SHARED MEMORY AND ACCELERATOR PROGRAMMING

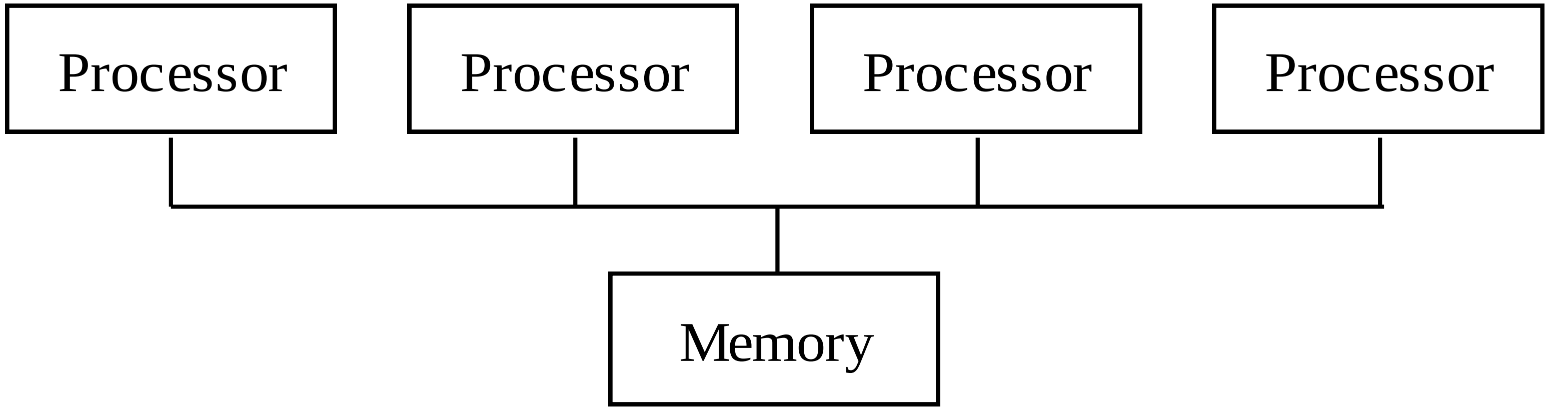
University of Virginia | Research Computing

SHARED MEMORY PROGRAMMING

SHARED-MEMORY PROCESSORS

- General system memory is shared by all the cores on a computer/node.
- Programming model is subprocesses, known as *threads.*
- Master process starts and controls subprocesses.
- Threads are created/destroyed as needed
- Each thread has a set of private variables. Other variables are shared by all threads.

SHARED-MEMORY MODEL



Processors interact and synchronize with each other through *shared* variables.

TYPES OF PARALLELISM

- Embarrassingly parallel (high-throughput computing)
  - Independent processes with little (or no) need to communicate.
- Data parallelism
  - Divide the data into smaller parts. Work on each part individually, then if necessary collect results and go to next phase.
- Task parallelism
  - Perform multiple tasks at the same time on the data.

DATA PARALLELISM

- Independent tasks apply same operation to different elements of a data set.
- Usually expressed as a loop.

```
for (i=0;i<imax;i++) {
       a[i]=b[i]+c[i]
}
```

- Must be safe to perform operations concurrently

REAL-LIFE EXAMPLE BRICKLAYING

- Step 1. Materials are delivered.
- Step 2. The foreman assigns the work.
- Step 3. Each mason lays brick in his assigned section.
  - Overlap of regions done by different masons must be managed.
- Step 4. Smooth joints between sections to make a unified whole.

