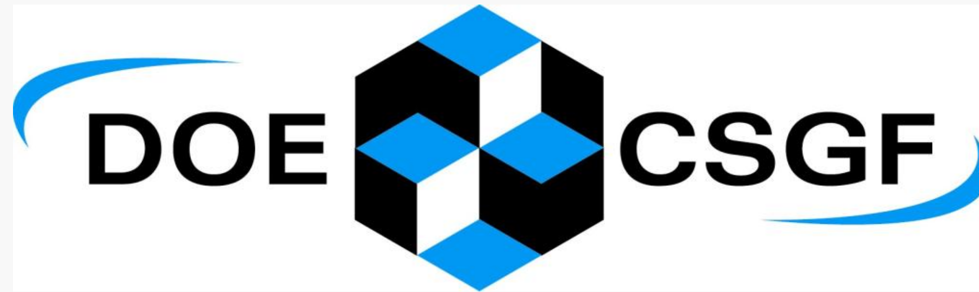


Programming Models for Scientific Computing

William Gropp

Programming Models

- Not just parallel programming
  - And not just "classical" programming languages – python, Matlab, multi-lingual programs
- (At least) Two goals
  - Clear, maintainable programs
    - —"Productivity"
  - Performance
    - —Otherwise, you don't need parallelism
- One more requirement
  - Interoperability with components (library routines) written in other languages
- Most parallel programming models consist of
  - A conventional single-threaded model
  - A parallel coordination layer

Single Threaded Languages

- Fortran, C, C++ (and many others)
  - No intrinsic parallelism
  - Do provide some features for memory hierarchies
- Programming for memory hierarchy
  - These provide some simple tools that can help the compiler produce better-performing code
- In C/C++
  - const – Data is not changed
  - restrict (pointers) – roughly, data is not accessed with a different pointer
- In Fortran
  - Arguments to routines are *required* to be distinct
    - As if they had C's restrict semantics
    - One of the reasons that Fortran is considered easier to optimize than C
  - Fortran provides intent as well (IN, OUT, INOUT). IN can help the compiler
- You should *always* use the correct declaration
  - Compilers continue to improve and to exploit this knowledge
  - Compiler may also check whether you told the truth
- One more issue - Aligned memory
  - Some special features require that operands be aligned on double-word (e.g., 16-byte) boundaries

Example: Using const and restrict

- Void daxpy(int n, const double * restrict x,
  double * restrict y, double a)

```
{
register int i;
for (i=0; i<n; i++)
    y[i] = a * x[i] + y[i];
}
```

- Using const and restrict allows a high-quality compiler to perform optimizations that *are not correct* when const and restrict are not present.
- Don’t let experiences with old, bad compilers convince you to ignore these
  - Factor of two benefits are possible

Illustrating the Programming Models

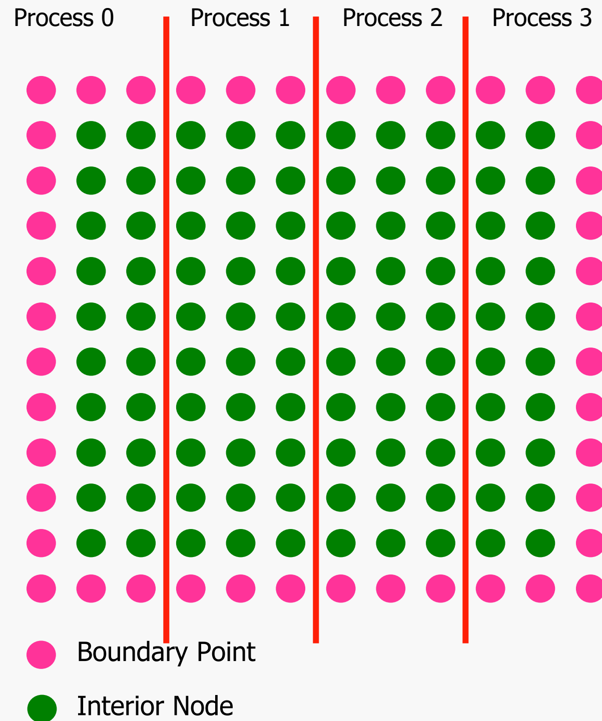
- Learning each programming model takes more than an hour ☺
  - This section will show samples of programming models, applied to one simple operation (matrix-vector multiply on a regular grid)
  - For more information, consider
    - —Tutorials (e.g., at SC09)
    - —Taking a parallel programming class
    - —Reading books on different models

