Fx User's Guide

The Fx Compiler Group

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

/afs/cs/project/iwarp/member/fx/doc/userguide/userguide.ps

September 9, 1994

1 Introduction

Fx is a prototype parallelizing Fortran compiler developed at Carnegie Mellon [1, 2, 3, 5]. The main idea behind the compiler is to integrate task and data parallelism within a uniform Fortran framework. Input is a Fortran 77 program augmented with HPF-like data layout directives, Fortran 90 array statements, a parallel loop construct, and directives for task parallelism. Output is a Fortran 77 program sprinkled with calls to communication routines.

2 Programming model

The target machine is a collection of nodes, $0, 1, ..., P-1$. Each node has its own local memory. An Fx program begins life as a single data parallel task running on all nodes. When the program reaches a *parallel section*, the nodes are partitioned among a collection of data parallel tasks. When all of these tasks terminate, the program resumes as a single data parallel task until the next parallel section is reached.

Data parallelism comes in the form of array assignment statements and parallel loops operating on scalars and arrays. At the discretion of the programmer, an array is either *replicated* on each node, or *distributed* (i.e. partitioned) across the nodes. Scalars are always replicated.

Task parallelism is expressed in special regions called *parallel sections*. Each procedure call in a parallel section identifies a *task*, which is a single thread of control. Tasks communicate with other tasks only through their input and output parameters. The communication is completely determined by the way the data would flow if the program was executed sequentially. A task waits for its inputs, executes, produces its outputs, and then terminates. Each task runs on one or more nodes.

3 Fortran 77 base language

Fx implements a subset of Fortran 77, including DO loops, conditionals, procedure calls, functions, single and double precision real, integer, and complex values, as well as the usual F77 intrinsics. The following are not implemented:

- EQUIVALENCE

4 Data layout directives

The programmer uses data layout directives to tell the compiler how to distribute arrays.

```
template <template_name>(<size>,<size>,...)
align <array>(<index>,...) with <template_name>(<index>,...)
distribute <template_name>(<distribution>,...)
```

The `template` directive defines an index space. The `align` directive associates elements of an array with elements of a template. The `distribute` directive assigns template elements in dimension *k* according to the *kth* `<distribution>`, which is one of

```
block_cyclic(b,p)
block(p)
cyclic(p)
```

where b is a block size and p is the number of processors over which to distribute the data. In the most general block_cyclic case, the array dimension is divided into chunks of size b, which are then dealt out cyclically to nodes $0, \ldots, p-1$. The block and cyclic cases are simply abbreviations for common values of b. For the block distribution, $b = \lceil n/p \rceil$, where n is the size of the array dimension. For the cyclic distribution, $b = 1$.

Example:

```
real a(n,n), b(n,n)
template t(n)
align a(i,j), b(j,i) with t(i)
distribute t(block(64))
```

This example distributes blocks of $\lceil n/p \rceil$ rows of a and blocks of $\lceil n/p \rceil$ columns of b across 64 nodes. For example, node 0 gets the first $\lceil n/p \rceil$ rows of a, node 1 gets the next $\lceil n/p \rceil$ rows of a, and so on.

Example:

```
real a(n,n)
template t(n,n)
align a(i,j) with t(i,j)
distribute t(block_cyclic(16,8),cyclic(8))
```

This example cyclically distributes blocks of size 16 in one dimension and blocks of size 1 in the other dimension over 64 nodes, 8 nodes per dimension.

5 Array assignment statements

Fx supports element-wise operations on whole arrays and on sections of arrays. Array sections are specified using Fortran 90 triplet notation:

```
<first>:<last>:<stride>
```

Each element in the triplet is optional. The defaults are first = 1, last = n, and stride = 1, where n is the size of the corresponding array dimension.

Examples:

```
a = b + c
a(:,j) = b(i,:)
a(:,2:n) = b(:,1:n-1)
a(:,1:n:2) = b(:,2:n:2)
```

The first statement adds two entire arrays element-wise and assigns the result to a third array. The second statement assigns the *ith* row of b to the *jth* column of a. The third statement assigns columns $1..n-1$ of b to columns $2..n$ of a. The last statement assigns the even columns of the b to the odd columns of a.

5.1 idxperm

One can transpose the dimensions of an array with the idxperm primitive:

```
idxperm(<array>,<dimension>,<dimension>,<dimension>,....)
```

idxperm (index permutation) takes an array name followed by the transpose dimension numbers. The transpose exists for the duration of the statement after which the array dimensions will resume initial state.

