## Programming Models

- **A *programming model* is an abstraction that we program by writing instructions for**
- **Programming models are implemented in languages and libraries**
- **Implementations of the “standard” serial model of a CPU**
  - Assembly language
  - Language models
    - C
    - C++
    - Fortran
- **Implementations of various parallel models**
  - For shared memory:  OpenMP (C and Fortran versions), pthreads library
  - For multiple-memory systems:  Message Passing (MPI)
  - Hybrid models for hybrid systems

## Higher-Level Models

- **Parallel Languages**
  - UPC
  - Co-Array Fortran
  - Titanium
- **Abstract, declarative models**
  - Logic-based (Prolog)
  - Spreadsheet-based (Excel)
- **The programming model research problem:  Define a model (and language) that**
  - Can express complex computations
  - Can be implemented efficiently on parallel machines
  - Is easy to use
- **It is hard to get all three**
  - Specialized libraries can implement very high-level, even application-specific models

## Parallel Programming Models

- **Multiple classes of models differ in how we think about communication and synchronization among processes or threads.**
  - Shared memory
  - Distributed memory
  - Some of each
  - Less explicit
- **Shared Memory (really globally addressable)**
  - Processes (or threads) communicate through memory addresses accessible to each
- **Distributed memory**
  - Processes move data from one address space to another via sending and receiving messages
- **Multiple cores per node make the shared-memory model efficient and inexpensive; this trend encourages all shared-memory and hybrid models.**

## Writing Parallel Programs

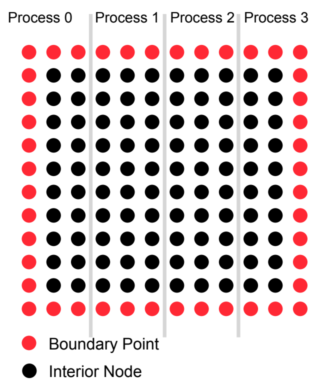
- **Parallel programming models are expressed:**
  - In libraries callable from conventional languages
  - In languages compiled by their own special compilers
  - In structured comments that modify the behavior of a conventional compiler
- **The new multicore chips are sparking a revolution in parallel programming languages and models**
  - OpenMP + MPI is one choice
  - MPI + ??? Is another
  - Or, a totally new paradigm/language
- **Here are some examples to get a feel for various languages**
  - (examples from Rusty Lusk, SC08 tutorial)

## The Poisson Problem

- **Simple elliptic partial differential equation**
- **Occurs in many physical problems**
  - Fluid flow, electrostatics, equilibrium heat flow
- **Many algorithms for solution**
- **We illustrate a sub-optimal one, since it is easy to understand and is typical of a data-parallel algorithm**

## Jacobi Iteration (Fortran Ordering)

- **Simple parallel data structure**



- Processes exchange columns with neighbors
- Local part declared as xlocal(m,0:n+1)

## Serial Fortran Version

```
real u(0:n,0:n), unew(0:n,0:n), f(1:n, 1:n), h

 ! Code to initialize f, u(0,*), u(n:*), u(*,0), and
 ! u(*,n) with g

 h = 1.0 / n
 do k=1, maxiter
   do j=1, n-1
     do i=1, n-1
       unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                      u(i,j+1) + u(i,j-1) - &
                      h * h * f(i,j) )
     enddo
   enddo
   ! code to check for convergence of unew to u.
   ! Make the new value the old value for the next iteration
   u = unew
 enddo
```

## MPI

- The Message-Passing Interface (MPI) is a standard library interface specified by the MPI Forum
- It implements the message passing model, in which the sending and receiving of messages combines both data movement and synchronization. Processes have separate address spaces.
- Send(data, destination, tag, comm) in one process matches Receive(data, source, tag, comm) in another process, at which time data is copied from one address space to another
- Data can be described in many flexible ways
- SendReceive can be used for exchange
- Callable from Fortran-77, Fortran-90, C, C++ as specified by the standard
  - Other bindings (Python, java) available, non-standard

