

A Transparent Parallel I/O Environment

Darren Erik Vengroff*
Brown University and Duke University

Abstract

We describe TPIE, a Transparent Parallel I/O Environment. TPIE is a system designed to bridge the gap between current theoretical knowledge about the construction of I/O-optimal algorithms on parallel disk systems and the design and implementation of parallel I/O systems. We discuss the design of TPIE and its interface, the structure of a typical implementation, applications of the system, our prototype, and future research directions.

The initial goal of our work is a prototype system to demonstrate: 1) that optimal algorithms can be made to run efficiently on parallel I/O devices; and 2) that high level hardware independent interfaces to the I/O paradigms required to implement such algorithms can be provided to application programmers. The TPIE interface is designed to be portable across a variety of parallel hardware platforms; thus code that runs efficiently on one machine will run efficiently on others.

Longer term goals for TPIE include extending the prototype in ways that will permit it to function as an extremely flexible parallel I/O system capable of supporting a wide range of applications, such as matrix computation, computational geometry, graph manipulation, and database management.

1 Introduction

As of today, gigabyte computer systems exist on desktops, and terabyte systems are not unheard of. In the not too distant future, systems designed to manage petabytes¹ of information will come on-line. The most important characteristic of such vast amounts of data is that they cannot possibly be stored in the primary memories of even the most powerful computers. Instead, they must be stored on secondary memory, such as magnetic disks, or tertiary memory, such as tapes and optical memory. Compared to CPUs and solid state random access memory, these devices are extraordinarily slow; the difference in access time is typically 2 to 5 orders of magnitude. Because of the low speed of secondary storage, good performance in the Input/Output (I/O) system that links secondary storage to main memory and the CPU or CPUs is critical if good performance is to be achieved overall. Performance can be further improved if many disks can be efficiently used in parallel. Unfortunately, existing I/O systems generally do not perform adequately [Pat].

In recent years, computer science theorists have studied the problem of efficiently using parallel disks to solve a variety of computational problems. At the same time, a number of parallel I/O systems have become available, though in most cases they have failed to take adequate advantage of the insights theorists have had to offer [CoK]. TPIE, the transparent parallel I/O environment that is the subject of this paper, is designed to bridge the gap between the theory and practice of parallel I/O systems. It is intended to demonstrate that a parallel I/O system can do all of the following simultaneously:

- Abstract away the details of how I/O is performed so that programmers need only deal with a simple high level interface.
- Implement I/O-optimal paradigms for large scale computation that are efficient not only in theory, but also in practice.

*Current address: Dept. of Computer Science, Box 90129, 241 North Building, Duke University, Durham, NC 27708-0129. Phone: (919) 660-6564. FAX: (919) 660-6519. Email: dev@cs.duke.edu. Support was provided in part by National Science Foundation research grant CCR-9007851 and by Army Research Office grant DAAL03-91-G-0035.

¹ 1 petabyte = 1,024 terabytes $\approx 10^{15}$ bytes.

- Remain 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.

The remainder of this paper is organized as follows: Section 2 discusses TPIE in the context of current theoretical results and existing systems. Section 3 introduces the overall design and structure of TPIE. Section 4 describes the high level interface programmers use to interact with TPIE, while Section 5 describes the lower level, machine dependent components. Section 6 describes the initial prototype TPIE system we plan to implement, while Section 7 outlines some future extensions to it. Section 7.3 introduces some remaining theoretical problems we plan to explore. Finally, Section 8 contains some concluding remarks.

2 Existing Systems and System Proposals

In recent years a number of parallel file systems for high performance computer systems have been built or proposed [CFP, Corb, DiS, Kotb, KSU, PGK, Pie]. At the same time, a number of I/O-optimal computation paradigms have been proposed [Cora, CoW, GTV, NoV, ViS].

Unfortunately, existing I/O systems tend not to adequately support the functionality required in order to implement I/O-optimal algorithms. In particular, many file systems do not support independent head movement on different disks or striping a file by putting complete blocks on individual disks rather than striping individual words of a file across multiple disks. Both of these are critical requirements for many I/O-optimal algorithms when many drives are used in parallel. For an detailed discussion of these and related issues surrounding the state of the relationship between theory and practice in high performance file systems the reader is directed to recent work by Cormen and Kotz [CoK].

Supporting block striping and independent head movement will obviously complicate an I/O system. It would also seem to complicate the lives of any programmers attempting to use the system, since they would have to write their code with multiple independent disks in mind. One of the things we hope to demonstrate with TPIE is that this need not be the case; instead, most of the complexity of I/O computation can be isolated at a low level that application programmers will not have to deal with directly.

As we will see, the programmer can think in terms of simple, high level concepts, such as scanning, divide and conquer via recursive distribution, merging, and filtering. A contrasting approach is that of Cormen [Corb] whose goal is to construct a new compiler capable of analyzing existing code and finding reference patterns and permutations that can be efficiently implemented on parallel I/O systems. This approach, if successful, has the advantage of being able to work with “dusty deck” numerical applications written in FORTRAN. The disadvantage of this approach is that it is extremely unlikely that any compiler will be able to develop sufficient understanding of the semantics of a program to recognize and generate optimally I/O efficient code for the type of combinatorial and geometric problems TPIE will be able to solve. This does not mean, however, that TPIE and a system based on Cormen’s approach cannot coexist. Though their goals are different, at a low level they work with many of the same I/O paradigms. Thus, one could implement Cormen’s approach by concentrating on the permutation recognition front end and relying on TPIE to provide a hardware independent back end at run time.

TPIE could also be used as a back end for I/O programs based on portable parallel languages like VCODE [BIC], since, as we shall see in Section 4.3, TPIE can be used to simulate many PRAM algorithms in secondary memory. This would allow code written in parallel languages such as NESL [Ble] to be made I/O efficient.

