INTRODUCTION TO OPENMP & OPENACC

Kadin Tseng
Boston University
Research Computing Services
Outline

- Introduction to OpenMP (for CPUs)
- Introduction to OpenACC (for GPUs)
Introduction to OpenMP (for CPUs)

• Types of parallel machines
  • Distributed memory
    • each processor has its own memory address space
    • variable values are independent
      \( x = 2 \) on one processor, \( x = 3 \) on a different processor
    • example: among nodes of the SCC
  • Shared memory
    • also called Symmetric Multiprocessing (SMP)
    • typically, parallel computing units are threads (or cores)
    • single address space for all threads
      • If one thread sets \( x = 2 \), \( x \) will also equal 2 on other threads (unless specified otherwise)
    • example: cores within each SCC node
Shared vs. Distributed Memory

- **Shared**
  - CPU 0
  - CPU 1
  - CPU 2
  - CPU 3
  - MEM 0
  - MEM 1
  - MEM 2
  - MEM 3

- **Distributed**
  - CPU 0
  - CPU 1
  - CPU 2
  - CPU 3
  - MEM

**Introduction to OpenMP & OpenACC**
Shared Computing Cluster (SCC)

Memories shared within each node.
Memories not shared (distributed) among nodes.
What is OpenMP?

- Application Programming Interface (API) for *multithreaded* parallelism consists of
  - Compiler directives (placed in source code by programmer)
  - Runtime utility functions and header files
  - Environment variables
- Languages supported: FORTRAN, C, C++
- Advantages
  - Easy to use
  - Incremental parallelization
  - Flexible -- from coarse grain to fine grain (loop level)
  - Portable -- on any SMP machine (e.g., each individual SCC node)
- Disadvantages
  - Shared-memory systems only (*i.e.*, not across SCC nodes)
Basics

• Goal – distribute work among threads

• Two methods to be discussed
  • Loop-level
    • Specific loops are parallelized
    • Used in automatic parallelization tools, like MATLAB PCT
  • Parallel regions
    • Also called “coarse-grained parallelism”
Basics (cont’d)

Loop-level

Parallel regions
Directive format

- FORTRAN

```fortran
!$omp parallel do default(none) private(i,j,k) shared(a,b,c,n) 
c$omp parallel do default(none) private(i,j,k) shared(a,b,c,n)
```

- C/C++

```c
#pragma omp parallel for default(none) private(i,j,k) shared(a,b,c,n)
```

Sentinel  directive name  clauses (optional)
parallel for (parallel do) directive

• parallel do (Fortran) or parallel for (C) directive

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

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

Use “c$” for fixed-format Fortran

• Suppose maxi = 1000 and 4 threads are available
  
  Thread 0 gets i = 1 to 250
  Thread 1 gets i = 251 to 500
  Thread 2 gets i = 501 to 750
  Thread 3 gets i = 751 to 1000

• Barrier (synchronization) imposed at end of loop
workshare

• For Fortran 90/95 array syntax, the parallel workshare directive is analogous to parallel do
• Previous example would be:

    !$omp parallel workshare
c = a + b
    !$omp end parallel workshare

• Also works for forall and where statements
• No equivalent directive for C/C++
**Shared vs. Private**

- In parallel region, variables are *shared* by default
- Loop indices are always *private* by default
- What is wrong with the following code segment?

```c
ifirst = 10        ! shared by all threads
 !$omp parallel do
 do i = 1, maxi    ! i is private
  i2 = 2*i         ! i2 is shared
  j(i) = ifirst + i2 ! j also shared
 enddo
 !$omp end parallel do
```

```c
ifirst = 10;       // shared by all threads
#pragma omp parallel for
for(i = 0; i < maxi; i++) { // i is private
  i2 = 2*i;                // i2 is shared
  j[i] = ifirst + i2;      // j also shared
}
```
Shared vs. Private (cont’d)

Need to declare \texttt{i2} with a private clause

\begin{verbatim}
ifirst = 10   !shared by all threads
 !$omp parallel do private(i2)
 do i = 1, maxi  ! i is private
    i2 = 2*i     ! i2 different on each thread
    j(i) = ifirst + i2
 enddo
 !$omp end parallel do

ifirst = 10;
#pragma omp parallel for private(i2)
for(i = 0; i < maxi; i++) {  // i is private
    i2 = 2*i;    // i2 different on each thread
    j[i] = ifirst + i2;
}
\end{verbatim}
Data Dependencies

• Data on one thread can be dependent on data on another thread
• This can result in wrong answers
  • thread 0 may require a variable that is calculated on thread 1
  • answer depends on timing – When thread 0 does the calculation, has thread 1 calculated it’s value yet?
Data Dependencies (cont’d)

- Example – Fibonacci Sequence 0, 1, 1, 2, 3, 5, 8, 13, …

\[
\begin{align*}
  a(1) &= 0 \\
  a(2) &= 1 \\
  \text{do } i &= 3, 100 \\
  &\quad a(i) = a(i-1) + a(i-2) \\
  \text{enddo}
\end{align*}
\]

Let's parallelize on 2 threads.

