Compiling High Performance Fortran (HPF)

Material taken from

- "A Linear Algebra Framework for Static HPF Code Distribution", by Ancourt, Irigoin, et al.

Overview of HPF

- Sequential fortran with annotations/directives.
- Supports "data-parallel" style of programming.
- Data-parallel = programmer specifies how the data is to be "parallelized". The compiler determines the rest.

Virtual and physical processors

- Templates - virtual processors

```
!HPF$ template T(0:99), T2(0:99)
```

- Processors

```
!HPF$ processors P(0:4)
```

Alignment

- Mapping array objects onto templates

```
!HPF$ align A(i,j,*) with T(i,j)
!HPF$ align B(i) with T(i,*)
```

- `T(i,*)` – replication.

HPF Rules:

- Each array index can be used at most once in a template subscript expression in any given alignment.
- Each subsript expression cannot contain more than one index.

Distribution

- Mapping virtual processors onto physical processors.

  `!HPF$ distribute T(block(20)), T2(*,cyclic(1)) onto P`

- `block` and `block(B)`
- `cyclic` and `cyclic(B)`

Parallel loops

```
!HPF$ INDEPENDENT(j,i)
do j=1, m
    do i=1, n
        A(i,j) = f(...)
    enddo
enddo

FORALL (i=1:n, j=1:m)
    A(i,j) = f(A, ...)
```

Computation alignment

Data-parallel,

- implicit – let the compiler decide.
- owner-computes rules - owner of lhs performs the computation.

Explicit computational alignment.

```
!HPF$ INDEPENDENT(j,i)
do j=1, m
    do i=1, n
        !HPF$ ON(HOME(A(i,j)))
        A(i,j) = f(...)
    enddo
enddo
```

Modelling directives using linear algebra

The usual suspects:

$$0 \leq a \leq D$$
$$0 \leq i \leq L$$
$$Fi = a$$

Other obvious stuff:

$$0 \leq t \leq T$$
$$0 \leq p \leq P$$

Modelling alignment (cont.)

- alignement w/o replication

$$t = Aa + s_0$$

example:

```
align A(*,i,j) with T(2*j-1,-i+7)
```

$$t_1 = 2a_3 - 1, \quad t_2 = -a_2 + 7$$

Modelling alignment

- alignement w/ replication

$$Rt = Aa + s_0$$

example:

```
align A(i) with T(2*i-1,*)
```

$$t_1 = 2a_1 - 1$$

Modelling distribution

- map from $t \rightarrow < p, c, l >$, where p is the processor number, c is the cycle number, and l is the offset within the block.
- block distribution,

  $$\Pi t = Cp + l$$

  where C is matrix with block sizes along diagonal, and Π is a projection matrix used when a `*` appears in the distribution.
- cyclic distribution,

  $$\Pi t = Pc + p$$

  where P is a matrix with the dimensions of the processor space along the diagonal.

Modelling distribution (cont.)

- block/cyclic distribution

$$\Pi t = CPc + Cp + l$$

example,

```
template T(0:99,0:99,0:99)
processors P(0:9,0:9)
distribute T(*,cyclic(4),block(13)) onto P
```

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} t = \begin{pmatrix} 40 & 0 \\ 0 & 130 \end{pmatrix} c + \begin{pmatrix} 4 & 0 \\ 0 & 10 \end{pmatrix} p + l$$

Code generation

- The alignments,

$$R_{\mathtt{X}} t = A_{\mathtt{X}} a + s_{\mathtt{X}0}$$

$$R_{\mathtt{Y}} t = A_{\mathtt{Y}} a + s_{\mathtt{Y}0}$$

$$R_{\mathtt{Z}} t = A_{\mathtt{Z}} a + s_{\mathtt{Z}0}$$

- The distribution,

$$\Pi t = CPc + Cp + l$$

- The original code:

```
!HPF$ INDEPENDENT(i)
forall (Li ≤ b_0(n))
     X(S_X i + a_X 0(n)) = f(Y(S_Y i + a_Y 0(n)),
                                  Z(S_Z i + a_Z 0(n)),...)
```

Sets for codegen

- Own set - array elements each processor "owns"

$$
\begin{aligned}
Own_{\mathtt{X}}(p) = \{a | \exists t, \exists c, \exists l, s.t. R_{\mathtt{X}} t = A_{\mathtt{X}} a + s_{\mathtt{X}0} \\
\Pi t = C_P c + C_p + l \\
0 \leq a \leq diag(D_{\mathtt{X}}) \\
0 \leq p \leq diag(P) \\
0 \leq l \leq diag(C) \\
0 \leq t \leq diag(T)\}
\end{aligned}
$$

Sets for codegen (cont.)

