11 Parallel programming models

Many different models for expressing parallelism in programming languages

- **Actor model**
  - Erlang
  - Scala

- **Coordination languages**
  - Linda

- **CSP-based (Communicating Sequential Processes)**
  - FortranM
  - Occam

- **Dataflow**
  - SISAL (Streams and Iteration in a Single Assignment Language)

- **Distributed**
  - Sequoia
  - Bloom

- **Event-driven and hardware description**
10.3 Assessing parallel programs

- Verilog hardware description language (HDL)

- **Functional**
  - Concurrent Haskell

- **GPU languages**
  - CUDA
  - OpenMP
  - OpenACC
  - OpenHMPP (HMPP for Hybrid Multicore Parallel Programming)

- **Logic programming**
  - Parlog

- **Multi-threaded**
  - Clojure

- **Object-oriented**
  - Charm++
  - Smalltalk

- **Message-passing**
  - MPI
  - PVM

- **Partitioned global address space (PGAS)**
  - High Performance Fortran (HPF)
11.1 HPF

Partitioned global address space parallel programming model
Fortran90 extension

- SPMD (*Single Program Multiple Data*) model
- each process operates with its own part of data
- HPF commands specify which processor gets which part of the data
- Concurrency is defined by HPF commands based on Fortran90
- HPF directives as comments:
  \[ !HPF$ \ <\text{directive}> \]
HPF example: matrix multiplication

PROGRAM ABmult
    IMPLICIT NONE
    INTEGER, PARAMETER :: N = 100
    INTEGER, DIMENSION (N,N) :: A, B, C
    INTEGER :: i, j
! HPF$ PROCESSORS square (2,2)
! HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: C
! HPF$ ALIGN A(i,*) WITH C(i,*)
! replicate copies of row A(i,*) onto proc.s which compute C(i,j)
! HPF$ ALIGN B(*,j) WITH C(*,j)
! replicate copies of col. B(*,j)) onto proc.s which compute C(i,j)
    A = 1
    B = 2
    C = 0
    DO i = 1, N
        DO j = 1, N
            ! All the work is local due to ALIGNs
            C(i,j) = DOT_PRODUCT(A(i,:), B(:,j))
        ENDDO
    ENDDO
    WRITE(*,*) C
END
HPF programming methodology

- Need to find balance between concurrency and communication
- The more processes the more communication
- Aiming to
  - Find balanced load based from the **owner calculates** rule
  - Data locality

**Easy to write a program in HPF** but **difficult to gain good efficiency**

Programming in HPF technique is more or less like this:

1. Write a correctly working serial program, test and debug it
2. Add distribution directives introducing as less as possible communication
OpenMP tutorial (http://www.llnl.gov/computing/tutorials/openMP/)

- Programming model based on thread parallelism
- Helping tool for a programmer
- Based on compiler directives
  - C/C++, Fortran*
- Nested Parallelism is supported
  (though, not all implementations support it)
- Dynamic threads
- OpenMP has become a standard
- **Fork-Join** model
OpenMP Example: Matrix Multiplication:


C******************************************************************************************************************
C OpenMp Example – Matrix Multiply – Fortran Version
C FILE: omp_mm.f
C DESCRIPTION:
C Demonstrates a matrix multiply using OpenMP. Threads share row iterations
C according to a predefined chunk size.
C LAST REVISED: 1/5/04 Blaise Barney
C******************************************************************************************************************

PROGRAM MATMULT

INTEGER NRA, NCA, NCB, TID, NTHREADS, I, J, K, CHUNK,
+ OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
C number of rows in matrix A
PARAMETER (NRA=62)
C number of columns in matrix A
PARAMETER (NCA=15)
C number of columns in matrix B
PARAMETER (NCB=7)

REAL*8 A(NRA,NCA), B(NCA,NCB), C(NRA,NCB)
Set loop iteration chunk size
CHUNK = 10

Spawn a parallel region explicitly scoping all variables
!$OMP PARALLEL SHARED(A,B,C,NTHREADS,CHUNK) PRIVATE(TID,I,J,K)

TID = OMP_GET_THREAD_NUM()
IF (TID .EQ. 0) THEN
   NTHREADS = OMP_GET_NUM_THREADS()
   PRINT *, 'Starting matrix multiple example with', NTHREADS,
   'threads'
   PRINT *, 'Initializing matrices'
END IF

