Appendix A

A. Enterprise Parallel Programming System

The Enterprise PPS [70] uses an analogical approach to help users develop their programs. It has successfully abstracted several parallelizing techniques so that the user code is written in conventional C and no special library functions are needed by the user. Enterprise does not analyze the code in order to parallelize it. Rather, by using the directions provided by the user, it inserts the necessary parallelizing code fragments according to predefined templates. Enterprise uses asynchronous message-passing where a message is defined by the formal parameter list of a user-defined asset. Analyzing the user's code identifies any synchronization points or futures.

The advantage of Enterprise is that the user has portable code, both from an architectural and a communication subsystem point of view. An Enterprise program provides similar behaviour regardless of the processor combination. Of course, performance will likely be different.

Appendix A.1 introduces the Enterprise programming model and explains how the user interacts with this PPS to create a working parallel application. A more detailed discussion is found in the Enterprise User's Manual [45]. What is presented here is pared down and is only sufficient for the parallel I/O discussion. More information about the different aspects of the Enterprise system can be found in [44, 49, 51, 64, 84]. Appendix A.2 examines the run-time libraries used by Enterprise to create the parallel computational behaviours. These libraries were completely redesigned and re-implemented in an object-oriented manner [51]. Because of this redesign, the implementation of parallel I/O in Enterprise was simplified.

A.1 Enterprise Programming Model

Enterprise seeks to separate the parallel implementation details from the user's source code. By means of compiler technology, the source code and the specifications for parallelism are blended to produce machine-generated source code reflecting the parallelism desired. The user has no special libraries of functions to learn; all user code is written in standard C. A graphical user interface allows the user to express the parallelism in an intuitive, visual fashion.

Enterprise uses a business analogy to represent the different parallel behaviours. A business is a naturally parallel object and can be viewed as containing *assets* that represent functions that can be remotely and concurrently executed. There is a corresponding well-defined chain of command progressing from the top to the bottom of the organization.

Starting with the initial *program* icon, the user modifies the asset type to represent a specific type of parallel behaviour. The choices of asset types are available from a menu. The assets are connected by a series of arcs to form the **control-flow** diagram for the parallelism in the application. The arcs joining these icons represent the communication paths for messages and control flow data. Each asset is associated with a user-defined function. To communicate over a given arc, the asset at the start of the arc invokes the **function** associated with the asset at the end of the arc. One of the current drawbacks to this model is the inability of the user to express peer-to-peer communication.

The assets have simple parallel behaviours and can be nested within one another to create more complex parallel behaviours. Currently, the available assets are: *individual*, *service*, *line*, *department*, and *division*. An *individual* asset corresponding to an individual worker consists of a sequential function. There is no special parallel behaviour associated with an individual other than indicating to the PPS that this function can be executed remotely. A *service* is similar to an *individual* except that a *service* cannot initiate remote calls to other assets and it is fully connected to all assets.

Next are *composite assets: line*, *department* and *division*. A composite asset contains other assets and has a specified parallel behaviour associated with it. A *line* asset represents the assembly-line in a business. Each worker does some work and then passes the partially completed work on to the next worker in the line. The starting worker in a line is considered the *receptionist*. All other workers initiate work requests through this worker.

A *department* has a *receptionist* who directs incoming calls to the most appropriate asset. The *receptionist* could also divide the work into heterogeneous portions and distributes it to all or some of its workers to execute concurrently.

The *division* is an attempt to represent the parallelism in divide-and-conquer problems. Every worker or *representative* in a division does the same job. By splitting the work into smaller chunks, eventually a particular *representative* will not make a remote call but will recursively perform the work itself. For a *division*, there is also a special *manager* asset whose purpose is to forward work to a *representative*. The user is not responsible for writing the code for this generic asset but should be aware of its existence at run-time.

Each asset has a collection of five attributes. First, and most important, each asset has some user-defined, sequential function associated with it. The **name** of the asset corresponds to the name of the function. Second, an asset has a **replication factor**. This is defined as a maximum number of replicas and a minimum number of replicas. If the asset is considered a composite asset, all the assets contained within this composite asset are replicated as well.

Third, an asset can have **ordered** or **unordered** parallel behaviour. Ordered behaviour tells the Enterprise run-time system that when multiple calls are made to one asset and a return reply is indicated, the order in which the calls were initiated is the same order in which the corresponding replies will be processed. Unordered behaviour tells the run-time system to process whichever reply is available to the caller.

