Appendix B: Intermediate Parallel Language

Overview

This appendix describes the Intermediate Parallel Language (IPL) produced by the Apply compiler to parallelize programs. IPL consists of the standard C language plus a few added parallel constructs. These parallel constructs are the means by which Apply's implicit parallelism is made explicit. IPL is designed to help programmers make full use of the iWarp system's distributed memory architecture. This architecture requires that specified parts of a program's data reside in the memories of the appropriate processors when parallel execution begins. Many details are involved in this process, and many of these details are crucial to avoiding deadlock. IPL handles these distributed memory details, freeing the programmer to develop the application.

IPL makes it possible to write high-performance parallel programs for many applications without having to manage the following details:

- mapping iWarp executables onto processors in the array
- routing data traffic between iWarp processors
- partitioning data arrays
- distributing computation over a set of iWarp processors
- avoiding deadlock
- generating execution tracefile data

In many applications, the IPL code need not be manipulated. In other applications you may want to modify the IPL code to add things like diagnostic print statements. It is also possible to write parallel programs directly in IPL to further customize applications, although there are some pitfalls associated with writing IPL programs from scratch.

IPL Extensions

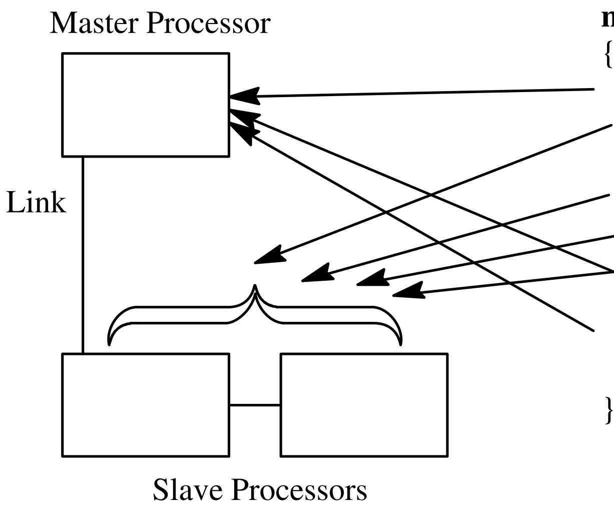
The IPL extensions to the standard C language are in the following three areas:

- distributed data allocation
- data movement
- parallel loops

IPL allocates data across the iWarp system's distributed memory by means of the **distributed** array construct. This construct allows you to manage storage for parallel programs in much the same way as for a conventional C program. The advantage is that although each participating processor is responsible for part of the array , you need not be concerned with the fraction of the **distributed** array actually assigned to each processor. There is also a **local** storage construct that allows each processor to have its own copy of an entire object. This is useful for things like tables of read-only data.

IPL is designed so a program will execute as if it were an ordinary sequential C program except within the body of **parallel for** loops, where computation proceeds in parallel. The compiler maps the iterations of these loops onto a group of processors, with all processors working on different parts of the iteration space concurrently . The compiler can do this for any size loop and for any number of iWarp cells. The number of loop iterations assigned to each processor depends on the number of processors used and the total number of loop iterations. The compiler maps the loop iterations onto processors in a way that is compatible with its mapping of **distributed** arrays.

Conventional C constructs are executed as part of the sequential computation, which is performed on a master iWarp processor. Parallel computation is performed on one or more slave processors at the master's direction. Each of the slave processors executes the same program, but works on different data and on different loop iterations. The compiler ensures that these two computational activities function together properly. The relation between master/slave processors and sequential/parallel code segments is illustrated in Figure B-1. The ten element array , x, is part of the sequential code on the master processor, while the distributed array , dx, is part of the parallel code on the set of slave processors. The **parallel for** is performed on the slave processors while **copy** is a cooperative enterprise that involves both master and slave processors.




```
main()
{
  int x[10];                          /* sequential */
 distributed int dx[10];             /* parallel   */

 parallel for(i=0; i<10; i++) /* parallel     */
    dx[i] = i;
  copy (dx, x, 2, 10,1,0,10,0,1);

 for (i=0; i<10; i++)                /* sequential */
    printf("x[5] = %d\n", x[5]);
}
```

Figure B-1: Sequential and parallel code executed on master and slave processors

Distributed Data Allocation: distributed, local

Arrays that are to be distributed across the slave processors are identified by the keyword **distributed** . Arrays without this keyword are allocated entirely on the master processor.

