>

6.4.10 Bit Permuting

<TO BE WRITTEN>

[3] See [24] for more details if required.

6.4.11 Dense Matrices

<TO BE WRITTEN>

6.4.12 Sparse Matrices

<TO BE WRITTEN>

6.4.13 Stacks

<TO BE WRITTEN>

6.4.14 Elementwise Arithmetic

<TO BE WRITTEN>

Chapter 7

Configuration and Performance Tuning

7.1 TPIE Configuration

[63]: LA: Should there be something about compiling in this chapter (and about e.g including ami.h)?

Certain behaviours of TPIE at run-time are controlled by compile-time variables, whose values should be defined before including any TPIE headers. Depending on the options desired, the values of these variables can be specified as early as when TPIE is installed, or as late as when an individual application program is compiled. Section 7.1.1 described the options available at installation time. Section 7.1.2 describes how TPIE can be configured differently for individual TPIE applications.

7.1.1 Installation Options

[64]: LA: We need to change this when we change logging

It is possible to customize certain TPIE behaviours by providing arguments to the configure script when TPIE is first installed (see Section 2.4. None of these arguments are necessary and the first time you build TPIE you should not need any of them. The arguments recognized are as follows:

`--enable-log-lib` Enable logging in TPIE library code. This can also be accomplished at compile time by defining the macro `TP_LOG_LIB` using the syntax `make lib TP_LOG_LIB=1`. This is useful for debugging the TPIE library, but slows it down. This option works by defining `TPL_LOGGING` when compiling the library. Section 7.3 discusses TPIE logging.

`--enable-assert-lib` Enable assertions in the TPIE library code for debugging purposes. This can also be accomplished at compile time by defining the macro `TP_ASSERT_LIB` using the syntax `make lib TP_ASSERT_LIB=1`. This option works by defining `DEBUG_ASSERTIONS`

when compiling the library.

`--enable-log-apps` and

`--enable-assert-apps` Similar to `--enable-log-lib` and `--enable-assert-lib`, but they apply to the test application code. Running `make test` with the options `TP_LOG_APPS=1` and/or `TP_ASSERT_APPS=1` accomplishes the same thing.

`--enable-expand-ami-scan` Expand the macros in the file `ami_scan.h` when making the include directory with the command `make include` (or `make all`). This is mainly useful for debugging the code in `ami_scan.h` itself, and is not normally needed by TPIE programmers. It may make compilation of TPIE programs slightly faster because the macro processor of the C++ compiler will have less work to do. In addition to the standard GNU tools mentioned in Section 2.3, this requires `perl`.

`--disable-*` Any of the options above can be explicitly disabled by using this syntax. For example `--disable-expand-ami-scan`.

7.1.2 Configuring TPIE for Individual Applications

Certain TPIE configuration options can be selected by setting compile-time variables in the file `test/app_config.h` which is then included in an application program. A typical example of this file can be found in the `test` directory. Selected parts of the file are shown and discussed below.[65]

[65]: LA: Something general about TPIE configuration as set up by the configure-script needs to be included here (e.g. discuss config.h file).

The File `test/app_config.h`

```
// Get the configuration as set up by the TPIE configure script.
#include <config.h>

// <><><><><><><><><><><><><><><><><><><><><><> //
// <><><><><><><> Developer use  <><><><><><><> //
// <><><><><><><><><><><><><><><><><><><><><><> //



// <><><><><><><><><><><><><><><><><><><><><><> //
// <><><> Choose default BTE COLLECTION  <><><> //
// <><><><><><><><><><><><><><><><><><><><><><> //

// Define only one (default is BTE_COLLECTION_IMP_MMAP)
#define BTE_COLLECTION_IMP_MMAP
//#define BTE_COLLECTION_IMP_UFS
//#define BTE_COLLECTION_IMP_USER_DEFINED


// <><><><><><><><><><><><><><><><><><><><><><> //
// <><><><><><> Choose BTE STREAM  <><><><><><> //
// <><><><><><><><><><><><><><><><><><><><><><> //

// Define only one (default is BTE_STREAM_IMP_UFS)
#define BTE_STREAM_IMP_UFS
//#define BTE_STREAM_IMP_MMAP
//#define BTE_STREAM_IMP_STDIO
//#define BTE_STREAM_IMP_USER_DEFINED


// <><><><><><><><><><><><><><><><><><><><><><><><> //
// <><> BTE_STREAM_MMAP configuration options  <><> //
// <><><><><><><><><><><><><><><><><><><><><><><><> //

#ifdef BTE_STREAM_IMP_MMAP
   // Define logical blocksize factor (default is 32)
#  define BTE_STREAM_MMAP_BLOCK_FACTOR 32
   // Enable/disable TPIE read ahead; default is enabled (set to 1)
#  define BTE_STREAM_MMAP_READ_AHEAD 1
   /* read ahead method, ignored unless BTE_STREAM_MMAP_READ_AHEAD is set to 1;
   if USE_LIBAIO is enabled, use asynchronous IO read ahead; otherwise
   use use mmap-based read ahead; default is mmap-based read ahead
   (USE_LIBAIO not defined) */
   //#define USE_LIBAIO
#endif


// <><><><><><><><><><><><><><><><><><><><><><><><> //
// <><> BTE_STREAM_UFS configuration options <><><> //
// <><><><><><><><><><><><><><><><><><><><><><><><> //
```

```
#ifdef BTE_STREAM_IMP_UFS
   // Define logical blocksize factor (default is 32)
#  define BTE_STREAM_UFS_BLOCK_FACTOR 32
   // Enable/disable TPIE read ahead; default is disabled (set to 0)
#  define BTE_STREAM_UFS_READ_AHEAD 0
   /* read ahead method, ignored unless BTE_STREAM_UFS_READ_AHEAD is set to 1;
   if USE_LIBAIO is set to 1, use asynchronous IO read ahead;
   otherwise no TPIE read ahead is done; default is disabled (set to
   0) */
#  define USE_LIBAIO 0
#endif



// <><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><> //
//                     logging and assertions;                      //
//              this should NOT be modified by user!!!              //
//       in order to enable/disable library/application logging,    //
//     run tpie configure script with appropriate options           //
// <><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><> //

// Use logs if requested.
#if TP_LOG_APPS
#define TPL_LOGGING 1
#endif

#include <tpie_log.h>

// Enable assertions if requested.
#if TP_ASSERT_APPS
#define DEBUG_ASSERTIONS 1
#endif
#include <tpie_assert.h>
```

Compile-Time Options in `test/app_config.h`

`BTE_STREAM_IMP_MMAP`

`BTE_STREAM_IMP_STDIO`

`BTE_STREAM_IMP_UFS` Used to choose which of the available Block Transfer Engine (see Section 6.2) implementations to use. Version 082902 of TPIE is distributed with three BTEs and the desired BTE is chosen by defining `BTE_STREAM_IMP_STDIO`, `BTE_STREAM_IMP_MMAP` or `BTE_STREAM_IMP_UFS`. See Section 6.2 for a discussion of the implementation details in these BTEs. The next section discusses how to choose an appropriate BTE for a given application in order to obtain maximal performance.