COMPUTATIONAL EXAMPLE: FINDING THE MAXIMUM

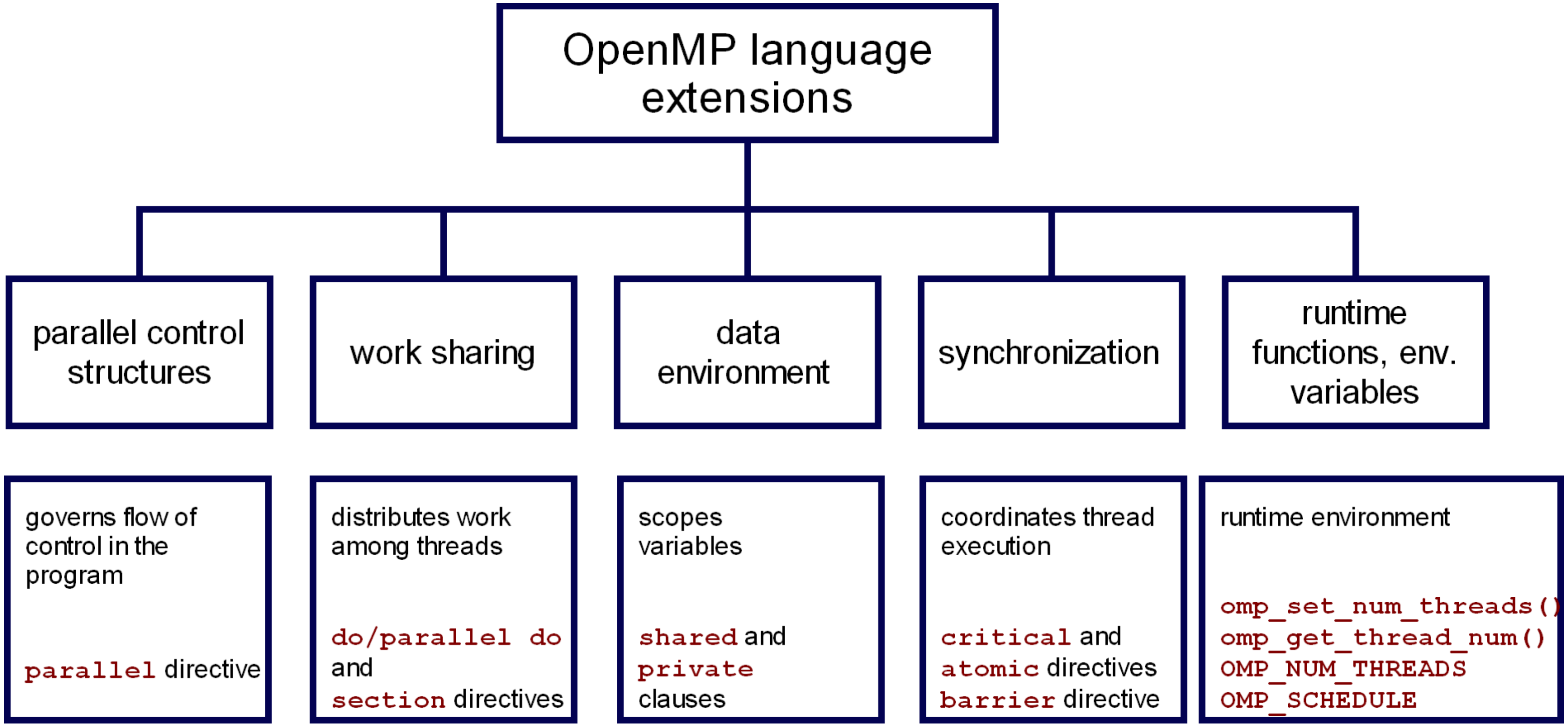
- Find the maximum of a function using a "brute force" method.
- Evaluate the function at a huge number of randomly-distributed points over a specified range of independent variables.
- Distribute these points out so that each process evaluates the function throughout the range.
- Each process computes the maximum of its sample.
- Individual maxima are returned to the master process, which selects the maximum of maxima as the result.

OPENMP

OPENMP

- OpenMP: An application programming interface (API) for parallel programming on multiprocessors
  - Compiler directives
  - Library of support functions
- OpenMP works in conjunction with Fortran, C, or C++
- Implemented within the compiler. Must be activated by a compile-time option.
- Python can use OpenMP via C or Fortran and wrap the result.

OPENMP


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

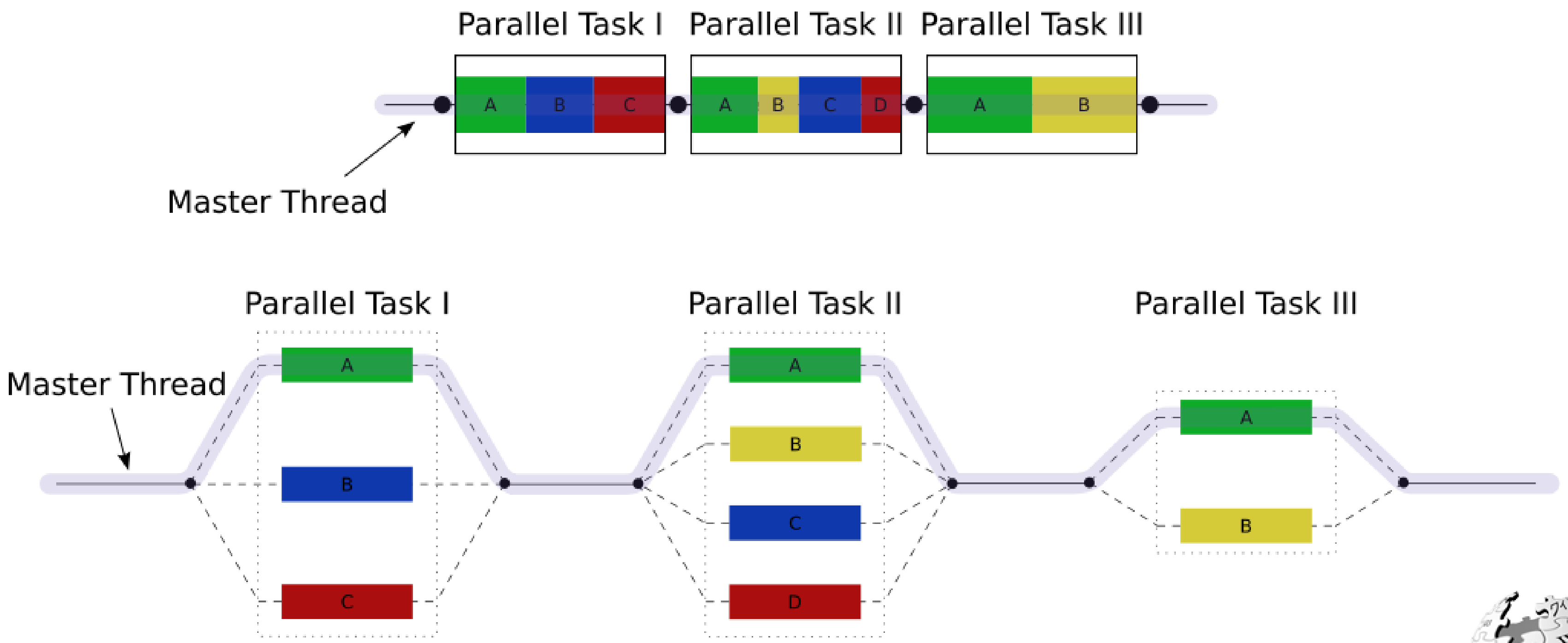
OPENMP USE CASES

- C/C++/Fortran + OpenMP sufficient to program shared-memory computers.
- C/C++/Fortran + MPI + OpenMP a good way to program distributed computers built out of shared-memory nodes.
  - Most modern clusters including Rivanna are of this type.
- OpenMP is easiest to use with data-parallel applications.