Example:

```
real a(n,n), b(n,n)

a = idxperm(b,2,1)
```

The above idxperm statement is equivalent to the following fortran loops

```
do i=1,n,1
   do j=1,n,1
      a(i,j) = b(j,i)
   enddo
enddo
```

The nesting order of the equivalent fortran loops is determined by the LHS dimension order of the idxperm assignment statement. Thus the loop nesting order will be reversed if the following statement is used instead.

```
idxperm(a,2,1) = b
```

The latter offers a performance advantage in cases where the array is distributed in the 2nd (column) dimension.

Example:

```
real a(n,n,n), b(n,n,n)
template t(n)
align a(i,j,k) with t(k)
align b(i,j,k) with t(k)
distribute t(block(64))

a = idxperm(b,1,3,2)
```

Since the array is distributed in the 3rd dimension, improved performance is achieved when the equivalent fortran outermost loop is the distributed dimension. This is accomplished by implementing an idxperm on the LHS with the 1st dimension of the transpose being the distributed dimension. Inorder to preserve correctness, the same transpose must also be performed on the RHS, resulting in nested idxperm statements shown below.

```
idxperm(a,3,1,2) = idxperm(idxperm(b,1,3,2),3,1,2)
```

6 Parallel loops

Fx uses a new parallel loop construct called the PDO [5]. The basic idea, from ADAPT [4], is to combine a user-defined merge operation with a parallel loop. As described in [5], the PDO can be used to implement a conventional FORALL-style parallel loop, an arbitrary user-defined parallel reduction operator, or a combination of the two.

The PDO uses copy-in/copy-out semantics. Conceptually, the entire global data space is copied into the local data space for each iteration. Then each iteration runs independently and in parallel. After all iterations have completed, variables that were modified are copied out into the global data space. Any variable modified by more than one loop iteration is merged using an arbitrary associative user-defined merge operation.

```
PDO ( <loop range> [, <loop range>] ... )
<declaration section>
PINIT
  <init section>
PBODY
  <body section>
PMERGE
  <merge section>
ENDPDO
```

Example: image histogram

```
real image(n,n), result(m)
template t(n)
align image(i,j) with t(i)
distribute t(block(64))

pdo (i=1:n)
pin image(i,:)
pmvars result
pinit
  result = 0
pbody
  do j = 1,n
    result(image(i,j)) = result(image(i,j)) + 1
  enddo
pmerge
  result = left(result) + right(result)
endpdo
```

Notes

1. `<loop range>`: Currently the PDO does not support non-unit strides.
2. `<declaration section>`: This section consists of any combination of the following:

   `PINOUT <list>`**:** variables to be copied in and out.

   `PIN <list>`**:** variables to be copied in only.

   `POUT <list>`**:** variables to be copied out only.

   `PMVARS <list>`**:** variables to be merged.

   `PCOMMUTATIVE`**:** asserts that the merge operator is commutative.

   The declarations can appear in any order. There can be multiple PINOUT, PIN, POUT, or PMVARS lists. If an array appears in both a PIN and a POUT list, it is assumed to be PINOUT.

3. `PINIT`: This denotes the init section, which initializes each merge variable to the identity of the associative merge operator. For example, if the merge operator is addition, then the merge variables are initialized to zero.

4. `PBODY`: This denotes the main body of the loop.

5. `PMERGE`: This denotes the merge section, which defines a binary associative merge tree. The parent is denoted by `mergeVar`. The left and right children are denoted by `LEFT(mergeVar)` and `RIGHT(mergeVar)`, respectively.

6. Miscellaneous: If there is no merge step, then the `PMVARS <list>`, the `<init section>`, and the `<merge section>` should be empty. For the one dimensional `PDO`, the loop header can also have the usual DO loop form:

```
PDO <induction variable> = <initial value>, <final value>
```

7 Task parallelism

Task parallelism is expressed inside a *parallel section*.

```
ENTER TASKING
  loop control statements and
  parallel subroutine specifications
EXIT TASKING
```

Each parallel subroutine specification has the following form:

```
CALL <subroutine>
<input spec>
<output spec>
<processor spec>
<origin spec>
```

where

```
<input spec>     =  INPUT (<varlist>)
<output spec>    =  OUTPUT (<varlist>)
<processor spec> = PROCESSOR <pair>
<origin spec>    = ORIGIN <pairlist>

<varlist>   = <var>,[<var>]*
<pairlist> = <pair>, [<pair>]*
<pair>     = (<value>, <value>)

<var>   = Name of a scalar or array variable
<value> = A non-negative integer
```