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
```

Adding SMP Parallelism

- Threads
  - A thread is an address space with a program counter.
  - A process contains one or more threads
  - Libraries such as pthreads (for POSIX threads) allow you to create and manipulate new threads
- Threads are a very complex and error-prone programming model.
  - It is better, when possible, to use a higher level SMP programming model…

OpenMP

- OpenMP provides both
  - Loop parallelism
  - Task parallelism
- OpenMP is a set of compiler directives (in comments) 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
- There are also subtle memory consistency issues (i.e., when is data written by one thread visible to another)

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
```

Distributed Shared Memory Parallelism

- Three major approaches
- Two-sided message passing (MPI-1)
  - Point-to-point
  - Collective
- One-sided (or Put/Get) models
  - SHMEM, Global Arrays, MPI-2 RMA
- Languages supporting distributed data structures (PGAS)
  - HPF (historical), UPC, CoArray Fortran
  - Future directions: ZPL (historical), Chapel, X10, Fortress

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

DOE CSGF

MPI Version – Two Sided

```
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
```

MPI Version – One Sided

```
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
real (kind=MPI_ADDRESS_KIND) off
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
! Code to create the MPI memory window
h = 1.0 / n
do k=1, maxiter
  ! Send downcall MPI_Win_fence( win, ierr )
  off = (je-js+2)*(n+1)
  call MPI_Put( u(1,js), n-1, MPI_REAL, nbr_down, off, n-1, MPI_REAL, win, ierr )
  ! Send up
  off = 0
  call MPI_Put( u(1,je), n-1, MPI_REAL, nbr_up, off, n-1, MPI_REAL, win, ierr )
  call MPI_Win_fence( win, 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
```

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 images
- Images have indices 1,2, …
- Number of images fixed during execution
- Each image has its own set of local 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
- A version of Co_Array Fortran is a part of the Fortran 2008 draft standard (mostly a subset of the full CAF)

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 pointers
- Provides distributed arrays
  - "Global view" simplifies programming
- Provides a "Put-Get" model of programming
  - You can dereference a shared pointer
- No standard, but an active community and multiple implementations, including some "vendor" implementations

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];
       }
     }
   }
}
```

Global Operations

- Some operations involve all processes
  - Or a collection, such as all processes in a “row” or “column”
- Example: checking for convergence
  - Need to compute the two-norm of the difference between successive iterations for the *entire d*ata array

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
```

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
```

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
```

The HPCS languages

- DARPA funded three vendors to develop next-generation languages for programming next-generation petaflops computers
  - Fortress (Sun) http://projectfortress.sun.com/Projects/Community
  - X10 (IBM) http://x10-lang.org/
  - Chapel (Cray) http://chapel.cs.washington.edu/
- 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.

Other Programming Approaches

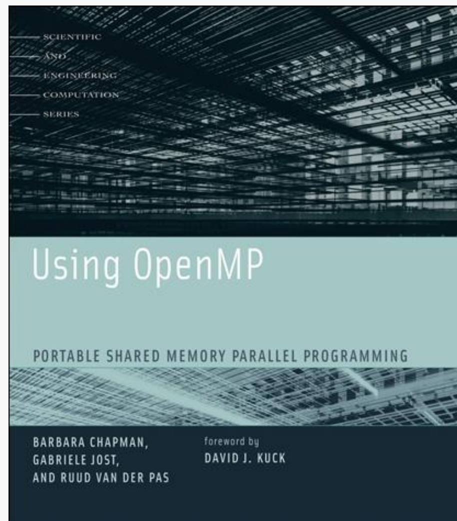
- Programming accelerators (GPGPUs)
  - Emerging (but low-level) languages such as CUDA, OpenCL
- Integrated Development Environments (Eclipse)
  - Powerful but with a high learning curve
- Automating code optimization and tuning (Annotations, Autotuning)
  - A number of research tools, particularly at the DOE Labs, but not generally ready for average users

Advice

- Short Term Projects
  - Use whatever best fits your application
    - —This might be libraries that provide all the needed methods
  - CAF good for regular data structures; UPC also has some good support
  - If you use a PGAS language, make sure
    - —You have access to CAF or UPC
    - —The environments produce efficient code
- Long Term Projects
  - MPI, possibly mixed with OpenMP, remains the reliable and efficient standard
  - But make use of existing software components as much as possible

Books

- Using MPI
- Using OpenMP
- Performance Optimization of Numerically Intensive Codes



Coming Up

- Scalable and Efficient Algorithms
  - Combines applied mathematics with architecture influences
    - —Picking algorithms with maximum FLOPS often wrong
    - —Picking algorithms with minimum FLOPs often wrong
  - David Keyes presents an overview of HPC algorithms next