Using PI/OT to Support Complex Parallel I/O

Ian Parsons, Jonathan Schaeffer, Duane Szafron and Ron Unrau
Department of Computing Science, University of Alberta
Edmonton, AB, CANADA T6G 2H1
{ian, jonathan, duane, unrau}@cs.ualberta.ca

Abstract

This paper describes PI/OT, a template-based parallel I/O system. In PI/OT, I/O streams have annotations associated with them that are external to the source code. These annotations specify an I/O behavior (template) and some modifiers (attributes). This paper shows how PI/OT attributes can be used to handle irregular data structures, and how the templates can be hierarchically composed to support complex I/O access patterns. PI/OT's separation of I/O specifications from the source code allows users to create these parallel I/O behaviors quickly and correctly. We demonstrate these capabilities by describing how PI/OT can be used to implement a biochemistry application and by discussing the performance results.

1. Introduction

The development of parallel applications has focused on computational parallelism, with the result that parallel I/O implementation techniques have lagged behind. Support for parallel I/O is typically obtained from library packages such as PIOUS [8] or MPI-IO [2]. While these libraries support a common and standardized interface respectively, they still require that the developer explicitly differentiate between parallel and sequential I/O streams at the source code level, and often require that application data be imported or exported to and from specialized file systems (e.g. Zebra [5]). As well, the computational parallelism must often be restructured to accommodate the model supported by the parallel I/O library. A different approach is to try to preserve the sequential I/O function interface. This has been successfully done for data parallel applications in the Stream* system [7]. However, in Stream*, the programmer must still explicitly describe the computational and I/O parallelism in the source code.

A more radical alternative does not explicitly embed any parallel behavior directly in the application. Instead, a parallel programming system (PPS) such as Enterprise [12] provides a set of pre-defined parallel behaviors, called templates, that are selected by the developer and applied to an application. Examples of templates for parallel computation are pipelines and task farms. The user supplies sequential code for the stages of the pipeline or tasks of the farm, and the PPS uses the templates and programmer-supplied code to generate a parallel program that implements the specified behavior.

This high level of abstraction is beneficial for five reasons. First, parallel behaviors are encapsulated into an easy to understand set of templates. Complex behaviors can be constructed by composing or nesting several templates (e.g. a pipeline of task farms). Second, the user specifies what parallelism is needed while the template determines how the parallelism is implemented. This shields the developer from the details of the parallel implementation and allows the solution to be optimized for different architectures. Third, parallel behavior can be changed with no (or minimal) changes to the source code. This allows developers to get first-draft code working quickly, and to experiment with different parallel models so that the one with the best performance can be selected. Fourth, correct behavior of the templates is guaranteed by the PPS. Fifth, the performance of the PPS generated code can be comparable to hand-coded solutions.

Our research extends the concept of a template system to include parallel I/O. The PI/OT Parallel I/O Templates system (pronounced Pilot) showed that I/O templates can be used to easily and efficiently describe the I/O of a program at a high level [11]. PI/OT is an integrated subsystem of a PPS (Enterprise). The computational part of a parallel program is developed as usual, and includes I/O as standard sequential calls to the *stdio* library. The I/O streams to be parallelized are then annotated, external to the source code, from a set of pre-defined I/O templates. The system generates the code to correctly implement the specified parallel I/O behavior. By not differentiating between sequential and parallel I/O streams in the code, the developer can use the annotations to quickly convert sequential I/O to parallel I/O. The generated parallel program has comparable performance to hand-tuned parallel I/O code that uses low-level I/O libraries [12, 13]. Mixing different parallel computational and I/O behaviors may affect the performance of the application, but their correct implementation is orthogonal to the particular templates used. In fact, in many cases the parallel I/O template can be changed and the parallel program rerun without recompiling the application.

In [11], we showed that PI/OT achieved good performance on applications with regular I/O patterns. Unfortunately, most parallel I/O systems cannot handle complex, irregular data access patterns well. For example, although PIOUS, Elfs [4], Galley [9] and Vesta [1] excel at regular I/O patterns (such as fixed size strides or

Copyright 1998 IEEE. Published in the Proceedings of IPPS/SPDP '98, March 30-April 3, 1998, in Orlando, Florida. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

segments), they are not tuned for irregular I/O. However, there is an important class of real-world applications where I/O operations must be performed on irregular data (for example, sparse matrices and dynamic lists). This paper describes PI/OT's two research contributions towards handling arbitrarily complex parallel I/O patterns.