If `BTE_STREAM_IMP_MMAP` or `BTE_STREAM_IMP_UFS` is defined, the following macros are used to control BTE options (how to set the options for maximal performance is discussed in the next section):

`BTE_STREAM_MMAP_BLOCK_FACTOR`

`BTE_STREAM_UFS_BLOCK_FACTOR` The value of this variable determines the logical blocksize used by the BTE as a multiple of the physical block size (refer to Section 6.2). A value of 1 indicates that the logical blocksize is the same as the physical blocksize of the OS.

`BTE_STREAM_MMAP_READ_AHEAD`

`BTE_STREAM_UFS_READ_AHEAD` Defining this variable instructs the corresponding BTE (`BTE_stream_mmap` or `BTE_stream_ufs`) to optimize for sequential read speed by reading blocks into main memory before the data they contain is actually needed.

`USE_LIBAIO` If `BTE_STREAM_MMAP_READ_AHEAD` is defined, defining `USE_LIBAIO` results in read ahead being performed using the asynchronous I/O library `libaio`. If the macro `USE_LIBAIO` is not defined the read ahead is done using `mmap` and double buffering in the case of `BTE_STREAM_IMP_MMAP` and not done at all in the case of `BTE_STREAM_IMP_UFS` (refer to Section 6.2).

The rest of the compile-time variables are normally not modified by TPIE application programmers:

`AMI_STREAM_IMP_SINGLE` This macro controls which Access Method Interface implementation (see Section 6.2) to use. Version 082902 of TPIE is distributed with a single AMI implementation, which stores the contents of a given stream on a single disk. This implementation is selected by defining `AMI_STREAM_IMP_SINGLE`.

[66]: LA: Is this correct?

`TPL_LOGGING` Set to a non-zero value to enable logging of TPIE's internal behavior[66]. By default, information is logged to the log file `/tmp/TPLOG_XXX` where `XXX` is a unique system dependent identifier. Typically it encodes the process ID of the TPIE process that produced it in some way. See Section 7.3 for information on exactly what TPIE writes to the log file.

`DEBUG_ASSERTIONS` Define to enable TPIE assertions. These assertions check for inconsistent or erroneous conditions within TPIE itself. They are primarily intended to aid in the debugging of TPIE. Some overhead is added to programs compiled with this macro set.

`DEBUG_CERR` Defining this macro tells TPIE to write all internal assertion messages to the C++ standard error stream `cerr` in addition to the TPIE log file.

`DEBUG_STR` Defining this macro enables certain debugging messages that report on the internal behavior of TPIE but do not necessarily indicate error conditions. In some cases this can increase the size of the log dramatically.

7.1.3 Environment Variables

In version 082902 of TPIE there is only one environment variable. The variable is called `AMI_SINGLE_DEVICE` and defines where TPIE places temporary streams. The default location is `/var/tmp`. If a different location is desired, `AMI_SINGLE_DEVICE` must be set accordingly. For example (in C-shell): setenv AMI_SINGLE_DEVICE /usr/project/tmp/.

7.2 TPIE Performance Tuning

7.2.1 Choosing and Configuring a BTE Implementation

[67]: LA: This needs an overhaul!

Choosing an appropriate BTE implementation (and BTE parameter settings) for best performance is both application and system dependent. (See section 6.2 for a description of the three existing BTEs). Theoretically, `BTE_stream_mmap` should have the best performance for most applications, because space and copy time is saved relative to `BTE_stream_stdio` and `BTE_stream_ufs` as steam objects do not have to pass through kernel level buffer space when accessed. On the other hand, buffering and prefetching has to be explicitly implemented in `BTE_stream_mmap` whereas it is (typically) done by the OS in `BTE_stream_stdio` and `BTE_stream_ufs`. Also theoretically, `BTE_stream_ufs` (and `BTE_stream_mmap`) should perform better than `BTE_stream_stdio` because of fewer kernel calls and because of the (possible) larger logical block size. However, in practice the performance of the three BTE's are very system (and application) dependent. This is for example due to different implementations of the `fread()`, `fwrite()`, `read()`, `write()`, `mmap()`, and `munmap()` calls on different machines[68].

[68]: LA: Other reasons?

The most important BTE configuration parameter is `BTE_LOGICAL_BLOCKSIZE_FACTOR` (but note that this parameter only applies to `BTE_stream_mmap` and `BTE_stream_ufs`). The size of each buffer and the size of each I/O in the BTE stream is `BTE_LOGICAL_BLOCKSIZE_FACTOR` times the operating system blocksize, so this roughly corresponds to the amount of data brought in or written out at the cost of a single disk operation. Increasing this parameter, therefore, can reduce the number of I/O operations required to read through a stream from beginning to end.

However, the amount of memory dedicated to a stream is either `BTE_LOGICAL_BLOCKSIZE_FACTOR` times the operating system blocksize (no prefetching) or twice `BTE_LOGICAL_BLOCKSIZE_FACTOR` times the operating system blocksize (if prefetching is enabled). So the value of the `BTE_LOGICAL_BLOCKSIZE_FACTOR` parameter, together with available memory, determines the number of BTE streams (and hence AMI streams) that can be active at the same time. This gives an upper bound on the arity of a multi-way merge or a multi-way distribution operation that can be undertaken by a TPIE application. This in turn can have a crucial impact on (say, the number of passes required in external sorting and hence the) net running time.

A large value for `BTE_LOGICAL_BLOCKSIZE_FACTOR` increases performance due to fewer kernel calls and due to the (track) buffering and prefetching in the disk controller. Too large a value results in decreased performance due to the BTE's use of main memory. Thus this parameter should be chosen carefully.

As far at the other BTE configuration parameters (prefetching) are concerned, the default settings in the `app_config.h` file in the `test` directory are normally the best.

In order to help in deciding which BTE to choose for a given application/system, as well as deciding on what logical block size to use (in `BTE_stream_mmap` and `BTE_stream_ufs`), we have included a C program in the `test` directory of the TPIE distribution called `bte_test.c`. This program can be used to determine the streaming speeds attained by `BTE_stream_stdio`, `BTE_stream_mmap`, and `BTE_stream_ufs` streams on a given system. The program simulates the buffering and I/O mechanisms used by each of the BTE stream implementations so that the "raw" (in the sense that there is no TPIE layer between the program and the filesystem) streaming speed of an I/O-buffering mechanism combination can be determined. To use the program, define one of `MMAP_TEST`, `READ_WRITE_TEST`[69] or `STDIO_TEST` in the program depending on whether you want to test the streaming speed of `BTE_stream_mmap`, `BTE_stream_ufs` or `BTE_stream_stdio`. Also define the `BLOCKSIZE_BASE` parameter to be equal to the underlying operating system blocksize.[70] Compile the program using a C compiler. In order to test the streaming performance of BTE streams of objects of size `ItemSize`, the program first writes out some specified number `NumStreams` of BTE streams containing a specified number `NumItems` of items. Then it carries out a perfect `NumStreams`-way interleaving of the streams via a simple merge like process, writing the output to an output stream. During the computation, each of the `NumStreams` streams input to the merge, as well as the stream being output by the merge uses either one (when `READ_WRITE_TEST` or `STDIO_TEST` are set to 1) or two (when `MMAP_TEST` is set to 1) buffers. In case of `STDIO_TEST`, the buffers are not maintained in the program but by the stdio library. In the case of `MMAP_TEST` or `READ_WRITE_TEST`, each buffer is set to be of size `block_factor` times `BLOCKFACTOR_BASE`, and each I/O operation corresponds to a buffer-sized operation. To test the streaming performance of a BTE stream with `items_in_block` items in each block simply execute:

```
bte_test NumItems ItemSize NumStreams block_factor items_in_block DataFile
```

The output of the program (streaming speed) is appended to the file `DataFile`. The streaming speed, alternatively called I/O Bandwidth, is given in units of MB/s, and can be used to decide which BTE to use and how to configure it.