Thread 0 gets \(i = 3\) to 51
Thread 1 gets \(i = 52\) to 100

Follow calculation for \(i = 52\) on thread 1. What will be values of \(a\) at \(i - 1\) and \(i - 2\) ?

\[
\begin{align*}
  a[1] &= 0; \\
  a[2] &= 1; \\
  \text{for}(i &= 3; i <= 100; i++)
  &\quad a[i] = a[i-1] + a[i-2]; \\
\end{align*}
\]
More clauses

- Can make **private** the default rather than **shared**
  - *Fortran only*
  - handy if most of the variables are private
  - can use continuation characters for long lines

```fortran
ifirst = 10
!$omp parallel do &
!$omp default(private) &
!$omp shared(ifirst,maxi,j)
do i = 1, maxi
   i2 = 2*i
   j(i) = ifirst + i2
endo
do !$omp end parallel do
```
More clauses (cont’d)

- Can use **default none**
  - Must declare all variables’ status (forces you to account for them)
  - Any variable not declared will receive a complaint from compiler.
ifirst = 10

!$omp parallel do &
!$omp default(none) &
!$omp shared(ifirst,maxi,j) private(i2)
do i = 1, maxi
  i2 = 2*i
  j(i) = ifirst + i2
endo
!$omp end parallel do

ifirst = 10;
#pragma omp parallel for \\default(none) \\shared(ifirst,maxi,j) private(i2)
for(i = 0; i < maxi; i++){ 
i2 = 2*i;
  j[i] = ifirst + i2;
}

Introduction to OpenMP & OpenACC
Firstprivate

- Suppose we need a running total for each index value on each thread

```c
iper = 0
do i = 1, maxi
    iper = iper + 1
    j(i) = iper
endo
```

- if `iper` were declared `private`, the initial value would not be carried into the loop

```c
iper = 0;
for(i = 0; i < maxi; i++){
    iper = iper + 1;
    j[i] = iper;
}
```
Firstprivate (cont’d)

- Solution – firstprivate clause
- Creates private memory location for each thread
- Copies value from master thread (thread 0) to each memory location

```c
iper = 0
!$omp parallel do &
!$omp firstprivate(iper)
do i = 1, maxi
  iper = iper + 1
  j(i) = iper
enddo
!$omp end parallel do
```

```c
iper = 0;
#pragma omp parallel for \ firstprivate(iper)
for(i = 0; i < maxi; i++){
  iper = iper + 1;
  j[i] = iper;
}
```
Lastprivate

- saves value corresponding to the last loop index
- "last" in the serial sense

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

```c
#pragma omp parallel for lastprivate(i)
for(i = 0; i < maxi; i++){
   a[i] = b[i];
}
a[i] = b[0];
```
Reduction

- Following example won’t parallelize correctly
  - different threads may try to write to \( s \) simultaneously

```c
s = 0.0
!$omp parallel do
do i = 1, maxi
    s = s + a(i)
Enddo
!$omp end parallel do
```

```c
s = 0.0;
#pragma omp parallel for
for(i = 0; i < maxi; i++){
    s = s + a[i];
}
```
Reduction (cont’d)

• Solution is to use the reduction clause

```c
s = 0.0
!$omp parallel do reduction(+:s)
do i = 1, maxi
  s = s + a(i)
enddo
!$omp end parallel do
```

```c
s = 0;
#pragma omp parallel for reduction(+:s)
for(i = 0; i < maxi; i++){
  s = s + a[i];
}
```

• each thread performs its own reduction (sum, in this case)
• results from all threads are automatically reduced (summed) at the end of the loop
Reduction (3)

- Fortran operators/intrinsics: MAX, MIN, IAND, IOR, IEOR, +, *, -, .AND., .OR., .EQV., .NEQV.
- C operators: +, *, -, /, &, ^, |, &&, ||
- roundoff error may be different than serial case
Conditional Compilation

- For C, C++: conditional compilation performed with \_OPENMP macro name (defined during compilation with OpenMP turned on*)

  \#ifdef \_OPENMP

  \quad \ldots \ \text{do stuff} \ \ldots

  \#endif

- For Fortran: there are two alternatives
  - The above for C works if fortran file named with suffix .F90 or .F
  - Source lines start with !\$ become active with OpenMP turned on*
    \$ print*, ‘number of procs =’, nprocs

* How to turn on OpenMP is discussed in {	extit{Compile and Run}} page.
Basic OpenMP Functions

• `omp_get_thread_num()`
  • returns current thread ID; effective inside parallel region

• `omp_set_num_threads(nthreads)`
  • subroutine in Fortran
  • sets number of threads in next parallel region to `nthreads`
  • overrides `OMP_NUM_THREADS` environment variable
  • Effective outside parallel region

• `omp_get_num_threads()`
  • returns number of threads in current parallel region
Some Tips