Fourth, an asset can be **optimized** and/or **debugged**. To optimize an asset currently means the conventional compiler will use the `-O2` compiler flag. In the future, the source-to-source compiler will reorganize the user's code to optimize the parallelism. This could be done by loop reorganization, moving code to ensure more overlap of concurrent computations. If an asset has the debug attribute set, this currently means the conventional compiler will use the `-g` compiler flag. This debug attribute provides only the process level of debugging in the parallel environment.

Parallel debugging is a run-time option. When this option is enabled, **event monitoring** is done for all or some of the processes. Event monitoring generates trace files containing significant events in the parallel application: messages sent, received, processes being spawned and the like. The user can replay the application using these trace files to debug individual processes. Trace files are also used to visualize and debug application performance.

Fifth, the user defines sets of processors either the asset must be run on or cannot be run on as another attribute pair of the asset. The user can specify a mutually exclusive subset of the machine list for either attribute. When running the application, the run-time system tries first to place the process running the asset on the desired processor list. If this list is not specified, a processor is selected from the sub-list composed of the available machine names with the unacceptable machine names removed. Otherwise, the Enterprise run-time system will use the entire machine list. In all cases, the processor chosen will be the machine with the lightest load. The load for a processor is determined by the Enterprise run-time library and may require more information than just the number of active processes on a given processor. This could include information such as: the amount of network activity flowing to and from a processor, the level of I/O activity (network or local), the type of users (interactive, batch, or owner).

The interface of the first asset is predefined to have the same arguments as the `main` function in a sequential C program. This allows the user to utilize the command-line argument list to pass data to the application. As well, the first asset is not permitted to be replicated.

A.2 Enterprise Implementation

The Enterprise implementation is presented as three distinct components: the graph file, the pre-compiler, and the run-time system. Each component has been modified to implement the parallel I/O templates. What is being presented is sufficient for the parallel I/O model but is not complete as to the full capabilities of Enterprise.

A.2.1 The Graph File

The **graph file** is the heart of the Enterprise system. It is the textual representation or definition of the assets, their parallel behaviour and attributes. The graphical interface generates this file for the user. The pre-compiler and the run-time system use this file to generate the user-specified behaviours. Figure A-1 shows the generic layout for one asset with the bolded text being comments added to clarify and classify the actual text. The bolded text is not included in the actual graph file.

Five lines are used to represent a given asset. Lines two through five are always treated the same way, regardless of the asset type selected. However, the definition of the first line is dependent on the asset type selected. The first line is the same only for the first two entries. They are: the name of the function associated with the asset template (`assetName`), and the parallel behaviour associated with the function (`assetType`). The `assetType` can be one of six values: `line`, `department`, `division`, `individual`, `representative`, and `service`. Depending on the selection of the asset type, the balance of the line is different.

Composite assets (`line`, `department`, or `division`) have the number of internal assets defined. For all assets other than the `service` asset, the minimum and maximum replication factors are then defined. The first line is completed by all assets defining if ordering (`ORDERED` or `NORDERED`), debugging (`DEBUG`, or `NDEBUG`) and optimization (`OPTIMIZE` or `NOPTIMZE`) are enabled or disabled.

Figure A-2 shows an actual graph file with two entries. Sub-assets are defined in a depth-first fashion with service assets being appended to the end of the file. This layout is

```
All Assets:
  assetName assetType
Composite Assets (line, department, division):
  numberOfSubassets minReplication maxReplication Order Debug Optimize
Individual Assets (individual, representative):
  minReplication maxReplication Order Debug Optimize
Other Asset (service):
  Order Debug Optimize
All Assets:
  CFLAGS
  EXTERNAL
  INCLUDE
  EXCLUDE
```

Figure A-1 — Annotated graph file entry for one asset different from the user manual and reflects the changes made in the redesigned system.

