The Programming Interface

Libraries and languages make parallel programming possible, but rarely easy

Commentary on Infix form of PP

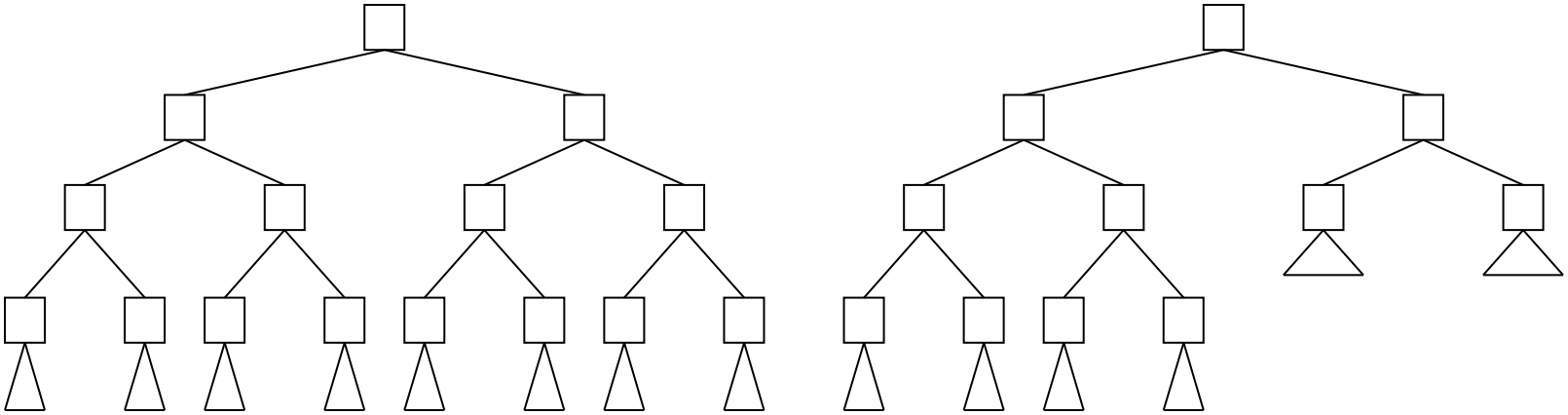
- What was your experience with formulating a parallel prefix computation as an infix operation?

From last time: Tree Algorithms

- Trees are an important component of computing
  - The "Schwartz tree" has been logical
  - Trees as data structures are complicated because they are typically more dynamic
  - Pointers are generally not available
  - Work well with work queue approach
  - As usual, we try to exploit locality and minimize communication

Breadth-first Trees

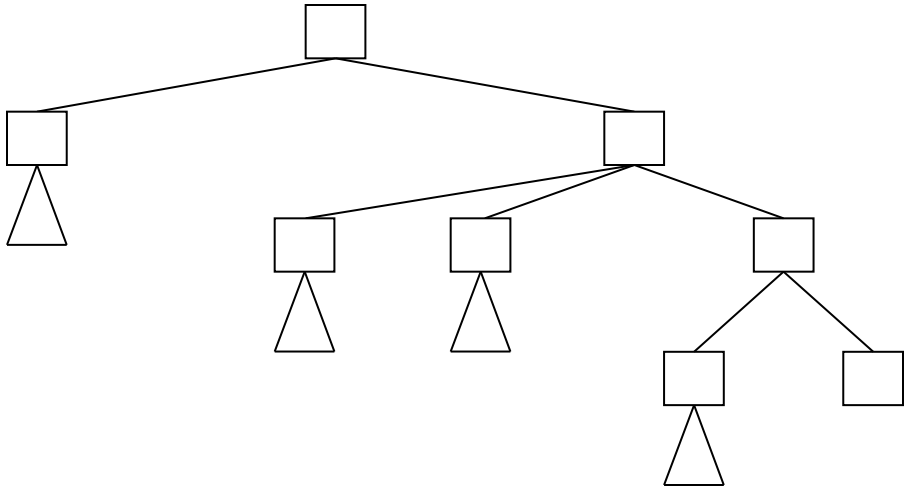
- Common in games, searching, etc



- Split: Pass 1/2 to other processor, continue
  - Stop when processors exhausted
  - Responsible for tree that remains
  - Ideal when work is localized

Depth-first

- Common in graph algorithms



- Get descendants, take one and assign others to the task queue

Key issue is managing the algorithm's progress

Coordination Among Nodes

- Tree algorithms often need to know how others are progressing
  - Interrupt works if it is just a search: Eureka!!
  - Record α-β cut-offs in global variable
  - Other pruning data, e.g. best so far, also global
  - Classic error is to consult global too frequently

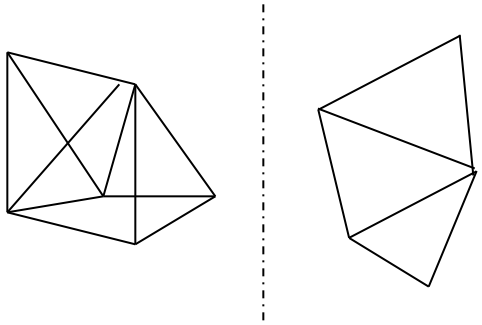
- Rethink: What is tree data structure's role?

Write essay: Dijkstra's algorithm is not a good… :)

Complications

- If coordination becomes too involved, consider alternate strategies:

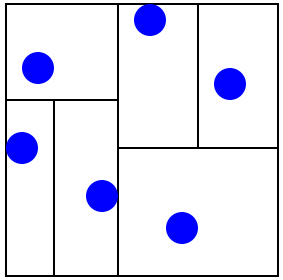
  Graph traverse => local traverse of partitioned graph