## MPI Version

```
use mpi
 real u(0:n,js-1:je+1), unew(0:n,js-1:je+1)
 real f(1:n-1, js:je), h
 integer nbr_down, nbr_up, status(MPI_STATUS_SIZE), ierr

 ! Code to initialize f, u(0,*), u(n:*), u(*,0), and
 ! u(*,n) with g

 h = 1.0 / n
 do k=1, maxiter
   ! Send down
   call MPI_Sendrecv( u(1,js), n-1, MPI_REAL, nbr_down, k &
                  u(1,je+1), n-1, MPI_REAL, nbr_up, k, &
                  MPI_COMM_WORLD, status, ierr )
   ! Send up
   call MPI_Sendrecv( u(1,je), n-1, MPI_REAL, nbr_up, k+1, &
                  u(1,js-1), n-1, MPI_REAL, nbr_down, k+1,&
                  MPI_COMM_WORLD, status, ierr )
   do j=js, je
     do i=1, n-1
       unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                        u(i,j+1) + u(i,j-1) - &
                        h * h * f(i,j) )
     enddo
   enddo
   ! code to check for convergence of unew to u.
   ! Make the new value the old value for the next iteration
   u = unew
 enddo
```

## HPF

- **HPF is a specification for an extension to Fortran 90 that focuses on describing the distribution of data among processes in structured comments.**
- **Thus an HPF program is also a valid Fortran-90 program and can be run on a sequential computer**
- **All communication and synchronization if provided by the compiled code, and hidden from the programmer**

## HPF Version

```
  real u(0:n,0:n), unew(0:n,0:n), f(0:n, 0:n), h
!HPF$ DISTRIBUTE u(:,BLOCK)
!HPF$ ALIGN unew WITH u
!HPF$ ALIGN f WITH u

   ! Code to initialize f, u(0,*), u(n:*), u(*,0),
   ! and u(*,n) with g

   h = 1.0 / n
   do k=1, maxiter
     unew(1:n-1,1:n-1) = 0.25 * &
                ( u(2:n,1:n-1) + u(0:n-2,1:n-1) + &
                  u(1:n-1,2:n) + u(1:n-1,0:n-2) - &
                     h * h * f(1:n-1,1:n-1) )
     ! code to check for convergence of unew to u.
     ! Make the new value the old value for the next iteration

     u = unew
   enddo
```

## OpenMP

- OpenMP is a set of compiler directives (in comments, like HPF) and library calls
- The comments direct the execution of loops in parallel in a convenient way.
- Data placement is not controlled, so performance is hard to get except on machines with real shared memory

## OpenMP Version

```
   real u(0:n,0:n), unew(0:n,0:n), f(1:n-1, 1:n-1), h

   ! Code to initialize f, u(0,*), u(n:*), u(*,0),
   ! and u(*,n) with g

   h = 1.0 / n
   do k=1, maxiter
!$omp parallel
!$omp do
     do j=1, n-1
       do i=1, n-1
         unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                     u(i,j+1) + u(i,j-1) - &
                     h * h * f(i,j) )
       enddo
     enddo
!$omp enddo
     ! code to check for convergence of unew to u.

     ! Make the new value the old value for the next iteration
     u = unew
!$omp end parallel
   enddo
```

## The PGAS Languages

- **PGAS (Partitioned Global Address Space) languages attempt to combine the convenience of the global view of data with awareness of data locality, for performance**
  - Co-Array Fortran, an extension to Fortran-90)
  - UPC (Unified Parallel C), an extension to C
  - Titanium, a parallel version of Java

## Co-Array Fortran

- SPMD – Single program, multiple data
- Replicated to a number of <u>images</u>
- Images have indices 1,2, …
- Number of images fixed during execution
- Each image has its own set of <u>local</u> variables
- Images execute asynchronously except when explicitly synchronized
- Variables declared as co-arrays are accessible of another image through set of array subscripts, delimited by [ ] and mapped to image indices by the usual rule
- Intrinsics:  this_image, num_images, sync_all, sync_team, flush_memory, collectives such as co_sum
- <u>Critical</u> construct