[69]: LA: Why not UFS_TEST?

[70]: LA: Why not automatic?

7.2.2 Other Factors Affecting Performance

In addition to the choice (and configuration) of BTE, a number of other factors, not all of which are TPIE specific, can effect the performance of a TPIE application.

Inlining operation management object methods Failing to inline the `operate()` method of operation management objects can be a major source of lackluster performance of an application, since `operate()` is called once for every object in a stream being scanned. Inlining of `operate()` is, however, just a suggestion to the compiler, which can choose to ignore it. In order to maximize the likelihood of inlining, it is a good idea to keep the function short and simple. One way of doing this is to wrap complex pieces of code that are called less often in separate functions.

`gcc` **optimization** We recommend using the `-O2` level of optimization of `gcc` in order to obtain the best overall performance. Although better performance can normally be obtained using `-O3`, this optimization leads to increased program size which can potentially result in decreased performance.

Memory size To insure that no disk swapping is done by the OS, the size of main memory used by TPIE (set by `MM_manager.set_memory_limit()`, see Section 4.13 and Section 5.1) should be set to a realistic value. The best value is usually much smaller than the size of the memory installed in the computer (due to memory use of operating system resources and daemons).

7.3 TPIE Logging

[71]: LA: The whole logging thing needs to be reevaluated and changed. DH: Now includes recent changes.

When logging is turned on (see Section 7), TPIE creates a log file in `/tmp/TPLOG_XXXXXX`, where `XXXXXX` is a unique system dependent identifier. TPIE writes into this file using a `logstream` class, which is derived from `ofstream` and has the additional functionality of setting a priority and a threshold for logging. If the priority of a message is below the threshold, the message is not logged. There are four priority levels defined in TPIE, as follows.

`TP_LOG_FATAL` is the highest level and is used for all kinds of errors that would normally impair subsequent computations. Errors are always logged;

`TP_LOG_WARNING` is the next lowest and is used for warnings.

`TP_LOG_APP_DEBUG` can be used by applications built on top of TPIE, for logging debugging information.

`TP_LOG_DEBUG` is the lowest level and is used by the TPIE library for logging debugging information.

By default, the threshold of the log is set to the lowest level, `TP_LOG_WARNING`. To change the threshold level, the following macro is provided:

`LOG_SET_THRESHOLD(`*level*`)`

where *level* is one of `TP_LOG_FATAL`, `TP_LOG_WARNING`, `TP_LOG_APP_DEBUG`, or `TP_LOG_DEBUG`.

The threshold level can be reset as many times as needed in a program. This enables the developer to focus the debugging effort on a certain part of the program.

The following compile-time macros are provided for writing into the log:

`LOG_FATAL(`*msg*`)` `LOG_FATAL_ID(`*msg*`)`

`LOG_WARNING(`*msg*`)` `LOG_WARNING_ID(`*msg*`)`

`LOG_APP_DEBUG(`*msg*`)` `LOG_APP_DEBUG_ID(`*msg*`)`

`LOG_DEBUG(`*msg*`)` `LOG_DEBUG_ID(`*msg*`)`

where *msg* is the information to be logged; *msg* can be any type that is supported by the `C++ fstream` class. Each of these macros sets the corresponding priority and sends *msg* to the log stream. The macros ending in `_ID` record the source code filename and line number in the log, while the corresponding macros without the `_ID` suffix do not.

Part III

Appendices

Appendix A

Test and Sample Applications

A.1 General Structure and Operation

The test and sample applications distributed with TPIE are in the `test` directory. The test programs are designed primarily to test the operation of the system to verify that it has been installed correctly and is as bug free as possible. These applications all have names of the form `test_*`. The sample applications are designed to demonstrate the use of TPIE in the solution of non-trivial problems.

The test and sample applications all share a small amount of common initialization and argument parsing code. They all include the header file `app_config.h`, which selects a particular implementation of streams at the AMI and BTE levels. They also all use the same argument parsing function `parse_args()`, which parses certain default arguments and then uses a callback function for arguments specific to the particular application.

Much of the functionality provided by the common initialization and argument passing code is intended to eventually be subsumed by operating system provided services. For example, the amount of main memory a particular application is permitted to use can be set via a command line argument. It is up to the user to be sure that this number is reasonable and does not exceed the true amount of main memory available to the application. In the future, it is hoped that this information will be provided by the operating system.

`parse_args()` is declared as follows:

```
parse_args();
```

The following is a summary of the common command line arguments that are parsed by `parse_args()`.

`-t testsize` Set the size of the test to be run to `testsize`. Typically this is the number of objects to be put into the application produced input stream. In matrix tests, however, it is the number of rows and columns in the test matrices. If this argument is not provided, then a default value of 8 MB is used.

`-m memsize` The number of bytes of main memory that the application is permitted to use. The TPIE Memory Manager will ensure that no more than this amount is used. If this option is not specified, then a default value of 2 MB is used.

`-z randomseed` Seed the random number generator with the value `randomseed`. This is useful for debugging or testing, when we want several runs of the application to rely on the same series of pseudo-random numbers. For applications that do not generate test data randomly, this has no effect.

`-v` Turns on verbose mode. When running in verbose mode, report major actions of the running program to `stdout`.

Each application specific argument appears in the string pointed to by `aso` as a single character, possibly followed by the single character '`:`', indicating that the argument requires a value. For example, if `aso` pointed to the string "`ax:z`" then the following command line arguments would all be parsed correctly:

```
-a

-x 123

-a -x 123

-ax123

-x123 -a
```

In each case, `parse_app()` would be called to take some application specific action for each of the arguments. It would be called once with `opt` set to ‘`a`’ and `optarg` set to `NULL`, and/or once with `opt` set to ‘`x`’ and `optarg` pointing to the string “`123`.” When multiple arguments are present on the command line, they are parsed from left to right.

The following is an example of how a test application, in this case `test_ami_sort`, can use application specific command line arguments to set up it’s global state.

```
static const char as_opts[] = "R:S:rsao";
void parse_app_opt(char c, char *optarg)
{
    switch (c) {
        case 'R':
            rand_results_filename = optarg;
        case 'r':
            report_results_random = true;
            break;
        case 'S':
            sorted_results_filename = optarg;
        case 's':
            report_results_sorted = true;
            break;
        case 'a':
            sort_again = !sort_again;
            break;
        case 'o':
            use_operator = !use_operator;
            break;
    }
}

int main(int argc, char **argv)
{
    parse_args(argc,argv,as_opts,parse_app_opt);

    ...

    return 0;
}
```

A.2 Test Programs

The test programs include with TPIE are as follows[72]:

`test_ami_merge` Test fixed way merging with direct calls to `AMI_merge()`, as described in Section 5.6.

