Lecture 21: Parallel Programming Models for Scientific Computing

William Gropp

www.cs.illinois.edu/~wgropp
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.
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
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

• We will survey some of each of these and consider a single example written in each
  ♦ Not an adequate tutorial on any of these approaches
  ♦ Many detailed sources are available
  ♦ Only trying to convey the “flavor” of each approach
Programming Models and Systems

• 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 systems consist of
  ♦ A conventional single-threaded model
  ♦ A parallel coordination layer
Single Threaded Languages

- **Fortran, C, C++ (and many others)**
  - No intrinsic parallelism until recently (C11 threads, Fortran coArrays)
  - 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
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 (sparse matrix-vector multiply on a regular grid)

♦ For more information, consider
  • Tutorials (e.g., at SC)
  • Taking a parallel programming class covering a specific programming model
  • 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 $x_{local}(n,0:m+1)$
The Computation

• These details are not important to showing the different programming systems, but may make some things clearer

• Approximation is

\[
\frac{u(x + h, y) + u(x - h, y) - 4u(x, y) + u(x, y + h) + u(x, y - h)}{h^2} = f(x, y)
\]

\[
4u(x, y) = (u(x + h, y) + u(x - h, y) + u(x, y + h) + u(x, y - h)) - h^2 f(x, y)
\]

\[
u(x, y+h) \text{ becomes } u(i,j+1) \text{ etc.}
\]
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
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

- We’ve seen how to use OpenMP for data parallelism in lecture17
- Here we’ll see it in Fortran
  - Since Fortran has no anonymous blocks, special comments (directives) are used to mark the blocks
- Note 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
     !$omp end do
     !$omp end parallel
     ! 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
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, C (and hence C++) as specified by the standard.
  - Other bindings (Python, Java) available, non-standard.
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

• No longer in much use, though some variations in use in Japan
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
endo
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
  - Co-Array Fortran, an extension to Fortran-90, and now part of Fortran 2008
  - 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
- Critical 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 with shared and local addresses
- Provides some simple distributions, similar to HPF
- Available on some large-scale systems
  - Including Blue Waters
```c
#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

• Example: checking for convergence
• In our case, it means computing

$$\left\| \mathbf{u} - \mathbf{u}_{\text{new}} \right\|^2_2$$

Locally, sum((u(i,j)-unew(i,j))**2)

• Then sum up the contributions across each processing element (node/process/thread)
• Often called a “global” sum
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
endo
dodo
twonorm = sqrt(twonorm)
if (twonorm .le. tol) ! ... declare convergence
use mpi
real u(0:n,js-1:je+1), unew(0:n,js-1:je+1), twonorm, &
twonorm_local
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_SUM, 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, ldiff

!..
    twonorm = 0.0
!$omp parallel
!$omp do private(ldiff,i) 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
!$omp end do
!$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, before Sun acquired by Oracle)
  ♦ 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://projectfortress.java.net
• Work continues at Cray (chapel.cray.com) and IBM (x10-lang.org)
Other Issues and Approaches

• Programming accelerators (GPGPUs)
  ♦ OpenCL, OpenACC, CUDA
• Annotations, autotuning
• Domain Specific Languages (DSLs) and embedded DSLs
• Integrated Development Environments (Eclipse)
• Automating code optimization and tuning (Annotations, Autotuning)
For More Information

- Using MPI (3rd edition)
- Using Advanced MPI
- Using OpenMP
To Find Out...

- Find out what programming systems are available on your platforms. Look for
  - MPI
  - UPC
  - CoArray Fortran (as a separate language)
  - Fortran 2008 including coArrays
  - SHMEM