- Local computation uses sequential tree algorithms directly … stitch together

Full Enumeration

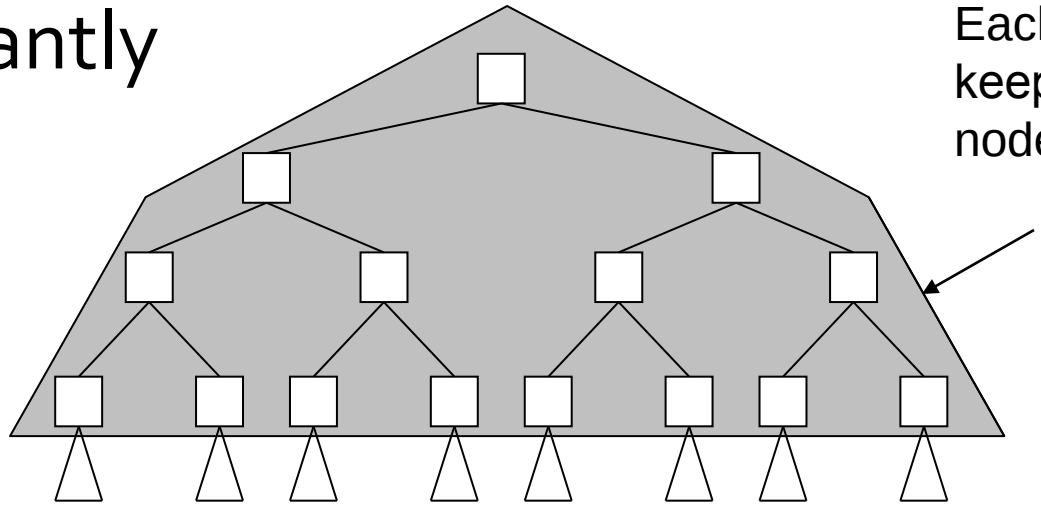
- Trees are a useful data structure for recording spatial relationships: K-D trees



- Generally, decomposition is unnecessary "all the way down" -- but this optimization implies two different regimes

Cap Reduces Communication

- The nodes near root can be stored redundantly



  - Processors consult local copy -- alert others to changes

Summary of Parallel Algorithms

- Reconceptualizing is often most effective
- Focus has not been on ||ism, but on other stuff
  - Exploiting locality
  - Balancing work
  - Reducing inter-thread dependences
- We produced general purpose solution mechanisms: UD-reduce and UD-scan
- We like trees, but recognize that direct application is not likely

The Programming Interface

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

John Backus, c. 1980

The Situation Today

- I have argued that a key property of a || programming system is that it embody an accurate (CTA) model of computation
- Recall why:
  - Wrong model leads to picking wrong algorithm
  - Communication costs -- they cannot be ignored
  - || programs must port, so pick universal model
- So, which of our present languages do that? Today, we'll see.

Parallel Programming Context

- At least 100 serious parallel programming languages have been developed in the last 2 decades … why isn't the problem solved?
  - Generalizing …
    - Most languages focused on a "silver bullet" solution, but the problem is more complex
    - Just a few of the languages were fully implemented
    - To be taken seriously, a language must
      - Run serious applications fast
      - Run on "all" parallel machines
      - Have substantial support (docs, compilers with libraries, tools such as debuggers and IDEs, 1-800 #)

Industry Backing

Not Surprisingly …

- No new languages crossed the bar
  - Performance challenge …
    - Serious applications programs are huge -- it is time consuming to write an equivalent program in any language, and it may require domain knowledge
    - Production programs are often well optimized -- competing on performance implies an effective compiler and performance debugging tools
    - "Linear speedup" goal (*P* processors will yield a *P*-fold speed-up) is naïve, but widely assumed
    - Doing well on one program is not persuasive
  - Portability challenges are similar
  - Will any programmer *learn* a new language?

Where We Stand Today

- Today, with few exceptions, we program using library-based facilities rather than languages
  - Sequential language + message passing in MPI or PVM
  - Sequential language + thread packages such as P-threads, or equivalently, Java-threads
  - OpenMP with a pragma-aware compiler for a sequential programming language
- Consider each briefly before discussing new developments

Message Passing

- Message passing is "the lowest of the low", but remains in widespread use because …
  - It works -- embodies the CTA || model
  - It is required for clusters, supercomputers, etc.
  - Achieving performance is definitely possible
  - Portability is essential for long-lived programs
- What is it?
  - Variations on primitive `send`/`receive`
  - Process spawning, broadcast, etc.
  - Programming goodies: reduce, scan, processor groups

Realities of Message Passing

- In message passing
  - There are few abstractions to simplify the work
  - Programmers must do everything except the physical layer
  - Experiments show that compared to "designed from first principles" parallel languages, MPI programs are 6 times larger … the extra code is the subtle, difficult to get right, and timing-sensitive
  - Consider dense matrix multiplication

MM in MPI -- 1

A “master--slave” solution

```
MPI_Status status;
main(int argc, char **argv) {
int numtasks,                   /* number of tasks in partition */
   taskid,                      /* a task identifier */
   numworkers,                  /* number of worker tasks */
   source,                      /* task id of message source */
   dest,                        /* task id of message destination */
   nbytes,                      /* number of bytes in message */
   mtype,                       /* message type */
   intsize,                     /* size of an integer in bytes */
   dbsize,                      /* size of a double float in bytes */
   rows,                /* rows of matrix A sent to each worker */
   averow, extra, offset,        /* used to determine rows sent to each worker */
   i, j, k,                     /* misc */
   count;
double a[NRA][NCA],  /* matrix A to be multiplied */
     b[NCA][NCB],    /* matrix B to be multiplied */
     c[NRA][NCB];                /* result matrix C */
```

MM in MPI -- 2

```
intsize = sizeof(int);
dbsize = sizeof(double);

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
numworkers = numtasks-1;

/************************** master task ************************************/
if (taskid == MASTER) {
for (i=0; i<NRA; i++)
   for (j=0; j<NCA; j++)
     a[i][j]= i+j;
 for (i=0; i<NCA; i++)
   for (j=0; j<NCB; j++)
     b[i][j]= i*j;
```

Create test data -- actually inputting data is harder

MM in MPI -- 3

```
/* send matrix data to the worker tasks */
  averow = NRA/numworkers;
  extra = NRA%numworkers;
  offset = 0;
  mtype = FROM_MASTER;
  for (dest=1; dest<=numworkers; dest++) {
    rows = (dest <= extra) ? averow+1 : averow;
    MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    count = rows*NCA;
    MPI_Send(&a[offset][0], count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    count = NCA*NCB;
    MPI_Send(&b, count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);

    offset = offset + rows;
    }
```

MM in MPI -- 4

```
/* wait for results from all worker tasks */
  mtype = FROM_WORKER;
  for (i=1; i<=numworkers; i++)      {
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    count = rows*NCB;
    MPI_Recv(&c[offset][0], count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD,&status);
}
/**************************** worker task ************************************/
if (taskid > MASTER) {
  mtype = FROM_MASTER;
  source = MASTER;
  MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  count = rows*NCA;
  MPI_Recv(&a, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
```

