Motivation for data parallel programming

- When we write an explicit parallel program, we need mechanisms for expressing parallelism
- So far we’ve examined
  - Communication
  - Synchronization
  - Data partitioning
- We have to think about new things, or think about known principles a bit differently
  - The distinction between global and local data
  - Optimization involves controlling granularity to avoid high overheads
- Today we’ll look at another approach to parallel programming which eliminates some concerns, or simplifies the programming process
Data parallel programming

- Earlier in the course we talked about SIMD parallelism
- Processors execute instructions in lock step under the direction of a control processor
- We may implement a relative of SIMD parallelism on MIMD architectures: data parallelism

The link with SIMD parallelism

- Historically, SIMD machines implemented data parallelism
  - ILLAC IV (1960’s)
  - Connection Machine (1980’s and early 1990’s)
- Intel re-introduced SIMD parallelism to support multimedia and graphics
  - SSE = Streaming SIMD extensions
  - 8 vector registers, each with 4 x 32 bit elements
  - Operations defined on vectors
  - http://x86.ddj.com/articles/sse_pt1/
The data parallel model

- A parallel data structure, e.g. an array, list, sequence
- Apply an operation uniformly over all processors in a single step
- Assign each array element to a virtual processor
- Implicit barrier synchronization between each step
- Program executes as if in a shared name space

\[
\begin{array}{c}
2 & 12 \\
8 & 25 \\
18 & 42 \\
\end{array}
= \\
\begin{array}{c}
1 & 4 \\
2 & 5 \\
3 & 6 \\
\end{array}
\begin{array}{c}
2 & 3 \\
4 & 5 \\
6 & 7 \\
\end{array}
\]

Practical data parallel languages

- APL (1962)
- Matlab
- Fortran 90, 95
- HPF (High Performance Fortran) - 1994
Bread and Butter: Array Operations

Parallel Assignment
A = 0 // scalar extension
Z = 3.7
D = C // array copy
T = [1 2 3 4 5] // An array literal

Binary array operators operate pointwise on \textit{conforming} arrays
- same size and shape
- The arrays could be multidimensional

Extension to array operations
- Scalars can be combined with arrays
- There are also specialized intrinsics
  T = [1 4 9 16 25]
  U = 3 + T  // 4 7 12 19 28
  Z = \text{sqrt}(T)  // Built in intrinsic extended to array
  // 1 2 3 4 5
  Y = \text{max}(T,10)  // 10 10 10 16 25
Array Sections

- Portion of an array defined by a triplet in each dimension
- May appear wherever an array is used

A(1:5) ! first five elements
A(1:10:2) ! odd elements
A(10:2:-2) ! Even elements in reverse order
B(2:4,2:4) ! 3 x 3 block
B(:, 1) ! first column
B(j,:) ! jth row

Example application

- Perform a sweep over a 1D mesh using nearest neighbor computation

  \[
  \text{unew}[1:N] = \frac{\text{uold}[0:N-1] + \text{uold}[2:N+1]}{2.0}
  \]

- Perform a sweep over a 2D mesh using nearest neighbor computation

  \[
  \text{unew}[1:N,1:N] = \frac{\text{uold}[0:N-1,1:N] + \text{uold}[2:N+1,1:N] + \text{uold}[1:N,0:N-1] + \text{uold}[1:N,2:N+1]}{4.0}
  \]
Ring algorithm for article computation

- Move a system of N bodies under mutual interaction
- The particles are described by
  - 3 position arrays x[], y[], z[]
  - A mass array m[
- Let the force law be a given function F(x,y,z,m,x0, y0, z0)

\[
mc = m, \quad xc = x, \quad yc = y, \quad zc = z \\
\text{do while} \quad (t < t_{\text{end}}) \\
\quad \text{for} \quad i = 1 : n \\
\quad \quad \text{force} = \text{force} + F(x,y,z,m,xc,yc,zc,mc) \\
\quad \quad xc = \text{CSHIFT}(xc,1) \ldots \\
\quad \text{end for} \\
\text{end do}
\]

Reduction Operators

Reduce an array to a scalar under an associative binary operation
- sum, product
- minval, maxval
- many others

\[
\text{do while} \quad (\text{maxdiff} < \epsilon) \\
\quad \text{unew}[1:N] = (\text{uold}[0:N-1] + \text{uold}[2:N+1]) / 2.0 \\
\quad \text{diff} = \text{unew} - \text{uold} \\
\quad \text{absdiff} = \text{abs}(\text{diff}) \\
\quad \text{maxdiff} = \text{maxval}(\text{absdiff}) \\
\text{enddo}
\]
Conditional Operations

- The following statement
  \[ \text{dist} = \max(\text{abs}(\text{fishp}), 0.01) \]
  is equivalent to
  \[
  \begin{align*}
  \text{where} & \quad (\text{abs}(\text{fishp}) \geq 0.01) \\
  \text{dist} & = \text{abs}(\text{fishp}) \\
  \text{elsewhere} & \\
  \text{dist} & = 0.01 \\
  \text{end where}
  \end{align*}
  \]

- Recall that in an SIMD architecture, processors can individually opt out of executing an operation.