3 System Design

In designing and implementing the TPIE prototype, we have five goals in mind: abstraction, efficiency, flexibility, portability, and extensibility. We hope to achieve all of these goals while keeping the system as simple as possible. Abstraction and efficiency will be achieved simultaneously by supporting paradigms of

I/O-optimal computation and by managing, to as great a degree as possible, all the computational resources (i.e. CPU, main memory, and disk) a given machine has to offer. Flexibility will be maintained by allowing application programmers a great deal of control over the functional details of the computation taking place within the supported paradigms. Portability will be achieved by isolating machine dependent code at lower levels of the system. Extensibility will be a result of modular design, well established interfaces, and advance consideration of the form and function of likely extensions.

3.1 Interacting with TPIE

On many systems the most difficult aspects of writing parallel code is managing multiple interacting threads of execution. Some systems avoid this complexity by operating on data in parallel using a single thread of control. TPIE is such a system. In the taxonomy of parallel systems, it is a SPMD (Single Program Multiple Data) system. As a SPMD system, TPIE is simpler for programmers to work with than a MPMD system, yet retains the ability to perform I/O-optimal computation for a wide variety of problems. In a sense, TPIE abstracts away not only the I/O it performs, but also the details of interprocessor communication and synchronization required for I/O-optimal computation.

A simple interface, using a popular and well understood programming language, will make TPIE easy to use. C++ has been chosen for a number of important reasons. First, C++'s built-in polymorphism and function and class templates enable TPIE to support a wide variety of applications with a small number of generic entry points. Second, C++ has become the language of choice for essentially all systems software and new non-numeric applications, and thus many programmers will need very little additional training in order to learn to use TPIE. TPIE applications written in C++ will have a single thread of control, though in most cases data will be processed in parallel by the underlying system.

In order to write programs using TPIE, programmers will define classes of objects (e.g., points, line segments, nodes, edges, etc.) on which their program will operate. They will then write functions to manipulate objects of these classes. Sets of objects and pointers to manipulation functions will then be passed to TPIE library functions which will perform fundamental operations such as distribution, merging, filtering, permutation routing, and PRAM simulation.

3.2 The Structure of TPIE

A simple, well structured, modular design will make TPIE easy to implement and maintain. Architecture specific code will be isolated in low-level modules. The system will consist of three components: a block transfer engine (BTE), a memory manager (MM) and an access method interface (AMI). The BTE handles block transfer for a single processor. The MM performs low level memory management across all the processors in the system. The AMI works on top of the MM and one or more BTEs, each running on a single processor, to provide a uniform interface for application programs. Applications that use this interface will be portable across hardware platforms, since they never have to deal with the underlying details of how I/O is performed on a particular machine.

The BTE is intended to bridge the gap between the I/O hardware and the rest of our system. It will work 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 will be 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 filtering paradigms, the buffer manager will provide stream oriented buffer replacement policies. To further improve performance, the system will 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. Although the BTE will run on a single processor, it will support concurrent access to multiple disks, allocating and managing buffer space for all of them concurrently.

The MM manages random access memory on behalf of TPIE. It is the most architecture-dependent component of the system. On a single processor or multiprocessor system with a single global address space, the MM will be relatively simple; its task will be to allocate and manage the physical memory used by the BTE. On a distributed memory system, the MM has the additional task of coordinating communication between processors and memory modules in order to support the primitives that the AMI provides.

The AMI is a layer between the BTE and user level processes that implements fundamental access methods, such as scanning, permutation routing, merging, distribution, and batch filtering. It also provides a consistent, object-oriented interface to application programs. The details of how these access methods are implemented will depend on the hardware on which the system is running. For example, recursive distribution will be done somewhat differently on a parallel disk machine than on a single disk machine. The AMI will abstract this fact away, allowing an application program that calls a function such as `AMI_recursive_distribute()` to work correctly regardless of the underlying I/O system.

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.

4 AMI Access Methods

The AMI is the programmer's point of contact with TPIE. It consists of a small number of very flexible polymorphic entry points which allow the programmer to work with a variety of I/O computation paradigms without having to worry about any low level I/O details.

With the exception some minor initialization and administrative functions, all of the AMI entry points operate on AMI streams. An AMI stream is nothing more than an abstraction of an ordered set of objects of any one class which is stored in secondary memory. In some applications, the elements of a stream might be nothing more than integers. In others, they might be objects of an arbitrarily complex user defined class. Creating a stream of objects of class `foo` called `foo_stream` requires nothing more than the definition²

```
AMI_STREAM<foo> foo_stream;
```

`AMI_STREAM` is a macro that resolves to a class template declared to match the underlying semantics of the target I/O architecture by using an appropriate BTE and MM.

In addition to specifying the streams on which to operate, the programmer will also typically specify a management object. A management object is an object that has methods for examining objects from an input stream or streams and writing them to an output stream or streams.

In the remainder of this section, we will introduce a number of the most common AMI entry points, divided into groups by their function.

4.1 Scanning and Selection

The most basic AMI entry points are for scanning and selecting. The polymorphic function `AMI_scan()` scans and selects from AMI streams on an element by element basis. The function `AMI_scan_by_memory_load()` is similar, but it works on as many elements of the stream as can fit into main memory at one time. The syntax of these entry points is described in Table 1.

`AMI_scan()` scans the elements of an AMI stream and applies the `operate()` member function of a user specified scan management object to each element or tuple of elements it encounters. This is best illustrated by an example, such as the following, in which each integer in an AMI stream is squared using a scan management object designed to square integers.

```
class square_scan : public 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;
        }
    }
};
```

²The reader is advised that this definition, like all the code contained in this paper, is based on a version of TPIE that is still under development. There may be slight changes in syntax between now and the time TPIE is actually released, though the fundamental concepts will not change.

```
AMI_scan(instream0, ... istreamN, s_obj, outstream0, ..., outstreamM)
```