Distributed arrays may be one- or two-dimensional arrays of type char, int, float, or double. Distributed arrays of type char or int may be further qualified as signed or unsigned.

An optional three-element parenthesized list may appear after the array identifier and before the bounds specifier(s). The first element indicates how much adjacent partitions must overlap. The second and third elements constrain partition sizes when Apply's sampling features are used. When the list is absent the compiler assumes there is no overlap and the sample parameters are both equal to 1.

The following example declaration allocates space on the slave processors for a 100x100 array of double precision numbers. Each partition would have an extra row at the top and bottom to allow for a 3 x N computation. Partitions are constrained to a number of rows that is divisible evenly by 2.

```
distributed double x (3,2,2)[100][100];
```

The array is partitioned by rows in a block oriented fashion onto four slave processors as follows:

rows 0–24	slave processor 0
rows 25–49	slave processor 1
rows 50–74	slave processor 2
rows 75 –99	slave processor 3

The size of the array is specified by the programmer whereas the number of processors is specified as a compiler command line parameter. The IPL compiler resolves cases where the size of the array does not map onto the processor evenly.

Distributed array declarations can only appear at the beginning of the outer–most compound statement in a function body. Distributed arrays cannot be static.

Local data is declared by preceding the array declaration by the keyword **local** . An entire copy of the local data item is allocated on each slave processor. Local data is useful for tables of constants and for temporaries. Local data may be scalars of any type or arrays having any number of dimensions.

Data Movement: copy

Once space for data arrays is allocated, it is necessary to move data between master and slave processors as called for by the application. Apply procedures can move image data into one or more distributed arrays and broadcast constant data into local arrays, prior to beginning the parallel computation. Results are returned to the master processor just prior to the function return.

Data is moved between sequential and parallel parts of the program by a special **copy** construct. There are three forms of data movement depending on the kind of arrays involved:

- moving data from an ordinary array to a distributed array is referred to as a scatter operation
- moving data from a distributed array to an ordinary array is referred to as a gather operation
- moving data from an ordinary array to a local array is referred to as a broadcast operation

All three forms require that source and destination arrays have the same declared size. The following sections describe the syntax for each form of the **copy** construct.

NOTE

Copy operators may appear only between **parallel for** loops, not inside them.

Scatter Form

```
copy(ordinary_array, distributed_array, border, fill,
      up, down, left, right);
```

The qualifiers are defined as follows:

ordinary_array	name of a 1 or 2 dimensional array
distributed_array	name of a 1 or 2 dimensional distributed array
border	border specifier; one of: 0 = fill border with value of *fill* 1 = replicate edge of image into border 2 = mirror the edge value into border 3 = do nothing to border
fill	value to be used with border type = 0. This parameter must be of type double.
up	upper bound of array specification for local operator computation.
down	lower bound of array specification
right	right–most bound of array specification
left	left–most bound of array specification

NOTE

border, fill, up, down, right, and *left* must be integer constants or variables. They can not be expressions.

Gather Form

```
copy(distributed_array, ordinary_array, n_rows, n_cols,
      row_start, row_count, col_start, col_count);
```

The qualifiers are defined as follows:

distributed_array	name of a 1 or 2 dimensional distributed array
ordinary_array	name of a 1 or 2 dimensional array
n_rows	declared height of destination array
n_cols	declared width of destination array
row_start	first row to be transferred
row_count	number of rows to be transferred
col_start	first column to be transferred
col_count	number of columns to be transferred

NOTE

n_rows, n_cols, row_start, row_count, col_start, and *col_count* must be integer constants or variables. They can not be expressions.

Broadcast Form

```
copy(ordinary_array, local_array, n_rows, n_cols);
```

The qualifiers are defined as follows:

ordinary_array	name of ordinary array *(any number of dimensions)*
l*ocal_array*	name of local array
n_rows	height of section being transferred
n_columns	width of section being transferred

NOTE

n_rows and *n_cols* must be integer constants or variables. They can not be expressions.