Recalling the example given in Chapter 1.1, the parallel behaviour of a parent-child relationship is easy to implement in the graph file. Figure A-2 shows the resultant graph file. It describes the computational parallelism of the application. The first asset is a composite asset called `Parent` and contains the function `Parent`. The signature for `Parent` is: `int, char **`. This is to conform with the sequential C interface to the user found in the function, `main`. `Parent` makes at least one call to its sub-asset, the asset `Child`, to execute some work. Otherwise, a compile-time error will occur. `Parent` is not replicated (nor is it allowed to replicate since it is the first asset). This asset is to be treated as having ordered behaviour with no debugging information needed, nor is the asset to be optimized. When run, the asset is to be run on the processor called **sherwoodpk** (if available) and cannot be run on the processor known as **maligne-lk**.

```
Parent line 1 1 1 ORDERED NDEBUG NOPTIMIZE
CFLAGS
EXTERNAL
INCLUDE sherwoodpk
EXCLUDE maligne-lk
Child individual 3 4 NORDERED NODEBUG NOPTIMIZE
CFLAGS
EXTERNAL
INCLUDE
EXCLUDE sherwoodpk
```

Figure A-2 — An example graph file.

`Child` asset is an `individual` asset that will be run as a separate process. It can only be called by the asset known as `Parent`. The function, `Child`, will be replicated starting with a minimum of three but expandable to a maximum of four processes. Successive return values from `Child` do not have to be processed in the original call order but rather whenever work is received by `Parent`. No debugging information is needed nor is the asset to be optimized. No `Child` asset process is allowed to run on the processor called **sherwoodpk**.

A.2.2 The Precompiler and Static Analysis

The Sage++ tool kit [8] is used to build the source-to-source compiler or *precompiler*, for Enterprise. Each asset is stored in a separate file. The precompiler examines each asset source file. It searches for function calls to other assets as defined in the graph file. It replaces the function call with a call to the wrapper function or *stub* that will pack the message containing the function parameters and send it to the remote process.

The precompiler identifies *futures* (variables that will be modified by a reply message) and inserts Enterprise-specific code fragments to ensure that futures are resolved prior to using them. Currently, any user source code that is not explicitly defined to be parallelized is not examined by this tool.

This wrapper code crosses the boundary from the user's code written in the standard C language into the run-time system which is written in C++. This *stub* function is a small function that contains sufficient information about each of the formal parameters of the remote function that permit the runtime libraries to marshal the formal parameters into a single message, identify the appropriate process, send the data over to the remote process for processing, add to the *futures* list when necessary, demarshal return values, and resolve any *futures* when a return message is received from a remote process.

A.2.3 The Run-time Libraries

MacDonald [51] completely redesigned and re-implemented the run-time system using C++. One of the significant changes was the use of behaviour classes to determine the parallel action appropriate for the asset. There are **single**, **worker**, **manager** and **root** asset class behaviours. During program execution, a process running an asset can exhibit different behaviours. For example, a process could be initialized as a **single** asset. Later on, a need develops for several copies of the asset running. The process promotes itself to **manager** of this asset and spawns the necessary number of **worker** assets. When the task is finished, the **worker** assets are dismissed and the **manager** demotes itself to become the **single** asset again. The **root** asset is responsible for user-interface and run-time system interactions.

The graph file is read and transformed into the **asset graph**. Recalling that the graph file is considered the heart of the Enterprise system, the asset graph should be considered the brains of the Enterprise parallel run-time system. It maintains both the static information contained in the graph file as well as the dynamic information acquired at run-time. By asking the graph, a process can determine what asset it is supposed to represent, the current, minimum, and maximum replication factors, what asset can call it and more importantly, what asset it can call, which process to send the message to, and what futures are outstanding.

The asset graph consists of the different components that make up a parallel program. This includes the user's asset definitions as well as the extra generic assets that are necessary to run the parallel application. These generic assets are the **root** and **manager** assets.

The **root** asset is responsible for being the interface between the GUI and the run-time system. The root asset is responsible for starting up and shutting down the parallel application. It spawns and starts the first asset by packing the command line arguments and sending it the resultant message. If event monitoring is needed, it collects events from all processes and either forwards them on to the GUI for real-time monitoring or stores them in a file for post-run analysis. It is responsible for coordinating the debugging of the entire application at the event level. Because of its responsibilities, the **root** asset is always a standalone process.

The other generic asset, the **manager** asset, uses a simple store-and-forward process to ensure even workload distribution. It is responsible for recruiting and disposing of new worker assets. One optimization of this asset is that it can be collapsed so that a single process contains more than one asset: one user-defined and one or more managers. Typically, this happens when there is only one caller of a replicated asset. The intent is to save one or two network messages. There are two situations when this manager asset cannot be collapsed. This occurs when more than one asset calls a replicated asset or when an asset is a *division*.