Where `instreamI` is a pointer to a stream of objects of type `tI`, `outstreamJ` is a pointer to a stream of objects of type `uJ`, and `s_obj` is a pointer to an object of a user defined subclass of `AMI_scan_object` with `initialize()` and `operate()` member functions defined as in one of the following:

```
class user_scan_object : public AMI_scan_object {
public:
    AMI_err initialize(void);
    AMI_err operate(const type0 &t0, ..., const typeN &tN,
                   AMI_SCAN_FLAG *inf,
                   U0 *u0, ..., U1 *uJ, AMI_SCAN_FLAG *outf);
};
```

```
class user_scan_take_object : public AMI_scan_take_object {
public:
    AMI_err initialize(void);
    AMI_err operate(const type0 &t0, ..., const typeN &tN,
                   AMI_SCAN_FLAG *inf, AMI_SCAN_FLAG *takef,
                   U0 *u0, ..., U1 *uJ, AMI_SCAN_FLAG *outf);
};
```

After initializing the scan management object by calling `s_obj->initialize()`, the AMI reads items from each input stream and presents them to the scan management object through its member function `s_obj->operate()`. `inf[]` is an array of flags specifying which of the inputs contain valid data. Not all inputs are necessarily valid, since the end of some input streams may have been reached before the end of others. `outf[]` is an array of flags set by `s_obj->operate()` to indicate which outputs it generated. These flags allow scan managers to function not only as scanners, but as selectors which only generate output corresponding to certain input objects.

The difference between the two types of scan management objects is the `takef` argument to the `operate()` member function. When `s_obj` points to an object of a subclass of `AMI_scan_take_object`, `takef` points to an array of flags that `s_obj->operate()` sets to indicate which of the inputs `tI` it processed. Those that are not processed will be presented to the object again on the next call. When `s_obj` points to an object of a subclass of `AMI_scan_object`, input can not be deferred by the scan management object.

The scan management object `s_obj` may store a finite amount of state information, and thus it may have more output to generate even after all input has been passed to it. The AMI will thus continue to call it, with no input, as long as it keeps returning `AMI_SCAN_CONTINUE`. The scan ends only when the scan object returns either an error or `AMI_SCAN_DONE`.

```
AMI_scan_by_memory_load(instream, s_obj, outstream)
```

Scan the input stream `instream` by reading as much data as possible into an array in main memory. When main memory is full, call the scan management object member function `s_obj->operate()` with a pointer to the main memory array and a count of the number of element in the array. When the member function returns, write the new contents of main memory to the output stream. Continue until the input stream is exhausted.

In some cases the scan management object may require auxiliary main memory in which to do its computation. Before beginning to read the input stream, `AMI_scan_by_memory_load()` will query the object to determine its memory needs and only use as much main memory as remains available. For example, if `*s_obj` claims that it needs one auxiliary object for each object it is processing, then `AMI_scan_by_memory_load()` will only ask it to operate on an array filling half of main memory at a time.

Table 1: AMI entry point for element-wise scans and selection and memory load-wise scans.

```

        } else {
            return AMI_SCAN_DONE;
        }
    }
};

void do_the_squaring(AMI_STREAM<int> *instream, AMI_STREAM<int> *outstream)
{
    square_scan my_scan;
    AMI_err ae = AMI_scan(instream, &my_scan, outstream);
}

```

The `operate()` method of the scan management object `my_scan_manager` is applied to every object in the input stream. The result of each application, which `operate()` returns in `*out`, is put into the corresponding element of the output stream. Once the input has been exhausted, `operate()` is called with `*sfin` set to zero. `operate()` recognizes that there is nothing left to do, and returns `AMI_SCAN_DONE`.

Let us now consider a case in which we are selecting as we are scanning, meaning that output is not generated for every input. The flag `*sfout` is set to zero when no output is generated. Output flags can be used in combination with stored internal state to build more advanced scan management objects that generate output based not only on current input objects, but on objects they have previously seen in the input stream. In the following example, we define a class of scan management object that builds an output stream consisting only of running maxima, that is values that are larger than any previously seen in the input stream. If an object of this class were used to scan an input stream containing the integers 5, 2, 3, 6, 7, 2, 5, 9, 7, for example, it would produce an output stream containing 5, 6, 7, 9.

```

class running_max : public AMI_scan_object {
private:
    unsigned int max;
public:
    AMI_err initialize(void) { max = 0; return AMI_ERROR_NO_ERROR; };
    AMI_err operate(const unsigned int &in, AMI_SCAN_FLAG *sfin,
        unsigned int *out, AMI_SCAN_FLAG *sfout) {
        if (*sfin) {
            *sfout = ((in > max) && (*out = max = in));
            return AMI_SCAN_CONTINUE;
        } else {
            return AMI_SCAN_DONE;
        }
    };
};

```

The `initialize()` method is called by `AMI_scan()` before the processing of a scan begins. This allows the scan management object to reset any internal state in preparation for scanning a new stream. In our example, it was used to reset the running maximum.

More advanced polymorphs of `AMI_scan` can work on multiple streams in parallel, selecting elements based on functions of the corresponding elements of the various input streams. It is also possible to produce multiple output streams. For example, the following code takes two input streams and produces four output streams, containing running sums of the values in the input streams and their squares:

```

class sum_scan : public AMI_scan_object {
private:
    long int sx, sy, sx2, sy2;
public:
    AMI_err initialize(void) {
        sx = sy = sx2 = sy2 = 0;
        return AMI_ERROR_NO_ERROR;
    };
    AMI_err operate(const int &x, const int &y, AMI_SCAN_FLAG *sfin,
        long int *sumx, long int *sumy,

```

```

        long int *sumx2, long int *sumy2,
        AMI_SCAN_FLAG *sfout) {

    AMI_err ae = AMI_SCAN_DONE;

    if (sfin[0]) {
        sfout[0] = sfout[2] = 1;
        *sumx = sx += x; *sumx2 = sx2 += x * x;
        ae = AMI_SCAN_CONTINUE;
    }
    if (sfin[1]) {
        sfout[1] = sfout[3] = 1;
        *sumy = sy += y; *sumy2 = sy2 += y * y;
        ae = AMI_SCAN_CONTINUE;
    }
    return ae;
};

};

void do_the_summing(AMI_STREAM<int> *xstream, AMI_STREAM<int> *ystream,
                   AMI_STREAM<long int> *sxstream, AMI_STREAM<long int> *sx2stream,
                   AMI_STREAM<long int> *ystream, AMI_STREAM<long int> *sy2stream)
{
    sum_scan my_scan;
    AMI_err ae = AMI_scan(xstream, ystream, &my_scan,
                        sxstream, systream, sx2stream, sy2stream);
}

```

An example of the use of a scan management object that is a subclass of `AMI_scan_take_object` appears in Appendix A.