Irregular data structures. Parallel I/O templates make it easy to specify and support irregular data structures on disk. Each template supports a different I/O parallelization, and has attributes that refine the abstract model for a particular file structure. A PI/OT template can include a segment size attribute that can be fixed if the segment size is known in advance, or can be dynamic to accommodate variable sized records. This dynamic segmentation can be used even if the record size is not known until run-time (i.e. it is embedded as part of the file structure itself).

Hierarchical I/O. PI/OT presents a small number of templates for specifying parallel I/O. These can be viewed as building blocks for constructing more complex I/O behavior. PI/OT allows these templates to be hierarchically composed so that the developer can create an unlimited set of parallel I/O access patterns. We know of no other parallel I/O system that achieves this flexibility.

2. PI/OT templates

PI/OT currently contains the five parallel I/O templates shown as shaded boxes in Figure 1. The templates are based on Crockett's concepts of global, independent, and segmented file I/O [3]. The arcs in Figure 1 show which templates make use of which concepts. When *Independent I/O* is used, each participating process has its own file pointer that it can move independently. With *Segmented I/O* each of these independent file pointers is restricted to its own file segment. With *Global I/O*, the processes logically share a single global file pointer. However, since the processes are distributed across a network, each of the physical file pointers must be synchronized with the single global file pointer.

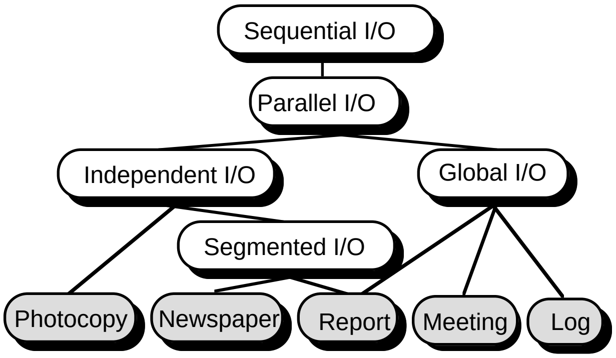

Figure 1. The PI/OT templates.

The PI/OT parallel templates add synchronization constraints to these basic abstract parallel concepts. This template collection is not complete, but can be extended if new properties are needed. However, it has not proved necessary to add new templates to the set at this point. One reason for this is that the current simple templates can be composed together to represent more complex I/O access patterns. These templates are intended to be integrated with parallel computational templates to create a parallel application.

2.1. Templates

A *photocopy* template uses independent file access, where client processes read independently, but all write operations are verified by an owner process before they become visible to other client processes. This template can be optimized by selectively replicating the file so that read operations become local.

A *newspaper* template divides a file into disjoint segments that can be read and written independently. This template is analogous to several individuals sharing a newspaper, where each individual has a section. A newspaper file segment always knows its starting point (base) in the file and the size of the segment (extent).

The *extent* is determined at compile-time by a user-supplied segmentation constant or at run-time by a user-supplied executable segmentation function. Segmentation functions support applications with irregular data structures, since each segment can have its own size that is dynamically computed at run-time. The segmentation function can even read data from disk to determine a dynamic segment size. A parent process executes the segmentation function to compute an extent before it calls a client function to process its segment.

Sometimes an extent cannot be determined before calling a client. The most common situation is for output, where the size of the segment is not known until the writing is complete. PI/OT marks such extents as *unknown*. When a client process has an unknown extent, its write operations are performed on its own local disk. When the client exits, the entire segment is sent to the parent process and written to the original file.

The *log* template uses a global file pointer with the added restriction that all writes take place at the end of the file. After a write, the global file pointer is left at the end of the file. However, any read or seek operation is free to proceed without synchronization because the data is always consistent. The end-of-file (EOF) marker is changed only when the process with write permission updates the global data structure. All other processes are limited to the last known value of the global EOF.