FORK/JOIN PARALLELISM

- Initially only master thread is active
- Master thread executes sequential code
- Fork: Master thread creates or awakens additional threads to execute parallel code
- Join: At end of parallel code created threads die or are suspended

HOW OPENMP WORKS



WIKIPEDIA
The Free Encyclopedia

PRAGMAS AND PSEUDOCOMMENTS

- Pragma: a compiler directive in C or C++
- Stands for “pragmatic information”
- A way for the programmer to communicate with the compiler
- The compiler is free to ignore pragmas
- Syntax:

  `#pragma omp` *<rest of pragma>*
- Pseudocomment: a pragma that otherwise looks like a comment

  `!$omp` *<rest of pseudocomment>*

COMPILING OPENMP PROGRAMS

- OpenMP is a *compiler-level* library.
- Gnu Compiler Collection
  - module load gcc
  - gcc/g++/gfortran
  - Use the –fopenmp flag when compiling.
- Intel's compilers
  - module load intel
  - the compilers are icc/icpc/ifort
  - Use the -qopenmp flag to compile
- Without the flag the pragmas/pseudocomments are ignored (but any OMP headers used won't be found).

RUNNING OPENMP PROGRAMS

- The default number of threads is implementation-dependent, but usually is the number of cores it sees on a node.
- The most common way to set the number of subprograms is to use an environment variable OMP_NUM_THREADS
- Example

```
gcc -fopenmp myopm.c
```

  - or

```
gfortran -fopenmp myomp.f90
```

- Run with

```
./a.out
export OMP_NUM_THREADS=4
./a.out
```

EXERCISE

- Using your choice of compiler, compile and run
  - `omphello.c`
  - or
  - `omphello.f90`
- Try setting different values for OMP_NUM_THREADS

PARALLEL FOR LOOPS

- C programs often express data-parallel operations as **`for`** loops. Fortran equivalent is **`do`**.

```
for (i = first; i < size; i += prime)
         marked[i] = 1;
```

- OpenMP makes it easy to indicate when the iterations of a loop may execute in parallel
- Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads

C/C++: PARALLEL FOR PRAGMA

- Format:

```
#pragma omp parallel for
for (i = 0; i < N; i++)
   a[i] = b[i] + c[i];
```

- Valid through the immediately following code block
- Compiler must be able to verify the run-time system will have information it needs to schedule loop iterations

FORTRAN: PARALLEL DO PSEUDOCOMMENT

- Format:

```
!$omp parallel do
 do i=1, N
   a(i) = b(i) + c(i)
 enddo
!$omp end parallel do
```

FORTRAN ONLY: WORKSHARE

- OpenMP provides the WORKSHARE option to support array operations in Fortran.

```
!$omp PARALLEL WORKSHARE
  A=1.
  B=42.
  C=2.*B
  A=B*C+D
!$omp END PARALLEL WORKSHARE
```

WARNING!!!

- You’re telling the compiler something that had better be true or else there will be lots of problems
- You’re telling the compiler there are no inter-iteration loop dependencies
  - i.e, the loop iterations are completely independent
- There cannot be statements that end the loop prematurely
  - No break, return, exit, or goto
  - But can have continues

SHARED AND PRIVATE VARIABLES

- Shared variable: has *same* address in execution context of every thread
- Private variable: has *different* address in execution context of every thread
  - A thread cannot access the private variables of another thread
- Default:
  - Shared:
    - C/C++: all static, all in whole-file scope
    - Fortran: COMMON, SAVE, variables in modules
  - Private:
    - (First) loop variables
    - Stack variables in functions (C/C++) or subroutines (Fortran) invoked in a parallel region
    - Fortran: Automatic variables within a statement block

DECLARING PRIVATE VARIABLES

```
for (i = 0; i < N; i++)
   for (j = 0; j <N; j++)
      a[i][j] = myMIN(a[i][j])+myMEAN(a[i][j];
```

- Either loop could be executed in parallel
- We prefer to make outer loops parallel, to reduce number of forks/joins
- We then must give each thread its own private copy of variable **`j`**

PRIVATE CLAUSE

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private
- Same syntax for C/C++ and Fortran

```
private ( <variable list> )
```

EXAMPLE USE OF PRIVATE CLAUSE

C/C++

```
#pragma omp parallel for private(j)
for (i = 0; i <N; i++)
   for (j = 0; j < N; j++)
      a[i][j] = myMIN(a[i][j)+myMEAN(a[i][j]);
```

Fortran

```
!$omp parallel do private(i)
do j=1,N
   do i=1,N
      a(i,j)=min(a(i,j),a(i,j)+tmp)
   enddo
enddo
!$omp end parallel do
```

RACE CONDITIONS

- Consider this C program segment to compute π using the rectangle rule:

