

Today's Plan

Higher level languages

- Raising the level of abstraction with HPF

“I don't know what the technical characteristics of the standard language for scientific and engineering computation in the year 2000 will be . . . but I know it will be called Fortran.”

- John Backus, c 1980

HPF: High Performance Fortran

Philosophy

- Automatic parallelization won't work
- For data parallelism, what's important are **data placement** and **data motion**
- Give the compiler help:
 - Extends Fortran with directives that guide data distribution
- Allow slow migration from **legacy codes**
 - The directives are only hints

Basic idea

- Each processor operates on part of the overall data
- Directives indicate which processor operates on which data
- Much higher level than message passing

HPF History

The beginning

- Designed by large consortium in the early 90's
- Participation by academia, industry, and national labs
 - All major vendors represented
 - Convex, Cray, DEC, Fujitsu, HP, IBM, Intel, Meiko, Sun, Thinking Machines
- Heavily influenced by Fortran-D from Rice
 - D stands for "Data" or "Distributed"
- HPF 2.0 specified in 1996

CS380P Lecture 14

HPF

3

Strategic Decisions

Context

- Part of early 90's trend towards consolidating supercomputing research
- To reduce risk, fund a few large projects rather than a lot of small risky projects
- Buoyed by the success of MPI
- Aware of the lessons of vectorizing compilers
 - Compilers can train programmers by providing feedback

CS380P Lecture 14

HPF

4

Vectorizing Compilers

Basic idea

- Instead of looping over elements of a vector, perform a single vector instruction

- Example

```
for (i=0; i<100; i++)
    A[i] = B[i] + C[i];
```

Scalar code

- Execute 4 instructions 100 times
 - 2 Loads
 - 1 Add
 - 1 Store

Vector code

- Execute 4 instructions once
 - 2 vector Loads
 - 1 vector Add
 - 1 vector Store

Advantages?

CS380P Lecture 14


HPF

5

Guidelines for Writing Vectorizable Code

1. Avoid conditionals in loops


```
for (i=0; i<100; i++)
    if (A[i] > MaxFloat)
        A[i] = MaxFloat;
```



```
for (i=0; i<100; i++)
    A[i] = min(A[i], MaxFloat);
```

2. Promote scalar functions

```
for (i=0; i<100; i++)
    foo (A[i], B[i]);
```



```
Foo(A, B);
```

- | | |
|--|--|
| <ul style="list-style-type: none"> – Lots of function calls inside a tight loop – Function call boundaries inhibit vectorization | <ul style="list-style-type: none"> – One function call – Body of this function call can be easily vectorized |
|--|--|

CS380P Lecture 14

HPF

6

Guidelines for Writing Vectorizable Code (cont)

3. Avoid recursion

4. Choose appropriate memory layout

- Depending on the compiler and the hardware, some strides are vectorizable while others are not

Other guidelines?

The point

- These are simple guidelines that programmers can learn
- The concept of a vector operation is simple

CS380P Lecture 14

HPF

7

Strategic Decisions (cont)

HPF: A community project

- Compiler directives don't change the program's semantics
- They only affect performance
- Allows different groups to conduct research on different aspects of the problem
- Even the "little guy" can contribute

Based on Fortran

- Why Fortran? Huge base of existing scientific software
- Fortran77 or Fortran90? Both

CS380P Lecture 14

HPF

8

Fortran 90

An array language

- Can operate with entire arrays as operands
 - Pairwise operators
 - Reduction operators
- Uses slice notation
 - `array1d(low: high: stride)` represents the elements of `array1` starting at `low`, ending at `high`, and skipping every `stride-1` elements
 - The stride is an optional operand
- Converts many loops into array statements

CS380P Lecture 14

HPF

9

Example Computation

Jacobi Iteration

- The elements of an array, initialized to 0.0 except for 1.0's along its southern border, are iteratively replaced with the average of their 4 nearest neighbors until the greatest change between two iterations is less than some epsilon.