Appendix B

B. PIOUS Test Application Codes

This appendix contains the PVM/PIOUS implementation for the various test applications discussed in the dissertation. Appendix B.1 contains the PIOUS code for the fine-grained example used in Chapter 5.1. Appendix B.2 contains the PIOUS code for the coarse-grained I/O example used in Chapter 5.2.

B.1 Small-Grained I/O Example Program.

From the original sequential code, there is a significant amount of new code that needs to be written. The original version is about 530 lines of code. Converting to parallel increases the program size by approximately 350 lines. The bold lines in the code represent the original code that the user had to write. For this example, the sequential code for `Child` did not have to be modified. As much as possible, the PVM code for the computational parallelism has been hidden away.

There are several functions that the `Parent` process calls to create (`CreateFileInPIOUS`), import files into (`ImportFileToPIOUS`), or export files from (`ExportFileFromPIOUS`) the PIOUS files system. These are tuned to the application granularity of an I/O segment and are specific to the application.

The `Parent` spawns all the `Child` processes (`CreatePVMChildren`) and waits for all the `Child` processes to finish (`WaitForChildrenToFinish`) before proceeding with the summary statistics (`Stats`). In the case of the segmented I/O, the `Parent` must also coordinate the access of the segments by the `Child` processes to ensure that all the segments are read and written only once. Note the distinction between the parallel and sequential I/O for the same file.

A `Child` process opens the global input and output files, copies the local segment of work (in one operation) to a temporary file, opens the temporary output file, performs the work and then exports the local output file back to the parallel output file (again one operation). This repeats until the global input file is exhausted. The `Parent` process is then informed and the `Child` process gracefully exits after cleaning up the temporary files.

```
#include <pvm3.h>
#include <pious1.h>
#include <stdio.h>
#define GROUP "iog"
#define MYMESSAGE 1234
#define MAXPATHLEN 1024
#define INBUFSIZE 352108
#define OUTBUFSIZE 18050
#define REGMODE  ( (pious_modet) ( PIOUS_IRUSR | PIOUS_IWUSR |
                                   PIOUS_IRGRP | PIOUS_IROTH ) )
main( int argc, char **argv )
{
   int myTID ;                                    /* My PVM handle */
  int myParentTID ;                      /* My parent's PVM handle */
  int *childTID ;                /* List of children's PVM handles */
  int dscnt ;                                     /* PIOUS handle */
  int nchild ;                          /* Number of children wanted */
  int infd, outfd ;                   /* The parallel file descriptors */
```

```
int i ;                                                     /* A counter */
int bufid, status;                         /* PVM buffer handle and status */
FILE *fp ;                            /* Local segment's input file handle */
FILE *ofp ;                          /* Local segment's output file handle */
char ibuffer[ INBUFSIZE ] ;                              /* Input buffer */
char obuffer[ OUTBUFSIZE ] ;                            /* Output buffer */
char inFile[ MAXPATHLEN ] ;                          /* Global input and */
char outFile[ MAXPATHLEN ] ;                         /* Output file path */
char myTmpInFile[ MAXPATHLEN ] ;                  /* Temporary input and */
char myTmpOutFile[ MAXPATHLEN ] ;                   /* Output file paths */
if ( ( myParentTID = pvm_parent() ) == PvmNoParent ) {
  /*
   * Parent -- spawn child processes
   * argv[0]:  process name            argv[1]:  input file name
   * argv[2]:  output filename         argv[3]:  Number of child processes
   */
  nchild = atoi( argv[3] ) ;            /* How many child processes are wanted */
  ImportFileToPIOUS( argv[1] ) ;          /* Import the UNIX file into PIOUS */
  CreateFileInPIOUS( argv[2] ) ;          /* Create the output file in PIOUS */
  childTID = (int *)malloc( nchild * sizeof( int ) ) ;  /* Allocate handle vector */
  CreatePVMChildren( childTID, argc, argv ) ;   /* Spawn all the child processes */
  WaitForChildrenToFinish( nchild ) ;   /* Wait for them to finshed processing */
  free( childTID ) ;                            /* Free up child handle vector */
  ExportFileFromPIOUS( argv[2] ) ;       /* Export the output file back to UNIX */
  fp = fopen( argv[2], "r" ) ;                        /* Open sequential file */
  Stats( fp ) ;                                      /* Calculate the totals */
  pvm_exit() ;                                                 /* Leave PVM */
} else {                                               /* I'm a child process */
  /* Child process -- wait for names of files to open */
  bufid = pvm_recv( myParentTID, MYMESSAGE ) ;
  status = pvm_upkstr( inFile ) ;                           /* The input file */
  status = pvm_upkstr( outFile ) ;                         /* The output file */
  dscnt = pious_sysinfo( PIOUS_DS_DFLT ) ;   /* Ask for PIOUS default information */
  /* Open the PIOUS input and output files */
  infd = pious_popen( GROUP, inFile, PIOUS_GLOBAL, INBUFSIZE,
                      PIOUS_VOLATILE, PIOUS_RDONLY, PIOUS_IRUSR, dscnt ) ;
  if ( infd < 0 )
    printError( status, "Opening input file: child" ) ;
  outfd = pious_popen( GROUP, outFile, PIOUS_GLOBAL, OUTBUFSIZE,
                       PIOUS_VOLATILE, PIOUS_RDWR | PIOUS_CREATE |
                       PIOUS_TRUNC, REGMODE, dscnt ) ;
  if ( outfd < 0 )
    printError( status, "Opening output file: child" ) ;
  /* Create local copy of input/output files for this segment */
  sprintf( myTmpInFile, "/tmp/in.%x", pvm_mytid() ) ;
  sprintf( myTmpOutFile, "/tmp/out.%x", pvm_mytid() ) ;
  while (1) {
    status = pious_read( infd, ibuffer, INBUFSIZE ) ;      /* Read the next block */
    if ( status < 0 ) {                                                /* Error */
      printError( status, "Reading input: child" ) ;
    } else if ( status == 0 ) {                                     /* All done */
      break ;
    } else if ( status > 0 ) {                              /* Normal situation */
      fp = fopen( myTmpInFile, "w+" ) ;                  /* Open local input file */
      fwrite( ibuffer, sizeof(char), INBUFSIZE, fp ) ;              /* Fill it up */
      rewind( fp ) ;                             /* Get it ready for the user's code */
      ofp = fopen( myTmpOutFile, "w+" ) ;              /* Open the local output file */
      Child( fp, ofp ) ;                                         /* Call user's code */
```

```
      rewind( ofp ) ;              /* Export the local output file to the global file */
      status = fread( obuffer, sizeof(char), OUTBUFSIZE, ofp ) ;
      status = pious_write( outfd, obuffer, status * sizeof(char) ) ;
      fclose( fp ) ;                        /* Close the local input and output files */
      fclose( ofp ) ;
    }
  }
  /* Shutdown this child and let the parent know */
  bufid = pvm_initsend( PvmDataRaw );                              /* A buffer please */
  status = pvm_send( myParentTID, MYMESSAGE ) ;                        /* Tell parent */
  status = pious_close( infd ) ;                           /* Close PIOUS input file */
  status = pious_close( outfd ) ;                         /* Close PIOUS output file */
  pvm_exit() ;                                                           /* Exit PVM */
  unlink ( myTmpInFile ) ;                            /* Remove the local input file */
  unlink ( myTmpOutFile ) ;                          /* Remove the local output file */
 }
 return 0 ;
}
```