- Computes set - iterations assigned to each processor
- Owner computes rule -

$$Compute(p) = \{i | S_{\mathtt{X}} i + a_{\mathtt{X}0}(n) \in Own_{\mathtt{X}}(p) \wedge Li \leq b_0(n)\}$$

- View Set - data elements accessors by processor

$$View_{\mathtt{Y}}(p) = \{a | \exists i \in Compute(p) s.t. a = S_{\mathtt{Y}} i + a_{\mathtt{Y}0}(n)\}$$

- Communication Sets

$$Send_{\mathtt{Y}}(p, p') = Own_{\mathtt{Y}}(p) \cap View_{\mathtt{Y}}(p')$$

$$Receive_{\mathtt{Y}}(p, p') = View_{\mathtt{Y}}(p) \cap Own_{\mathtt{Y}}(p')$$

Pseudo-code

```
real X'((c,l) ∈ Own_X(p)),
     Y'((c,l) ∈ Own_Y(p)),
     Z'((c,l) ∈ Own_Z(p))
forall(U ∈ {Y',Z',...})
    forall((p,p'),p ≠ p',Send_U(p,p') ≠ ∅)
        forall((l,c) ∈ Send_U(p,p'))
            send(p',U(l,c))
forall(U ∈ {Y',Z',...})
    forall((p,p'),p ≠ p',Receive_U(p,p') ≠ ∅)
        forall((l,c) ∈ Receive_U(p,p'))
            U(l,c) = receive(p')
if Compute(p) ≠ ∅
    forall((l,c) ∈ Compute(p))
        X(S_X' i+a_X' 0(n)) = f(Y(S_Y' i+a_Y' 0(n)), Z(S_Z' i+a_Z' 0(n)),...)
```

Generating the loops

- Simple method: Fourier-motzkin
- Heuristic: choose l, c loop order. Why?
- More sophisticated methods for arrays.

Blocking communication

```
forall((p,p'), p ≠ p', Send_U(p,p') ≠ ∅)
    bufinx=1
    forall((l,c) ∈ Send_U(p,p'))
        send_buffer(bufinx) = U(l,c)
        bufinx = bufinx + 1
    send(p',send_buffer)
forall((p,p'), p ≠ p', Receive_U(p,p') ≠ ∅)
    receive(p',recv_buffer)
    bufinx=1
    forall((l,c) ∈ Receive_U(p,p'))
        U(l,c) = recv_buffer(bufinx)
        bufinx = bufinx + 1
```

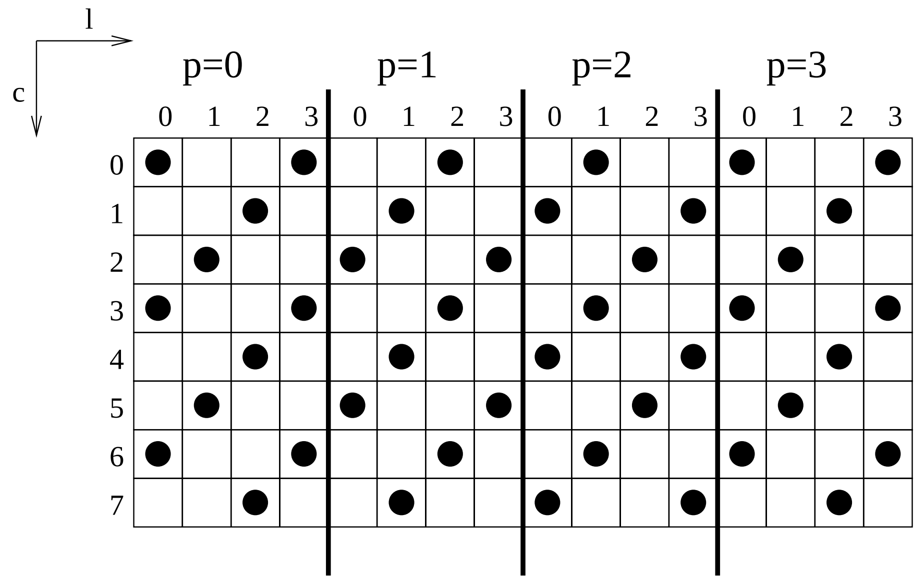
Generating the arrays

- Use Fourier-Motzkin to find bounds on l and c.
- Example,

```
      real A(0:42)
!HPF$ template T(0:127)
!HPF$ processors P(0:3)
!HPF$ align A(i) with T(3*i)
!HPF$ distribute T(cyclic(4)) onto P
```

Generating the arrays (cont.)