	0	0	0	0	
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
	1	1	1	1	

CS380P Lecture 14

HPF

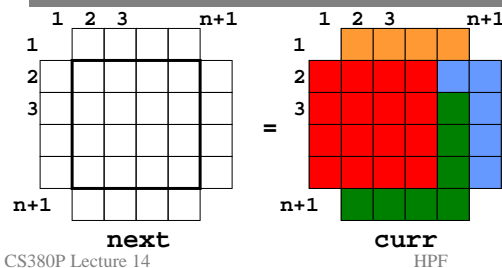
10

Jacobi Iteration in Fortran 90

Example

- The following statement computes the averaging step in the Jacobi iteration
- Assume that `next` and `curr` are 2D arrays

```
next(2:n, 2:n) = (curr(1:n-1, 2:n) +  
curr(2:n, 3:n+1) +  
curr(3:n+1, 2:n) +  
curr(2:n, 1:n-1)) / 4
```



CS380P Lecture 14

11

Block Data Distribution

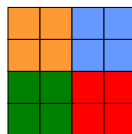
Block distribution of 1D array

```
!HPF$ PROCESSORS PROCS(4)  
!HPF$ DISTRIBUTE array1D(BLOCK) ONTO PROCS
```



Block distribution of 2D array

```
!HPF$ PROCESSORS PROCS(4)  
!HPF$ DISTRIBUTE array2D(BLOCK,BLOCK) ONTO PROCS
```



CS380P Lecture 14

HPF

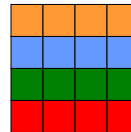
12

Block Data Distribution (cont)

1D block distribution

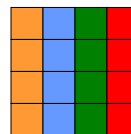
```
!HPF$ PROCESSORS PROCS(4)
!HPF$ DISTRIBUTE array2D(BLOCK, *) ONTO PROCS
```

- Do not distribute dimensions with '*'s



1D block distribution

```
!HPF$ DISTRIBUTE array2D(*, BLOCK) ONTO PROCS
```



CS380P Lecture 14

HPF

13

Cyclic Data Distribution

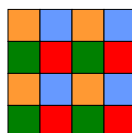
Cyclic distribution of 1D array

```
!HPF$ PROCESSORS PROCS(4)
!HPF$ DISTRIBUTE array1D(CYCLIC) ONTO PROCS
```



Cyclic distribution of 2D array

```
!HPF$ PROCESSORS PROCS(4)
!HPF$ DISTRIBUTE array2D(CYCLIC,CYCLIC) ONTO PROCS
```



CS380P Lecture 14

HPF

14

Cyclic Data Distribution (cont)

Cyclic distribution of 1D array

```
!HPF$ PROCESSORS PROCS(4)
!HPF$ DISTRIBUTE array1D(CYCLIC(2)) ONTO PROCS
```



- Cyclic with panels of width 2

CS380P Lecture 14

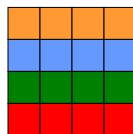
HPF

15

Cyclic Data Distribution (cont)

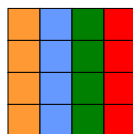
1D cyclic distribution

```
!HPF$ PROCESSORS PROCS(4)
!HPF$ DISTRIBUTE array2D(CYCLIC, *) ONTO PROCS
```



Cyclic distribution of a 2D array

```
!HPF$ DISTRIBUTE array2D(*, CYCLIC) ONTO PROCS
```



CS380P Lecture 14

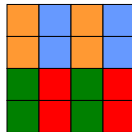
HPF

16

Block-Cyclic Data Distribution

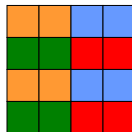
Block-cyclic distribution

```
!HPF$ PROCESSORS PROCS(4)
!HPF$ DISTRIBUTE array2D(BLOCK, CYCLIC) ONTO PROCS
```



Block-cyclic distribution

```
!HPF$ DISTRIBUTE array2D(CYCLIC, BLOCK) ONTO PROCS
```