B.2 Coarse-Grained I/O Example

From the original sequential code in Figure 5-7, it can be seen that there is a significant amount of new code that needs to be written. The sequential version is about 225 lines of code. Converting to parallel increases the program size by approximately 350 lines. The code shown has been clarified and shortened by removing the timing and resource utilization code.

In this application, the sequential code for `Child` (Figure 5-8) did have to be modified. The file pointers were changed to PIOUS file handles and the UNIX read and write functions were converted to PIOUS read and write functions. It is inefficient to cache the striped data to local disk and the re-read the local file into memory. As well, the **B** matrix file was too large to cache locally.

B.2.1 Source code for Parent.c

```
#include <pvm3.h>
#include <pious1.h>
#include <stdio.h>
#define GROUP "iog"
#define MYMESSAGE 1234
#define MOREWORK 4321
#define MAXPATHLEN 1024
#define REGMODE ((pious_modet)(  PIOUS_IRUSR | PIOUS_IWUSR | \
                                 PIOUS_IRGRP | PIOUS_IROTH ) )
int Child( int, int, int, int, int ) ;       /* Note the change in parameter type */
main( int argc, char **argv )
{
   int myTID;                                    /* The PVM tid for this process */
   int myParentTID ;                             /* The process who spawned me */
   int *childTID ;                                  /* A vector of children tids */
   int dscnt ;                               /* The default PIOUS configuration */
   int segment ;                          /* The current file segment to work on */
   int nsegments ;                               /* Number of segments in a file */
   int tid, length, msgTag ;             /* The tid, length and tag for a message */
   int nelemems, rowsPerBlock ;  /* The number of elements per row and rows/block */
   int IOBUFFERSIZE ;                                   /* The size of the I/O buffer */
   int nchild, numbt ;                            /* The number of child processes */
   int infd, outfd ;                                 /* The parallel file descriptors */
   int ainfd, binfd, coutfd ;                      /* Child parallel file descriptors */
```

```
int i,j, k ;                                                        /* Counters */
int inputNumber ;
int bufid, status ;                         /* Message handles and status variable */
FILE *fp ;                      /* UNIX file descriptor to import and export files */
char ainFile[ MAXPATHLEN ] ;                          /* The A matrix input file name */
char binFile[ MAXPATHLEN ] ;                          /* The B matrix input file name */
char coutFile[ MAXPATHLEN ] ;                        /* The C matrix output file name */
char *iobuffer ;
if ( ( myParentTID = pvm_parent() ) == PvmNoParent ) {
  /*
   * Parent -- spawn child processes
   * argv[0]:   Process name
   * argv[1]:   Matrix A input file name
   * argv[2]:   Matrix B input file name
   * argv[3]:   Matrix C output  filename
   * argv[4]:   Number of elements per row
   * argv[5]:   Number of rows per block (segments)
   * argv[6]:   Number of child processes
   */
  nelemems = atoi( argv[4] ) ;                       /* Number of Elements per row */
  rowsPerBlock = atoi( argv[5] ) ;                                /* Rows per block */
  nchild = atoi( argv[6] ) ;                             /* Number of child processes */
  nsegments = nelemems / rowsPerBlock ;           /* Precalculate number of segments */
  IOBUFFERSIZE  = nelemems * rowsPerBlock * sizeof(double) ;
  iobuffer = (char *)malloc( IOBUFFERSIZE ) ;
  /* Now, create the PIOUS files from the UNIX files */
  dscnt = pious_sysinfo(PIOUS_DS_DFLT) ;       /* Get default PIOUS configuration */
  /* Open the A matrix file and import it into PIOUS */
  infd = pious_popen( GROUP, argv[1], PIOUS_GLOBAL, IOBUFFERSIZE,
                      PIOUS_VOLATILE, PIOUS_RDWR | PIOUS_CREAT | PIOUS_TRUNC,
                      REGMODE, nsegments ) ;
  fp = fopen( argv[1], "r") ;                                  /* Open the UNIX file */
  while ( ! feof( fp ) ) {                       /* Read in until EOF is encountered */
    i = fread( iobuffer, sizeof( char ), IOBUFFERSIZE, fp ) ;   /* Read a block */
    i = pious_write( infd, iobuffer, i ) ;                         /* Export a block */