The *report* template segments a file like a newspaper. However, unlike a newspaper, a reader or writer can access segments other than the one it owns. However, there is a performance penalty for accessing another segment since the parent process that divides the report into segments must grant access to the external reader or writer.

The *meeting* template has a single global file pointer. Only one reader or writer has control at any one time.

2.2. Template attributes

In addition to its basic semantics, each template can have several attributes which refine its behavior. The first is the *order attribute*. It determines the access sequence for I/O operations that are executed by different processes and the visibility semantics of the updates. There are three values for the order attribute: *ordered*, *relaxed* and *chaotic*. In addition, read operations may be ordered independently from write operations so there is a read order and write order attribute for each template.

For example, the source code in Figure 2 performs six I/O operations. If the code is executed sequentially, the order of operations is: α_0, α_1, α_2, β_0, β_1, β_2. If `DoAlphaIO` and `DoBetaIO` are executed concurrently, the separate processes could potentially perform the operations in any order. If the ordered attribute is used for these I/O operations, the operations will occur in sequential order. If the relaxed attribute is used, the alpha operations will occur in arbitrary order, but will all be completed before any of the beta operations start. Then the beta operations will occur in arbitrary order. If the chaotic attribute is selected, all six operations will occur in arbitrary order. The visibility of the operations depends on both the order attribute and the I/O template. The visibility semantics are discussed in detail in [10].

```
AlphaBetaIO ( FILE *fp ) {
  int j; int alpha[3]; double beta[3];
  for ( j = 0 ; j < 3 ; j++ )
    alpha[j] = DoAlphaIO( fp );
  for ( j = 0 ; j < 3 ; j++ )
    beta[j] = DoBetaIO( fp );
  fclose( fp );
}
```

Figure 2. Code to illustrate I/O attributes.

The second attribute of each template is the I/O *transaction attribute* that specifies either atomic or block accesses. If a template has atomic I/O transactions then each I/O statement is a single transaction. For example, if a single write statement outputs two integers, no other I/O operation can output between them. However, with atomic I/O, if two adjacent write statements each output a single integer, another I/O operation from a different process may be interleaved between them. If the template has block I/O transactions, then all I/O operations from a block will be done in a single I/O transaction so that no interleaving between I/O statements is possible.

The final attribute is the *segmentation attribute*. It is only used by templates that support segmented I/O (newspapers and reports). The segmentation is either a constant that represents the extent of all segments for that template, or a function that is called at run-time to dynamically determine the extent of each individual segment. The function is an ordinary sequential function, supplied by the programmer, and is used to support irregular data structures. Section 3 contains a detailed example of a segmentation function and its application.

3. PI/OT with irregular data structures

To our knowledge, PI/OT is the only parallel I/O system that supports dynamic segmentation of files for irregular data structures. It provides this support in two ways. First, it allows the user to write a sequential function that is automatically called at run-time to compute the size of a segment. This function can even compute the segment size by reading from the data file itself. The parent process calls the segmentation function and then launches a child process that reads its data from that specific segment on disk. Since the child process is restricted to a particular disk segment, the parent process can call the segmentation function many times and launch many child processes concurrently, with no need for the child processes to synchronize their disk access.

Second, PI/OT provides a mechanism to support situations in which the segment size of an output file cannot be computed before the child process is launched. In this case, each child process writes its segment to a temporary file and then sends this data to the parent process. The parent process then writes the complete segment to the actual output file. Although this mechanism works more slowly than when the segment size is known in advance, it still provides speed-ups when the known segment size technique cannot be used, as is often the case for output files that contain irregular data.

To demonstrate these techniques, we present a biochemistry molecular docking application [6] that requires support for irregular data structures. The basic problem is to read variable length blocks of data from disk, do a computation for each block, and output a variable length block of data based on the computation. Each block represents a molecule that contains other molecules, each of which contains other molecules, etc. The standard way of performing I/O on objects is to let each object perform its own I/O. This approach can be applied to composite molecules. A composite object performs I/O by letting each of its components perform its own I/O. Therefore, each individual I/O operation is small, consisting of four to several hundred bytes. Each composite object has a different I/O block size. The experiments described in this paper are not the results of running the real application, but rather an abstracted form of the application that allows us to focus on the I/O and easily modify the granularity of the computations.