Example: 2D FFT

```
call gen_bit_reverse_table(brt)
call gen_W_table(W)
call timer_start()
enter tasking
do k=1,iters
  call cffts(a,brt,W)
  output (a)
  processor (8,4)  ! cffts() runs on an 8x4 subarray
  origin (0,0)     ! with the top left corner at cell (0,0)

  call rffts(a,brt,W)
  input (a)
  output (a)
  processor (8,4)  ! rffts() runs on an 8x4 subarray
  origin (0,4)     ! with top left corner at cell (0,4)
enddo
exit tasking
```

Also all subroutines that are called by a task-subroutine must contain a processor statement describing the shape of the processor that they will execute on, as the first statement inside the subroutine.

Notes

1. Parallel sections: A parallel section consists of a set of subroutine instances or tasks, each of which can potentially run in parallel with other tasks, subject to dependence constraints due to input and output specifications. Only one level of loop nesting is allowed inside a parallel section, and all loops must have constant bounds.

2. `INPUT`/`OUTPUT`: These describe the parameters that a subroutine can access for reading and writing respectively. The parameters can be arrays or scalars in the procedure call parameter list. (Some forms of array sections are also supported as parameters, but we omit the discussion here.)

3. `PROCESSOR`/`ORIGIN`: These specify the number of processors to be allocated to a parallel subroutine and its location in the processor array. Note that multiple origins can be specified, in which case multiple tasks due to a subroutine will execute at different locations in the array. Two tasks must either be mapped to exactly the same set of processors, or to disjoint sets of processors.

Restrictions

1. Our model is that a "main" program runs on all processors, until a parallel section is reached, at which time the processor array is divided among tasks. In the current implementation, there are two important restrictions to this, which we plan to remove with further development):

   - The main program can operate only on replicated objects and NOT on distributed objects.
   - Distributed objects that are declared in the main program (including all objects in input and output list of parallel procedures) must be declared in such a way that sufficient memory is allocated for operations inside tasks. For instance, if an array is distributed in blocked fashion, it must be distributed in the main program as blocked with a parameter reflecting a number of processors less than or equal to the task with the least number of processors.

2. Arrays that are parameters to tasks are allowed to have only one distributed dimension.

8 Subroutines and parameters

8.1 Distributed array parameters

Fx supports Fortran 77 subroutines and functions. Only scalars and whole arrays parameters are supported. Array slices are not supported. Passing distributed arrays can be done using four different models. Ordered in increasing safety and decreasing performance, they are:

- Programmer Redistribute Before Call
- Programmer Redistribute Before Call With Check
- Callee Redistribute
- Inherit

By default, each distributed array is passed using the *Programmer Redistribute Before Call With Check* model. In this model, before calling the subroutine the programmer must ensure that the distribution of the array matches what is expected by the subroutine. On entry, the distribution is checked and the program will terminate if it does not match. The syntax of declaring such a subroutine is a combination of the Fortran subroutine syntax and Fx's data layout directives:

```
subroutine colfft(a,n)
real a(n,n)
integer n

template t(n)
align a(i,j) with t(j)
distribute a(block(64))
```

The *Programmer Redistribute Before Call* model has the same semantics, but distribution checking is avoided for each distributed array in the `nocheck` list. For example, adding the line

```
nocheck a
```

after the `distribute` line in the above example would eliminate the run time distribution check of `a` and result in slightly faster (but less safe) code.

In the *Callee Redistribute* model, the subroutine not only checks the distribution of distributed arrays, but redistributes them if necessary. To indicate that an array parameter should be handled in this fashion, the `conform` directive is used:

```
conform a
```

By default, an array on the `conform` list is copied in at the beginning of the subroutine and copied out at the end. This means that a copy of the array is made, distributed as specified in the subroutine, manipulated by the subroutine, and copied back to the array of the caller. Obviously, not all conformed arrays need to be copied both in and out, so `copyin` and `copyout` directives are available. The following example is the header of a subroutine that copies columns of `a` into rows of `b`:

```
subroutine col2row(a,b,n)
real a(n,n), b(n,n)
integer n

template t(n)
align a(i,j), b(j,i) with t(j)
conform a, b
copyin a
copyout b
```

In the *inherit* model, distributed array parameters inherit whatever runtime distribution they have in the caller. To indicate that an array parameter should inherit distribution from the caller, the `inherit` directive is used:

```
inherit a
```

Note that it is incorrect to use the `align` directive on an inherited array. Finally, inherited arrays can make for extreme amounts of communication, so they should be used sparingly.