## CAF Version

```
real u( 0:n,js-1:je+1,0:1)[*], f (0:n,js:je), h
integer np, myid, old, new
np = NUM_IMAGES()
myid = THIS_IMAGE()
new = 1
old = 1-new
! Code to initialize f, and the first and last columns of u on the extreme
! processors and the first and last row of u on all processors
 h = 1.0 / n
 do k=1, maxiter
  if (myid .lt. np) u(:,js-1,old)[myid+1] = u(:,je,old)
  if (myid .gt. 0) u(:,je+1,old)[myid-1] = u(:,js,old)
  call sync_all
  do j=js,je
    do i=1, n-1
      u(i,j,new) = 0.25 * ( u(i+1,j,old) + u(i-1,j,old) + &
                    u(i,j+1,old) + u(i,j-1,old) - &
                    h * h * f(i,j) )
    enddo
  enddo
  ! code to check for convergence of u(:,:,new) to u(:,:,old).
  ! Make the new value the old value for the next iteration
  new = old
  old = 1-new
enddo
```

## UPC

- UPC is an extension of C (not C++) with shared and local addresses

## UPC Version

```
#include <upc.h>
#define n 1024
shared [*] double u[n+1][n+1];
shared [*] double unew[n+1][n+1];
shared [*] double f[n][n];
int main() {
   int maxiter = 100;
   //  Code to initialize f, u(0,*), u(n:*), u(*,0), and
   //  u(*,n) with g
   double h = 1.0 / n;
   for (int k=0; k < maxiter; k++) {
     for (int i=1; i < n; i++) {
       upc_forall (int j=1; j < n; j++; &unew[i][j]) {
         unew[i][j] = 0.25 * ( u[i+1][j] + u[i-1][j] +
                          u[i][j+1] + u[i][j-1] -
                          h * h * f[i][j] );
       }
     }
     upc_barrier;
     // code to check for convergence of unew to u.
     // Make the new value the old value for the next iteration
     for (int i = 1; i < n; i++) {
       upc_forall(int j = 1; j < n; j++; &u[i][j]) {
         u[i][j] = unew[i][j];
       }
     }
   }
}
```

## Titanium

- **Titanium is a PGAS language based on Java**
  - Implementations do not use the JVM
- **We show both a serial and parallel version**

## Titanium Serial Version

```
public class Poisson_seq {

   public static void main (String[] argv) {
      int n = 10;        // grid side length of f grid
      int maxiter = 100;  // number of iterations

      double [2d] u = new double [[0,0]:[n+1,n+1]];
      double [2d] unew = new double [u.domain()];
      double [2d] f = new double [u.domain().shrink(1)];
      double [2d] temp;  // used for switching arrays

       // initialize u and f

      double h = 1.0/n;
      for (int i = 0; i < maxiter; i++) {
         foreach (p in unew.domain().shrink(1)) {
           // perform computation
           unew[p] = 0.25 * (u[p + [ 1, 0]] + u[p + [-1, 0]]
                                  + u[p + [ 0, 1]] + u[p + [0, -1]]
                                  - h * h * f[p]);
         }

         // swap u and unew
         temp = unew;
         unew = u;
         u = temp;
      }
   }
}
```

## Titanium Version – Part 1

```
public class Poisson_par {
  public static single void main (String[] argv) {
    int n = 10;         // grid side length of f (RHS) grid
    int single maxiter = 100;   // number of iterations

    RectDomain<2> myDomain = [[0, Ti.thisProc() * n / Ti.numProcs()] :
                        [n+1, (Ti.thisProc()+1)* n / Ti.numProcs()+ 1]];
    RectDomain<2> myInterior = myDomain.shrink(1);

    // create distributed array (auto-initialized to zero)
    double [1d][1d] single [2d] allu = new double [0:1][0:Ti.numProcs()-1] single [2d];
    allu[0].exchange(new double [myDomain]);
    allu[1].exchange(new double [myDomain]);

    // create & initialize f
    double [2d] f = new double [myInterior];
    f.set(1.0);

    double h = 1.0/n;
    for (int single i = 0; i < maxiter; i++) {
      // fetch reference to local arrays
      double [2d] local u = (double [2d] local)allu[0][Ti.thisProc()];
      double [2d] local unew = (double [2d] local)allu[1][Ti.thisProc()];
```