MM in MPI -- 5

```
count = NCA*NCB;
 MPI_Recv(&b, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);

 for (k=0; k<NCB; k++)
  for (i=0; i<rows; i++) {
    c[i][k] = 0.0;
    for (j=0; j<NCA; j++)
      c[i][k] = c[i][k] + a[i][j] * b[j][k];
    }

 mtype = FROM_WORKER;
 MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
 MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
 MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);

 }  /* end of worker */
```

Actual Multiply

91 “Net” Lines

MPI Collective Communication

- Reduce and scan are called *collective* operations
- Reduce/scan apply to nodes, not values
- Basic operations +, *, min, max, &&, ||
- Processor groups simplify collective ops on logical structures like “rows”, “leaves”, etc
- MPI allows user-defined scans … these have probably never been used!
- Bottom Line: Message passing is painful to use but it works … which makes it a solution of choice

Threading Libraries

- The P-threads library, designed for concurrency, is now also used for parallelism
- Sharing is implemented by referencing shared memory
  - As mentioned, the memory not sequentially consistent
  - Not CTA; P-threads use RAM performance model, a greater concern as latencies have increased
  - Tends to promote very fine-grain sharing (recall `count_3s` example), which limits the work that can be used to amortize the overhead costs such as thread creation, scheduling, etc.
  - Scaling potential is limited

Writing threaded code using CTA principles usually gives good results

Threading Is Subtle

- It is difficult to get threaded programs right
  - Programmers are responsible for protecting all data references
  - Avoiding deadlock requires discipline and care -- and mistakes are easy to make, especially when optimizing
  - Timing errors can remain latent for a very long time before emerging

Main difficulties: Lots of work for small ||ism; poor scaling prospects

Sample P-thread Code: Dot-Product

```
# define NUMTHRDS 4
double sum;
double a[256], b[256];
int status;
int n = 256;
pthread_t thds[NUMTHRDS];
pthread_mutex_t mutex_sum;

int main ( int argc, char *argv[] );

void *dotprod ( void *arg );
int main ( int argc, char *argv[] ) {
 int i;
 pthread_attr_t attr;
 for ( i = 0; i < n; i++ ) {
  a[i] = i * 0.5;
  b[i] = i * 2.0;
 }
```

P-threads Dot #2

```
  pthread_mutex_init ( &mutex_sum, NULL );
  pthread_attr_init ( &attr );
  pthread_attr_setdetachstate ( &attr, PTHREAD_CREATE_JOINABLE );

  for ( i = 0; i < NUMTHRDS; i++ ) {
      pthread_create ( &thds[i], &attr, dotprod, ( void * ) i );
  }
  pthread_attr_destroy ( &attr );
  for ( i = 0; i < NUMTHRDS; i++ ) {
      pthread_join ( thds[i], ( void ** ) &status );
  }

  printf ( "  Sum = %f\n", sum );
  pthread_mutex_destroy ( &mutex_sum );
  pthread_exit ( NULL );
  return 0;
}
```

P-threads

```
void *dotprod ( void *arg ) {
  int i, my_first, my_last, myid;
  double sum_local;
  myid = ( int ) arg;
  my_first = myid * n / NUMTHRDS;
  my_last = ( myid + 1 ) * n / NUMTHRDS;

  sum_local = 0;
  for ( i = my_first; i <= my_last; i++ ) {
    sum_local = sum_local + a[i] * b[i];
  }

  pthread_mutex_lock ( &mutex_sum );
  sum = sum + sum_local;
  pthread_mutex_unlock ( &mutex_sum );

  pthread_exit ( ( void * ) 0 );
}
```

OpenMP

- Developed as easy access to multi-threading
- Has second life with multi-core (Intel and others push)
- Approach
  - Add pragmas to C or Fortran code
  - Pragma-aware compiler links in appropriate library calls
  - Pragma-unaware compiler -- no change from sequential
  - All responsibility for parallel == sequential left to programmer
- Main benefit: little effort, some benefit
- Main liability: tight binding to sequential semantics

Note OpenMP Conflict

- The program is sequential
  - When there is no compiler to interpret the pragmas, the code is sequential
  - When there is no parallelism available, the sequential code runs
  - When there is a compiler AND parallel processors the sequential code runs
- But, we often observe that there IS usually a conceptual difference between sequential and parallel algorithms

Sample Code -- Dot Product

```
double dotProduct()  {
      int I; double sum_p;
      double result = 0;
      #pragma omp parallel shared(a, b, result) private(sum_p)
      {
         sum_p=0;
      #pragma omp parallel for private(i)
         for(i=0; i<n; i++) {
              sum_p += a[i]*b[i];
          }
       #pragma omp critical
          {
          result += sum_p;
          }
       }
       return result;
   }
```

OpenMP Compiler

- 4 Processor Sun Enterprise running the NAS PB written in C with OpenMP

Block Tridiagonal
Conjugate Gradient
Embarrassingly ||
Fast Fourier Trans
Integer Sort
LU Decomposition
Multigrid Iteration
Sparse Matrix-Vector

Program	Class	1 thread	2 threads	4 threads
BT	W	119.19 (1.00)	61.28 (1.95)	36.65 (3.25)
	A	2900.02 (1.00)	1546.70 (1.87)	1024.93 (2.83)
CG	W	14.61 (1.00)	6.05 (2.41)	3.12 (4.68)
	A	49.65 (1.00)	26.01 (1.91)	15.14 (3.28)
EP	W	33.36 (1.00)	16.74 (1.99)	8.45 (3.95)
	A	267.39 (1.00)	133.73 (2.00)	67.98 (3.93)
FT	W	6.07 (1.00)	3.20 (1.90)	1.85 (3.28)
	A	113.96 (1.00)	60.55 (1.88)	34.73 (3.28)
IS	W	0.76 (1.00)	0.47 (1.62)	0.38 (2.00)
	A(*1)	17.05 (1.00)	9.25 (1.84)	5.81 (2.93)
LU	W	194.90 (1.00)	101.42 (1.92)	54.43 (3.58)
	A	1810.94 (1.00)	775.63 (2.33)	411.07 (4.41)
MG	W	13.56 (1.00)	6.58 (2.06)	3.34 (4.06)
	A	101.29 (1.00)	50.68 (2.00)	26.67 (3.80)
SP	W	329.05 (1.00)	175.04 (1.88)	110.83 (2.97)
	A	2127.84 (1.00)	1157.58 (1.84)	762.07 (2.79)

Critique of OpenMP

- The easy cases work well; harder cases are probably much harder
- Requires that the semantics of sequential computation be preserved
  - Directly opposite of our thesis in this course that algorithms must be rethought
  - Compilers must enforce the sequentially consistent memory model
  - Limited abstractions