- An SIMD machine needs 2 steps to execute the statement.

- What about an MIMD machine?

Forall

**FORALL** (triplet, triplet,...,mask) assignment

Evaluate entire RHS for all index values (in any order) and assign to a temporary.

Perform all assignments (in any order) using the temporary.

No more than one value for each element on the left hand side.

```c
forall (i=1:n) x[i] = (i*2.0/n)-1.0
forall (i = 1:n) D[\text{Indx}[i] = C[i]
forall (i=1:n, j = 1:m) X[i,j] = 1.0/(i+j)
forall (i=1:n, j=1:N, i == j) X[i,j] = 0 // Guard
forall (i=1:n, j=1:n, k=1:n) C(i,j) = C(i,j) + A(i,k) * B(k,j)
  // Why not?
```
Subtle semantics

Evaluate entire RHS for all index values (in any order) and assign to a temporary array

Assign temporary to LHS of assignment

`forall` is not like `for`

But it is like array section assignment

\[
A, B, C = \{3 \ 2 \ 0 \ 1\}
\]

\[
forall (i=2:4) \ A[i] = A[i-1] \quad ! \{3 \ 3 \ 2 \ 0\}
\]

\[
for (i=2:4) \quad B[i] = B[i-1] \quad ! \{3 \ 3 \ 3 \ 3\}
\]

\[
C[2:4] = C[1:3] \quad ! \{3 \ 3 \ 2 \ 0\}
\]

An example

- Compute the following loop using array statements

  \[
  \text{for } i := 0 \text{ to } N-1 \text{ do} \\
  \quad B[i] := B[N-i-1];
  \]

- If \(B = \{0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7\}\) initially
- Result is \(\{7 \ 6 \ 5 \ 4 \ 4 \ 5 \ 6 \ 7\}\)
What happens with forall?

- We try
  \[ \text{forall } (i=0:N-1) \ B[i] = B[N- i-1] \]

- Which returns \{7 6 5 4 3 2 1 0 \}

  instead of \{7 6 5 4 4 5 6 7\}

A solution

- The original loop:

  \[
  \text{for } i := 0 \text{ to } N-1 \text{ do} \\
  \quad B[i] := B[N- i-1];
  \]

- For 0 to \(N/2-1\), \(B[N- i-1]\) references initial data, but updated data for remaining values of \(i\)
Preserving semantics

• We split the loop accordingly

  for i := 0 to N/2 - 1 do
    B[i] := B[N - i - 1];
  for i := N/2 to N do
    B[i] := B[N - i - 1];

• Introducing the array statements

  forall (i=0:N/2-1) B[i] = B[N-i-1]

• We didn’t need to second loop

HPF Data Distribution (layout) directives

• DISTRIBUTE arrays over processors
• HPF offers additional control
  –Create abstract processor geometries and templates (not discussed)
  –ALIGN arrays with other arrays for affinity: elements that are operated on together should be stored together
• Cyclic, block_cyclic, block
**Layouts on Processor Grids**

- (Block, *)
- (*, Block)
- (Block, Block)
- (Cyclic, *)
- (Cyclic, Cyclic)
- (Cyclic, Block)

**Implicit Communication**

Operations on conformable array sections may induce communication:

$A(1:7) = B(2:8)$  \! \! A and B are distributed BLOCKwise
Procedures

- Communication may occur if formal and actual parameters have different layouts

- Consider A(1:7) = C(2:8), where
  A is BLOCK distributed and C is CYLIC

- Important to be aware of this activity

Global Communication

\[
X = X(n:1:-1) \quad \quad ! \text{permutation (reverse)} \\
B = A(\text{Indx(:)}) \quad \quad ! \text{“gather”} \\
C(\text{Indx(:)}) = B \quad \quad ! C = “scatter:” \\
\quad \quad \quad ! \text{no duplicates on left!}
\]
Specialized Communication

\begin{align*}
\text{CSHIFT(} & \text{array, dim, shift)} \quad \text{! cyclic shift in one dimension} \\
\text{TRANSPOSE(} & \text{matrix)} \quad \text{! matrix transpose}
\end{align*}

\begin{align*}
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
\end{array} & \rightarrow & \begin{array}{cccc}
4 & 1 & 2 & 3 \\
8 & 5 & 6 & 7 \\
\end{array}
\end{align*}

source \quad \text{target}

Parallel prefix (scan) operations

\begin{align*}
X = [4 \ 5 \ 6 \ 7 \ 8 \ 9] \\
\text{SUM\_PREFIX(X)}
\end{align*}

\begin{align*}
\text{SUM\_PREFIX(X, MASK=[T T F T F T])}
\end{align*}

\begin{align*}
\text{SUM\_PREFIX(X, SEGMENT=[T T F F T])}
\end{align*}