  }                                                        /* Written the whole file out */
  status = pious_close( infd ) ;                               /* Closing the PIOUS file */
  fclose( fp ) ;                                                  /* Close the UNIX file */
  /* Open the B matrix file and import it into pious */
  infd = pious_popen(  GROUP, argv[2], PIOUS_GLOBAL, IOBUFFERSIZE,
                       PIOUS_VOLATILE, PIOUS_RDWR | PIOUS_CREAT | PIOUS_TRUNC,
                       REGMODE, dscnt ) ;
  fp = fopen( argv[2], "r") ;                                  /* Open the UNIX file */
  while ( ! feof( fp ) ) {                       /* Read in until EOF is encountered */
    i = fread( iobuffer, sizeof( char ), IOBUFFERSIZE, fp ) ;     /* Read block */
    i = pious_write( infd, iobuffer, i ) ;                           /* Export block */
  }                                                        /* Written the whole file out */
  status = pious_close( infd ) ;                               /* Closing the PIOUS file */
  fclose( fp ) ;                                                  /* Close the UNIX file */
  /* Open and create the C Matrix output file */
  for ( i = 0 ; i < nsegments; i++ ) {                          /* Create each segment */
    outfd = pious_popen( GROUP, argv[3], PIOUS_SEGMENTED, i, PIOUS_VOLATILE,
                         PIOUS_RDWR | PIOUS_CREAT | PIOUS_TRUNC, REGMODE,
                         nsegments ) ;
    status = PIOUS_close( outfd ) ;
  }
  numbt = SpawnWorkers( argv[0], &childTID, nchild ) ;    /* Spawn the workers */
  /* Broadcast to all child processes a startup message */
```

```
bufid = pvm_initsend( PvmDataRaw );                                 /* A buffer please */
status = pvm_pkstr( argv[1] ) ;                                 /* PIOUS A matrix FILE */
status = pvm_pkstr( argv[2] ) ;                                 /* PIOUS B matrix FILE */
status = pvm_pkstr( argv[3] ) ;                                 /* PIOUS C matrix FILE */
status = pvm_pkint( &nelemems, 1, 1 ) ;                         /* The elements per row */
status = pvm_pkint( &rowsPerBlock, 1, 1 ) ;                      /* The rows per block */
status = pvm_pkint( &nsegments, 1, 1 ) ;                            /* Total segments */
status = pvm_mcast( childTID, nchild, MYMESSAGE ) ;            /* Send to all children */
status = pvm_freebuf( bufid );                                             /* Clean up */
/*
 * There are nsegments of work to be done.
 * There are nchild processes to do the work
 * Each child process asks for work, gets segment to work on, does work
 *  until there is no more in segment then repeats the process
 * Each request for work (nsegments) each child process told to die (nchild)
 * Total: nsegments + nchild messages out
 */
for ( i = 0 ; i < nsegments; i++ ) {
  bufid = pvm_recv( -1, MOREWORK ) ;                              /* Request for work */
  status = pvm_bufinfo( bufid, &length, &msgTag, &tid ) ;           /* Who from ? */
  status = pvm_freebuf( bufid ) ;                                          /* Clean up */
  status = pvm_initsend( PvmDataRaw ) ;                           /* New buffer please */
  status = pvm_pkint( &i, 1, 1 ) ;                                  /* Remaining work */
  status = pvm_send( tid, MOREWORK ) ;              /* Tell child segment to work on */
  status = pvm_freebuf( bufid ) ;                                          /* Clean up */
}
k = -1;                                   /* No more work to be done tell all children */
for ( i = 0 ; i < nchild; i++ ) {
  bufid = pvm_recv( -1, MOREWORK ) ;                              /* Request for work */
  status = pvm_bufinfo( bufid, &length, &msgTag, &tid ) ;           /* Who from ? */
  status = pvm_freebuf( bufid ) ;                                          /* Clean up */
  status = pvm_initsend( PvmDataRaw ) ;                           /* New buffer please */
  status = pvm_pkint( &k, 1, 1 ) ;                                    /* No more work */
  status = pvm_send( tid, MOREWORK ) ;                               /* Tell child that */
  status = pvm_freebuf( bufid ) ;                                          /* Clean up */
}
for ( i = 0 ; i < nchild; i++ ) {  /* Receive the die message from each child */