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

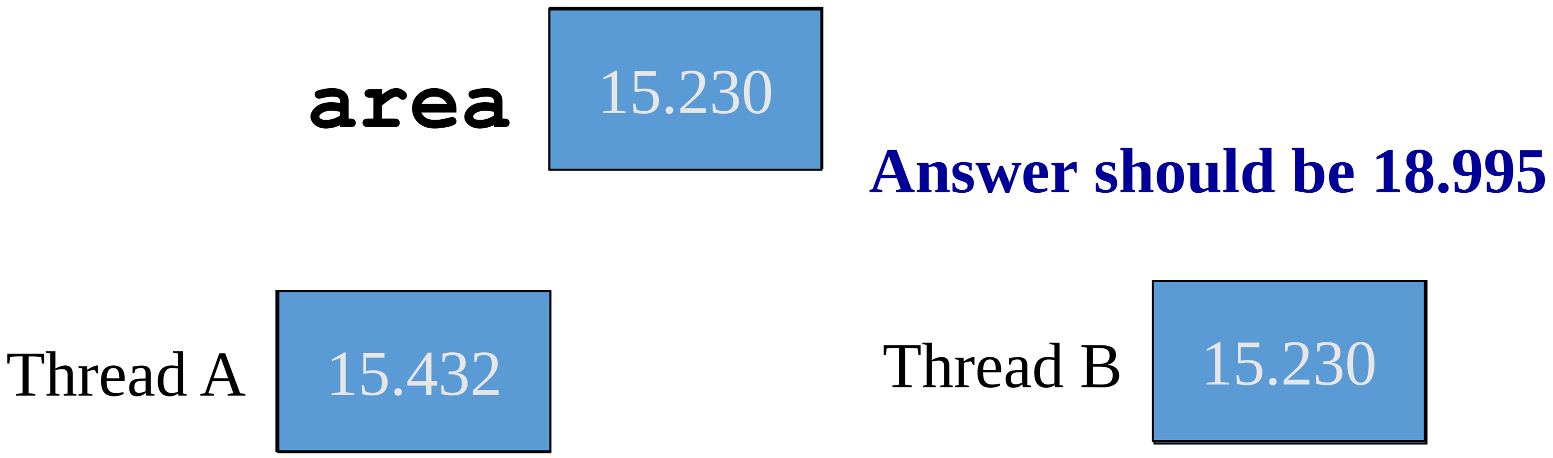
RACE CONDITION (CONT.)

- If we simply parallelize the loop…

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

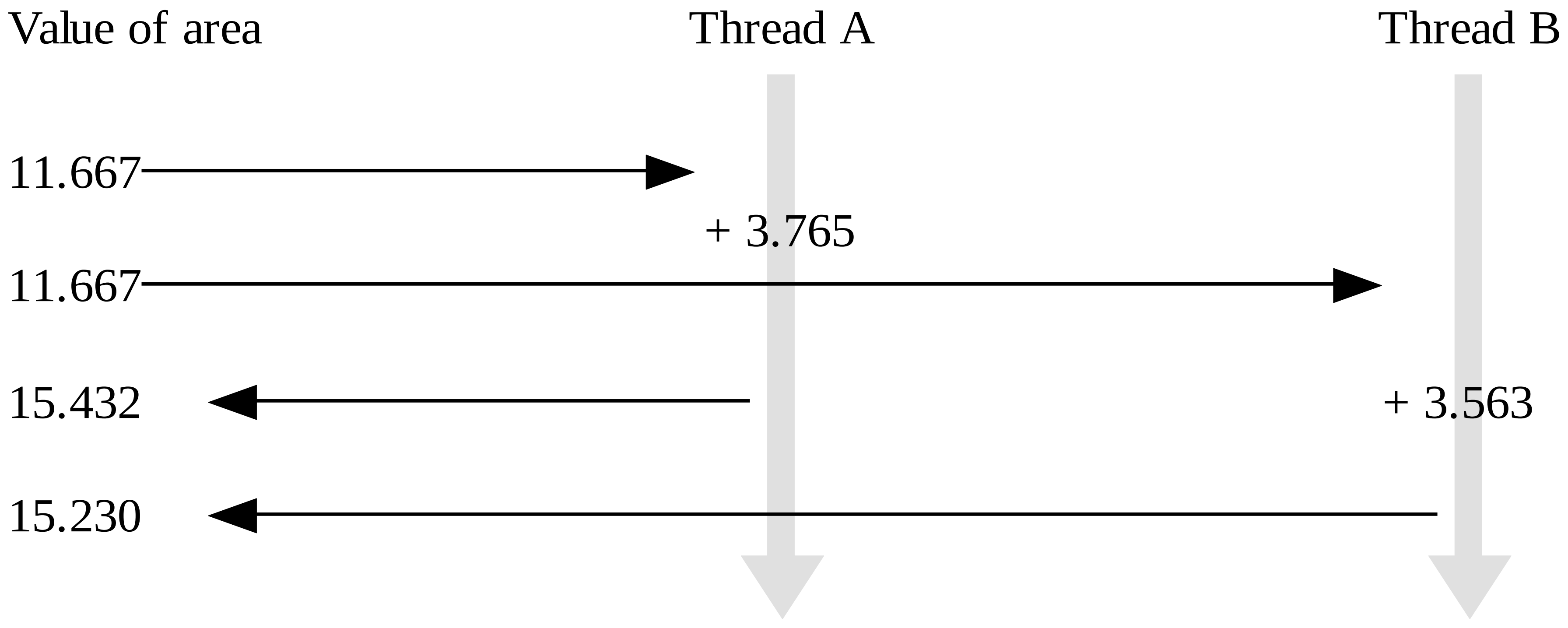
RACE CONDITION (CONT.)