HPF: High Performance Fortran

- Philosophy
  - Automatic parallelization won't work
  - For data parallelism, what's important is data placement and data motion
  - Give the compiler help:
    - Extends Fortran with directives to guide data distribution
  - Allow slow migration from legacy codes
    - The directives are only hints
- Basic idea
  - Processors operate on only part of overall data
  - Directives say 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

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

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];
```

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

- Scalar code
  - Execute 4 insts 100 times, 2 Loads, 1 Add, 1 Store
- Advantages?

Rules 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);
```

  - One function call
  - Body of this function call can be easily vectorized

  - Lots of function calls inside a tight loop
  - Function call boundaries inhibit vectorization

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

Strategic Decisions (cont)

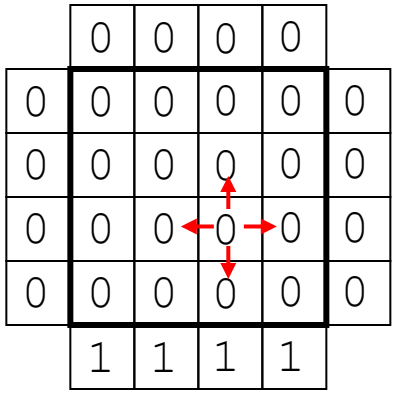
- A community project
  - Compiler directives don't change the program's semantics
  - They only affect performance
  - Allows different groups to conduct research on different aspects of the problem
  - Even the "little guy" can contribute

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

Example Computation

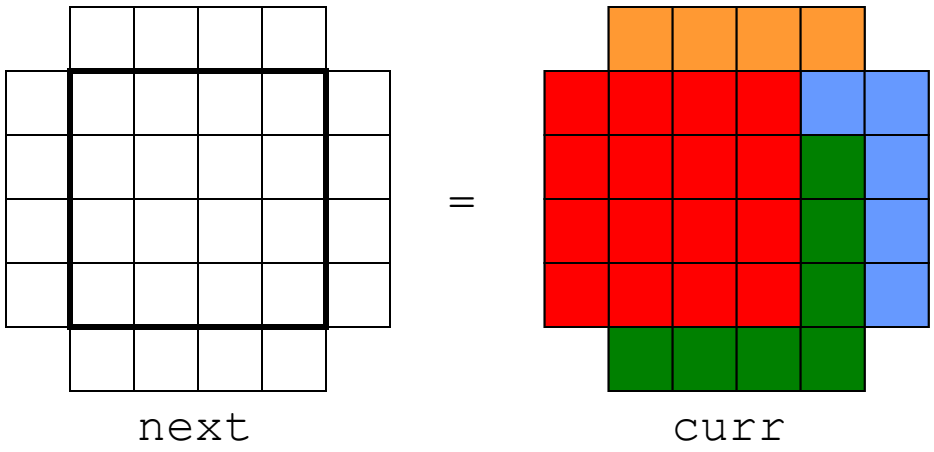
- Jacobi Iteration
  - The elements of an array, initialized to 0.0 except for 1.0's along its southern border, are iteratively replaced with the average of their 4 nearest neighbors until the greatest change between two iterations is less than some epsilon.



Jacobi Iteration in Fortran 90

- Example
  - The following statement computes the averaging step in the Jacobi iteration

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



Block Data Distribution

- Block distribution of 1D array

Keywords in caps

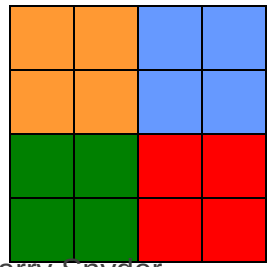
Number of virtual processors

Name of array

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

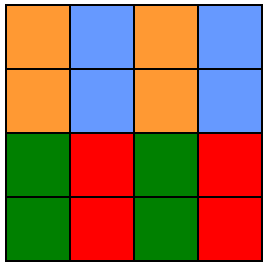
Block distribution of 2D array

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



Block-Cyclic Data Distribution

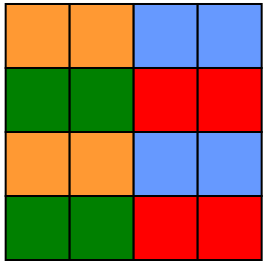
- Block-cyclic distribution



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

Block-cyclic distribution

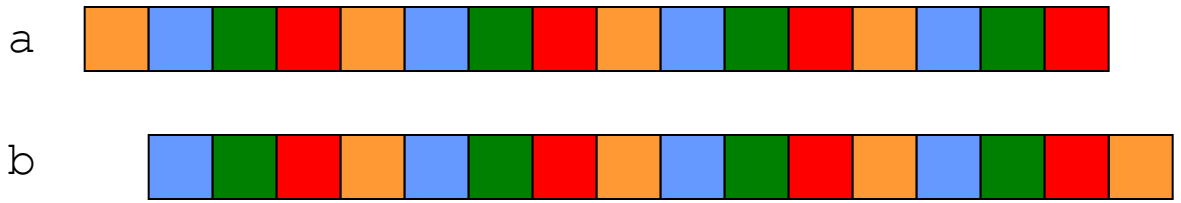
```
!HPF$   DISTRIBUTE array2D(CYCLIC, BLOCK) ONTO PROCS
```



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

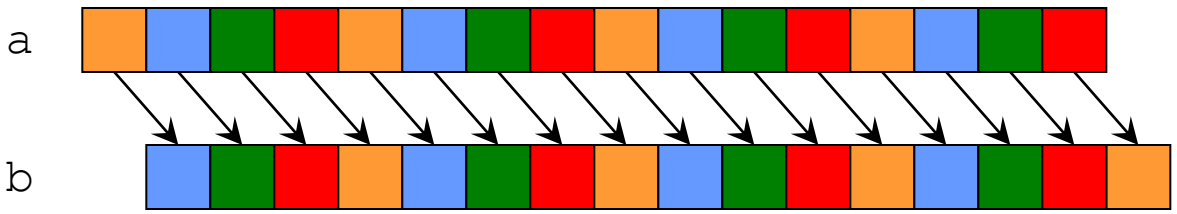
```
!HPF$  ALIGN a (i) WITH b(i-1)
```



Comm Implied by Distribution

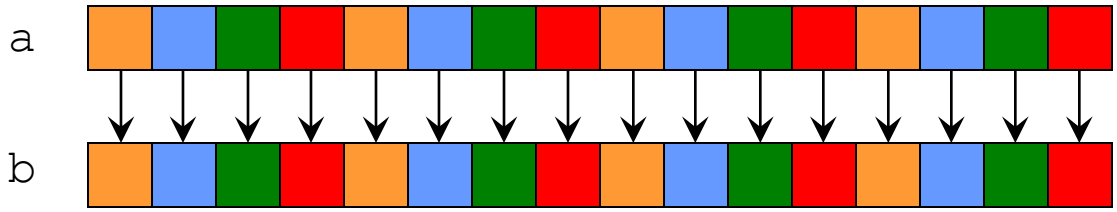
This alignment and assignment require all elements to be communicated to a different processor

```
!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)
```