`test_ami_pmerge` Test many-way merging using `AMI_partition_and_merge()`, as described in Section 5.6.

`test_ami_sort` Test sorting using `AMI_sort()` as described in Section 5.10.

[72]: LA: Is this still correct?

`test_ami_gp` Test general permutation using `AMI_general_permute()` as described in Section ??. The program generates an input stream consisting of sequential integers, and outputs a stream consisting of the same integers, in reverse order.

`test_ami_bp` Test bit permutations using `AMI_BMMC_permute()` as described in Section ??. The program generates an input stream consisting of sequential integers, and outputs a stream consisting of a permutation of these integers, as described in the example given in the Tutorial, Section 4.9.2.

`test_matrix`

`test_bit_matrix` Test main memory matrix manipulation and arithmetic. This is used both by the bit permuting code described in Section ?? and the dense matrix multiplication code described in Section ?? for internal manipulation of sub-matrices of external memory matrices.

`test_ami_matrix_pad` Test padding of external matrices. This is the preprocessing step for the external dense matrix multiplication algorithm TPIE uses, which is described in Section ??.

`test_ami_matrix` Test external dense matrix arithmetic as described in Section ??.

`test_ami_sm` Test external sparse matrix arithmetic as described in Section ??.

`test_ami_stack` Test external memory stacks as described in Section 5.12.

`test_ami_arith` Test element-wise arithmetic on external memory streams as described in Section ??. The program generates an input stream consisting of sequential integers, squares them, and performs elementwise division between the resulting stream and the input stream.

A.3 Sample Applications

The sample applications included with TPIE are as follows:

`ch2` Two dimensional convex hull program using Graham's scan. It is implemented using a scan management object that maintains the upper and lower hull internally as external memory stacks. Much of the code in this application appears in Section B.1.

`lr` An implementation of an asymptotically optimal list ranking algorithm. The idea of geometrically decreasing computation is used. Much of the code in this application appears in Section B.2.

`nas_ep` An I/O-efficient implementation of the NAS EP parallel benchmark. This benchmark generates pairs of independent Gaussian random variates.

`nas_is` An I/O-efficient implementation of the NAS IS parallel benchmark. This benchmark sorts integers using one of a variety of approaches.

Detailed descriptions of the NAS parallel benchmarks are available from the NAS Parallel Benchmark Home Page at URL `http://www.nas.nasa.gov/NAS/NPB/`.

Appendix B

Additional Examples

This chapter contains some additional annotated examples of TPIE application code.[73]

[73]: LA: Is this chapter still ok?

B.1 Convex Hull

The convex hull of a set of points in the plane is the smallest convex polygon which encloses all of the points. Graham's scan is a simple algorithm for computing convex hulls. It should be discussed in any introductory book on computational geometry, such as [49]. Although Graham's scan was not originally designed for external memory, it can be implemented optimally in this setting. What is interesting about this implementation is that external memory stacks are used within the implementation of a scan management object.

First, we need a data type for storing points. We use the following simple class, which is templated to handle any numeric type.

```
template<class T>
class point {
public:
    T x;
    T y;
    point() {};
    point(const T &rx, const T &ry) : x(rx), y(ry) {};
    ~point() {};

    inline int operator==(const point<T> &rhs) const {
        return (x == rhs.x) && (y == rhs.y);
    }
    inline int operator!=(const point<T> &rhs) const {
        return (x != rhs.x) || (y != rhs.y);
    }

    // Comparison is done by x.
    int operator<(const point<T> &rhs) const {
        return (x < rhs.x);
    }

    int operator>(const point<T> &rhs) const {
        return (x > rhs.x);
    }

    friend ostream& operator<<(ostream& s, const point<T> &p);
    friend istream& operator>>(istream& s, point<T> &p);
};
```

Once the points are s by their x values, we simply scan them to produce the upper and lower hulls, each of which are stored as a stack pointed to by the scan management object. We then concatenate the stacks to produce the final hull. The code for computing the convex hull of a set of points is thus

```
template<class T>
AMI_err convex_hull(AMI_STREAM< point<T> > *instream,
                    AMI_STREAM< point<T> > *outstream)
{
    AMI_err ae;

    point<T> *pt;

    AMI_stack< point<T> > uh((unsigned int)0, instream->stream_len());
    AMI_stack< point<T> > lh((unsigned int)0, instream->stream_len());

    AMI_STREAM< point<T> > in_sort;

    // Sort the points by x.

    ae = AMI_sort(instream, &in_sort);

    // Compute the upper hull and lower hull in a single scan.

    scan_ul_hull<T> sulh;

    sulh.uh_stack = &uh;
    sulh.lh_stack = &lh;

    ae = AMI_scan(&in_sort, &sulh);

    // Copy the upper hull to the output.

    uh.seek(0);

    while (1) {
        ae = uh.read_item(&pt);
        if (ae == AMI_ERROR_END_OF_STREAM) {
            break;
        } else if (ae != AMI_ERROR_NO_ERROR) {
            return ae;
        }

        ae = outstream->write_item(*pt);
        if (ae != AMI_ERROR_NO_ERROR) {
            return ae;
        }
    }

    // Reverse the lower hull, concatenating it onto the upper hull.

    while (lh.pop(&pt) == AMI_ERROR_NO_ERROR) {
        ae = outstream->write_item(*pt);
        if (ae != AMI_ERROR_NO_ERROR) {
            return ae;
        }
    }

    return AMI_ERROR_NO_ERROR;
}
```