4.2 Distribution and Merging

The next set of AMI entry points are for distribution and merging, which are summarized in Table 2. Distribution is a process whereby data is read sequentially from a single input stream, and each item is written to an output stream determined by a user defined distribution object. It can be very useful in sorting and related problems where elements are ordered [GTV, NoV, ViS]. Merging is essentially the reverse of distribution; it takes a number of ordered input streams and merges them into a single ordered output stream. All items from a single input stream remain in the same order in the output stream, although the different input streams may be interleaved in arbitrary ways in the output stream.³ The way in which this interleaving takes place is under the control of a user supplied merge management object.

The following example is a sketch of how a distribution management object is defined and used. The object generates a set of medians from the input stream and uses them as the basis of the distribution of the input. Note that at initialization time `AMI_distribute()` informs the object how many output streams it should prepare to generate using the argument `out_arity`. This value will depend on the underlying hardware, particularly the amount of main memory and number of independent disks available.

```

class dist_medians : AMI_dist_object {
private:
    int *medians;
    arity_t arity;
    AMI_err generate_medians(arity_t out_arity, AMI_base_stream<int> *instream);
public:

```

³In actuality, the AMI does not strictly enforce this. A strangely constructed user merge management object could do a limited amount of reordering of elements from a single input stream by maintaining an internal state consisting of previous input objects. The number of elements out of position that an element could move could be at most a constant. A user defined distribution management object could also do a similarly bounded amount of reordering of the elements that it distributes. This behavior, though technically possible, would be highly atypical.

```
AMI_distribute(instream, dist_obj, outstreams, out_arity)
AMI_recursive_distribute(instream, dist_obj, outstreams, out_arity)
```

Distribute the elements of the input stream `instream`, putting them into a hardware dependent number of output streams. Return a pointer to an array of newly created output streams in `*a_outstreams` and the number of output streams created in `*arity`. The number of output streams will depend on factors such as the amount of available main memory and the number of independent disks.

`dist_obj` is a distribution management object. This is a user supplied object which the AMI will use to manage the distribution process. A detailed description of the interfaces and protocols associated with user defined distribution objects will be included in the complete technical specification of the AMI.

`AMI_recursive_distribute()` works just like `AMI_distribute()` except that, before returning, it checks the size of each of the streams it created and recurses on any that are too large to fit in main memory.

```
AMI_merge(instreams, in_arity, outstream, merge_obj)
AMI_merge_by_offset(instream, in_arity, offsets, outstream, merge_obj)
```

`AMI_merge()` merges the elements of the `in_arity` input streams pointed to by `instreams`, writing them to the output stream `outstream`. `merge_obj` is a user supplied merge management object, which is analogous to the distribution management object used by `AMI_distribute()`.

`AMI_merge_by_offset()` merges `in_arity` sub-streams of a single input stream `instream`. The `i`'th sub-stream begins with the `offsets[i]`th item in `instream` and ends with the `offsets[i+i] - 1`st. If the number of input streams is more than can be merged at one time given the main memory available, then merging is done recursively, in as many passes as are needed to merge all of the original input streams or sub-streams into a single output stream.

Table 2: AMI entry points for distributing and merging.

AMI entry point	Input Arity	Output Arity	Multiple Types
<code>AMI_scan()</code>	small constant	small constant	Yes
<code>AMI_distribute()</code>	1	hardware dependent	No
<code>AMI_merge()</code>	hardware dependent	1	No

Table 3: The differences between `AMI_scan()`, `AMI_distribute()`, and `AMI_merge()`.

```
AMI_err initialize(arity_t out_arity, AMI_base_stream<int> *instream) {
    return generate_medians(out_arity, instream);
};
AMI_err operate(const int &in, arity_t *dest) {
    *dest = binary_search_index(medians, in);
    return AMI_ERROR_NO_ERROR;
};
~dist_medians(void) {
    if (medians) delete [] medians;
};
};
```

It is important to note exactly what the differences between the `AMI_distribute()` and `AMI_merge()` entry points and `AMI_scan()`, particularly since `AMI_scan()` can do a limited amount of distribution or merging simply by operating on one input stream and generating several output streams or vice versa. If the user is only interested in distributing an input stream to a constant number of output streams rather than a hardware dependent number, then indeed `AMI_scan()` should be used rather than `AMI_distribute()`. Similarly, `AMI_scan()` should be used rather than `AMI_merge()` when the number of streams to be merged

<p><code>AMI_sort(instream, outstream, compar)</code></p> <p>Sort the input stream <code>instream</code>, putting the sorted result in the output stream <code>outstream</code>. Use the function <code>*compar()</code> to compare elements to establish the sorted order. The algorithms used are described in [NoV, ViS].</p>
<p><code>AMI_BMMC_permute(instream, outstream, permut)</code> <code>AMI_BPC_permute(instream, outstream, permut)</code></p> <p>Permute the elements of the input stream <code>instream</code>, putting them into the output streams <code>outstream</code>. <code>permut</code> is an object describing the BMMC or BPC permutation to be performed. Algorithms described in [CoW] are used.</p>
<p><code>AMI_general_permute(instream, outstream, dest)</code></p> <p>Permute the elements of the input stream <code>instream</code>, putting them into the output streams <code>outstream</code>. <code>*dest()</code> is a function that, when passed an element of <code>instream</code>, returns an integer specifying the position that object should appear in in <code>outstream</code>. Routing a general permutation will take asymptotically as long as sorting the input stream. No attempt is made to recognize and route simpler, (e.g. BMMC or BPC) permutations as described in [Cora].</p>

Table 4: AMI entry points for sorting and permuting.

is a constant. However, if the users goal is to make most effective use of hardware resources my merging from or distributing to as many streams as the hardware possibly can at a time, then `AMI_merge()` and `AMI_distribute()` are the entry points of choice.