There is a shortened broadcast form that is useful when individual scalars are to be broadcast to local variables. This is done by means of a simple assignment statement that has a local variable as the left hand side. Such scalar broadcast assignment statements can appear anywhere a **copy** operator can appear. The following is an example of the shortened broadcast form:

```
main()
{
   int x; = 5;
   local lx;
   lx = x;       /* scalar broadcast assignment */
   . . .
}
```

Parallel Loops: parallel for

The preceding two language extensions set the stage for actually performing arithmetic in parallel on an array of iWarp cells using the **parallel for** construct. This construct schedules loop iterations onto the slave array in a way that is closely matched to the preceding extensions. There are several restrictions on the use of parallel loops.

1. The **parallel for** statement must be of the form:

   ```
   parallel for (i=n; i<m; i++)
   ```

   where:

   i is a **local** scalar
   n is an integer constant or variable
   m is an integer constant or variable
   the induction variable, *i*, must be incremented by 1

   No other forms are acceptable.

2. The body of a **parallel for** statement is a compound statement that can contain any legal C expression involving distributed array references, local variable references, or math library function calls.

3. Parallel loops cannot be nested. Functions appearing inside a **parallel for** body must not contain **parallel for** constructs.
4. Distributed array references must be subscripted (no pointer references).
5. The first subscript of a distributed array reference must be of the form

   ```
   i*a + b;
   ```

   where:

   i is parallel loop induction variable
   a is the vertical sample factor.
   b is a small integer constant <... –2, –1, 0, 1, 2, ...> associated with overlap between partitions.

   Manually generated IPL code will ordinarily have $a = 1$ and $b = o$.
6. There must be no loop–carried dependence between successive iterations of the loop.

The compiler determines which loop iterations should be assigned to each slave processor and how distributed array subscripts should be remapped to correspond with this assignment.

Language Restrictions

IPL has the following language restrictions:

- IPL compounds must have at least one executable statement
- the `main()` function can not contain parallel constructs
- return statements can appear only at the end of a function body
- calls to exit( ) can only appear at the end of the main function

In addition, the only calls in the main IPL function should be calls to Apply procedures and separately-compiled C libraries. These libraries must be linked in the same way as sequential C libraries, using the –L and –l options as in the following example:

```
apply -L. -lsample  example.app
```

This example assumes that an archive named libsample has been created in the current working directory.

A Program Example — Numerical Integration in parallel

The following example illustrates all of the IPL language extensions with the exception of broadcast operations. This example computes the definite integral from 0 to 1 of the function 1/(1+x*x), which has the numerical value π/4. Rectangular integration is used to approximate the actual integral. Figure B-2 provides an illustration of this process.

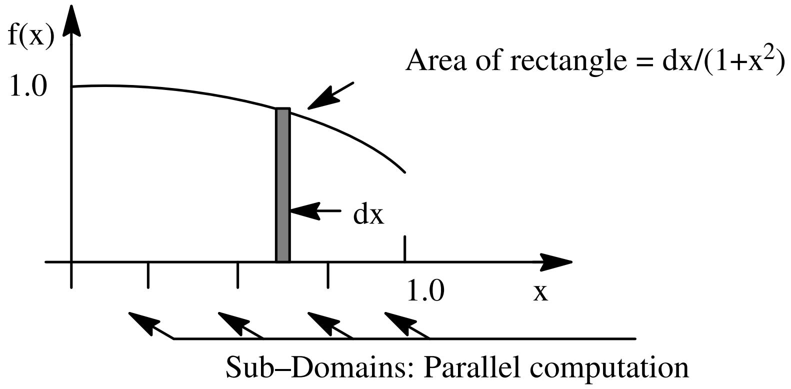


Figure B-2: Computing an integral through rectangular approximation

The parallel computation is organized as follows:

- Rectangular integration is performed in parallel across four sub–domains.
- Four partial sums are copied back to the master processor, where they are merged into a final sum suitable for printing.