The only thing that remains is to define a scan management object that is capable of producing the upper and lower hulls by scanning the points. According to the Graham's scan algorithm, we produce the upper hull by moving forward in the x direction, adding each point we encounter to the upper hull, until we find one that induces a concave turn on the surface of the hull. We then move backwards through the list of points that have been added to the hull, eliminating points until a convex path is reestablished. This process is made efficient by storing the points on the hull so far in a stack. The code for the scan management object, which relies on the function `ccw()` to actually determine whether a corner is convex or not, is as follows:

```
template<class T>
class scan_ul_hull : AMI_scan_object {
public:
    AMI_stack< point <T> > *uh_stack, *lh_stack;

    scan_ul_hull(void);
    virtual ~scan_ul_hull(void);
    AMI_err initialize(void);
    AMI_err operate(const point<T> &in, AMI_SCAN_FLAG *sfin);
};

template<class T>
scan_ul_hull<T>::scan_ul_hull(void) : uh_stack(NULL), lh_stack(NULL)
{
}

template<class T>
scan_ul_hull<T>::~scan_ul_hull(void)
{
}

template<class T>
AMI_err scan_ul_hull<T>::initialize(void)
{
    return AMI_ERROR_NO_ERROR;
}


template<class T>
AMI_err scan_ul_hull<T>::operate(const point<T> &in,
                                 AMI_SCAN_FLAG *sfin)
{
    AMI_err ae;

    // If there is no more input we are done.
    if (!*sfin) {
        return AMI_SCAN_DONE;
    }

    if (!uh_stack->stream_len()) {

        // If there is nothing on the stacks then put the first point
        // on them.
        ae = uh_stack->push(in);
        if (ae != AMI_ERROR_NO_ERROR) {
            return ae;
        }

        ae = lh_stack->push(in);
        if (ae != AMI_ERROR_NO_ERROR) {
            return ae;
        }

    } else {

        // Add to the upper hull.

        {
            // Pop the last two points off.

            point<T> *p1, *p2;

            tp_assert(uh_stack->stream_len() >= 1, "Stack is empty.");

            uh_stack->pop(&p2);

            // If the point just popped is equal to the input, then we
            // are done.  There is no need to have both on the stack.

            if (*p2 == in) {
                uh_stack->push(*p2);
                return AMI_SCAN_CONTINUE;
            }

            if (uh_stack->stream_len() >= 1) {
                uh_stack->pop(&p1);
            } else {
                p1 = p2;
            }

            // While the turn is counter clockwise and the stack is
            // not empty pop another point.

            while (1) {
                if (ccw(*p1,*p2,in) >= 0) {
                    // It does not turn the right way.  The points may
                    // be colinear.
                    if (uh_stack->stream_len() >= 1) {
                        // Move backwards to check another point.
                        p2 = p1;
                        uh_stack->pop(&p1);
                    } else {
                        // Nothing left to pop, so we can't move
                        // backwards.  We're done.
                        uh_stack->push(*p1);
                        if (in != *p1) {
                            uh_stack->push(in);
                        }
                        break;
                    }
                } else {
                    // It turns the right way.  We're done.
                    uh_stack->push(*p1);
                    uh_stack->push(*p2);
                    uh_stack->push(in);
                    break;
                }
            }
        }
        // Add to the lower hull.

        {
            // Pop the last two points off.

            point<T> *p1, *p2;

            tp_assert(lh_stack->stream_len() >= 1, "Stack is empty.");

            lh_stack->pop(&p2);

            // If the point just popped is equal to the input, then we
            // are done.  There is no need to have both on the stack.

            if (*p2 == in) {
                lh_stack->push(*p2);
                return AMI_SCAN_CONTINUE;
            }

            if (lh_stack->stream_len() >= 1) {
                lh_stack->pop(&p1);
            } else {
                p1 = p2;
            }

            // While the turn is clockwise and the stack is
            // not empty pop another point.

            while (1) {
                if (ccw(*p1,*p2,in) <= 0) {
                    // It does not turn the right way.  The points may
                    // be colinear.
                    if (lh_stack->stream_len() >= 1) {
                        // Move backwards to check another point.
                        p2 = p1;
                        lh_stack->pop(&p1);
                    } else {
                        // Nothing left to pop, so we can’t move
                        // backwards.  We’re done.
                        lh_stack->push(*p1);
                        if (in != *p1) {
                            lh_stack->push(in);
                        }
                        break;
                    }
                } else {
                    // It turns the right way.  We’re done.
                    lh_stack->push(*p1);
                    lh_stack->push(*p2);
                    lh_stack->push(in);
                    break;
                }
            }
        }
    }

    return AMI_SCAN_CONTINUE;
}
```

The function ccw() computes twice the signed area of a triangle in the plane by evaluating a 3 by 3 determinant. The result is positive if and only if the the three points in order form a counterclockwise cycle.

```
template<class T>
T ccw(const point<T> &p1, const point<T> &p2, const point<T> &p3)
{
    T sa;

    sa = ((p1.x * p2.y - p2.x * p1.y) -
          (p1.x * p3.y - p3.x * p1.y) +
          (p2.x * p3.y - p3.x * p2.y));

    return sa;
}
```

B.2 List-Ranking

List ranking is a fundamental problem in graph theory. The problem is as follows: We are given the directed edges of a linked list in some arbitrary order. Each edge is an ordered pair of node ids. The first is the source of the edge and the second is the destination of the edge. Our goal is to assign a weight to each edge corresponding to the number of edges that would have to be traversed to get from the head of the list to that edge.

The code given below solves the list ranking problem using a simple randomized algorithm due to Chiang *et al.* [19]. As was the case in the code examples in the tutorial in Chapter 4, #include statements for header files and definitions of some classes and functions as well as some error and consistency checking code are left out so that the reader can concentrate on the more important details of how TPIE is used. A complete ready to compile version of this code is included in the TPIE source distribution.

First, we need a class to represent edges. Because the algorithm will set a flag for each edge and then assign weights to the edges, we include fields for these values.

```
class edge {
public:
    unsigned long int from;        // Node it is from
    unsigned long int to;          // Node it is to
    unsigned long int weight;      // Position when ranked.
    bool flag;            // A flag used to randomly select some edges.

    friend ostream& operator<<(ostream& s, const edge &e);
};
```

As the algorithm runs, it will sort the edges. At times this will be done by their sources and at times by their destinations. The following simple functions are used to compare these values:

```
int edgefromcmp(CONST edge &s, CONST edge &t)
{
    return (s.from < t.from) ? -1 : ((s.from > t.from) ? 1 : 0);
}

int edgetocmp(CONST edge &s, CONST edge &t)
{
    return (s.to < t.to) ? -1 : ((s.to > t.to) ? 1 : 0);
}
```

The first step of the algorithm is to assign a randomly chosen flag, whose value is 0 or 1 with equal probability, to each edge. This is done using AMI_scan() with a scan management object of the class random_flag_scan, which is defined as follows:

```
class random_flag_scan : AMI_scan_object {
public:
    AMI_err initialize(void);
    AMI_err operate(const edge &in, AMI_SCAN_FLAG *sfin,
                    edge *out, AMI_SCAN_FLAG *sfout);
};

AMI_err random_flag_scan::initialize(void) {
    return AMI_ERROR_NO_ERROR;
}

AMI_err random_flag_scan::operate(const edge &in, AMI_SCAN_FLAG *sfin,
                                  edge *out, AMI_SCAN_FLAG *sfout)
{
    if (!(sfout[0] = *sfin)) {
        return AMI_SCAN_DONE;
    }
    *out = in;
    out->flag = (random() & 1);

    return AMI_SCAN_CONTINUE;
}
```