A further distinction is that `AMI_scan()` allows the streams on which it operates to be of different types (as in the `sum_scan` example in Section 4.1) whereas all streams presented to a given invocation of `AMI_merge()` or `AMI_distribute()` must be of the same type. The differences between the entry points are summarized in Table 3.

4.3 Sorting and Permuting

The next set of AMI entry points are for sorting and permuting the elements of an AMI stream. These functions are useful in a wide variety of settings. They are summarized in Table 4.

One novel use of optimal sorting in a parallel I/O system is to produce I/O-optimal algorithms based on certain PRAM algorithms, which may themselves not be optimal. This technique, which we will briefly outline here, is discussed in detail in a recent paper on I/O efficient graph algorithms [CGG].

The central idea is to take a parallel algorithm that starts with an input of size n and then, in constant time and linear work on n processors, reduces the problem to one of size αn , for some constant $0 < \alpha < 1$. After $O(\log n)$ recursive stages, the original problem is reduced to a problem of constant size than can be solved in a constant amount of time. The total time is thus $O(\log n)$ and the total work is $O(n \log n)$ on n processors. We can implement a similar algorithm in I/O by simulating each stage of the parallel algorithm. To simulate a stage, we first sort the input so that the operands used by each processor are in adjacent locations in the stream. We then scan the stream, applying whatever operations the parallel processors would apply. They may need to be repeated a constant number of times. We then select the αn intermediate results that make up the recursive problem and recurse. When we have recursed down to a problem instance that can fit in main memory, we read it in using a linear number of I/Os, solve it in main memory and then write it out to disk using a linear number of I/Os. The time spent by our I/O algorithm is given by the recurrence:

$$\begin{aligned}
 T(n) &= O(\text{sort}(n)) + T(\alpha n) \\
 &= O(\text{sort}(n)),
 \end{aligned}$$

<pre>AMI_batch_filter(instream, outstreams, filter)</pre> <p>Filter the input in <code>instream</code> through the filter pointed to by <code>filter</code>, placing the results in the output streams pointed to by <code>outstreams</code>. <code>filter</code> is an object representing a planar DAG, as described in [GTV]</p>
<pre>AMI_dist_sweep(instream, outstreams, dist_sweep_obj) AMI_recursive_dist_sweep(instream, outstreams, dist_sweep_obj)</pre> <p>Perform either a single phase of distribution sweeping as described in [GTV] or a complete recursive sweep of the input stream. <code>*dist_sweep_obj</code> is a distribution sweeping object that specifies the details of the distribution sweep to be performed.</p>

Table 5: AMI entry points for filtering and distribution sweeping operations. These are useful in computational geometry, as described in [GTV].

where $sort(n)$ is the I/O complexity of sorting n items. In cases where the block size is not extremely small, any problem that can require an arbitrary permutation to solve takes at least $\Omega(sort(n))$ I/O operations [AgV]. Thus, TPIE can optimally solve any problem that can be solved by an appropriately structured PRAM algorithm, yet requires arbitrary permutations. Furthermore, it has recently been shown that many interesting problems that don't require arbitrary permutations still require enough permutation that the $\Omega(sort(n))$ lower bound holds [CGG].

An example of a problem on which this approach can be used is list ranking. Sample TPIE code which solves the list ranking problem can be found in Appendix A.

4.4 Geometric Computation

The AMI also has a pair of entry points designed to support I/O efficient computation on geometric objects, as described in [GTV]. They can be used in optimal algorithms for a variety of problems, including nearest neighbors, orthogonal segment intersection, and computing convex hulls. These entry points are summarized in Table 5.

4.5 Miscellaneous

In addition to the entry points described above, there are a small number of initialization and administrative entry points. These functions are all fairly straightforward and will not be described in detail here. They include functions to query various system parameters and functions to copy streams.

Another set of entry points will allow the programmer to access a specific block or blocks within an AMI stream. These are provided primarily for advanced programmers who may want the flexibility to implement their own computational paradigms on top of TPIE. Such programmers, should they desire to do so, will be able to micro-manage the contents of blocks within a stream by accessing them individually, manipulating their contents in whatever way they chose, and then returning them to disk. To access a set of blocks, one from each parallel disk, in a single I/O operation the programmer will simply specify the logical address of each block on each disk. Obviously this requires that the program have knowledge of exactly how many disks there are at run time. This information will be available through one of the aforementioned administrative entry points.

5 The BTE and MM

Although programmers working with TPIE will interact primarily with the AMI layer, it is largely just a uniform programming interface; most of the real work in TPIE is done by the lower levels, namely the BTE

and the MM. The details of how these layers function will depend a great deal on the underlying hardware. This is not to say, however, that there is not a well defined interface between the BTE and the MM, for there is. This interface exists both to promote modularity and because, as we shall see, different computing environments may operate most efficiently with different BTE/MM combinations. Because their precise implementations are highly machine dependent, the BTE and MM will not be discussed in great detail here. Instead, we will outline their basic roles and discuss at a high level how they interact with each other, with the hardware, and, through the AMI, with the programmer.

The BTE is lowest layer of TPIE. It is the layer that is ultimately responsible for moving blocks of data from physical disk devices to main memory and back. We hope that in most cases it will be possible for the BTE to work with device drivers provided by the machine vendor's operating system. In some cases, however, new drivers will undoubtedly have to be written. The BTE is also responsible for maintaining the integrity of streams striped across multiple disks attached to a single CPU, which it will do as described in [ViS]. The BTE is not, however, responsible for coordinating the actions of multiple CPU's and the disks attached to them. A separate instance of the BTE will run on each such CPU, and their actions will be coordinated by a single multi-threaded MM running at a higher level. The reason for the functional split between the two levels is that it will likely be advantageous to be able to use a single BTE written for a specific piece of hardware with more than one MM, for example, one MM written for a homogeneous environment and one for a heterogeneous environment.