Initialize matrices
!$OMP DO SCHEDULE(STATIC, CHUNK)
DO 30 I=1, NRA
   DO 30 J=1, NCA
      A(I,J) = (I-1)+(J-1)
   30 CONTINUE
!$OMP DO SCHEDULE(STATIC, CHUNK)
DO 40 I=1, NCA
   DO 40 J=1, NCB
      B(I,J) = (I-1)*(J-1)
   40 CONTINUE


```c
!$OMP DO SCHEDULE(STATIC, CHUNK)
   DO 50 I=1, NRA
   DO 50 J=1, NCB
       C(I,J) = 0
      50 CONTINUE

C Do matrix multiply sharing iterations on outer loop
C Display who does which iterations for demonstration purposes
PRINT *, 'Thread', TID, 'starting matrix multiply ...

!$OMP DO SCHEDULE(STATIC, CHUNK)
   DO 60 I=1, NRA
      PRINT *, 'Thread', TID, 'did row', I
   DO 60 J=1, NCB
       DO 60 K=1, NCA
           C(I,J) = C(I,J) + A(I,K) * B(K,J)
       60 CONTINUE

C End of parallel region
!$OMP END PARALLEL

C Print results
PRINT *, '*******************************
PRINT *, 'Result Matrix: '
DO 90 I=1, NRA
```

DO 80 J = 1, NCB
   WRITE(*,70) C(I,J)
70 FORMAT(2x,f8.2,$)
80 CONTINUE
PRINT *, '           
90 CONTINUE
PRINT *, '************************************************************
PRINT *, 'Done.'

END
11.3 GPGPU

General-purpose computing on graphics processing units

11.3.1 CUDA

- Parallel programming platform and model created by NVIDIA
- CUDA gives access to
  - virtual instruction set
  - memory on parallel computational elements on GPUs
- based on executing a large number of threads simultaneously
- CUDA-accelerated libraries
  - CUDA-extended programming languages (like nvcc)
    - C, C++, Fortran
CUDA Fortran Matrix Multiplication example (for more information, see http://www.pgroup.com/lit/articles/insider/v1n3a2.htm)

Host (CPU) code:

```fortran
subroutine mmul( A, B, C )
    use cudafor
    real, dimension(:, :) :: A, B, C
    integer :: N, M, L
    real, device, allocatable, dimension(:, :) :: Adev, Bdev, Cdev
    type(dim3) :: dimGrid, dimBlock
    N = size(A,1); M = size(A,2); L = size(B,2)
    allocate( Adev(N,M), Bdev(M,L), Cdev(N,L) )
    Adev = A(1:N,1:M)
    Bdev = B(1:M,1:L)
    dimGrid = dim3(N/16, L/16, 1)
    dimBlock = dim3(16, 16, 1)
    call mmul_kernel<<<dimGrid, dimBlock>>>( Adev, Bdev, Cdev, N,M,L )
    C(1:N,1:M) = Cdev
deallocate( Adev, Bdev, Cdev )
end subroutine
```
GPU code:

```
attributes(global) subroutine MMUL_KERNEL( A,B,C,N,M,L)
    real, device :: A(N,M),B(M,L),C(N,L)
    integer, value :: N,M,L
    integer :: i,j,kb,k,tx,ty
    real, shared :: Ab(16,16), Bb(16,16)
    real :: Cij
    tx = threadIdx%x ; ty = threadIdx%y
    i = (blockIdx%x-1) * 16 + tx
    j = (blockIdx%y-1) * 16 + ty
    Cij = 0.0
    do kb = 1, M, 16
        ! Fetch one element each into Ab and Bb; note that 16x16 = 256
        ! threads in this thread-block are fetching separate elements
        ! of Ab and Bb
        Ab(tx,ty) = A(i,kb+ty-1)
        Bb(tx,ty) = B(kb+tx-1,j)
        ! Wait until all elements of Ab and Bb are filled
        call syncthreads()
        do k = 1, 16
            Cij = Cij + Ab(tx,k) * Bb(k,ty)
        enddo
        ! Wait until all threads in the thread-block finish with
```
11.3.2 OpenCL

Open Computing Language (OpenCL):

- Heterogeneous systems of
  
  - CPUs (central processing units)
  - GPUs (graphics processing units)
  - DSPs (digital signal processors)
  - FPGAs (field-programmable gate arrays)
  - and other processors
• language (based on C99)
  – kernels (functions that execute on OpenCL devices)
  – plus application programming interfaces (APIs)
  – ( fortrancl )