In the example, note the use of a local variable for the **parallel for** index and for the first subscript of assignment to elements of the distributed array, dsum **.**

```
#define N 32                             /* number of rectangles          */
#define dx (1.0/N)                       /* width of a rectangle          */
#define M 4                              /* number of domains             */

demo ()                                  /* main program                  */
{
  int i;                                 /* iteration var. for final sum  */
  float pi4 = 0.0;
  float gsum[M];                         /* sums                          */

  local int j, k;                        /* domain & iteration variables  */
  local float x;                         /* integration variable          */
  distributed float dsum[M];             /* dist array for partial sums   */

  parallel for(j=0; j<M; j++)            /* integrate concurrent domains  */
  {
    dsum[j] = 0.0;                       /* initialize partial sum        */
    for(k = 0; k<N/M; k++)               /* integrate over a sub–domain   */
    {
      x = (N/M*j + k + 0.5) * dx;        /* get x from loop indices       */
      dsum[j] += dx / (1 + x*x);         /* add area of one rectangle     */
    }
  }
  copy( dsum, gsum, M, 1, 0, M, 0, 1 );  /* partial sums -> seq code      */

  for (i=0; i<M; i++)  pi4 += gsum[i];  /* combine partial sums          */
  printf("pi = %f\n", 4.0*pi4);         /* print result                  */
}
```

Computing Model Used by IPL

IPL carries out its communication and control functions over a ring type communication network established at runtime when the master and slave programs are initialized. Routing software supplied by the compiler determines the route from each processor to its successor on the basis of the size of the communication ring and the size of the iWarp processor array.

Figure B-3 illustrates how a 16-element ring is mapped onto a 4x4 iWarp hardware array:

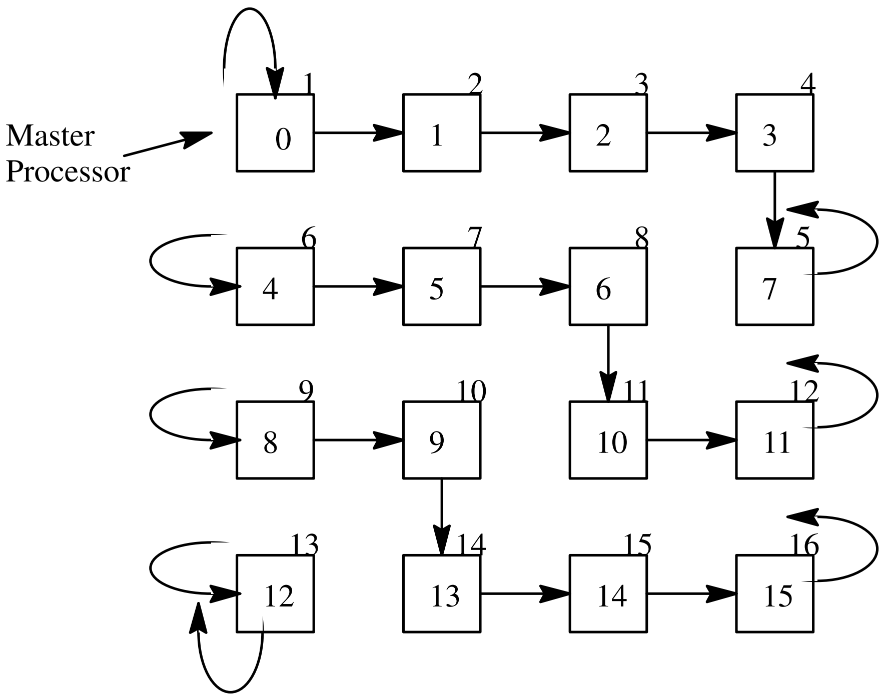


Figure B-3: 16-element ring mapped onto a 4x4 iWarp array

An element's position within the ring is indicated by the number at the upper right of each box, while the processor number assigned by the iWarp runtime software is indicated by the number inside the box. The master and slave programs each contain calls to the compiler support routines, **_ap_setup()** and **_ap_open_apply_chan()**. Together, these routines convert ring element position into processor number, compute the communication route, open the necessary connections, and bind gates to the connections. Refer to the iWarp Macro Specification for further information about connections and gates. Corresponding routines close the connections when the program completes execution.

The master processor uses this network to control operations on the slave processors and to move data between master and slave processors. High performance is achieved by overlapping movement of data from the master processor to the slave processor array with computation on the slave processors and with movement of finished results back to the master processor.