The MM is the layer of TPIE that sits between the AMI interface and the BTE. Its primary role is managing main memory, including memory that may be distributed across multiple physical machines. The performance of many of the AMI stream operations, such as sorting, permuting, merging, and distribution depend critically on the efficient use of main memory. The first thing the MM will have to do to achieve this is bypass the virtual memory system provided by UNIX and related operating systems. The second thing it has to do is bypass the traditional UNIX buffer cache and take charge of managing the blocks of data provided by the BTE. In some cases, operating system kernels will have to be modified in order for the MM to do its job. In modern micro-kernel operating systems, however, the MM may be able to operate entirely as a user level process.

In multiple CPU environments, the job of the MM will be complicated by the need to manage multiple banks of memory. In tightly coupled homogeneous parallel environments, this task is likely to be made far simpler by existing hardware and operating system support. In distributed, and in particular in heterogeneous environments, the MM will have to work with various network protocols and drivers to accomplish its task.

It seems clear that the bulk of the complexity in any implementation of TPIE will be in the MM layer. Luckily, memory management tasks similar to those the MM must perform have received a great deal of research attention. Much of this research comes to us by way of the database community, which has built memory management systems for relational databases [Epp, SaG] and object oriented databases [RuC, ZdM], among others.

6 Prototype Implementation

The first TPIE prototype will be implemented to run on Sun SparcStations. There will be two versions. The first will run on a single CPU with parallel disks. The second will run on multiple CPUs linked by a local area network; each CPU will have multiple local disks. We expect to release a beta version of the single CPU implementation in late 1994 or early 1995. The release will include a complete technical specification of the TPIE interface as well as a detailed discussion of the implementation itself.

In addition to the TPIE system itself, we plan to implement a number of sample applications for performance tuning and benchmarking purposes. Benchmarks will be performed in a number of areas, including sorting and general permutation, special case BMMC and BPC permutation, computational geometry, general PRAM simulation, and graph theory.

7 Future Extensions

As currently proposed, TPIE will be an efficient system for performing a wide variety of I/O computation. There are, however, a number of ways in which the system could be extended. We plan to explore these extensions in order to determine how best to incorporate them into future versions of TPIE. Some of these, such as persistence, are high priority extensions that will unquestionably add value to the system. Others, such as fault tolerance will become more important when we move from a prototype to a full production system. Code optimization lies somewhere in between.

7.1 Persistence

As it stands, TPIE will be a prototype I/O computation environment, but has no support for persistent storage of either the input data or results of the computation it performs. Obviously in a complete system this will be necessary. A related issue is that of concurrency. Once streams are persistent we must establish consistent mechanisms for handling concurrent access to them. This issue is further complicated in the TPIE environment by the fact that the MM will have to manage requests for main memory from more than one concurrently running application.

There are two different ways in which persistence may be supported in TPIE. The first would be an extension of the UNIX tree structured file system. This is an approach that has been considered extensively in the literature of parallel file systems [Kota]. It has the advantages of familiarity and backwards compatibility. The second approach is to treat AMI streams as large persistent objects and manage them in the context of a persistent object base. An implementation based on this approach would borrow heavily from techniques developed in the database community [ZdM]. At the moment, we are leaning towards the latter approach.

7.2 Code Optimization

We can view the problem of code optimization in TPIE as being similar to the problem of query optimization in a relational database. We do so by observing that once we have persistence, TPIE will be functionally similar to a parallel relational database system. A database oriented reader may have noticed that the scan and select operations defined in Section 4 bear a great deal of resemblance to the project (π) and select (σ) operations supported by relational database systems. Furthermore, joins can be performed by sorting and scanning. TPIE will thus support a superset of the semantics of a relational database, though it will do so with a slightly different set of fundamental operations than a traditionally indexed database system. If we look at persistent TPIE as a database, we can view a program or program fragment written using TPIE as a query. When we view TPIE in this way, it makes sense to apply query optimization techniques to produce more efficient TPIE code. In fact, this can be done even before persistence is added to TPIE, to increase the efficiency of TPIE programs. One form of code optimization of this type could be done by a preprocessor which would examine the C++ source code of a TPIE program and combine or restructure AMI stream operations to increase efficiency. Additional optimization could be done at run time by predicting the sizes of intermediate AMI streams through random sampling.

Another database derived approach to improving the efficiency of TPIE applications is to add indices to AMI streams. The most obvious way to do this is through the addition of an indexing mechanism within the AMI. We do not, however, believe that it is necessary to add much additional code to the AMI itself to support this. Instead, we feel that the AMI is robust enough that indexing can be provided at the application program level, much as a small database system might be implemented by writing indices into a standard UNIX file system. Indices would be managed through the direct block manipulation functions mentioned in Section 4.5. The only extension to the AMI that is likely to be needed is a mechanism for the application program to communicate with the MM in order to provide it with hints on which blocks to cache and which to allow to fall out of main memory.

7.3 Further Theoretical Work

In addition to designing and building a prototype system, we plan to continue theoretical research aimed at developing I/O efficient algorithms for new problems and domains. We expect that as such algorithms are developed it will be relatively easy to add support for them to our system. In many cases it will be possible

to do this at the application level; in others, new access methods will have been developed and some small additions will be made to the AMI.

8 Conclusions

TPIE is designed to prove the concept that by wrapping I/O-optimal computation paradigms in a straightforward, polymorphic interface, it is possible to construct an I/O computation environment that is easy for programmers to learn and use, yet efficient, flexible and portable. By basing TPIE on C++, we can use existing compilers and tools with which many programmers are already familiar. Furthermore, the polymorphism of C++ allows us to implement a very flexible interface for programmers. Behind this interface, the system supports computational paradigms that have been proven to be optimally efficient. Within the framework of these paradigms, the programmer has the flexibility to operate on arbitrary classes of objects by applying arbitrary user defined functions to them. Finally, the system is designed so as to be portable across a variety of hardware platforms.