CS380P Lecture 14

HPF

17

Alignment Directives

Arrays can be aligned with one another

- Aligned elements will reside on the same physical processor
- Alignment can reduce communication
- Can align arrays of different dimensions

Example

```
!HPF$ ALIGN a (i) WITH b(i-1)
```



CS380P Lecture 14

HPF

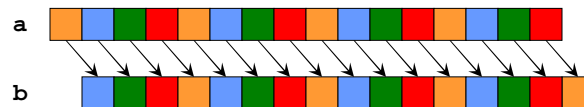
18

Communication is Implied by the Distribution

Example

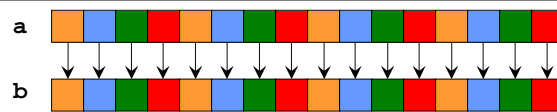
- The following alignment and assignment requires every element to be communicated to a different processor

```
!HPF$ ALIGN a(i) WITH b(i-1)  
a(1:n) = b(1:n)
```



- The following induces no communication

```
!HPF$ ALIGN a(i) WITH b(i)
```



CS380P Lecture 14

HPF

19

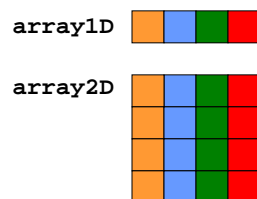
Arrays of Different Dimensions Can Be Aligned

Higher-dimensional arrays can be collapsed

- Non-collapsed dimensions are then aligned with lower-dimensional array

Example

```
!HPF$ ALIGN array2D(*,:) WITH array1D(:)
```



CS380P Lecture 14

HPF

20

Arrays of Different Dimensions Can Be Aligned (cont)

Lower-dimensional arrays can be replicated

- Replicated array is then aligned with higher-dimensional array

Example

```
!HPF$ ALIGN array1D(:) WITH array2D(*,:)
```

array1D  } replicated values

array2D  Distribution again induces communication

CS380P Lecture 14

HPF

21

Data Distribution and Procedure Calls

Parameter passing

- Must bind actuals to formals
- We now need to worry about binding data distributions as well as values
- There are many options
 - Serial execution for “assumed-size arrays”
 - Also, “assumed-shape arrays” and “explicit-shape arrays”

Consider two cases

- Procedure is oblivious to the data distribution
 - Compiler needs to implement general code that will work for any data distribution
 - Difficult to reason about communication and locality
- Procedure defines an explicit data distribution
 - Compiler must dynamically redistribute data between actuals and formals– how expensive is this operation?

CS380P Lecture 14

HPF

22

New Statements

FORALL loop (for Fortran 77)

- Provides [array language semantics](#)
- Evaluate entire rhs of a statement before assigning to lhs
- Implied barrier between every statement
- Now part of the ANSI Fortran standard

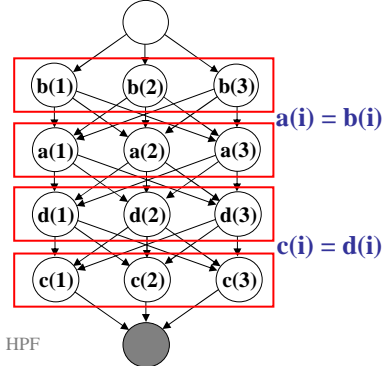
Example

```
FORALL (i = 1:3)
  a(i) = b(i)
  c(i) = d(i)
END FORALL
```

Fortran90 equivalent?

```
a(1:3) = b(1:3)
c(1:3) = d(1:3)
```

Dependence graph



CS380P Lecture 14

HPF

23

FORALL Loops vs. DO Loops

Example

- For the given initial values, what do the following compute?

```
FORALL (i = 2:5)
  a(i) = a(i-1)
END FORALL
```

```
DO (i = 2:5)
  a(i) = a(i-1)
END DO
```