8.2 Foreign subroutines and functions

This section is very preliminary

It is possible to call subroutines and functions that are not written in Fx, but some caveats should be kept in mind. First, Fx will generally add distribution descriptor arguments for each subroutine argument to the call if even one of the arguments is distributed. A foreign subroutine should look at the global variables `WasArgListDoubled` and `PosOf1stDistArg` to determine how to handle incoming arguments. Finally, to be callable in a `pdo`, a foreign function should be serial since nested parallelism is not permitted in a `pdo`. The following is legitimate if `fft` is a foreign, serial subroutine:

```
pdo i=1,n
pin image(:,i)
pout colfftresult(:,i)
pbody
  call fft(image(:,i),colfftresult(:,i),n)
endpdo
```

9 Input/Output

9.1 Standard Fortran I/O

Fx supports the input/output commands of Fortran 77. For example,

```
    read (5,100) x, y
    write (6,200) x*y
    print *,"x times y is ", x*y

100 format(I2,",",I2)
200 format(I3, "is result")
```

will read x and y from unit id five - the stdin file associated with the processor, assuming x and y are separated by a comma and each take up two columns. It then writes their product to unit id six, the stdout file associated with the processor, formatted into three columns followed by the "is a result" string. The `print` statement prints to the stdout file with no (*) formatting. "*" can also be used as a unit id. Reading from * is the same as reading from reading from the stdin file. For a more detailed description of Fortran 77 I/O see any book on general Fortran programming.

Distributed arrays cannot be printed using standard Fortran I/O - it is necessary to copy them into a replicated array and then print that array. Standard Fortran I/O also cannot be used to read into a distributed array. Printing distributed arrays is a feature that will soon be widely available.

9.2 Yuppy I/O library

The Yuppy I/O Library is basically a clone of the C stdio library with extensions for reading and writing distributed arrays. The only major limitation is that arrays may only be distributed along a single dimension and the size of the distributed dimension must evenly divide (number of participating processors * block size).

For more information on the Yuppy I/O library, consult:

```
/afs/cs/project/iwarp/member/fx/doc/io/yuppy-io.doc
```

and

```
/afs/cs/project/iwarp/member/fx/examples/io/io.fx
```

10 Examples

10.1 Hello, world

Get this running and you are home free.

```
program main
print *,"hello, world"
end
```

10.2 Successive Over Relaxation (SOR)

Notice that PDO loop iterations can work on overlapping input data:

```
pdo x = 2, nx-1
pin A(x-1:x+1,:)
pout A(x,:)
  A(x,2:ny-1) = c1*A(x-1, 2:ny-1) + c2 * A(x+1, 2:ny-1) +
$ c3*A(x,1:ny-2) + c4*A(x, 3:ny) + c5*A(x,2:ny-1)
endpdo
```

10.3 Matvec

This is an example of using the PDO to compute the matrix-vector product $y = y + Ax$.

```
y = 0
pdo j=1,n
pin a(:,j),x(j)
pmvars result
pinit
  result = 0.0
pbody
  result = result + a(:,j)*x(j)
pmerge
  result = left(result) + right(result)
endpdo
y = y + result
```

10.4 Dynamically redistributing a matrix

Often we want to start with an array that is distributed by rows, perform independent row-wise operations on the array, redistribute it by columns, and then perform independent column-wise operations. One of our favorite techniques in Fx is to perform the redistribution using an array assignment statement. For example, consider the outer loop of a 2D FFT:

```
real a(2,n,n), b(2,n,n)
template t(n)
align a(i,j,k) with t(j)    ! distributed by cols
align b(i,j,k) with t(i)    ! distributed by rows
distribute t(block(64))

call gen_bit_reverse_table(brt)
call gen_W_table(W)
do k=1,iters
  call rffts(b, brt, W)
  a = b
  call cffts(a, brt, W)
enddo
```