At the present time, we have plans to implement a prototype TPIE system which, in addition to providing proof that our concept works, will be designed to serve as a foundation on which extensions to the current TPIE design can be built. We expect the area of parallel I/O systems to be fertile ground for research for quite some time to come, and are eager to see an evolving TPIE among the fruits it produces.

Acknowledgments

The author would like to thank Eddie Grove and Jeff Vitter for their insightful comments on early drafts of this paper.

References

- [AgV] A. Aggarwal and J. S. Vitter, “The Input/Output Complexity of Sorting and Related Problems,” *Comm. ACM* 31 (1988), 1116–1127.
- [Ble] G. E. Blelloch, *NESL: A Nested Data-Parallel Language (Version 2.6)* Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, 1993.
- [BIC] G. E. Blelloch and S. Chatterjee, “VCODE: A Data-Parallel Intermediate Language,” *Proc. 3rd Symp. Frontiers of Massively Parallel Computation* (1990), 471–480.
- [CGG] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter, “External-Memory Graph Algorithms,” 1994, (Submitted).
- [CFP] P. F. Corbett, D. G. Feitelson, J-P. Prost, and S. J. Baylor, “Parallel Access to Files in the Vesta File System,” *Supercomputing* (1993).
- [Cora] T. H. Cormen, *Virtual Memory for Data Parallel Computing* Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.
- [Corb] T. H. Cormen, “Implementing Virtual Memory for Data-Parallel Computing,” 1993, White Paper.
- [CoK] T. H. Cormen and D. Kotz, *Integrating Theory and Practice in Parallel File Systems* Technical Report PCS-TR93-188, Department of Math and Computer Science, Dartmouth University, 1993, Also appears in *Proc. DAGS '93 Symposium*, 64–74.
- [CoW] T. H. Cormen and L. F. Wisniewski, “Asymptotically Tight Bounds for Performing BMMC Permutations on Parallel Disk Systems,” *Proc. 5th Annual ACM Symp. on Parallel Algorithms and Architectures* (1993), 130–139.
- [DiS] P. C. Dibble and M. L. Scott, *The Parallel Interleaved File System: A Solution to the Multiprocessor I/O Bottleneck*, Computer Science Department, University of Rochester.
- [Epp] J. Eppinger, *Virtual Memory Management for Transaction Processing Systems* Ph.D. Thesis, Carnegie Mellon University, 1988.
- [GTV] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, “External-Memory Computational Geometry,” *Proc. 34th Annual IEEE Symp. on Foundations of Computer Science* (1993), 714–723.
- [Kota] D. Kotz, *Multiprocessor File System Interfaces* Technical Report PCS-TR92-179, Department of Math and Computer Science, Dartmouth University, 1992, Abstract appeared in 1992 USENIX Workshop on File Systems.
- [Kotb] D. Kotz, *Throughput of Existing Multiprocessor File Systems (An Informal Study)* Technical Report PCS-TR93-190, Department of Math and Computer Science, Dartmouth University, 1993.
- [KSU] O. Krieger, M. Stumm, and R. Unrau, “The Alloc Stream Facility: A Redesign of Application-Level Stream I/O,” *IEEE Computer* 27 (March 1994), 75–82.
- [NoV] M. H. Nodine and J. S. Vitter, “Paradigms for Optimal Sorting with Multiple Disks,” *Proc. 5th Annual ACM Symp. on Parallel Algorithms and Architectures* (1993).
- [Pat] Y. N. Patt, “The I/O Subsystem: A Candidate for Improvement,” *IEEE Computer* 27 (March 1994), 15–16.
- [PGK] D. A. Patterson, G. Gibson, and R. H. Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID),” *Proc. 1988 ACM-SIGMOD Conf. on Management of Data* (1988), 109–116.
- [Pie] P. Pierce, “A Concurrent File System for a Highly Parallel Mass Storage System,” *Fourth Conf. Hypercube Concurrent Comput. and Applications* (1990), 155–160.
- [RuC] V. F. Russo and R. H. Campbell, “Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques,” *OOPSLA* (1989), 267–278..
- [SaG] K. Salem and H. Garcia-Molina, “Disk Striping,” *Proc. 2nd. IEEE Data Engrg. Conf.* (1986).
- [ViS] J. S. Vitter and E. A. M. Shriver, “Algorithms for Parallel Memory I: Two-Level Memories,” to appear in a special issue of *Algorithmica* on Large-Scale Memories, summary appears in “Optimal Disk I/O with Parallel Block Transfer” *Proc. 22nd ACM STOC* (1990), 159–169.
- [ZdM] S. B. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems*, Morgan Kauffman, 1990.

A Sample TPIE Code for List Ranking

The following code is intended to illustrate the use of TPIE to solve a non-trivial problem, namely list ranking. We are given an input stream of links in a linked list. These links are not necessarily given in the order they appear in the linked list, and our job is to put them in that order. The function `list_rank()`, with the help of the various objects and functions defined along with it, does this. When a list is ranked, the `weight` field of each edge is set to the number of nodes known to come before it in the list.

List ranking was chosen not only because it is a fundamental building block for many more advanced graph theoretic algorithms, but also because, like many graph algorithms, the pattern of memory accesses used by a straightforward main memory implementation are extremely erratic and highly dependent on the input problem. If implemented on a virtual memory system and run on large inputs, such an algorithm would thrash terribly because of the lack of locality in data references. Even a smart compiler designed to optimize code for I/O would be unlikely to be able to recognize the underlying semantics of the program and rewrite it efficiently for I/O. Before TPIE, the only hope a programmer would have would be to actually work directly with the I/O system of the particular machine on which the program was running. This would undoubtedly be complicated and frustrating, and yield a highly non-portable application.

In the list ranking code we present, the AMI's sorting, scanning, and merging primitives, along with support objects and functions, are used to write I/O efficient code without ever thinking about I/O.