Figure 3 shows a high-level view of the main sequential program for our abstraction of the docking application. The `ComputeMolecule` function reads an object from a file (`fin`), performs calculations and writes the results to an output file (`fout`). Not only does the size of the input objects vary, but the size of the results are different for each object. Once the input is exhausted, the main program rereads the output file to analyze the results (`Stats`). Each object has its own specific read and write functions and knows which sub-objects it contains.

```
main( int argc, char **argv ) {
  FILE *fin, *fout;
  fin = fopen( argv[1], “r” );
  fout = fopen( argv[2], “w+” );
  while ( ! feof( fin ) ) {
    ComputeMolecule( fin, fout ) ;
    /* THESE CALLS CAN BE DONE IN PARALLEL */
  }
  fclose( fin ); rewind( fout );
  Stats( fout ); fclose( fout );
  return 0;
}
```

Figure 3. Sequential main program.

Since each `ComputeMolecule` call is independent of the others, multiple processes can be used to execute multiple copies of the `ComputeMolecule` function. There is no need to preserve the sequential input file order or the sequential output file order. It is only necessary to ensure that each molecule is analyzed precisely once.

For the experiments in this paper, the molecule structure of Figure 4 was used. The input file consists of a sequence of B molecules, where each B starts with a sequence of N molecule sub-sequences consisting of the molecules C, E, D and E. Each B molecule has a final E molecule at the end. For our tests, the input file contains 50 B molecules, the average size of each B molecule is about 350 KBytes and the total file is about 17 MBytes.

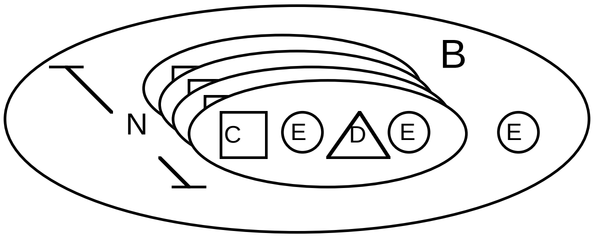

Figure 4. Structure of B molecule.

Figure 5 shows an outline of the code that does the molecule I/O. The code for molecules `C` and `D` is omitted since it is similar to the code for `E`. Notice that this code is purely sequential code with ordinary *stdio* calls. The user makes no explicit changes to this source code in parallelizing the I/O using PI/OT.

Since it does not matter which process does which piece of work, segmenting the input file avoids the inefficiency of having to synchronize file access. Each process reads a contiguous segment in the file, although the size of the segments varies. This I/O behavior is represented by the newspaper template described in Section 2. The following annotation is sufficient to automatically generate the newspaper parallel I/O code for the input file:

```
fin NEWSPAPER ro wo b ComputeMolecule=BSize
```

This annotation for file pointer `fin`, has ordered read order (`ro`), ordered write order (`wo`), block I/O transaction size (`b`), function name `ComputeMolecule` and segmentation function `BSize`. Note that the sequential user code does not call the segmentation function BSize explicitly. Instead the parallel I/O code inserted by PI/OT calls this segmentation function to split the file into segments. When PI/OT is used with a parallel programming system like Enterprise, the annotation is automatically generated from the graphical user interface when the user fills in a template attribute dialog box. Figure 6 shows the programmer supplied sequential code for the segmentation function `BSize`.

```
int ComputeMolecule( FILE *bfin, int nfin,
  FILE *bfout, int nfout ) {
  /* Process one instance of B */
  int N, i; char type ='B';
  if ( fread( &N, sizeof(int), 1, bfin ) != 1)
    return 1;
  if ( feof( bfin ) == 0 ) {
    fwrite( &type, sizeof(char), 1, bfout );
    fwrite( &N, sizeof(int), 1, bfout );
    for ( i = 0; i < N; i++ )
      CEDE( bfin, bfout );
      E( bfin, bfout );
    }
  }
  fflush( bfout ); return 0;
}
void CEDE( FILE *fin, FILE *fout ) {
  if ( feof( fin ) == 0 ) {
    C( fin, fout ); E( fin, fout );
    D( fin, fout ); E( fin, fout );
  }
}
void E( FILE *fin, FILE *fout ) {
  if ( feof(fin) == 0 ) {
    if ( ReadEData( fin, &Data ) == -1 )
      return;
    ProcessEData( &Data );
    WriteEData( fout, &Data );
  }
}
```