- … we set up a race condition in which one process may “race ahead” of another and not see its change to shared variable **`area`**



```
area += 4.0/(1.0 + x*x)
```

RACE CONDITION TIME LINE



CRITICAL PRAGMA

- Critical section: a portion of code that only *one* thread at a time may execute
- We denote a critical section by putting the pragma

```
#pragma omp critical
```

in front of a block of C code.

For Fortran the equivalent is

```
!$omp critical
!$end omp critical
```

CORRECT, BUT INEFFICIENT, CODE

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
#pragma omp critical
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

SOURCE OF INEFFICIENCY

- Update to `area` inside a critical section
- Only one thread at a time may execute the statement; i.e., it is sequential code
- Time to execute statement significant part of loop
- Speedup will be severely constrained by this serialization

REDUCTIONS

- This pattern is called a *reduction*. Reductions are so common that OpenMP provides support for them.
- May add reduction clause to `parallel for` pragma.
- Specify reduction operation and reduction variable.
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop.

REDUCTION CLAUSE

- The reduction clause has this syntax:
  `reduction (<op> :<variable>)`
- Operators

`+`	Sum
`*`	Product
`&`	Bitwise and
`|`	Bitwise or
`^`	Bitwise exclusive or
`&&`	Logical and
`||`	Logical or

- Fortran, and C/C++ for OpenMP 3.1 or greater

```
max
min
```

π-COMPUTING CODE WITH REDUCTION CLAUSE

```
double area, pi, x;
int i, n;
...
area = 0.0;
//pragma should be all on one line
#pragma omp parallel for
       private(x) reduction(+:area)
for (i = 0; i < n; i++) {
   x = (i + 0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

PARALLEL PRAGMA

- The `parallel` pragma launches a team of threads.
- Execution is replicated among all threads.
- More general than parallel for or parallel do. Permits break/exit in a loop.

FOR PRAGMA

- The `parallel for` pragma will fork the threads, and split the `for` loop into parts
- The `parallel` pragma will fork the threads, and execute the *same* for loop for each thread (i.e. not split any loops into parts)
- But if you have already split the threads (via a `parallel` pragma), and want to split a for loop among the *already existing* threads (as opposed to executing the entire loop in all threads), then use a `for` pragma:

```
#pragma omp for
!$omp do .. !$omp end do
```

EXAMPLE USE OF FOR PRAGMA

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
      printf ("Exiting (%d)\n", i);
      break;
   }
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

BARRIER PRAGMA

- When a thread encounters a barrier pragma it will wait till all threads have reached the barrier.
  - `#pragma omp barrier`
  - `$!omp barrier`
- This enables the threads to synchronize but does tend to serialize the code.

MASTER AND SINGLE PRAGMAS

- Suppose we only want to see the output once
- The **`single`** pragma directs compiler that only a single thread should execute the block of code the pragma precedes
- Syntax:

```
#pragma omp single {}
!$omp single
!$omp end single
```

- With single the first thread who reaches it executes the segment. The **`master`** pragma causes only the master thread to execute the section.

```
#pragma omp master {}
!$omp master
!$omp end master
```

USE OF MASTER PRAGMA

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp master
   printf ("Exiting (%d)\n", i);
      break;
   }
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

ATOMIC OPERATIONS

- What if we can't use a reduction but need to update a variable without a race condition?
- The atomic directive ensures that a variable is accessed by one thread at a time.

```
#pragma omp atomic
  count+=1
```

```
!omp atomic
  Count=count+1
```

- The Fortran version does not need an `end` because atomic applies to only *one* statement.

USEFUL FUNCTIONS

- C/C++: #include <omp.h>
- Fortran: use omp_lib
- omp_get_num_threads()
  - This function returns the number of active threads
  - Returns 1 in serial regions

```
int omp_get_num_threads (void)
integer function omp_get_num_threads()
```

- omp_get_thread_num()
  - This function returns the thread identification number
  - If there are *t* threads, the ID numbers range from 0 to *t*-1
  - The master thread has ID number 0

```
int omp_get_thread_num (void)
integer function omp_get_thread_num()
```

TIMING FUNCTION

- C/C++

```
double t1, etime;
t1=omp_get_wtime();
   compute
etime=omp_get_wtime()-t1;
```

- Fortran

```
double precision :: t1, etime
t1=omp_get_wtime()
   compute
etime=omp_get_wtime()-t1
```

OPENMP SLURM RESOURCE REQUEST

- You will request a single node and some number of cores on the node. Be sure that the number of cores requested matches the OMP_NUM_THREADS specified in your script.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=20
#SBATCH -p instructional
#SBATCH -A rivanna-training
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./myexec
```

EXERCISE

- Fortran and C/C++:
  - Download omparea.c or omparea.f90
  - Compile and run them as is
  - Add the correction to get the right answer
- Python:
  - Download and run multi.py