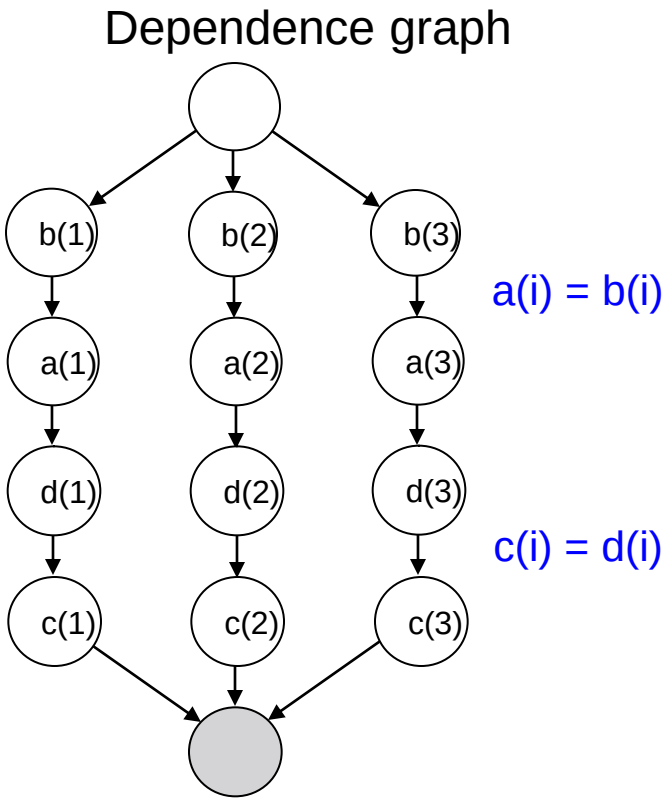
Break

Independent Loops

- INDEPENDENT directive
  - Loop iterations are independent
  - No implied barriers

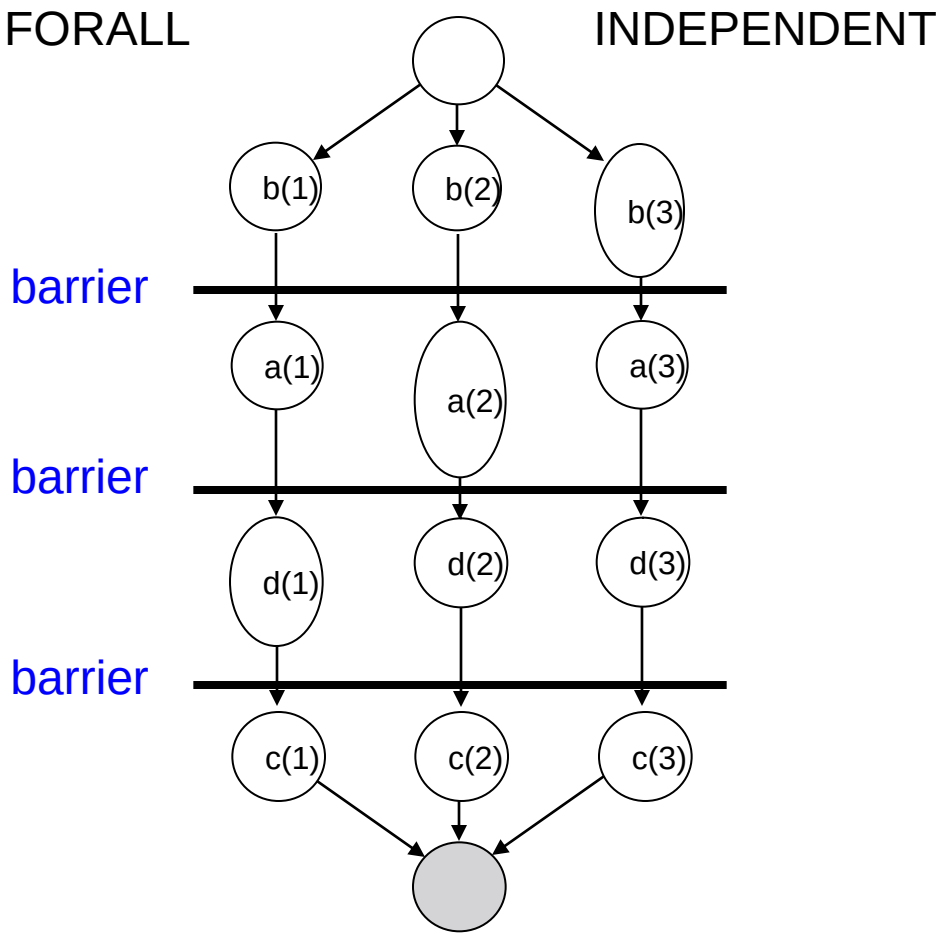
```
!HPF$ INDEPENDENT
DO (i = 1:3)
  a(i) = b(i)
  c(i) = d(i)
END DO
```

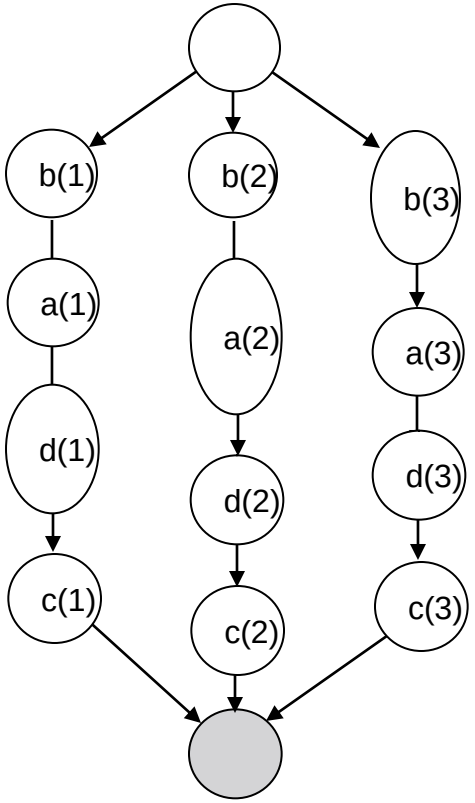
Fortran90 equivalent?
- None



FORALL Loops vs. Independent Loops

- Is there a difference?