The algorithm used is an example of a PRAM simulation algorithm as described in Section 4.3. In each stage of the algorithm the problem is reduced from n edges to X_n edges, where X_n is a binomial random variable with expectation $3n/4$. The expected running time of the algorithm is thus $O(\text{sort}(n))$. Details of the algorithm used can be found in [CGG].

Our implementation uses the following data structure to represent an edge in the list:

```
class edge {
public:
    long int from;        // Node it is from
    long int to;         // Node it is to
    long int weight;     // Position when ranked.
    int flag;           // A flag used to randomly select some edges.
};
```

Our algorithm is randomized, and requires that we be able to flip a coin and set the flag of each edge in a stream accordingly. This is done with a simple scanning object.

```
extern int flip_coin(void);

class random_flag_scan : public 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::operate(const edge &in, AMI_SCAN_FLAG *sfin,
                                  edge *out, AMI_SCAN_FLAG *sfout)
{
    if (!*sfin) return AMI_SCAN_DONE;
    *out = in;
    out->flag = flip_coin();
}
```

In order to construct a recursive subproblem, we use 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 $e_1 = (x, y, w_1, f_1)$ be the first edge and $e_2 = (y, z, w_2, f_2)$ the second. If e_1 's flag (f_1) is set and e_2 's (f_2) is not, then we write $(x, z, w_1 + w_2, ?)$ to the active list and e_2 to the cancel list. The effect of this is to bridge over the node y with the new active edge. f_2 , which was the second half of the bridge, is saved in the cancelation 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 : public AMI_scan_object {
public:
    AMI_err initialize(void) { return AMI_ERROR_NO_ERROR; };
    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::operate(const edge &e1, const edge &e2,
                                           AMI_SCAN_FLAG *sfin,
                                           edge *active, edge *cancel,
                                           AMI_SCAN_FLAG *sfout)
{
    if (!sfin[0]) return AMI_SCAN_DONE;

    sfout[0] = 1;
    if (e1.flag && !e2.flag) {
        active->from = e1.from;
        active->to = e2.to;
        active->weight += e2.weight;
        *cancel = e2;
        sfout[1] = 1;
    } else {
        *active = e1;
    }
    return AMI_SCAN_CONTINUE;
}

```

After we recursively rank the active list, we will merge the cancel list back into it, ranking the canceled edges based on the recursively computed weights of the active edges. The following scan class does this. Note that we use scanning rather than merging since we are only merging two streams together. The first stream 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.

```

class interleave_active_cancel : public AMI_scan_take_object {
private:
    int state;           // Nonzero if we are holding an input from
                        // last time.
    edge held_edge;     // An edge waiting to go in the output stream.
public:
    AMI_err initialize(void);
    AMI_err operate(const edge &in_active, const edge &in_cancel,
                   AMI_SCAN_FLAG *inf, AMI_SCAN_FLAG *takenf,
                   edge *out_edge);
};

AMI_err interleave_active_cancel::initialize(void)
{
    state = 0;
    return AMI_ERROR_NO_ERROR;
}

AMI_err interleave_active_cancel::operate(const edge &in_active, const edge &in_cancel
                                           AMI_SCAN_FLAG *inf,
                                           AMI_SCAN_FLAG *takenf,
                                           edge *out_edge)

```

```

{
    long int lost_weight;

    // If we're holding an edge from last time, output it.
    if (state) {
        state = 0;
        *out_edge = held_edge;
        takenf[0] = takenf[1] = 0;
        return AMI_SCAN_OUTPUT;
    }

    if (!inf[0])
        return AMI_SCAN_DONE;

    // If the active edge goes to a smaller node than the cancelled one,
    // skip it.
    if (in_active.to < in_cancel.to) {
        *out_edge = in_active;
        takenf[0] = 1; takenf[1] = 0;
    } else {

        // The edges should go to the same node.
        tp_assert(in_active.to == in_cancel.to,
            "Edges don't go the same place.");

        takenf[0] = takenf[1] = 1;
        held_edge = in_active;
        *out_edge = in_cancel;
        out_edge->to = held_edge.from;
        lost_weight = held_edge.weight;
        held_edge.weight = out_edge->weight;
        out_edge->weight -= lost_weight;
        state = 1;
    }

    return AMI_SCAN_OUTPUT;
}

```

We have to sort the edges before we scan them using the scan objects above. To do this, we need two simple comparison functions.

```

static int edgefromcmp(edge *s, edge *t)
{ return (s->from < t->from) ? -1 : ((s->from < t->from) ? 1 : 0); }

static int edgetocmp(edge *s, edge *t)
{ return (s->to < t->to) ? -1 : ((s->to < t->to) ? 1 : 0); }

```

Finally, we can write the recursive function that ranks the list.

```

AMI_STREAM<edge> *list_rank(AMI_STREAM<edge> *edges)
{
    AMI_STREAM<edge> *edges1;
    AMI_STREAM<edge> *active;
    AMI_STREAM<edge> *cancel;

    // Scan management objects.
    random_flag_scan my_random_flag_scan;
    separate_active_from_cancel my_separate_active_from_cancel;
    interleave_active_cancel my_interleave_active_cancel;
}

```

```

// Check if the recursion has bottomed out.

if (edges->stream_len() * sizeof(edge) < AMI_mem_size())
    return(main_mem_list_rank(edges));

// Flip coins for each node, setting the flag to 0 or 1 with equal
// probability.

AMI_scan(edges, &my_random_flag_scan, edges);

// Make a copy of the stream.

AMI_copy_stream(edges, edges1);

// Sort one stream by where the edge is from, the other by where it is
// to.

AMI_sort(edges, edgetocmp);
AMI_sort(edges1, edgefromcmp);

// Scan to produce an active list and a cancel list.

AMI_scan(edges, edges1, &my_separate_active_from_cancel, active, cancel);

// Recurse on the active list. It will return sorted by terminal (to)
// vertex.

active = list_rank(active);

// Sort the cancel list by where the edges go to.

AMI_sort(cancel, edgetocmp);

// Now merge the two lists.

AMI_scan(active, cancel, edges1, &my_interleave_active_cancel);

return edges1;
}

```