The next step of the algorithm is to separate the edges into an active list and a cancel list. In order to do this, we sort one copy of the edges by their sources (using `edgefromcmp`) and sort another copy by their destinations (using `edgetocmp`). We then call `AMI_scan()` to scan the two lists and produce an active list and a cancel list. A scan management object of class `separate_active_from_cancel` is used.

```
////////////////////////////////////////////////////////////////////////
// separate_active_from_cancel
//
// A class of scan object that takes two edges, one to a node and one
// from it, and writes an active edge and possibly a canceled edge.
//
// Let e1 = (x,y,w1,f1) be the first edge and e2 = (y,z,w2,f2) the second.
// If e1's flag (f1) is set and e2's (f2) is not, then we write
// (x,z,w1+w2,?) to the active list and e2 to the cancel list.  The
// effect of this is to bridge over the node y with the new active edge.
// f2, which was the second half of the bridge, is saved in the cancellation
// list so that it can be ranked later after the active list is recursively
// ranked.
//
// Since all the flags should have been set randomly before this function
// is called, the expected size of the active list is 3/4 the size of the
// original list.
////////////////////////////////////////////////////////////////////////
class separate_active_from_cancel : AMI_scan_object {
public:
    AMI_err initialize(void);
    AMI_err operate(CONST edge &e1, CONST edge &e2, AMI_SCAN_FLAG *sfin,
                    edge *active, edge *cancel, AMI_SCAN_FLAG *sfout);
};

AMI_err separate_active_from_cancel::initialize(void)
{
    return AMI_ERROR_NO_ERROR;
}

// e1 is from the list of edges sorted by where they are from.
// e2 is from the list of edges sorted by where they are to.
AMI_err separate_active_from_cancel::operate(CONST edge &e1,
                                             CONST edge &e2,
                                             AMI_SCAN_FLAG *sfin,
                                             edge *active, edge *cancel,
                                             AMI_SCAN_FLAG *sfout)
{
    // If we have both inputs.
    if (sfin[0] && sfin[1]) {
        // If they have a node in common we may be in a bridging situation.
        if (e2.to == e1.from) {
            // We will write to the active list no matter what.
            sfout[0] = 1;
            *active = e2;
            if (sfout[1] = (e2.flag && !e1.flag)) {
                // Bridge.  Put e1 on the cancel list and add its
                // weight to the active output.
                active->to = e1.to;
                active->weight += e1.weight;
                *cancel = e1;
                sfout[1] = 1;
            } else {
                // No bridge.
                sfout[1] = 0;
            }
        } else {
            // They don't have a node in common, so one of them needs
            // to catch up with the other.  What happened is that
            // either e2 is the very last edge in the list or e1 is
            // the very first or we just missed a bridge because of
            // flags.
            sfout[1] = 0;
            if (e2.to > e1.from) {
                // e1 is behind, so just skip it.
                sfin[1] = 0;
                sfout[0] = 0;
            } else {
                // e2 is behind, so put it on the active list.
                sfin[0] = 0;
                sfout[0] = 1;
                *active = e2;
            }
        }
        return AMI_SCAN_CONTINUE;
    } else {
        // If we only have one input, either just leave it active.
        if (sfin[0]) {
            *active = e1;
            sfout[0] = 1;
            sfout[1] = 0;
            return AMI_SCAN_CONTINUE;
        } else if (sfin[1]) {
            *active = e2;
            sfout[0] = 1;
            sfout[1] = 0;
            return AMI_SCAN_CONTINUE;
        } else {
            // We have no inputs, so we're done.
            sfout[0] = sfout[1] = 0;
            return AMI_SCAN_DONE;
        }
    }
}
```

The next step of the algorithm is to strip the cancelled edges away from the list of all edges. The remaining active edges will form a recursive subproblem. Again, we use a scan management object, this time of the class strip_active_from_cancel, which is defined as follows:

```
////////////////////////////////////////////////////////////////////////
//
// strip_cancel_from_active
//
// A scan management object to take an active list and remove the
// smaller weighted edge of each pair of consecutive edges with the
// same destination.  The purpose of this is to strip edges out of the
// active list that were sent to the cancel list.
//
////////////////////////////////////////////////////////////////////////
class strip_cancel_from_active : AMI_scan_object {
private:
    bool holding;
    edge hold;
public:
    AMI_err initialize(void);
    AMI_err operate(const edge &active, AMI_SCAN_FLAG *sfin,
                    edge *out, AMI_SCAN_FLAG *sfout);
};

AMI_err strip_cancel_from_active::initialize(void) {
    holding = false;
    return AMI_ERROR_NO_ERROR;
}

// Edges should be sorted by destination before being processed by
// this object.
AMI_err strip_cancel_from_active::operate(const edge &active,
                                  AMI_SCAN_FLAG *sfin,
                                  edge *out, AMI_SCAN_FLAG *sfout)
{
    // If no input then we're done, except that we might still be
    // holding one.
    if (!*sfin) {
        if (holding) {
            *sfout = 1;
            *out = hold;
            holding = false;
            return AMI_SCAN_CONTINUE;
        } else {
            *sfout = 0;
            return AMI_SCAN_DONE;
        }
    }

    if (!holding) {
        // If we are not holding anything, then just hold the current
        // input.
        hold = active;
        holding = true;
        *sfout = 0;
    } else {
        *sfout = 1;

        if (active.to == hold.to) {
            if (active.weight > hold.weight) {
                *out = active;
            } else {
                *out = hold;
            }

            holding = false;
        } else {
            *out = hold;
            hold = active;
        }
    }

    return AMI_SCAN_CONTINUE;
}
```

After recursion, we must patch the cancelled edges back into the recursively ranked list of active edges. This is done using a scan with a scan management object of the class `interleave_active_cancel`, which is implemented as follows:

```
////////////////////////////////////////////////////////////////////////
// interleave_active_cancel
//
// This is a class of merge object that merges two lists of edges
// based on their to fields.  The first list of edges should be active
// edges, while the second should be cancelled edges.  When we see two
// edges with the same to field, we know that the second was cancelled
// when the first was made active.  We then fix up the weights and
// output the two of them, one in the current call and one in the next
// call.
//
// The streams this operates on should be sorted by their terminal
// (to) nodes before AMI_scan() is called.
//
////////////////////////////////////////////////////////////////////////

class patch_active_cancel : AMI_scan_object {
private:
    bool holding;
    edge hold;
public:
    AMI_err initialize(void);
    AMI_err operate(CONST edge &active, CONST edge &cancel,
                    AMI_SCAN_FLAG *sfin,
                    edge *patch, AMI_SCAN_FLAG *sfout);
};

AMI_err patch_active_cancel::initialize(void)
{
    holding = false;
    return AMI_ERROR_NO_ERROR;
}

AMI_err patch_active_cancel::operate(CONST edge &active, CONST edge &cancel,
                                     AMI_SCAN_FLAG *sfin,
                                     edge *patch, AMI_SCAN_FLAG *sfout)
{
    // Handle the special cases that occur when holding an edge and/or
    // completely out of input.
    if (holding) {
        sfin[0] = sfin[1] = 0;
        *patch = hold;
        holding = false;
        *sfout = 1;
        return AMI_SCAN_CONTINUE;
    } else if (!sfin[0]) {
        *sfout = 0;
        return AMI_SCAN_DONE;
    }

    if (!sfin[1]) {
        // If there is no cancel edge (i.e. all have been processed)
        // then just pass the active edge through.
        *patch = active;
    } else {
        if (holding = (active.to == cancel.to)) {
            patch->from = active.from;
            patch->to = cancel.from;
            patch->weight = active.weight - cancel.weight;
            hold.from = cancel.from;
            hold.to = active.to;
            hold.weight = active.weight;
        } else {
            *patch = active;
            sfin[1] = 0;
        }
    }

    *sfout = 1;
    return AMI_SCAN_CONTINUE;

}
```