Evaluation

- Your thoughts on HPF?
  - 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
  - Small changes in distribution can have large performance impact

Does the following code induce communication?

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

Evaluation (cont)

- No performance model
  - Complex language ⇒ 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

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
```

ZPL

- Philosophy
  - Provide performance portability for data-parallel programs
  - Allow users to reason about performance
  - Start from scratch
    - Parallel is fundamentally different from sequential
    - Be willing to **throw out** conveniences familiar to sequential programmers
- Basic idea
  - An array language
  - Implicitly parallel

ZPL History

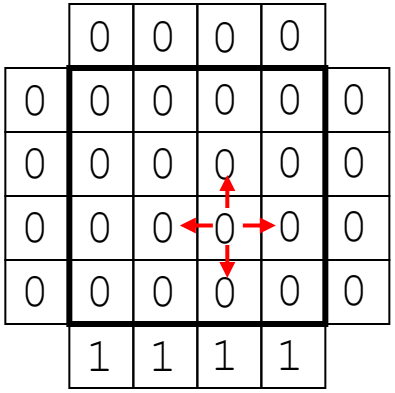
The beginning

- Designed by a small team beginning in 1993
- Compiler and runtime released in 1997

- Claims
  - Portable to any MIMD parallel computer
  - Performance comparable to C with message passing
  - Generally outperforms HPF
  - Convenient and intuitive

Recall Our 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.



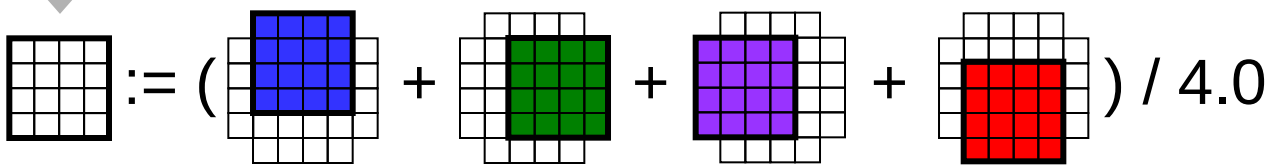
Jacobi Iteration– The Main Loop

```
program Jacobi;
config   var n : integer = 512;
       epsilon : float = 0.00001;
region         R = [1..n, 1..n];
var     A, Temp : [R] float;
```

Naming Convention:
Arrays begin with upper case letters
Scalars begin with lower case letters

Reductions:
max<< returns the maximum of an array expression

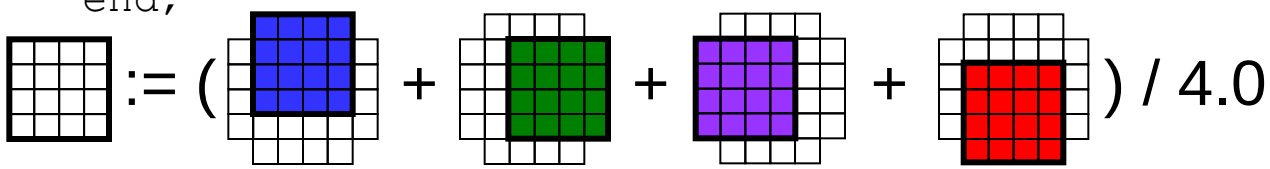
```
[north of R] A := 0.0; [west of R]  A := 1.0;
[east of R]  A := 0.0; [south of R] A := 0.0;
          repeat
            Temp := (A@north + A@east + A@west + A@south)/4.0;
            err  := max<< abs(Temp - A);
            A    := Temp;
          until err < epsilon;
        end;
end;
```



Jacobi Iteration– The Region

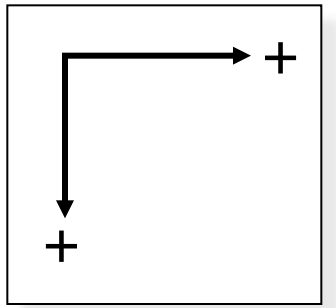
```
program Jacobi;
config   var n : integer = 512;
       epsilon : float = 0.00001;
region         R = [1..n, 1..n];
var      A, Temp : [R] float;
             err : float;
direction north = [-1, 0];      south = [ 1,  0];
          east  = [ 0, 1];      west  = [ 0, -1];
procedure Jacobi();
        [R] begin
              A := 0.0;
[north of R] A := 0.0; [west of R]  A := 1.0;
[east of R]  A := 0.0; [south of R] A := 0.0;
            repeat
              Temp := (A@north + A@east + A@west + A@south)/4.0;
              err  := max<< abs(Temp - A);
              A    := Temp;
            until err < epsilon;
          end;
end;
```

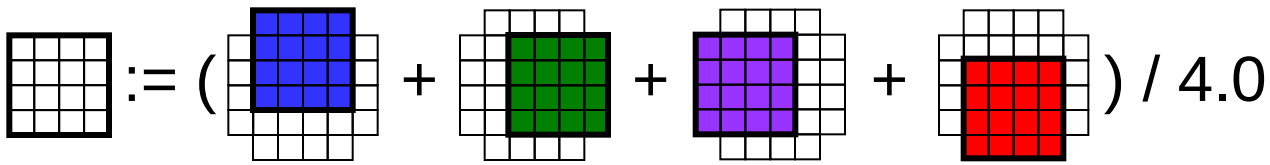
```
          end;
```



Jacobi Iteration– The Direction

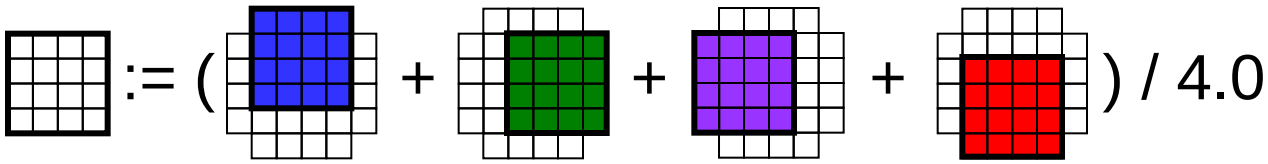
```
program Jacobi;
config   var n : integer = 512;
       epsilon : float = 0.00001;
region         R = [1..n, 1..n];
var     A, Temp : [R] float;
             err : float;
direction north = [-1, 0];     south = [ 1,  0];
          east  = [ 0, 1];     west  = [ 0, -1];
procedure Jacobi();
        [R] begin
              A := 0.0;
[north of R] A := 0.0; [west of R]  A := 1.0;
[east of R]  A := 0.0; [south of R] A := 0.0;
            repeat
              Temp := (A@north + A@east + A@west + A@south)/4.0;
              err  := max<< abs(Temp - A);
              A    := Temp;
            until err < epsilon;
          end;
end;
```

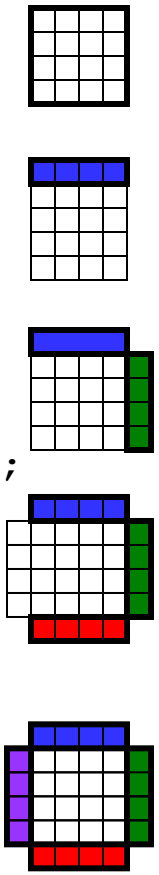




Jacobi Iteration– The Border

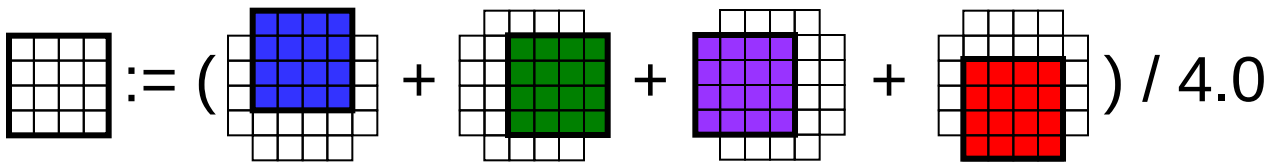
```
program Jacobi;
config   var n : integer = 512;
       epsilon : float = 0.00001;
region         R = [1..n, 1..n];
var     A, Temp : [R] float;
              err : float;
direction north = [-1, 0];      south = [ 1,  0];
          east  = [ 0, 1];      west  = [ 0, -1];
procedure Jacobi();
        [R] begin
              A := 0.0;
[north of R] A := 0.0;  [west of R]  A := 1.0;
[east of R]  A := 0.0;  [south of R] A := 0.0;
            repeat
              Temp := (A@north + A@east + A@west + A@south)/4.0;
              err  := max<< abs(Temp - A);
              A    := Temp;
            until err < epsilon;
          end;
end;
```





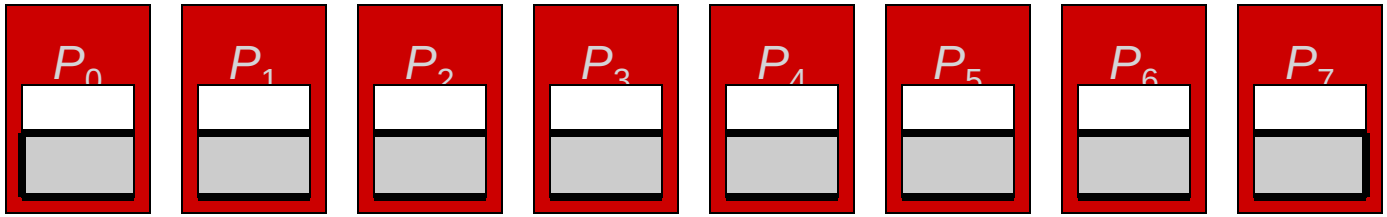
Jacobi Iteration– Remaining Details