## Titanium Version – Part 2

```
      // update ghost cells
      if (Ti.thisProc() > 0)
        allu[0][Ti.thisProc()-1].copy(u.restrict(myInterior));
      if (Ti.thisProc()+1 < Ti.numProcs())
        allu[0][Ti.thisProc()+1].copy(u.restrict(myInterior));
      Ti.barrier();

      // perform computation
      foreach (p in myInterior) {
        unew[p] = 0.25 * (u[p + [ 1, 0]] + u[p + [-1, 0]]
                        + u[p + [ 0, 1]] + u[p + [0, -1]]
                         - h * h * f[p]);
      }
      // swap u and unew
      double [1d] single [2d] temp = allu[0];
      allu[0] = allu[1];
      allu[1] = temp;
    }
  }
}
```

## Global Operations

- **Example: checking for convergence**

## Serial Version

```
real u(0:n,0:n), unew(0:n,0:n), twonorm

! ...
  twonorm = 0.0
  do j=1, n-1
    do i=1, n-1
      twonorm = twonorm + (unew(i,j) - u(i,j))**2
    enddo
  enddo
  twonorm = sqrt(twonorm)
  if (twonorm .le. tol) ! ... declare convergence
```

## MPI Version

```
use mpi
real u(0:n,js-1:je+1), unew(0:n,js-1:je+1), twonorm
integer ierr

! ...

twonorm_local = 0.0
  do j=js, je
    do i=1, n-1
      twonorm_local = twonorm_local + &
                (unew(i,j) - u(i,j))**2
    enddo
  enddo
  call MPI_Allreduce( twonorm_local, twonorm, 1, &
            MPI_REAL, MPI_SUMM, MPI_COMM_WORLD, ierr )
  twonorm = sqrt(twonorm)
  if (twonorm .le. tol) ! ... declare convergence
```

## HPF Version

```
   real u(0:n,0:n), unew(0:n,0:n), twonorm
!HPF$ DISTRIBUTE u(:,BLOCK)
!HPF$ ALIGN unew with u
!HPF$ ALIGN f with u

 ! ...
     twonorm = sqrt ( &
          sum ( (unew(1:n-1,1:n-1) - u(1:n-1,1:n-1))**2) )
     if (twonorm .le. tol) ! ... declare convergence
   enddo
```

## OpenMP Version

```
real u(0:n,0:n), unew(0:n,0:n), twonorm

  ! ..
    twonorm = 0.0
!$omp parallel
!$omp do private(ldiff) reduction(+:twonorm)
    do j=1, n-1
      do i=1, n-1
        ldiff = (unew(i,j) - u(i,j))**2
        twonorm = twonorm + ldiff
      enddo
    enddo
!$omp enddo
!$omp end parallel
    twonorm = sqrt(twonorm)
  enddo
```

## The HPCS languages

- **DARPA funded three vendors to develop next-generation languages for programming next-generation petaflops computers**
  - Fortress (Sun)
  - X10 (IBM)
  - Chapel (Cray)
- **All are global-view languages, but also with some notion for expressing locality, for performance reasons.**
  - They are more abstract than UPC and CAF in that they do not have a fixed number of processes.
- **Sun's DARPA funding was discontinued, and the Fortress project made public. See http://fortressproject.sun.com**
- **Work continues at Cray and IBM**

## OpenCL

- A new standard Platform for Heterogeneous Parallel Computers
- For programming GPUs, CPUs, etc. in one model
- Supports data- and task- parallel compute models
- Based on C
- See upcoming tutorials by Tim Mattson, and the OpenCL Working Group, et al.