10.5 Checking results

The `PDO` is handy for quickly checking results stored in a distributed array:

```
pdo i = 2, nx-1
pin a(i,:)
pmvars errors
pinit
  errors = 0
pbody
  do j = 2, ny-1
    if (a(i,j) .ne. 1.25) errors = errors + 1
```

```
    enddo
  pmerge
    errors = left(errors) + right(errors)
  endpdo
  print *, \"errors=\", errors
```

10.6 Getting performance numbers

This example shows how to instrument your code to get performance numbers:

```
  pclocks = 0 !processing time in clocks
  cclocks = 0 !communication time in clocks

c this section does no communication
  call timer_start()
  call rffts(b, brt, W)
  pclocks = pclocks + timer_stop()

c this section does an all-to-all personalized communication
  call timer_start()
  a = b
  cclocks = cclocks + timer_stop()

c this section does no communication
  call timer_start()
  call cffts(a, brt, W)
  pclocks = pclocks + timer_stop()
```

11 Tricks

11.1 Faking global constants in Fortran

Because Fortran has no global variable or constant space outside of `common` blocks placing "magic numbers" into code is commonplace. While there is a `parameter` keyword, parameters are only local, so the magic numbers simply move the the beginning of subroutine declarations. Because Fx firsts runs code through the C preprocessor, `#define` can be used:

```
  #define NUMPROCS 64

  distribute t(block(NUMPROCS))
```

Note that it is possible for preprocessed code to have lines that extend beyond column 72, when the original code did not. For example,

```
  #define PI 3.14159265359

  t = PI * PI * PI * PI * PI * PI * PI * PI * PI * PI
```

may not behave as expected.

11.2 Dealing with undeclared variables in Fortran

Fortran does not require that variables be declared before using them. This can lead to a variety of hard to find bugs and is generally bad form. Adding the line

```
implicit none
```

to the beginning of your main program and each subroutine or function will cause Fx to print an error message for each undeclared variable used in that module. If you want Fx to complain about using undeclared variables without using `implicit`, use the `-u` option. Note, however, that `-u` may not be portable to other compilers, but `implicit` is.

11.3 Bitwise operators

Fortran 77 does not provide any facility to do bitwise operations. However, `iwf77`, Fx's host compiler provides the following bitwise functions:

```
AND(x,y)
OR(x,y)
XOR(x,y)
NOT(x)
LSHIFT(thingtoshift,howmuch)
RSHIFT(thingtoshift,howmuch)
```

References

[1] STICHNOTH, J. Efficient compilation of array statements for private memory multicomputers. Tech. Rep. CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, Feb. 1993.

[2] STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing 21*, 1 (1994), 150–159.

[3] SUBHLOK, J., STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Exploiting task and data parallelism on a multicomputer. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)* (San Diego, CA, May 1993), pp. 13–22.

[4] WEBB, J. Steps toward architecture independent image processing. *IEEE Computer 25*, 2 (February 1992), 21–31.

[5] YANG, B., WEBB, J., STICHNOTH, J., O'HALLARON, D., AND GROSS, T. Do&Merge: Integrating parallel loops and reductions. In *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing* (Portland, OR, Aug. 1993), vol. 768 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 169–183.

A Using Fx on the CMU iWarps

There are two iWarp systems at CMU for running Fx programs: Victory and Enterprise. Each system is attached to SunOS host workstation by the same name. You can compile and remotely execute Fx programs from any of the iWarp SunOs systems. See

```
/iwarp/machines
```

or equivalently

```
/afs/cs/project/iwarp/machines
```

for a list of these workstations. You must be logged in to a SunOS system, and be authenticated (using kinit) to the appropriate iWarp host. To set up your environment, put

```
setpath -i /iwarp
setpath -i /iwarp/member/fx
```

in your .login file.

```
source /iwarp/member/fx/{Victory,Enterprise}
```

sets up the iWarp environment variables to run one of the iWarps. The Fx driver is in

```
/iwarp/member/fx/bin/fx
```

To compile and run, see

```
/iwarp/member/fx/bin/makefile.iwarp
```

To get a handy Fx editing mode in Gnu Emacs, add this your .emacs file:

```
(setq auto-mode-alist
  (append '(("\\.fx$" . fortran-mode)) auto-mode-alist))
(autoload 'fortran-mode "fx" "Major mode for editing Fx code." t)
(setq load-path (append '("/iwarp/member/fx/misc/lisp") load-path))
```

There are numerous Fx example programs in

```
/iwarp/member/fx/examples
```