SHARED MEMORY PROGRAMMING IN PYTHON

THE GIL

- Standard Python implements a GIL (global interpreter lock).
- Threads cannot be started within a single interpreter.
- It can be faked but it's slow.
- Better: Just start another process.
- Multiprocessing is standard in Python 2.7 and up.
- Next few examples are from the documentation at

http://docs.python.org/2/library/multiprocessing.html

MULTIPROCESSING

- Import the package

```
from multiprocessing import Process
```

- Define a function

```
def f(name):
   print('hello from '+name)
```

- Multiprocessing can *only* run as main

```
if __name__ == '__main__':
  ncpus=2
  for i in range(ncpus):
     p=Process(target=f,args=('str(i)',))
     p.start()
  p.join()
```

WORKER POOLS

- For manager-worker problems, we can start a pool of workers.

```
from multiprocessing import Pool
def f(x):
  return x*x
if __name__ == '__main__':
   pool = Pool(processes=4)
   result = pool.apply_async(f, [10])
   print(result.get(timeout=1))
   print(pool.map(f, range(10)))
```

EXAMPLE

```
import multiprocessing as MP
import os
from pylab import *
import random

# This is the function that we want to
# compute for various different parameters
def spikes(weight):
    number=weight*random.random()
    return number

if __name__ == '__main__':
    ncpus=int(os.getenv("NUM_THREADS"))
    pool = MP.Pool(processes=ncpus)
    weights = linspace(0, 3.5, 100)
    volt=0.2
    args = [w * volt for w in weights]
    # launches multiple processes
    results = pool.map(spikes, args)
    plot(weights, results, '.')
    show()
```

MORE REALISTIC EXAMPLE

- From http://kmdouglass.github.io/posts/learning-pythons-multiprocessing-module.html

```
import multiprocessing

def runSimulation(params):

  """This is the main processing function. It will contain
whatever code should be run on multiple processors. """
  param1, param2 = params
  # Example computation
  processedData = []
  for ctr in range(1000000):
    processedData.append(param1*ctr - param2**2)
  return processedData
```

REALISTIC EXAMPLE (CONT)

```
if __name__ == '__main__':
    ncpus=4
   # Define the parameters to test
   param1 = range(100)
   param2 = range(2, 202, 2)
   # Zip the parameters because pool.map() takes
  only one iterable
   params = zip(param1, param2)
   pool = multiprocessing.Pool(processes=npcus)
   results = pool.map(runSimulation, params)
   print(results)
```

ADD TIMING INFORMATION

```
#add import time to top of file, use time.time() rather than clock
#Run under Python 2 for consistency of map behavior
from __future__ import print_function
if __name__ == '__main__':
      ncpus=4
    # Define the parameters to test
    param1 = range(100)
    param2 = range(2, 202, 2)
    # Zip the parameters because pool.map() takes only one iterable
    params = zip(param1, param2)
    pool = multiprocessing.Pool(processes=ncpus)
    tic=time.time ()
    results = pool.map(runSimulation, params)
    toc=time.time ()
    print("Parallel time "+str(toc-tic))
    tic=time.time ()
    results=map(runSimulation, params)
    toc=time.time ()
    print("Serial time "+str(toc-tic))
    pool.close(); pool.join()
```

RESULT

- On my workstation with Python 2.7.11 this results in

```
    Parallel time 9.66964006424
    Serial time 25.3560800552
```

- This is a speedup of a factor of 2.6
- The efficiency is thus 0.66

WARNING

- If no processes argument is given to Pool it starts as many processes as it detects cores on your machine.
- If using a shared resource this can be bad. You must tell it how many to use and it must match your request.
- We can use `os.getenv('Envvar')` to get the value of an environment variable, if we have an appropriate variable.

PYTHON MP SLURM RESOURCE REQUEST

- You will request a single node and some number of cores on the node. You will need to set some environment variable, then use os.getenv() in your code to capture it.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --cpus-per-task=20
#SBATCH -p instructional
#SBATCH -A rivanna-training
export NUM_THREADS=${SLURM_CPUS_PER_TASK}
module load mpi4py/3.0.0-py3.6
python mymulti.py
```

IN THE CODE

```
import os
import multiprocessing
…
def f(name):
   print("Greetings from "+str(name))

ncpus=int(os.getenv('NUM_THREADS'))
pool=multiprocessing.Pool(processes=ncpus)
pool.map(f,range(ncpus))
pool.close()
pool.join()
```

SAMPLE RESULT

```
Greetings from 0
Greetings from 1
Greetings from 3
Greetings from 2
```

- Why are they not in order??
  - Multiprocessing is not deterministic! The output depends on in which order the processes complete and can access the standard output.
- The map method of Pool does **force** determism for the *return* values. So if we change it a bit:

KEEP IT ORDERED

```python
import os
from multiprocessing import Pool

def f(name):
    return "Greetings from "+str(name)

ncpus=int(os.getenv('SLURM_NTASKS'))
pool=Pool(processes=ncpus)
result=pool.map(f,range(ncpus))
print(';'.join(result))
pool.close()
pool.join()
```

- Output:

```
Greetings from 0;Greetings from 1;Greetings from 2;Greetings from 3
```

EXERCISE

- Write a multiprocessing program that computes the sum of the cubes of the numbers from 1.0 to 1000.0 by increments of 0.1
- Add the timing routines to compare the parallel and serial times. You can use the cluster frontend if you do not have a Python 2 or 3 environment, with