Figure 5. Sequential code for molecule I/O.

```
unsigned long BSize ( FILE * fp, int pmin, int
  pmax, int pcurrent ) {
  /* Segmentation function for B = n(CEDE)E */
  /* Read the number of CEDE sequences; For */
  /* each CEDE record, add its size. Finally,*/
  /* add the size of the trailing E record. */
  unsigned long offset; int N, i;
  i = fread( &N, sizeof(int), 1, fp );
  /* number of CEDE molecules */
  offset = sizeof(int);
  if ( i != 1 )
    /* End of file or file error */
    return (unsigned long) -1;
  for ( i = 0 ; i < N ; i++ ) {
    offset += AtomicSize( fp ); /* C size */
    offset += AtomicSize( fp ); /* E size */
    offset += AtomicSize( fp ); /* D size */
    offset += AtomicSize( fp ); /* E size */
  }
  offset += AtomicSize( fp ); /* E size */
  return offset;
}
unsigned long AtomicSize ( FILE * fp ) {
  /* Size function for each Atom.          */
  unsigned long offset; int M;
  /* Scratch buffer big enough for any atom */
  char buff[4096]; /* Maximum atom size */
  offset = sizeof(int);
  fread( &M, sizeof(int), 1, fp );
  /* number of elements in each vector */
  offset += M * (sizeof(int) + sizeof(char));
  /* Skip over the vectors */
  fread( buff, sizeof(char), M * (sizeof(int)
    + sizeof(char)), fp);
  return offset;
}
```

Figure 6. A segmentation function.

Output file access also needs to be coordinated. The sequential program outputs one result for each molecule B. To reduce synchronization, we can select a newspaper template to obtain a segmented output file. Since the order of the results is not important in this application, synchronization can be further reduced by assigning a chaotic attribute to the newspaper template. In this way, a call to `ComputeMolecule` does not have to wait for all previous calls to write their results before it can write its result. The annotation for the output file is:

```
fout NEWSPAPER rc wc b ComputeMolecule=0
```

This annotation for file pointer `fout`, has chaotic read order (`rc`), chaotic write order (`wc`), block I/O transaction size (`b`), function name `ComputeMolecule` and segmentation unknown (`=0`). Alternately, a log template can be used for the output file. This choice will be discussed at the end of this section.

The size of the result disk block varies for each B molecule that is processed. However, we cannot use a programmer-supplied segmentation function like we did for the input records since segmentation functions are called by the parent process before a child process is called. In this application, the parent process, `main`, cannot pre-compute the segment size of the output file, since the size of an output record is not known until the child process, `ComputeMolecule`, reads its molecule from disk and performs its computation. Therefore, we use an unknown segment size for the output file. The child process writes its output data to a temporary scratch file (local to the process if possible) as its I/O operations are executed. When the child process is finished executing, it reads the contents of the temporary file and sends this complete segment in a single message to the parent process, `main`. In this application, since the output file template is chaotic, the parent process writes these segments to the actual output file in a first-come-first-served order.

Segmenting both the input and output files eliminates the need for the `ComputeMolecule` processes to internally synchronize their concurrent activities. However, they must synchronize before the sequential `Stats` function can be called. The `rewind` function in Figure 3 serves as a barrier to guarantee that all the results are in the output file. `Stats` does a sequential read of the output file, summarizing each output record.