  bufid = pvm_recv( -1, MYMESSAGE ) ;                                /* From anybody */
  status = pvm_freebuf( bufid ) ;                                          /* Clean up */
}
free( childTID );                                          /* Free the children handles */
/* Open the output file in global mode and reread it for export */
outfd = pious_popen( GROUP, argv[3], pious_GLOBAL, IOBUFFERSIZE,
                     PIOUS_VOLATILE, PIOUS_RDONLY, REGMODE, nsegments ) ;
if ( outfd >= 0 ) {                                        /* If the output file exists */
  fp = fopen( argv[3], "w+");                                  /* Open the UNIX file */
  while (1) {                                                           /* Until done */
    status = pious_read( outfd, iobuffer, IOBUFFERSIZE ) ;            /* Read a block */
    if ( status == 0 ) break ;                                            /* All done */
    status = fwrite( iobuffer, sizeof(char), IOBUFFERSIZE, fp ) ; /* Write it */
  }
  status = pious_close( outfd ) ;                              /* Close the output file */
  fclose( fp ) ;                                                 /* Close the UNIX file */
  status = pious_unlink( argv[3] );   /* Remove the C matrix file from PIOUS */
}
status = pious_unlink( argv[2] );      /* Remove the B matrix file from PIOUS */
status = pious_unlink( argv[1] );      /* Remove the A matrix file from PIOUS */
pvm_exit() ;                                                  /* Gracefully leave PVM */
```

```
  } else {                                                   /* I'm a child process */
    /* CHILD process -- wait for file names in order to open */
    bufid = pvm_recv( myParentTID, MYMESSAGE ) ; /* Get a message from my parent */
    status = pvm_upkstr( ainFile ) ;                    /* The A matrix file name */
    status = pvm_upkstr( binFile ) ;                    /* The B matrix file name */
    status = pvm_upkstr( coutFile ) ;                   /* The C matrix file name */
    status = pvm_upkint( &nelemems, 1, 1 ) ;              /* The elements per row */
    status = pvm_upkint( &rowsPerBlock, 1, 1 ) ;          /* The rows per segment*/
    status = pvm_upkint( &nsegments, 1, 1 ) ;               /* The total segments */
    IOBUFFERSIZE = nelemems * rowsPerBlock * sizeof(double) ;
    dscnt = pious_sysinfo(PIOUS_DS_DFLT) ;
    binfd = pious_popen(  GROUP, binFile, PIOUS_INDEPENDENT, IOBUFFERSIZE,
                          PIOUS_VOLATILE, PIOUS_RDONLY , REGMODE, dscnt ) ;
    while ( 1 ) {
      bufid = pvm_initsend( PvmDataRaw ) ;                        /* Buffer please */
      status = pvm_send( myParentTID, MOREWORK ) ;   /* Ask parent for some work */
      status = pvm_freebuf( bufid ) ;                                  /* Clean up */
      bufid = pvm_recv( myParentTID, MOREWORK ) ;                 /* Get some work */
      status = pvm_upkint( &segment, 1, 1 ) ;        /* Which segment to work on */
      status = pvm_freebuf( bufid ) ;                                  /* Clean up */
      if ( segment == -1 ) break                                       /* All done */
      ainfd = pious_popen( GROUP, ainFile, PIOUS_SEGMENTED, segment,
                           PIOUS_VOLATILE, PIOUS_RDONLY, REGMODE, nsegments ) ;
      coutfd = pious_popen( GROUP, coutFile, PIOUS_SEGMENTED, segment,
                            PIOUS_VOLATILE, PIOUS_WRONLY, REGMODE, nsegments ) ;
      while (1) {                                               /* Read until done */
        status = Child( ainfd, binfd, coutfd, nelemems, rowsPerBlock ) ;
        if ( status < 1 ) break ;
      }
      status = pious_close( ainfd ) ;          /* Close the A segmented input file */
      status = pious_close( outfd ) ;         /* Close the C segmented output file */
    }
    status = pious_close( binfd ) ;         /* Close the independent B input file */
    bufid = pvm_initsend( PvmDataRaw );                         /* A buffer please */
    status = pvm_send( myParentTID, MYMESSAGE ) ;                /* Tell my parent */
    pvm_exit() ;                                             /* Gracefully exit PVM */
  }                                              /* End of if child or parent process */
  exit(0) ;                                               /* End of matrix multiply */
}
```