```
module load anaconda/5.2.0-py2.7
```

- or

```
module load anaconda/5.2.0-py3.6
```

PROGRAMMING NEW HARDWARE

ACCELERATORS

- Accelerators include
  - General-purpose GPUs (GPGPUs)
  - Intel MIC
- These are programmed with OpenACC (GPGPUs) or OpenMP (MIC, GPGPU extensions to OpenMP are in newer versions).

GENERAL PURPOSE GRAPHICAL PROCESSING UNITS

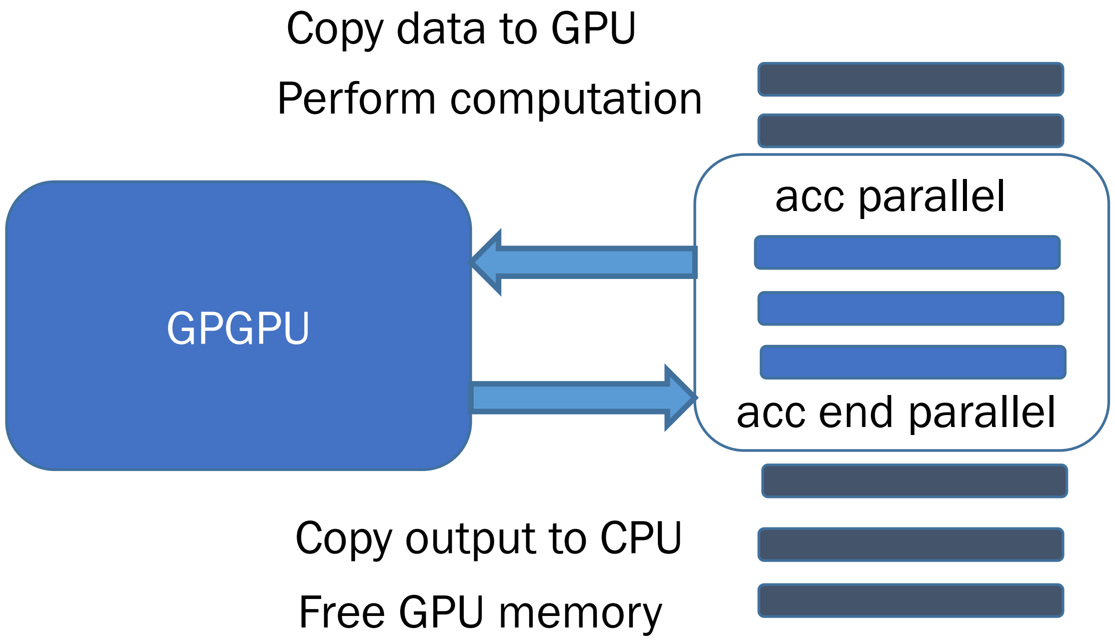
GPGPU ARCHITECTURE

- GPUs have
  - thousands of ALUs (Arithmetic Logic Units) compared to 4-8 for a CPU
  - very limited instruction sets
  - very fast memory bandwidth
    - but not that much memory
  - Most GPGPUs do not have hardware for double precision floats.
    - K80 does, P100 does not

PROGRAMMING MODELS

- CUDA
  - CUDA is a library from NVIDIA that allows general-purpose computations on devices originally designed for graphics
- OpenMP
  - OpenMP will support both NVIDIA and AMD GPGPUs but compiler support is still lagging
- OpenACC
  - Developed by the Portland Group compiler vendors for NVIDIA devices.
    - Bought by NVIDIA, will soon become the “NVIDIA HPC SDK”.
  - Similar to OpenMP (pragma/pseudocomment based)
  - Will be our example

PROGRAMMING MODEL



OPENACC

- OpenACC is available in Portland Group compilers and in Gnu compilers starting with 5.0
- Supports C/C++/Fortran
- Pragmas similar to OpenMP
- #pragma acc
- !$acc
- PGI also supports CUDA and provides Fortran bindings for it.

PARALLEL REGIONS

- Pragmas are the same as for OpenMP with acc instead of omp
- These include
- #pragma acc parallel
- #pragma acc parallel if (<conditional>)
- #pragma acc parallel private (varlist)
- OpenACC uses a generic loop rather than for/do
- #pragma acc parallel
- #pragma acc loop
- Or
- !$acc parallel
- !$acc loop
- !$acc end parallel

REDUCTIONS

- #pragma acc parallel reduction(operator:list)
  - Provides a max and min for C/C++ as well as Fortran
  - C operators are max, min, +, *, &, |, &&, ||, ^
  - Fortran operators are max, min, +, *, iand, ior, .and., .or., .eqv., .neqv.

EXAMPLE - SAXPY

- single precision `a` times `x` plus `y`
- `a` scalar, `x` and `y` one-dimensional arrays
- array version of "fused multiply add" and should be fast for good performance of many algorithms.

SERIAL VERSION

Fortran

```fortran
subroutine saxpy(n, a, x, y)
real, dimension(n), intent(in)::x,y
real                , intent(in) :: a
integer, intent (in)            :: n
integer                         :: i
  do i=1,n
     y(i) = a*x(i)+y(i)
  enddo
end subroutine saxpy
...
! Perform SAXPY on 1M elements
  call saxpy(2**20, x_d, y_d)
```

C

```c
void saxpy(int n, float a,
float *x, float *restrict y){
//restrict prohibits aliasing
   for (int i=0;i<n;++i)
      y[i]=a*x[i]+y[i];
}
...
//Perform SAXPY on 1M
//elements
   saxpy(1<<20,2.0,x,y);
```

OPENMP VERSION

Fortran