The parallel speedup for this application is highly dependent on the computational granularity of the specific molecules used. If the computation time for molecules is short, there is only a small speedup. As the computation time increases, so does the speedup. Figure 7 shows the performance results for various computational granularities and number of processors. One additional process was used for the `main` program. The execution time of the sequential program is also given for comparison. The different granularity numbers indicate average CPU times taken to compute the results for a single B molecule (10 seconds, 37 seconds and 147 seconds), since the nature of these computations can be changed. In the original application, a typical molecule took 37 seconds. A maximum of 16 processors were used in this set of experiments. Fourteen of them were Sun4 ELCs and the other two were slower Sun4 IPCs. The faster processors were always used first. All processors had a local disk for swap and temporary files and were connected by a 10 megabit per second Ethernet network.

no. of CM	Computational Granularity (secs)		
processes	10	37	147
sequential	479	1853	7339
2	512	1361	4875
5	251	535	1824
10	187	324	1062
15	148	255	679

Figure 7. Performance results.

As the computational granularity increased, parallelization produced a better speed-up since a larger portion of the time was spent doing computation as compared to I/O. Figure 8 shows the speedups.

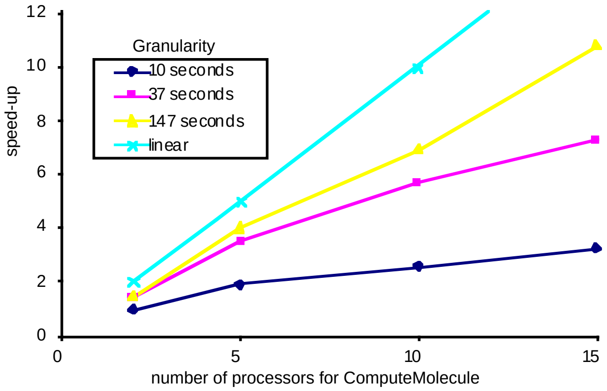

Figure 8. Speedup.

It is important to clearly understand the role that PI/OT plays in improved application performance. The speed-up shown in Figure 8 is a measure of the speed of a parallel application that uses parallel I/O, versus a sequential application that uses sequential I/O. There are two components to this speed-up: one due to parallel computation and the other due to parallel I/O. We are not claiming that all of the speed-up in Figure 8 is directly due to parallel I/O. In fact, it is possible to write a parallel application that relies on sequential I/O since each parallel process could send its I/O to a single I/O process that could do all of the physical I/O sequentially. The programmer could explicitly insert all necessary I/O synchronization in the source code. If the granularity of the computations was high enough and the times of the I/O operations did not overlap very much, most of the speed-up would result from computational parallelism. However, we think that dividing speed-up responsibility is irrelevant. The bottom line is that PI/OT allows a programmer to rapidly match an I/O template to a computational structure so that parallel I/O is painless to implement. The alternative of hand-crafting I/O to match a parallel computational structure is unappealing.

The dynamic segmentation function shown in Figure 6 is fairly complicated since, although each atomic object (molecule) stores its size, each non-atomic molecule must compute its size by calling the size function of each of its components. This results in each molecule being essentially read twice, once to determine its size and once when the data is actually read. Would a simpler segmentation function increase performance? This application shows that in practice, the actual segmentation function is relatively insignificant. For example, we replaced the complicated segmentation function, `BSize`, of Figure 6 by one that simply reads the length of molecule B from disk in one read operation (after changing the format of the molecule on disk so that the length is there). With this change, the program speed improvement is only 1%. In fact, if the size of all of the B molecules is made the same, and the segmentation function is then replaced by a segmentation constant, then there is no further measurable improvement. Therefore, our approach of calling a user-supplied dynamic segmentation function supports dynamic molecule configurations without any significant segmentation function performance penalty.

In this paper we have presented an application that uses a single parallel I/O template, the newspaper, to demonstrate template-based parallel I/O. We have done this for simplicity so that the reader would not have to learn the semantics of more than one template. However, it is easy to experiment with different templates to compare their relative performance for any particular application. For example, a programmer could select a log template instead of a newspaper, for the output file of the `ComputeMolecule` process. Unfortunately, for the docking problem, a log turns out to be a poor choice since the variable size of the output records causes too much synchronization overhead and results in a 25% slowdown from the sequential time. Nevertheless, it is easy to determine that a log template is a poor choice, since no code has to be rewritten to try it. It took only a few minutes to change the template to a log and re-run the program. It takes hours to modify the I/O source code in a non-template-based system like PIOUS to produce a similar change in functionality.