B.2.2 Source code for Child.c

```
int Child( int fa, int fb, int fc, int nelems, int nblocks )
{
      double *A, *B, *C ;                         /* Pointer to the three matrices */
      int k, n, j, status ;
      /* Allocate space for each block of A, B, and C */
      A = (double *)malloc( nblocks * sizeof( double ) * nelems ) ;
      B = (double *)malloc( nblocks * sizeof( double ) * nelems ) ;
      C = (double *)malloc( nblocks * sizeof( double ) * nelems ) ;
      /* Read in the block of A for this call to Child */
      status = pious_read( fa, A, sizeof( double ) * nelems * nblocks ) ;
      if ( status < 1 ) return status;                       /* End of file or problems*/
      k = 0 ;
      while ( 1 ) {
        /* Read in all of B, one block at a time, until the read fails */
        status = pious_read( fb, B, sizeof( double ) * nelems * nblocks ) ;
        if ( status < 0 )                                          /* Problems abort */
          return status ;
```

```
      else if ( status == 0 )                              /* All done reading B */
        break ;
      for ( n = 0; n < nblocks ; n++ ) {                  /* Compute the C matrix */
        for ( j = 0 ; j < nblocks; j++ ) {
          C[ n*nelems+k+j ] = DotProduct( &A[i*nelems], &B[j*nelems], nelems ) ;
        }
      }
      k += nblocks ;
    }
    /* Write out the completed block of C */
    status = pious_write( fc, C, sizeof( double ) * nelems * nblocks ) ;
    free( A ) ;                                      /* Free up the allocated memory */
    free( B ) ;
    free( C ) ;
    return status ;
}
```