```
program Jacobi;
config   var n : integer = 512;
       epsilon : float = 0.00001;
region         R = [1..n, 1..n];
var     A, Temp : [R] float;
              err : float;
direction north = [-1, 0];     south = [ 1,  0];
          east  = [ 0, 1];     west  = [ 0, -1];
procedure Jacobi();
        [R] begin
              A := 0.0;
[north of R] A := 0.0; [west of R]  A := 1.0;
[east of R]  A := 0.0; [south of R] A := 0.0;
            repeat
              Temp := (A@north + A@east + A@west + A@south)/4.0;
              err  := max<< abs(Temp - A);
              A    := Temp;
            until err < epsilon;
          end;
end;
```



Recent Notable Efforts: PGAS

- Greatest potential to assist programmer comes from hiding communication calls
  - Compilers can generate the calls
  - Need interface to specify which are local/global
- Concept: Partitioned Global Address Space
  - Overlay global addressing on separate memories
  - PGAS tends to use 1-sided comm as simplification



Extend Languages

- Three PGAS languages

CAF	UPC	Ti
Co-Array Fortran Numrich & Reed Extends Fortran	Universal Parallel C El Ghazawi, Carlson & Draper Extends C	Titanium Yelick Extends Java

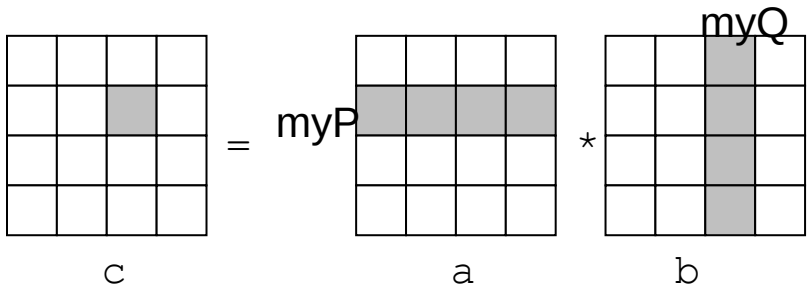
- Developed around 2000 +/- & Implemented
  - Similarities: GAS, comm handled by compiler/rt, programmer controls work/data assignment
  - Differences: Most everything else

Co-Array Fortran

- Incredibly elegant (for Fortran) extension

```
real, dimension(n,n)[p,*]:: a,b,c
...
do k=1,n
  do q=1,p
      c(i,j)[myP,myQ]=c(i,j)[myP,myQ]+a(i,k)[myP, q]*b(k,j)[q,myQ]
  enddo
enddo
```

Co-array



UPC

- Data can be allocated either shared or private; shared is assigned **cyclically** or **BC**
- Pointers are an issue

Property of pointer

Property of reference	Private	Shared
Private	Private-Private, `p1`	Private-Shared, `p2`
Shared	Shared-Private, `p3`	Shared-Shared, `p4`