4. Support for complex I/O patterns

The templates are sufficient to represent the I/O patterns in many parallel applications. However, especially in applications that perform computations on complex data structures, more complex patterns can sometimes result in improved performance. For example, in the molecular docking application of Section 3, when the granularity of the CPU computations increases beyond 10 seconds per molecule, it is more efficient for a `ComputeMolecule` process to launch separate processes for computing multiple CEDE chains. Figure 9 shows a new process diagram that represents this change in computational parallelism.

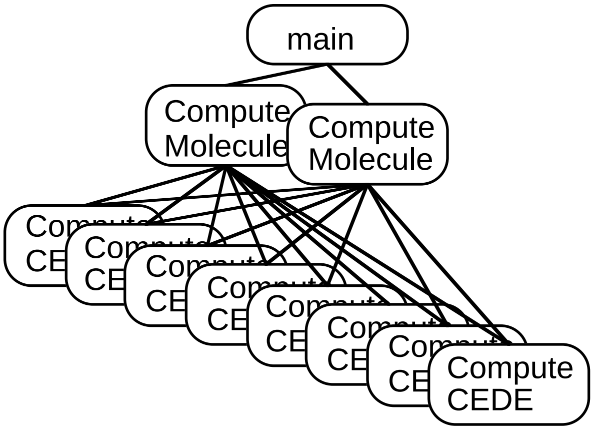

Figure 9. Composite template processes.

If the programmer was using a template-based parallel programming system, then this change in computational parallelism could be achieved by changing the computational template. For example, in the Enterprise system, this change could be accomplished by drawing a diagram similar to Figure 9 and by moving the code for the sequential function `ComputeCEDE` to a separate source file.

In this case, it is useful to modify the I/O templates so that the new processes perform their own parallel I/O. That is, the parallel I/O behaviors should be modified to match the new computational behavior. Note that since PI/OT uses the sequential source code of the original `ComputeMolecule` function, no I/O code needs to be changed. PI/OT supports the hierarchical composition of I/O templates that can be used to rapidly mirror changes in the computational parallelism. In this case, the N segment newspaper template of the input file of the `ComputeMolecule` function can be modified to contain a newspaper template for `ComputeCEDE`. Similarly the output file can be transformed to a newspaper of newspapers. The following annotations defines the hierarchy of newspaper templates for the input and output files for the process diagram of Figure 9:

```
fin NEWSPAPER ro wo b ComputeMolecule=BSize
fin NEWSPAPER ro wo b ComputeCEDE=CEDESize
fout NEWSPAPER rc wc b ComputeMolecule=0
fout NEWSPAPER rc wc b ComputeCEDE=0
```

These annotations are the only changes that a programmer must make to the parallel I/O in order to use a newspaper of newspapers instead of a simple newspaper. No changes to the code are necessary.

Figure 10 contains performance results for the composed newspaper I/O templates used in the process diagram of Figure 9. We used the same input file that was used to generate the performance results in Figure 7, with 147 second computational time. Four different strategies are shown for distributing the `ComputeMolecule` and `ComputeCEDE` processes among 10 of the processors. In addition, one processor is used to execute the `main` process, one processor is used to manage the multiple processes for ComputeMolecule and one processor is used to manage the multiple processes for ComputeCEDE. Notice that using 10 processes for ComputeMolecule and 0 for ComputeCEDE corresponds to the 1062 second time in Figure 7. The results in Figure 10 are the average of 5 trials with less than 1% variation between trials.

CMolecule replication	CCEDE replication	Time (sec)
10	0	1062
2	8	1023
3	7	1158
4	6	1344

Figure 10. Composed I/O templates.

These strategies can be quickly tested since no code needs to be re-compiled. Only computational annotations need to be changed. The details of these annotations are not germane to this discussion except that the number of processes allocated to ComputeMolecule and ComputeCEDE can easily be changed subject to the constraint that a total of 10 processes are used. In this case, we can see from Figure 10 that allocating 2 processes to ComputeMolecule and 8 processes to ComputeCEDE is best. This means that the hierarchical process structure of Figure 10 has a 4% better performance result than the simple process structure of Section 3. Previously, you could argue that the increased difficulty in coding a more complex process structure would not justify such a marginal performance improvement. However, with templates for both computation and I/O, the code does not change and it is easy to experiment with different process hierarchies.