Finally, here is the actual function to rank the list.

```
////////////////////////////////////////////////////////////////////////
// list_rank()
//
// This is the actual recursive function that gets the job done.
// We assume that all weights are 1 when the initial call is made to
// this function.
//
// Returns 0 on success, nonzero otherwise.
////////////////////////////////////////////////////////////////////////

int list_rank(AMI_STREAM<edge> *istream, AMI_STREAM<edge> *ostream)
{
    AMI_err ae;

    off_t stream_len = istream->stream_len();

    AMI_STREAM<edge> *edges_rand;
    AMI_STREAM<edge> *active;
    AMI_STREAM<edge> *active_2;
```

```
AMI_STREAM<edge> *cancel;
AMI_STREAM<edge> *ranked_active;

AMI_STREAM<edge> *edges_from_s;
AMI_STREAM<edge> *cancel_s;
AMI_STREAM<edge> *active_s;
AMI_STREAM<edge> *ranked_active_s;

// Scan/merge management objects.
random_flag_scan my_random_flag_scan;
separate_active_from_cancel my_separate_active_from_cancel;
strip_cancel_from_active my_strip_cancel_from_active;
patch_active_cancel my_patch_active_cancel;

// Check if the recursion has bottomed out.  If so, then read in the
// array and rank it.

{
    size_t mm_avail;

    mm_avail = MM_manager.memory_available();

    if (stream_len * sizeof(edge) < mm_avail / 2) {
        edge *mm_buf = new edge[stream_len];
        istream->seek(0);
        istream->read_array(mm_buf,&stream_len);
        main_mem_list_rank(mm_buf,stream_len);
        ostream->write_array(mm_buf,stream_len);
        return 0;
    }
}

// Flip coins for each node, setting the flag to 0 or 1 with equal
// probability.

edges_rand = new AMI_STREAM<edge>;

AMI_scan(istream, &my_random_flag_scan, edges_rand);

// Sort one stream by source.  The original input was sorted by
// destination, so we don’t need to sort it again.

edges_from_s = new AMI_STREAM<edge>;

ae = AMI_sort(edges_rand, edges_from_s, edgefromcmp);

// Scan to produce and active list and a cancel list.

active = new AMI_STREAM<edge>;
cancel = new AMI_STREAM<edge>;

ae = AMI_scan(edges_from_s, edges_rand,
              &my_separate_active_from_cancel,
              active, cancel);

delete edges_from_s;
delete edges_rand;

// Strip the edges that went to the cancel list out of the active list.
```

```
    active_s = new AMI_STREAM<edge>;

    ae = AMI_sort(active, active_s, edgetocmp);

    delete active;

    active_2 = new AMI_STREAM<edge>;

    ae = AMI_scan(active_s,
                  &my_strip_cancel_from_active,
                  active_2);

    delete active_s;

    // Recurse on the active list.  The list we pass in is sorted by
    // destination.  The recursion will return a list sorted by
    // source.

    ranked_active = new AMI_STREAM<edge>;

    list_rank(active_2, ranked_active);

    delete active_2;

    cancel_s = new AMI_STREAM<edge>;

    AMI_sort(cancel, cancel_s, edgetocmp);

    delete cancel;

    // The output of the recursive call is not necessarily sorted by
    // destination.  We'll make it so before we try to merge in the
    // cancel list.

    ranked_active_s = new AMI_STREAM<edge>;

    AMI_sort(ranked_active, ranked_active_s, edgetocmp);

    delete ranked_active;

    // Now merge the recursively ranked active list and the sorted
    // cancel list.

    ae = AMI_scan(ranked_active_s, cancel_s,
                  &my_patch_active_cancel, ostream);

    delete ranked_active_s;
    delete cancel_s;

    return 0;
}
```

Our recursion bottoms out when the problem is small enough to fit entirely in main memory, in which case we read it in and call a function to rank a list in main memory. The details of this function are omitted here.

```
////////////////////////////////////////////////////////////////////////
// main_mem_list_rank()
```

```
//
// This function ranks a list that can fit in main memory.  It is used
// when the recursion bottoms out.
//
//////////////////////////////////////////////////////////////////////////

int main_mem_list_rank(edge *edges, size_t count)
{
    // Rank the list in main memory

    ...

    return 0;
}
```

B.3 NAS Parallel Benchmarks

<TO BE EXTENDED>

Code designed to implement external memory versions of a number of the NAS parallel benchmarks is included with the TPIE distribution. Examine this code for examples of how the various primitives TPIE provides can be combined into powerful applications capable of solving real-world problems.

Detailed descriptions of the NAS parallel benchmarks are available from the NAS Parallel Benchmark Home Page at URL `http://www.nas.nasa.gov/NAS/NPB/`.

B.4 Spatial Join

<TO BE WRITTEN>

[74]: LA: Distribution sweeping, SSSJ, ect

[75]: LA: Someting about R-tree building and drainage networks at some point

Appendix C

TPIE Error Codes

C.1 AMI Error Codes

[76]AMI entry points typically return error codes of the enumerated type AMI_err. Member functions of operation management objects also typically return this type. Possible values for error codes include those listed below. It is expected that in future releases of TPIE, many of these error codes will be replaced by exceptions.

[76]: LA: Should we move this section to an appendix? Or just move it to the end of this section? What about appendix for management object return codes ect.?

AMI_ERROR_NO_ERROR No error occurred. The call the the entry point returned normally.

AMI_ERROR_IO_ERROR A low level I/O error occurred.

AMI_ERROR_END_OF_STREAM An attempt was made to read past the end of a stream or write past the end of a substream.

AMI_ERROR_READ_ONLY An attempt was made to write to a read-only stream.

AMI_ERROR_OS_ERROR An unexpected operating system error occurred. Details should appear in the log file if logging is enabled. See Section 7.3.

AMI_ERROR_BASE_METHOD An attempt was made to call a member function of the virtual base class of AMI_STREAM. This indicates a bug in the implementation of AMI streams.

AMI_ERROR_BTE_ERROR An error occurred at the BTE level.

AMI_ERROR_MM_ERROR An error occurred within the memory manager.

AMI_ERROR_OBJECT_INITIALIZATION An AMI entry point was not able to properly initialize the operation management object that was passed to it. This generally indicates a bug in the operation management object's initialization code.

AMI_ERROR_INSUFFICIENT_MAIN_MEMORY The MM could not make adequate main memory available to complete the requested operation. Many operations adapt themselves to use whatever main memory is available, but in some cases, when memory is extremely tight, they may not be able to function.

AMI_ERROR_INSUFFICIENT_AVAILABLE_STREAMS The AMI could not allocate enough intermediate streams to perform the requested operation. Certain operating system restrictions limit the number of streams that can be created on certain platforms. Only in unusual circumstances, such as when the application itself has a very large number of open streams, will this error occur.

AMI_ERROR_ENV_UNDEFINED An environment variable necessary to initialize the AMI was not defined.

AMI_ERROR_BIT_MATRIX_BOUNDS A bit matrix larger than the number of bits in an offset into a stream was passed to ami_gp().

