Compiling High Performance Fortran (HPF)

Material taken from

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
  
  \[!\text{HPF} \text{ align } A(i,j,* \text{) with } T(i,j)\]
  
  \[!\text{HPF} \text{ align } B(i) \text{ with } T(i,*)\]

- \( T(i,*) \) – replication.

HPF Rules:

- Each array index can be used at most once in a template subscript expression in any given alignment.

- Each subscript expression cannot contain more than one index.
Distribution

- Mapping virtual processors onto physical processors.
  
  \[!\text{HPF}\$ \text{distribute} ~ T(\text{block}(20)), ~ T2(*,\text{cyclic}(1)) \text{ onto } P\]

- 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.

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


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 \( T(2j-1,-i+7) \)

\[ t_1 = 2a_3 - 1, \quad t_2 = -a_2 + 7 \]
Modelling alignment

- alignment w/ replication

\[ Rt = Aa + s_0 \]

example:

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

\[ t_1 = 2a_1 - 1 \]
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 $\Pi$ 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 \( T(*, \text{cyclic}(4), \text{block}(13)) \) onto \( P \)

\[
\begin{pmatrix}
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
t \\
c \\
p
\end{pmatrix} =
\begin{pmatrix}
40 & 0 \\
0 & 130
\end{pmatrix}
c +
\begin{pmatrix}
4 & 0 \\
0 & 10
\end{pmatrix}p + l
\]
Code generation

• The alignments,

\[ R_X t = A_X a + s_{X0} \]
\[ R_Y t = A_Y a + s_{Y0} \]
\[ R_Z t = A_Z a + s_{Z0} \]

• The distribution,

\[ \Pi t = CPc + Cp + l \]

• The original code:

\`!
HPF$ INDEPENDENT(i)
forall (L_i \leq 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”

\[
\text{Own}_X(p) = \{a | \exists t, \exists c, \exists l, s.t. R_X t = A_X a + s_{X0} \\
\Pi t = C_P c + C_P + l \\
0 \leq a \leq \text{diag}(D_X) \\
0 \leq p \leq \text{diag}(P) \\
0 \leq l \leq \text{diag}(C) \\
0 \leq t \leq \text{diag}(T) \}
\]
Sets for codegen (cont.)

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

\[
Compute(p) = \{i | S_X i + a_{X0}(n) \in Own_X(p) \land Li \leq b_0(n) \}
\]

- View Set - data elements accessors by processor

\[
View_Y(p) = \{a | \exists i \in Compute(p) s.t. a = S_Y i + a_{Y0}(n) \}
\]

- Communication Sets

\[
Send_Y(p, p') = Own_Y(p) \cap View_Y(p')
\]
\[
Receive_Y(p, p') = View_Y(p) \cap Own_Y(p')
\]
**Pseudo-code**

real $X'((c, l) \in Own_X(p)),$  
$Y'((c, l) \in Own_Y(p)),$  
$Z'((c, l) \in Own_Z(p))$

forall ($U \in \{Y', Z', \ldots\}$)

forall ($((p, p'), p \neq p', \text{Send}_U(p, p') \neq \emptyset)$)

forall (($l, c) \in \text{Send}_U(p, p')$)

send ($p', U(l, c)$)

forall ($U \in \{Y', Z', \ldots\}$)

forall ($((p, p'), p \neq p', \text{Receive}_U(p, p') \neq \emptyset)$)

forall (($l, c) \in \text{Receive}_U(p, p')$)

$U(l, c) = \text{receive}(p')$

if $\text{Compute}(p) \neq \emptyset$

forall (($l, c) \in \text{Compute}(p)$)

$X(S_{X'i + a_{X'i}} 0(n)) = f(Y(S_{Y'i + a_{Y'i}} 0(n)), Z(S_{Z'i + a_{Z'i}} 0(n)), \ldots)$
Generating the loops

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

\[
\begin{align*}
&\text{forall}( (p,p'), p \neq p', \text{Send}_U(p,p') \neq \emptyset) \\
&\text{bufinx}=1 \\
&\text{forall}( (l,c) \in \text{Send}_U(p,p')) \\
&\quad \text{send\_buffer(bufinx)} = U(l,c) \\
&\quad \text{bufinx} = \text{bufinx} + 1 \\
&\quad \text{send}(p',\text{send\_buffer}) \\
&\text{forall}( (p,p'), p \neq p', \text{Receive}_U(p,p') \neq \emptyset) \\
&\text{receive}(p',\text{recv\_buffer}) \\
&\text{bufinx}=1 \\
&\text{forall}( (l,c) \in \text{Receive}_U(p,p')) \\
&\quad U(l,c) = \text{recv\_buffer(bufinx)} \\
&\quad \text{bufinx} = \text{bufinx} + 1
\end{align*}
\]
Generating the arrays

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

```plaintext
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

\[
F = \begin{pmatrix}
0 & 0 & A & -R & 0 \\
I & CP & 0 & -I & 0 \\
0 & 0 & -I & 0 & S \\
\end{pmatrix}
\]

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 \( x = Uv \),

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

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

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

Let,

$$\tilde{F} = \begin{bmatrix} F & f \\ 0 & 1 \end{bmatrix} \quad \tilde{C} = [C \ c] \quad \tilde{D} = [D \ d]$$
\[ \tilde{C} - \tilde{D}\tilde{F} = 0 \]

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

- \( \tilde{C} \) 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

\[
V = \begin{bmatrix}
C_1 & \ldots & D_1 & \ldots
\end{bmatrix} \quad U = \begin{bmatrix}
\ldots & U_{kjm} & \ldots
\end{bmatrix}
\]

\[
U_{kjm} = \begin{bmatrix}
0 & \ldots & 0 & I & 0 & \ldots & 0 & -\tilde{F}_k & 0 & \ldots & 0
\end{bmatrix}^T
\]
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] ⇒ execute everything sequentially.
- Have to drop $C = DF$ constraints.
- Heuristics,
  - If constraints differ only by $f$, then use only one of them.
  - Pick constraints to maximize parallelism, then use replication.
  - Pick constraints for largest arrays first.