Function Calls Inserted by IPL

The IPL compiler may insert calls to one or more of the following support library functions:

_ap_setup();	Network setup
_ap_open_apply_chan();	Open connections for network
_ap_close_apply_chan();	Close connections for network
_ap_app_debug();	Write tracefile data
_ap_copyin_m();	Sends data to slave processors
_ap_copyin_s();	Receives data into slave processors
_ap_copyout_m();	Receives data from slave processors
_ap_copyout_s();	Sends data to master processor
_ap_copybc_m();	Broadcasts data to slave processors
_ap_copybc_s();	Receives broadcast data

Program Flow of Control

Computationally-expensive operations are done in parallel at the direction of the master processor. The master processor controls program flow by broadcasting a token to all slave processors indicating what they are to do next. The slave processors respond by branching to the appropriate case in a switch statement where the parallel work is performed. When the work is finished, the slave processors process the next token. The final token causes the slave processes to terminate, after which the master terminates.

The following slave processor activities can be initiated in this fashion:

- allocate space for distributed and local variables
- execute a **parallel for** loop
- execute a **copy** operation
- execute a scalar broadcast assignment
- terminate execution

Linking Functions that Contain Parallel Constructs

Functions that contain parallel constructs and the main program that calls them must be processed as a unit by the Apply compiler. All of this software may be contained in a single program file, or a separate file may be used for each function. Each of the separate program files are combined using the C pre–processor's **#include** feature. Once included, these functions can perform parallel operations by means of the control mechanism described in the previous section.

Debugging

The IPL extensions have been designed to look a great deal like corresponding constructs in the standard C language. You can use this similarity in early phases of the application development to test a sequential prototype of the program. In the prototype, **distributed** and **local** variables are replaced by their ordinary equivalents, and **parallel for** loops are replaced by standard for loops. Once you resolve algorithm design and arithmetic correctness, you can develop a parallel version of the program with greater confidence.

The following example shows how the parallel program example shown earlier would be converted to sequential form. The sequential version of the program can be examined using the Sun debug tool, “dbxtool.”

```
#define N 32                            /* number of rectangles         */
#define dx (1.0/N)                      /* width of a rectangle         */
#define M 4                             /* number of domains            */

main ()                                 /* main program                 */
{
  int i;                                /* iteration var. for final sum  */
  float pi4 = 0.0;
  float gsum[M];                        /* sums                         */

  int j, k;                             /* domain & iteration variables  */
  float x;                              /* integration variable         */
  float dsum[M];                        /* dist array for partial sums   */

  for(j=0; j<M; j++)                    /* integrate concurrent domains  */
  {
    dsum[j]= 0.0;                       /* initialize partial sum       */
    for(k = 0; k<N/M; k++)              /* integrate over a sub-domain   */
    {
      x = (N/M*j + k + 0.5) * dx;       /* get x from loop indices      */
      dsum[j] += dx / (1 + x*x);        /* add area of one rectangle    */
    }
  }
  for(i=0; i<M; i++) gsum[i] = dsum[i]; /* sim. part. sums -> seq code */
  copy( dsum, gsum, M, 1, 0, M, 0, 1 );

  for (i=0; i<M; i++)  pi4 += gsum[i];  /* combine partial sums         */
  printf("pi = %f\n", 4.0*pi4);         /* print result                 */
}
```

Print Statements

You can use diagnostic **printf()** statements anywhere in an IPL program, including inside the body of **parallel for** loops. However, text from print statements executed in parallel may not appear in the usual sequential order. If sequential order is important, you can label the print traffic with the current value of the loop iteration variable and sort the print traffic afterwards.

Tracefile Information

You can direct the IPL compiler to generate a trace of control traffic corresponding to operations called for by the master processor and operations undertaken by the slave processors in response. The tracefile feature is intended for those who are interested in the details of how the Apply compiler manages parallel computation. As such, it may be of limited value to many programmers.

To use the tracefile feature, it is necessary to compile the program using the –g option and to run the program using either the –a or the –A option. Refer to Chapter 3 for descriptions of these command options.

Examining Output Text

A third means for identifying problems is to examine the master and slave programs produced by the Apply compiler. This requires a certain amount of familiarity with the master/slave method of program control described previously. Like the tracefile mechanism, this technique is also designed for those who are interested in the details of how parallel computation is managed. It is necessary to use the –k compiler option to retain these program files after compilation.