These experiments were re-run on a heterogeneous network consisting of one Sparc 10 (SS10) dual processor unit, two Sun4 Classics, five Sun ELCs and five Sun SLCs. The same general pattern emerged in that the 2/8 split performed the best. However, in this case the 3/7 split was also better than the 10/0 split. This difference is due to the widely varying processing speeds of the machines. The important point here is that the template I/O approach allows you to rapidly experiment to find the best configuration for the machines you have on hand.

5. Conclusions and future work

Parallelizing the I/O part of a distributed parallel application is difficult for three reasons. First, the data files must often be segmented to allow parallel access. Segmentation is especially difficult for irregular data structures whose structure can only be determined at run-time. Second, the I/O accesses must be coordinated with the computational parallelism. Third, the low-level details of distributed I/O, such as file pointer management and synchronization, are difficult to implement correctly.

PI/OT addresses all of these difficulties in the context of an integrated template-oriented parallel programming system. High level parallel I/O behaviors can be described using the pre-defined templates. Multiple templates can be composed to extend the descriptive power of the template behavior, and static or dynamic segmentation is supported through the template attributes. The I/O can be expressed as sequential accesses in the parallel program, making it easier to implement the access semantics correctly. The templates are then applied automatically by the PPS to parallelize the accesses in concert with the parallel computational structure. Low-level file pointer management and synchronization is correctly and automatically handled by the system.

We have demonstrated these properties on a sample application from biochemistry that illustrates PI/OTs capabilities for handling dynamic segmentation of irregular data structures. We have also shown how the compositional power of templates is easily incorporated to support complex parallel computation and I/O structures.

References

[1] P. Corbett, S. Baylor, and D. Feitelson, "Overview of the Vesta Parallel File System," IPPS '93 Workshop on I/O in Parallel Computer Systems, pp. 1-16, 1993.

[2] P. Corbett, D. Feitelson, Y. Hsu, et. al., “Overview of the MPI-IO Parallel I/O Interface,” Third Workshop on I/O in Parallel Distributed Systems, pp. 1-15, 1995.

[3] T. Crockett, “File Concepts For Parallel I/O” Supercomputing'89, pp. 574-579,1989.

[4] A. Grimshaw and E. Loyot Jr., “ELFS: Object-Oriented Extensible File Systems,” University of Virginia, Computer Science Report TR-91-14, July 1991.

[5] J. Hartman and J. Ousterhout, “The Zebra Striped Network File System,” *ACM Transactions on Computer Systems*, 13(3), pp. 274-310, 1995.

[6] T. Hart and R. Read, "A Multiple-Start Monte Carlo Docking Method," *Proteins: Structure, Function and Genetics*, 13, pp. 206-222, 1992.

[7] J. Moore, P. Hatcher, and M. Quinn, "Stream*: Fast, Flexible, Data-Parallel I/O," Proceedings of Parallel Computing 95, pp. 287-294, 1995.

[8] S. Moyer and V. Sunderam, “Scalable Concurrency Control for Parallel File Systems,” Third Workshop on I/O in Parallel and Distributed Systems, pp. 90-106, 1995.

[9] N. Nieuwejaar and D. Kotz, “Performance of the Galley File System,” Fourth Annual Workshop on I/O in Parallel and Distributed Systems, pp. 83-94, 1996.

[10] I. Parsons, "PI/OT: A Template Approach to Parallel I/O," Ph.D. Thesis, Department of Computing Science, University of Alberta, 1997.

[11] I. Parsons, R. Unrau, J. Schaeffer, and D. Szafron, "PI/OT: Parallel I/O Templates," *Parallel Computing*, Vol. 23, No. 4-5, pp. 543-570, May 1997.

[12] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons, “The Enterprise Model for Developing Distributed Applications,” *IEEE Parallel & Distributed Technology*, 1(3), pp. 85-96, 1993.