- Lots of holes. . .



Generating the arrays (cont.)

- Let $x = (l, c, a, t, i)$, then $Fx = f_0(n, p)$, where

$$F = \begin{pmatrix} 0 & 0 & A & -R & 0 \\ I & CP & 0 & -I & 0 \\ 0 & 0 & -I & 0 & S \end{pmatrix} \text{ and } f_0(n, p) = \begin{pmatrix} s_0 \\ -Cp \\ a_0(n) \end{pmatrix}$$

- Find the lattice.

Finding the lattice

- Hermite Normal Form: $L = FU$

$$Fx = f_0(n,p)$$
$$FUU^{-1}x = f_0(n,p)$$

- Let $x = Uv$,

$$Lv = f_0(n,p)$$
$$\begin{pmatrix} \tilde{L} & 0 \end{pmatrix} \begin{pmatrix} v_0(n,p) \\ v' \end{pmatrix} = f_0(n,p)$$

Finding the lattice (cont.)

- $\tilde{L}$ is unimodular.

$$v_0(n,p) = \tilde{L}^{-1} f_0(n,p)$$

- The final form,

$$x = Qv = \begin{pmatrix} Q_0 & F' \end{pmatrix} \begin{pmatrix} v_0(n,p) \\ v' \end{pmatrix}$$

$$x = Q_0 v_0(n,p) + F'v'$$

- Additional tricks to turn trapezoids into ragged rectangles.

Computing Alignment Automatically

Material taken from

- "Solving Alignment Using Elementary Linear Algebra", by Kotlyar, et al.

Collocation

What does it mean for iterations and data to be collocated?

- Array access function, $Fi + f = a$
- Computation alignment, $t_i = Ci + c$
- Data alignment, $t_a = Da + d$
- therefore, $\forall i, Ci + c = D(Fi + f) + d$

Collocation

$$\begin{bmatrix} C & c \end{bmatrix} = \begin{bmatrix} D & d \end{bmatrix} \begin{bmatrix} F & f \\ 0 & 1 \end{bmatrix}$$

Let,

$$\tilde{F} = \begin{bmatrix} F & f \\ 0 & 1 \end{bmatrix} \qquad \tilde{C} = \begin{bmatrix} C & c \end{bmatrix} \qquad \tilde{D} = \begin{bmatrix} D & d \end{bmatrix}$$

$$\tilde{C} - \tilde{D}\tilde{F} = 0$$

$$\begin{bmatrix} \tilde{C} & \tilde{D} \end{bmatrix} \begin{bmatrix} I \\ -\tilde{F} \end{bmatrix} = 0$$

- $\tilde{C}$ and $\tilde{D}$ are the unknowns.
- "Solve" by finding vectors that span the null space.

Multiple Loops and Arrays

Set of contraints,

$$\begin{bmatrix}\tilde{C}_k & \tilde{D}_j\end{bmatrix}\begin{bmatrix}I \\ -\tilde{F}_m\end{bmatrix} = 0$$

as $VU = 0$, where

$$V = \begin{bmatrix}C_1 & \ldots & D_1 \ldots\end{bmatrix} \qquad U = \begin{bmatrix}\ldots U_{kjm} \ldots\end{bmatrix}$$

$$U_{kjm} = \begin{bmatrix}0 & \ldots & 0 & I & 0 & \ldots & 0 & -\tilde{F}_k 0 & \ldots & 0\end{bmatrix}^T$$

Multiple Loops and Arrays (cont.)

$$VU = 0$$

- V is the set of unknowns
- Find vectors to span the null space
- Reduce the solution basis (make sure the dimension of C_k is correct).

Replication

$$RC = DF$$

Non-linear

Heuristic,

- Solve $C = DF$ for non-replicated arrays
- Solve $RC = DF$.

Heuristic

- Null space is [0] $\Rightarrow$ execute everything sequentially.
- Have to drop $C = DF$ contraints.
- Heurisitc,
  - If contraints differ only by f, then use only one of them.
  - Pick contraints to maximize parallelism, then use replication.
  - Pick contraints for largest arrays first.