```
int *p1;                  /* private ptr pointing locally */
shared int *p2;           /* private ptr pointing into shared space */
int *shared p3;           /* shared  ptr pointing locally */
shared int *shared p4; /* shared  ptr pointing into shared space */
```

UPC Code for Vector Sum

```
shared int v1[N], v2[N], v1v2sum[N];

void main()
{
    int i;
    shared int *p1, *p2;
    p1=v1;
    p2=v2;
    upc_forall(i=0; i<N; i++, p1++, p2++;i)
    {
        v1v2sum[i] = *p1 + *p2;
    }
}
```

Affinity

Titanium

- Java extensions including
  - "*regions*, which support safe, performance-oriented memory management as an alternative to garbage collection."
  - *foreach* is an unordered iteration, which logically raises the concurrency:

```
foreach ( … ) {    }
```

  - Used with the concept of a *point*, tuple of integers that range over a *domain*

Titanium Code for MM

```
public static void matMul(double [2d] a,
                          double [2d] b,
                          double [2d] c)
{
   foreach (ij in c.domain())
   {
      double [1d] aRowi = a.slice(1, ij[1]);
      double [1d] bColj = b.slice(2, ij[2]);
      foreach (k in aRowi.domain())
      {
         c[ij] += aRowi[k] * bColj[k];
      }
   }
}
```

Summarizing PGAS Languages

- The languages improve on the alternative--base language + MPI
- Compiler provides significant help, but the need to be attuned to subtle detail remains
- Deep issues
  - Global address space+private are good, but how they "play together" remains unclear
  - Better abstractions to reduce detail

New Parallel Languages

- DARPA has supported three new "high productivity" parallel languages
  - Is productivity really the issue?
  - Project coupled with design of a new machine
- The final competitors:
  - Cray's Cascade High Productivity Language, Chapel
  - IBM's X10
  - Sun's Fortress

Chapel

- Chapel is a multithreaded language supporting
  - Data ||ism, task ||ism, nested ||ism
  - Optimizations for locality of data and computation
  - Object oriented and generic programming techniques
  - Parallel implementation is nearing completion
- Designed for experts, production programmers

Chapel: 1D 4-ary FFT

```
for(str, span) in genDFTPhases(numElements, radix) {
  forall (bankStart, twidIndex) in (ADom by 2*span, 0..) {
    var wk2 = W(twidIndex),
        wk1 = W(2*twidIndex),
        wk3 = (wk1.re - 2 * wk2.im * wk1.im,
                2 * wk2.im * wk1.re - wk1.im):elemType;
     forall lo in bankStart + [0..str) do
        butterfly(wk1, wk2, wk3, A[[0..radix)*str + lo]);
     wk1 = W(2*twidIndex+1);
     wk3 = (wk1.re - 2 * wk2.re * wk1.im, 2 * wk2.re * wk1.re -
     wk1.im):elemType;
     wk2 *= 1.0i;
     forall lo in bankStart + span + [0..str) do
        butterfly(wk1, wk2, wk3, A[[0..radix]*str + lo]);
  }
}
```

Fortress

- Developed at Sun, Fortress pushes the envelop in expressivity
  - Focus on new programming ideas rather than parallel programming ideas: components and test framework assist with powerful compiler optimizations across libraries
  - Textual presentation important -- subscripts and superscripts -- mathematical forms
  - Transactions, locality specification, implicit ||ism
  - Extendibility

Fortress

- Conjugate gradient program in Fortress
- Features
  - := / =
  - Sequential
  - Mathematical

```
conjGrad[Elt extends Number, nat N,
         Mat extends Matrix [Elt, N×N],
         Vec extends Vector [Elt, N]
        ](A: Mat, x: Vec):(Vec, Elt)
cgit_max = 25
z : Vec = 0
r : Vec = x
p : Vec = r
r : Elt = r^T r
for j ← seq(1: cgit_max) do
  q = Ap
  α = ρ / (p^T q)
  z := z + α p
  r := r − α q
  ρ_0 = ρ
  ρ := r^T r
  β = ρ / ρ_0
  p := r + β p
end
(z, ||x − A z||)
```

X-10

- IBM's X10 is a type safe, distributed object oriented language in the PGAS family -- its "accessible to Java programmers"
- Many goodies including regions (a la ZPL), places (for locality), asynch, futures, foreach, ateach, atomic blocks and global manipulation of data structures

X-10 Jacobi Computation

```
public class Jacobi {
const int N=6;
const double epsilon = 0.002;
const double epsilon2 = 0.000000001;
const region R = [0:N+1, 0:N+1];
const region RInner= [1:N, 1:N];
const distribution D = distribution.factory.block(R);
const distribution DInner = D | RInner;
const distribution DBoundary = D - RInner;
const int EXPECTED ITERS=97;
const double EXPECTED ERR=0.0018673382039402497;
double[D] B = new double[D] (point p[i,j])
        {  return DBoundary.contains(p)
            ? (N-1)/2 : N*(i-1)+(j-1); };
public double read(final int i, final int j) {
      return future(D[i,j]) B[i,j].force(); }
public static void main(String args[]) {
      boolean b= (new Jacobi()).run();
      System.out.println("++++++ " + (b? "Test succeeded." :"Test failed."));
      System.exit(b?0:1);
  }
}
```

X-10 Jacobi (continued)

```
public boolean run()  {
  int iters = 0;
  double err;
  while(true) {
     double[.] Temp =
            new double[DInner] (point [i,j])
            {return (read(i+1,j)+read(i-1,j)
            +read(i,j+1)+read(i,j-1))/4.0; };
     if((err=((B | DInner) - Temp).abs().sum()) < epsilon)
          break;
     B.update(Temp);
     iters++;
   }
   System.out.println("Error="+err);
   System.out.println("Iterations="+iters);
   return Math.abs(err-EXPECTED ERR) < epsilon2 && iters==EXPECTED ITERS;
}
```

Summary

- Language is key tool to express parallelism
- State of the art is libraries –
  - threads, message passing, OpenMP
- There has been tremendous experimentation with alternative language approaches
  - ZPL, HPF, CAF, UPC, Titanium
- The next generation is here
  - Chapel, X10, Fortress

HW 6

- Using online research become familiar with a parallel programming language and critique it
  - NOT allowed: ZPL, Chapel, libraries
  - The critique must include a small code example
  - Relevant topics to discuss might include
    - Execution model (data parallel, task, etc.), mem model
    - Mechanisms for creating threads, communicating, etc.
    - Brief history, if known
    - Evidence of performance, scalability, portability, etc.
  - Any length OK, but ~2 pages is intended scale; refs