An Outline Proposal for the PCRC Library Interface

D.B. Carpenter

Northeast Parallel Architectures Center

Syracuse University

111 College Place

Syracuse NY 13244-4100

Introduction

Will sketch a possible interface to the PCRC run-time library.

Give concrete procedure interfaces in C.

"Concrete syntax" will change later (eg, need systematic naming convention for the routines). For now, important agree on the underlying semantics of the library

The library contains two types of routine.

- "Housekeeping" routines: local procedures for address translation, enumeration of local subranges, etc.
- Collective operations: collective communications, and collective arithmetic operations.

Process coordinates

There are two important run-time operations associated with process arrays (processor arrangements).

1. Determine if *this* processor holds *any* element of of the logical process array, and
2. Determine the position (the *process coordinates*) of this processor within the process array. (Assumes max one logical process/processor)

```
int procs(int rank, int* shape) ;

int on(int procsHandle) ;

void crds(int* coords, int procsHandle) ;
```

`procs` returns an integer handle to a process array. `on` return a non-zero value if local processor holds any element of the process array. `crds` returns coordinates of local process in vector `coord`.

Address translation

Routines define mapping between HPF-program-level global subscripts and pairs of the form

```
(coord, l_i)
```

`l_i` a local index, `coord` is a process coordinate

Primitive conversion routines

```
void globToLoc(int* coord, int* l_i, int g_i,
  int g_lb, int g_ub, int offset, int stride,
  int t_lb, int t_ub, int distCode, int numProcs) ;

void locToGlob(int* g_i, int coord, int l_i,
  int g_lb, int g_ub, int offset, int stride,
  int t_lb, int t_ub, int distCode, int numProcs) ;
```

The procedure `globToLoc` takes `g_i` and returns `*coord` and `*l_i`. `locToGlob` does reverse translation.

More convenient interfaces are

```
void globToLoc(int* coord, int* l_i, int g_i,
               DIM* range) ;

void locToGlob(int* g_i, int coord, int l_i,
               DIM* range) ;
```

Work in terms of the `DIM` structure in the DAD.

The HPF fragment

```
a(i) = 27
```

could be translated (to C) as

```
globToLoc(&coord, &l_i, i, a_DAD.axes) ;

if(coord == locCoord)
  a_LOC [l_i - 1] = 27 ;
```

`locCoord` is the coordinate of the local processor,
`a_LOC` is the C array holding the local segment of `a`.

Parallel iteration

```
void locBlock(int* l_lb, int* l_ub, int* l_stride,
  int g_lb, int g_ub, int offset, int stride,
  int t_lb, int t_ub, int distCode, int numProcs,
  int coord) ;
```

computes the parameters `l_lb`, `l_ub` and `l_stride` of segment of array held on processor with coordinate `coord`. A more convenient interface is

```
void locBlock(int* l_lb, int* l_ub, int* l_stride,
                  DIM* range, int coord) ;
```

BLOCK, numProcs 5, t_lb 1, t_ub 32, g_lb 8, g_ub 15, offset 4, stride 3

coord	l_lb	l_ub	l_stride
0	5	5	3
1	1	7	3
2	3	6	3
3	2	5	3
4	1	-2	3

l_i\coord	0	1	2	3	4
1	-	9	-	-	-
2	-	-	-	14	-
3	-	-	12	-	-
4	-	10	-	-	-
5	8	-	-	15	-
6	-	-	13	-	-
7	-	11	-	-	-

```
CYCLIC, numProcs 6, t_lb 1, t_ub 42, g_lb 1, g_ub 10, offset 40, stride -4
```

coord	l_lb	l_ub	l_stride
0	3	7	2
1	1	0	1
2	2	6	2
3	1	0	1
4	1	7	2
5	1	0	1

l_i\coord	0	1	2	3	4	5
1	-	-	-	-	10	-
2	-	-	9	-	-	-
3	8	-	-	-	7	-
4	-	-	6	-	-	-
5	5	-	-	-	4	-
6	-	-	3	-	-	-
7	2	-	-	-	1	-

`locBlock` used to initialise the `l_lb` and `l_ub` fields in the DAD, or used directly to translate parallel loops. Will use the example

```
FORALL (i = l : u : s)
  a(i) = ...
END FORALL
```

First translation:

```
DIM* j = a_DAD.axes ;
locBlock(&l_lb, &l_ub, &l_stride,
    l, u,
    offset + (l - j->g_lb) * j->stride, s * j->stride,
    j->t_lb, j->t_ub, j->distCode, j->numProcs,
    locCoord) ;
for(l_i = l_lb ; l_i <= l_ub ; l_i += l_stride)
  a_LOC [l_i - 1] = ...
```

Second translation:

```
DIM i ;
rangeSection(&i, a_DAD.axes, l, u, s, locCoord) ;
locBlock(&l_lb, &l_ub, &l_stride, &i, locCoord) ;
for(l_i = l_lb ; l_i <= l_ub ; l_i += l_stride)
  a_LOC [l_i - 1] = ...
```

where introduced new constructor function

```
void rangeSection(DIM* range, DIM* parent,
                  int l, int u, int s, int locCoord) ;
```

Third translation:

```
DIM i ;
rangeSection(&i, a_DAD.axes, l, u, s) ;
for(l_i = i.l_lb ; l_i <= i.l_ub ; l_i += i.l_stride)
  a_LOC [l_i - 1] = ...
```

Depends on to extensions to the DAD (add `locCoord` and `l_stride` fields to `DIM`. Removed redundant last argument of constructor `rangeSection`.)