• parallel computing using
  – task-based and data-based parallelism
  – GPGPU

• open standard by Khronos Group (Apple, AMD, IBM, Intel and Nvidia)
  – + adopted by Altera, Samsung, Vivante and ARM Holdings

Example: Matrix Multiplication
/* kernel.cl
 * Matrix multiplication: C = A * B. (( Device code ))
 */

// OpenCL Kernel
__kernel void
matrixMul(__global float* C,
          __global float* A,
          __global float* B,
          int wA, int wB)
{
    // 2D Thread ID
    // Old CUDA code
    // int tx = blockIdx.x * TILE_SIZE + threadIdx.x;
    // int ty = blockIdx.y * TILE_SIZE + threadIdx.y;
    int tx = get_global_id(0);
    int ty = get_global_id(1);

    // value stores the element that is
    // computed by the thread
    float value = 0;

    for (int k = 0; k < wA; ++k)
    {
        float elementA = A[ty * wA + k];
        float elementB = B[k * wB + tx];
11.3.3 OpenACC

- standard (Cray, CAPS, Nvidia and PGI) to simplify parallel programming of heterogeneous CPU/GPU systems
- using annotations (like OpenMP), C, C++, Fortran source code
- code started on both CPU and GPU automatically
- OpenACC to merge into OpenMP in a future release of OpenMP
! A simple OpenACC kernel for Matrix Multiplication

! $acc kernels

    do k = 1, n1
        do i = 1, n3
            c(i, k) = 0.0
            do j = 1, n2
                c(i, k) = c(i, k) + a(i, j) * b(j, k)
            enddo
        enddo
    enddo

! $acc end kernels


program matrix_multiply
    use omp_lib
    use openacc
    implicit none
    integer :: i, j, k, myid, m, n, compiled_for, option
    integer, parameter :: fd = 11
    integer :: t1, t2, dt, count_rate, count_max
    real, allocatable, dimension(:, :) :: a, b, c
real :: tmp, secs

open(fd, file='wallclocktime', form='formatted')
option = compiled_for(fd) ! 1−serial, 2−OpenMP, 3−OpenACC, 4−both

!omp parallel
!$  myid = OMP_GET_THREAD_NUM()
!$  if (myid.eq.0) then
!$    write(fd,"( 'Number of procs is ',i4')") OMP_GET_NUM_THREADS()
!$  endif
!omp end parallel

call system_clock(count_max=count_max, count_rate=count_rate)
do m=1,4  ! compute for different size matrix multiplies
call system_clock(t1)
n = 1000*2**(m-1)  ! 1000, 2000, 4000, 8000
allocate( a(n,n), b(n,n), c(n,n) )

! Initialize matrices

do  j=1,n
  do  i=1,n
    a(i,j) = real(i + j)
    b(i,j) = real(i − j)
  enddo
endo
do m=1,4  ! compute for different size matrix multiplies
   call system_clock(t1)
n = 1000*2**(m-1)  ! 1000, 2000, 4000, 8000
allocate( a(n,n), b(n,n), c(n,n) )

!omp parallel do shared(a,b,c,n,tmp) reduction(+: tmp)
!acc data copyin(a,b) copy(c)
!acc kernels
Compute matrix multiplication.

```fortran
! Enables ACC parallelism for k-loop
do k=1,n
  tmp = tmp + a(i,k) * b(k,j)
enddo

C(i,j) = tmp
enddo
```

! $acc end kernels
! $acc end data
! $omp end parallel do

call system_clock(t2)
dt = t2 - t1
secs = real(dt)/real(count_rate)
write(fd, "( 'For n= ', i4 , ', wall clock time is ', f12.2 , ' seconds ')") &
n, secs
deallocate(a, b, c)
enddo
close(fd)
end program matrix_multiply

integer function compiled_for(fd)
implicit none
integer :: fd

#if defined _OPENMP && defined _OPENACC
    compiled_for = 4
    write(fd, "('This code is compiled with OpenMP & OpenACC')")
#elif defined _OPENACC
    compiled_for = 3
    write(fd, "('This code is compiled with OpenACC')")
#elif defined _OPENMP
    compiled_for = 2
    write(fd, "('This code is compiled with OpenMP')")
#else
    compiled_for = 1
    write(fd, "('This code is compiled for serial operations')")
#endif
end function compiled_for