Initial values

a 7 8 9 10 11

Final values

a 7 7 8 9 10

Final values

a 7 7 7 7 7

CS380P Lecture 14

HPF

24

Independent Loops

INDEPENDENT directive

- Loop iterations are independent
- No implied barriers

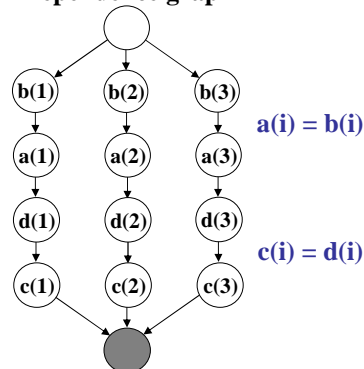
Example

```
!HPF$ INDEPENDENT
DO (i = 1:3)
  a(i) = b(i)
  c(i) = d(i)
END DO
```

Fortran90 equivalent?

- None

Dependence graph



CS380P Lecture 14

HPF

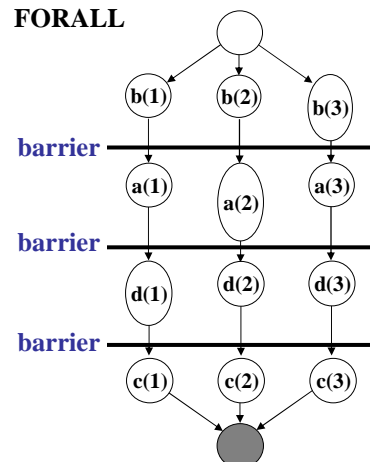
25

FORALL Loops vs. Independent Loops

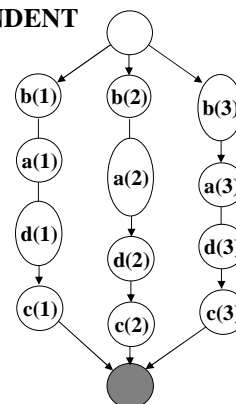
Is there a difference?

- Independent loops may offer more parallelism

FORALL



INDEPENDENT



CS380P Lecture 14

HPF

26

FORALL Loops vs. Independent Loops (cont)

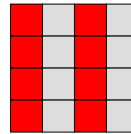
Is there a difference?

- FORALL loops are concise

Example

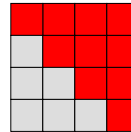
```
FORALL (i=1:n:2)
  a(i) = b(i)
END FORALL
```

Strided iteration



```
FORALL (i=1:n, j=1:n, j>=i)
  a(i) = b(i)
END FORALL
```

Upper triangular iteration



CS380P Lecture 14

HPF

27

Evaluation

Your thoughts on HPF?

- How does it compare with MPI?
- Is this a convenient language to use?
- Can programmers get good performance?

No performance model

- To understand locality and communication, need to understand complex interactions among distributions
 - Procedure calls are particularly bad
- Many hidden costs

Does the following code induce communication?

```
a(i) = b(i)
```

- Small changes in distribution can have large performance impact

CS380P Lecture 14

HPF

28

Evaluation (cont)

No performance model

- Complex language \Rightarrow Difficult language to compile
 - Large variability among compilers
 - Kernel HPF: A subset of HPF “guaranteed” to be fast

An accurate performance model is essential

- Witness our experience with the PRAM

Common user experience

- Play with random different distribution in an attempt try to get good performance

CS380P Lecture 14

HPF

29

Evaluation (cont)

Language is too general

- Difficult to obey an important system design principle:
 - “Optimize the common case”
 - What is the common case?
- Sequential constructs inherited from Fortran77 and Fortran90 cause problems
 - For example, the following code forces compiler to perform matrix transpose

```
FORALL (i=1:n, j=1:n)
  a(i, j) = a(j, i)
END FORALL
```

CS380P Lecture 14

HPF

30

Next Time

Distributed Work Queue Homework

- Due Wednesday 11:59pm

Assignment 5

- Available Wednesday night
- Due in a while