Assumes constructor intialises `l_i`, etc.

Miscellaneous constructors

Constructors are functions that initialise fields of a data structure, eg `rangeSection`.

Another `DIM` constructor,

```
void range(DIM* range,
  int g_lb, int g_ub, int offset, int stride,
  int t_lb, int t_ub, int distCode,
  int numProcs, int locCoord) ;
```

which initialises all fields of a `DIM` structure. Calls `locBlock` internally to initialise the `l_lb`, `l_lu` fields.

DAD constructors also needed. Eg, copy constructor. Also, section constructor:

```
void arraySection(DAD* array, DAD* parent,
                  int l1, int u1, int s1,
                  int l2, int u2, int s2,
                  ...)
```

New DAD describes the section

```
a (l1 : u1 : s1, l2 : u2 : s2, ...)
```

Collective operations

Three broad categories

- Operations that copy data held in regular sections to other regular sections with different mapping. Eg shifts, transpose, broadcast.
- Gather or scatter operations. Eg array assignments with vector subscript, forall with non-linear subscripts.
- Collective arithmetic operations. Eg reductions, matrix arithmetic.

Regular data remapping

Prototype operation in this category

```
void remap(DAD* dst, DAD* src) ;
```

Copies the values in the array described by `*src` to the array described by `dst`.

Example

```
FORALL (i = l : u : s)
  a(i) = b(i) + ...
END FORALL
```

where `a` and `b` have unrelated mappings.

Translation:

```
arraySection(&bSect_DAD, &b_DAD, l, u, s) ;
arraySection(&tmpSect_DAD, &tmp_DAD, l, u, s) ;

remap(&tmpSect_DAD, &bSect_DAD) ;

DIM* x = tmpSect_DAD.axes ;
for(l_i = x->l_lb ; l_i < x->l_ub ;
                           l_i += x->l_stride)
  a_LOC [l_i] = tmp_LOC [l_i] + ...
```

Compiler sets up `tmp` with same mapping as `a`.

Because no replicated alignments, `remap` does not directly implement broadcast operations.

An array element can be broadcast by

```
void broadcast(void* dst, DAD* src,
               int i1, int i2, ...) ;
```

`i1`, `i2`, ... are global subscripts.

May or may not be necessary to provide lower-level operation which broadcasts arbitrary datum from specified processor.

Streamlined versions of `remap` can be provided for special cases, eg

```
void remapShift(DAD* dst, DAD* src, int dim) ;
```

`dst` and `src` have identical alignments except in dimension `dim`. In this dimension alignments only differ in their offset.

Also provied F90 intrinsics `CSHIFT`, `EOSHIFT`, `TRANSPOSE`, `SPREAD`, etc.

Gather and scatter operations

Implement parallel assignments with irregular patterns of subscripting.

Eg `a` is an `n`-element array, `b` is a rank-2 array:

```
FORALL (i = 1 : n)
  a(i) = b(f(i), g(i))
END FORALL
```

`f` and `g` are arbitrary functions.

Translation:

```
DIM* i = a_DAD.axes ;

/* Compute subscript arrays, `tmp1', `tmp2' */
for(l_i = i->l_lb ; l_i < i->l_ub ; l_i += i->l_stride) {
  int g_i ;
  locToGlob(&g_i, locCoord, l_i, i) ;

  tmp1_LOC [l_i - 1] = f(g_i) ;
  tmp2_LOC [l_i - 1] = g(g_i) ;
}
...
```

Translation (continued):

```
DAD* subs [2] ;
...

/* gather data */
subs [0] = &tmp1_DAD ;
subs [2] = &tmp2_DAD ;
gather(a, b, subs) ;
```

The general `gather` operation has interface

```
void gather(DAD* gath, DAD* scat, DAD* [] subs) ;
```

Subscript arrays have same shape and mapping as *gathered data*. Number of subscripts equal to rank of *scattered data*.

Similarly

```
void scatter(DAD* gath, DAD* scat, DAD* [] subs) ;
```

Collective arithmetic operations

Reduction operations and matrix arithmetic operations (and other operations in the HPF library) can be implemented directly.

Eg, F90 `MATMUL` function supported through the procedure

```
void matmul(DAD* result, DAD* a, DAD* b) ;
```

Mappings of the three arguments unrelated.

Problems with the DAD

- Does not distinguish triplet subscripts of extent one from scalar subscripts.
- Includes non-default lower bounds on global indices. Causes problems at procedure boundaries, where lower bounds change.
- `DIM` does not have a `l_stride` field.
- `DIM` does not have a `locCoord` field.