```
subroutine saxpy(n, a, x, y)
real, dimension(n), intent(in)::x,y
real              , intent(in) :: a
integer, intent (in)           :: n
integer                        :: I
!$omp parallel do
  do i=1,n
     y(i) = a*x(i)+y(i)
  enddo
!$end omp parallel do
end subroutine saxpy
...
! Perform SAXPY on 1M elements
  call saxpy(2**20, x_d, y_d)
```

C

```
void saxpy(int n, float a,
float *x, float *restrict y){
#pragma omp parallel for
   for (int i=0;i<n;++i)
      y[i]=a*x[i]+y[i];
}
...
//Perform SAXPY on 1M
//elements
   saxpy(1<<20,2.0,x,y);
```

OPENACC VERSION

Fortran

```
subroutine saxpy(n, a, x, y)
real, dimension(n), intent(in)::x,y
real              , intent(in) :: a
integer, intent (in)           :: n
integer                        :: I
!$acc parallel loop
  do i=1,n
     y(i) = a*x(i)+y(i)
  enddo
!$end acc parallel loop
end subroutine saxpy
...
! Perform SAXPY on 1M elements
  call saxpy(2**20, x_d, y_d)
```

C

```
void saxpy(int n, float a,
float *x, float *restrict y){
#pragma acc parallel loop
   for (int i=0;i<n;++i)
      y[i]=a*x[i]+y[i];
}
...
//Perform SAXPY on 1M
//elements
   saxpy(1<<20,2.0,x,y);
```

COMPILING

- With the PGI compiler

```
module load pgi
pgcc -acc mysaxpy.c
pgf90 -acc mysaxpy.f90
pgCC -acc mysaxpy.cxx
```

SPECIAL CONSIDERATIONS FOR GPGPUS

- GPGPUs have a small amount of memory with very high bandwidth
- Management of data movement to/from the device is critical for performance
- These pragmas do not have (basic) OpenMP equivalents.

KERNELS AND DATA

- Kernels are implemented on the device
    - #pragma acc kernels
    - !$acc kernels
    - !$acc end kernels
- Data constructs are regions where data is accessible to the device
    - #pragma acc data
    - !$acc data
    - !$acc end data

KERNELS

- With the keyword kernel the compiler determines what can be offloaded to the gpu

```
!$acc kernels
do i=1,n
   a(i) = 0.0
   b(i) = 1.0
   c(i) = 2.0
end do
do i=1,n
   a(i) = b(i)+c(i)
enddo
!$acc end kernels
```

DATA MOVEMENT

- Enter data
  - Data are allocated on and moved to the device
  - #pragma acc enter data
  - !$acc enter data
- Exit data
  - The data will stay on the device to the end of the program or to the next exit data pragma
  - #pragma acc exit data
  - !$acc exit data
- Update data
  - Copies data between memory for the thread and the device
  - #pragma acc update
  - !$acc update

EXCESSIVE COPIES SLOW DOWN CODE

- Example:

```
while (error>tol && i<maxIter) {
#pragma acc parallel loop reduction(max:err)
         Data transfer into GPU
     for (something) {do things with A, Anew}
         Data transfer out of GPU
}
```

- Data Regions
  - `!$acc data`
  - `!$acc end data`
  - `#pragma acc data`

DATA DIRECTIVES

- `copy (list)`
  - copy items in the list to the GPU on entry; to host on exit
- `copyin (list)`
  - copy items in list to the GPU
- `copyout (list)`
  - copy items in list from the GPU
- `create (list)`
  - allocate GPU memory but do not copy (good for temporary variables not needed outside)
- `present (list)`
  - assert that items in list are already present in the GPU's memory

IMPROVING THE DOUBLE LOOP

```
#pragma acc data copy(A), create(Anew)
while (error>tol && i<maxIter) {
#pragma acc parallel loop reduction(max:err)
   for (something) {do things with A, Anew}
}
```

PYTHON AND GPGPUS

- Numba
  - `from numba import cuda`
- We use the cuda.jit decorator
- Check for a gpu
  - `print(cuda.gpus())`
- `https://numba.pydata.org/numba-doc/latest/cuda/overview.html`

NUMBA EXAMPLE

- From their documentation

```
from numba import cuda, float32

# Controls threads per block and shared memory usage.
# The computation will be done on blocks of TPBxTPB elements.
TPB = 16

@cuda.jit
def fast_matmul(A, B, C):
    # Define an array in the shared memory
    # The size and type of the arrays must be known at compile time
    sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
    sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)

    x, y = cuda.grid(2)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bpg = cuda.gridDim.x    # blocks per grid

    if x >= C.shape[0] and y >= C.shape[1]:
        # Quit if (x, y) is outside of valid C boundary
        return
```

```
 # Each thread computes one element in the result matrix.
 # The dot product is chunked into dot products of TPB-long
#vectors.
    tmp = 0.
    for i in range(bpg):
        # Preload data into shared memory
        sA[tx, ty] = A[x, ty + i * TPB]
        sB[tx, ty] = B[tx + i * TPB, y]

        # Wait until all threads finish preloading
        cuda.syncthreads()

        # Computes partial product on the shared memory
        for j in range(TPB):
            tmp += sA[tx, j] * sB[j, ty]

        # Wait until all threads finish computing
        cuda.syncthreads()

    C[x, y] = tmp
```

PYCUDA

- This is a free product
- Requires the CUDA libraries, which must be installed by the user (or whoever administers the system)
- Example from their documentation

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))
```