`AMI_ERROR_NOT_POWER_OF_2` The length of a stream on which a bit permutation was to be performed is not a power of two.

`AMI_MATRIX_BOUNDS` An attempt was made to perform a matrix operation on matrices whose bounds did not match appropriately.

C.2 Return Values for Scan Management Objects

More information on the precise semantics of these values appears in Section 5.3.

`AMI_SCAN_CONTINUE` Tells `AMI_scan()` to continue to call the `operate()` member function of the scan management object with more data.

`AMI_SCAN_DONE` Tells `AMI_scan()` that the scan is complete.

C.3 Return Values for Merge Management Objects

More information on the precise semantics of these values appears in Section 5.6.

`AMI_MERGE_CONTINUE` Tells `AMI_merge()` to continue to call the `operate()` member function of the scan management object with more data.

`AMI_MERGE_DONE` Tells `AMI_merge()` that the scan is complete.

`AMI_MERGE_OUTPUT` Tells `AMI_merge()` that the last call generated output for the output stream.

`AMI_MERGE_READ_MULTIPLE` Tells `AMI_merge()` that more than one input object was consumed and thus the input flags should be consulted.

Appendix D

The GNU General Public License, Version 2

June, 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
```

```
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Bibliography

[1] J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. In *Proc. Annual European Symposium on Algorithms, LNCS 1461*, pages 332–343, 1998.

[2] P. K. Agarwal, L. Arge, G. S. Brodal, and J. S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 1999.

[3] P. K. Agarwal, L. Arge, J. Erickson, P. Franciosa, and J. Vitter. Efficient searching with linear constraints. In *Proc. ACM Symp. Principles of Database Systems*, pages 169–178, 1998.

[4] P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 117–126, 1998.

[5] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. V. er. A framework for index bulk loading and dynamization. In *Proc. 28th Intl. Colloq. Automata, Languages and Programming (ICALP)*, 2001.

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

[7] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.

[8] L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995. A complete version appears as BRICS technical report RS-96-29, University of Aarhus.

[9] L. Arge. *Efficient External-Memory Data Structures and Applications.* PhD thesis, University of Aarhus, February/August 1996.

[10] L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS.* Springer-Verlag, Lecture Notes in Computer Science 1340, 1997.

[11] L. Arge, P. Ferragina, R. Grossi, and J. Vitter. On sorting strings in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 540–548, 1997.

[12] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. IEEE International Conf. on Very Large Databases*, 1998.

[13] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 685–694, 1998.

[14] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. Annual European Symposium on Algorithms.*

[15] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica (to appear in special issues on Geographical Information Systems)*, 1998. Extended abstract appears in Proc. of Third European Symposium on Algorithms, 1995.

[16] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 560–569, 1996.

[17] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 381–392, 1995.

[18] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.

[19] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[20] A. Cockcroft. *Sun Performance and Tuning. SPARC & Solaris.* Sun Microsystems Inc., 1995.

[21] T. H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1-2):41–57, 1993.

[22] T. H. Cormen and M. T. Goodrich. Position Statement, ACM Workshop on Strategic Directions in Computing Research: Working Group on Storage I/O for Large-Scale Computing. *ACM Computing Surveys*, 28A(4), Dec. 1996.

[23] T. H. Cormen and D. Kotz. Integrating Theory and Practice in Parallel File Systems. Technical Report TR94-223, Dartmouth College, Computer Science, Hanover, NH, 1994.

[24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[25] T. H. Cormen and D. M. Nicol. Performing Out-of-Core FFTs on Parallel Disk Systems. *Parallel Computing*, 24(1):5–20, 1998.

[26] A. Crauser and P. Ferragina. On the construction of suffix arrays in external memory. Manuscript, 1998.

[27] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for geometric problems. In *ACM Symposium on Computational Geometry*, 1998.

[28] A. Crauser and K. Mehlhorn. LEDA-SM: Extending LEDA to secondary memory. 1999.

[29] R. F. Cromp. An intellegent information fusion system for handling the archiving and querying of terabyte-sized spatial databases. In *S. R. Tate ed., Report on the Workshop on Data and Image Compression Needs and Uses in the Scientific Community, CESDIS Technical Report Series, TR–93–99*, pages 75–84, 1993.

[30] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, June 1997.

[31] F. Dehne, D. Hutchinson, and A. Maheshwari. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *International Parallel Processing Symposium*, pages 14–20, April 1999.

[32] H. M. Deitel and P. J. Deitel. *C++ How To Program.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1994.

[33] M. A. Ellis and B. Stroustrup. *The Annotated C++ Rerenence Manual.* Addison–Wesley, 1990, 1990.

[34] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proc. ACM Symp. on Theory of Computation*, pages 693–702, 1995.

[35] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 373–382, 1996.

[36] E. Feuerstein and A. Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. In *Proc. Int. Symp. on Algorithms and Computation*, 1993.

[37] P. G. Franciosa and M. Talamo. Orders, k-sets and fast halfplane search on paged memory. In *Proc. Workshop on Orders, Algorithms and Applications (ORDAL'94), LNCS 831*, pages 117–127, 1994.

[38] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.

[39] R. Grossi and G. F. Italiano. Efficient cross-tree for external memory. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization.* American Mathematical Society Press, Providence, RI, 1999.

[40] D. Hutchinson, A. Maheshwari, J.-R. Sack, and R. Velicescu. Early experiences in implementing the buffer tree. Workshop on Algorithm Engineering, 1997. Electronic proceedings available at http://www.dsi.unive.it/˜wae97/proceedings/.

[41] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences*, 52(3):589–612, 1996.

[42] B. Kobler and J. Berbert. NASA earth observing systems data information system (EOSDIS). In *Digest of Papers: 11'th IEEE Symp. on Mass Storage Systems*, 1991.

[43] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.

[44] S. Meyers. *Effective C++.* Addison-Wesley, 1992.

[45] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[46] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 120–129, 1993.

[47] M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, pages 919–933, 1995.

[48] I. Pohl. *C++ for C Programmers.* Benjamin-Cummings, 1994.

[49] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[50] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. Principles of Database Systems*, pages 25–35, 1994.

[51] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995.

[52] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3:331–360, 1991.

[53] D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 192–201, 1996.

[54] J. S. Vitter. External memory algorithms (invited tutorial). In *Proc. of the 1998 ACM Symposium on Principles of Database Systems*, pages 119–128, 1998.

[55] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. American Mathematical Society Press, Providence, RI, 1999. Available via the author's web page `http://www.cs.duke.edu/~jsv/`. Earlier shorter versions entitled "External Memory Algorithms" appear as an invited tutorial in *Proceedings of the 17th ACM Symposium on Principles of Database Systems*, Seattle, WA, June 1998, 119–128, and as an invited paper in *Proceedings of the 6th Annual European Symposium on Algorithms*, Venice, Italy, August 1998, 1–25, published in Lecture Notes in Computer Science, **1461**, Springer-Verlag, Berlin.

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

[57] B. Zhu. Further computational geometry in secondary memory. In *Proc. Int. Symp. on Algorithms and Computation*, pages 514–522, 1994.

Index