- OpenMP will do what you tell it to do
  - If you try to parallelize a loop with a dependency, it will go ahead and do it! (but gives wrong answer)
- Generally, no benefit to parallelize short/shallow loops
- Maximize number of operations performed in parallel
  - parallelize outer loops where possible
- For Fortran, add “use omp_lib” to include header
- For C, header file is omp.h
Compile and Run on SCC

- Portland Group compilers:
  - Compile with `-mp` flag to turn on OpenMP
  - `scc1% pgfortran -o myprog myprog.f90 -mp -O3`
  - `scc1% pgcc -o myprog myprog.c -mp -O3`

- GNU compilers:
  - Compile with `-fopenmp` flag to turn on OpenMP
  - `scc1% gfortran -o myprog myprog.f90 -fopenmp -O3`
  - `scc1% gcc -o myprog myprog.c -fopenmp -O3`

- Run interactive job (up to 16 threads; 4 on login node)
  - `scc1% setenv OMP_NUM_THREADS 4`
  - `scc1% myprog`
Parallel

- parallel do / for can be separated into two directives.

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

is the same as

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

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

```c
#pragma omp parallel
#pragma omp for
for(i=0; i<maxi; i++){
   a[i] = b[i];
}
```
Parallel (cont’d)

• Note that an end parallel directive is required.
• end do not needed
• Everything within the parallel region will run in parallel.
• The do/for directive indicates that the loop indices will be distributed among threads rather than duplicating every index on every thread.
Parallel (3)

- Multiple loops in parallel region:

```c
#$omp parallel
#$omp do
do i = 1, maxi
   a(i) = b(i)
enddo
#$omp do
do i = 1, maxi
   c(i) = a(2)
enddo
#$omp end parallel
```

```c
#pragma omp parallel
#pragma omp for
for(i=0; i<maxi; i++){
   a[i] = b[i];
}
#pragma omp for
for(i=0; i<maxi; i++){
   c[i] = a[2];
}
#pragma omp end parallel
```

- `parallel` directive has a significant overhead associated with it.

- The above example has the potential to be faster than using two `parallel do/parallel for` directives.
Coarse-Grain Parallelism

• OpenMP is not restricted to loop-level, or fine-grained, parallelism.
• The !$omp parallel or #pragma omp parallel directive duplicates subsequent code within its scope on all threads.
• Parallelization similar to MPI style.
Coarse-Grain Parallelism (cont’d)

```c
$omp parallel &
$omp private(myid, istart, iend, nthreads, nper)
nthreads = omp_get_num_threads()
nper = max1/nthreads
myid = omp_get_thread_num()
istart = myid*nper + 1
iend = istart + nper - 1
call do_work(istart, iend)
do i = istart, iend
   c(i) = a(i)*b(i) + ...
enddo
$omp end parallel
```

```c
#pragma omp parallel
#pragma omp private(myid, istart, iend, nthreads, nper)
nthreads = omp_get_num_threads();
nper = max1/nthreads;
myid = omp_get_thread_num();
istart = myid*nper;
iend = istart + nper - 1;
do_work(istart, iend);
for(i=istart; i<=iend; i++){
   c[i] = a[i]*b[i] + ...
}
```
Thread Control Directives
Barrier

- **barrier** synchronizes threads

```c
!$omp parallel private(myid,istart,iend)
call myrange(myid,istart,iend)
do i = istart, iend
   a(i) = a(i) - b(i)
enddo
!$omp barrier
myval(myid+1) = a(istart) + a(1)
!$omp end parallel
```

```c
#pragma omp parallel private(myid,istart,iend)
myrange(myid,&istart,&iend);
for(i=istart; i<=iend; i++){
   a[i] = a[i] - b[i];
}
#pragma omp barrier
myval[myid] = a[istart] + a[0]
```

- Here **barrier** assures that a(1) or a[0] is available before computing myval
Master

- if you want part of code to be executed only on master thread, use `master` directive
- “non-master” threads will skip over `master` region and continue
Master Example - Fortran

```fortran
!$OMP PARALLEL PRIVATE(myid, istart, iend)
call myrange(myid, istart, iend)
do i = istart, iend
   a(i) = a(i) - b(i)
enddo
!$OMP BARRIER
!$OMP MASTER
write(21) a
!$OMP END MASTER
call do_work(istart, iend)
!$OMP END PARALLEL
```
Master Example - C

```
#pragma omp parallel private(myid, istart, iend)
myrange(myid, &istart, &iend);
for (i = istart; i <= iend; i++){
    a[i] = a[i] - b[i];
}
#pragma omp barrier
#pragma omp master
fwrite(fid, sizeof(float), iend - istart + 1, a);
#pragma omp end master
do_work(istart, iend);
#pragma omp end parallel
```
Single

If you:
• want part of code to be executed only by a single thread
• don’t care whether or not it’s the master thread

The use **single** directive
• Unlike the **end master** directive, **end single** has barrier
Single Example - Fortran

```fortran
!$OMP PARALLEL PRIVATE(myid,istart,iend)
call myrange(myid,istart,iend)
do i = istart, iend
   a(i) = a(i) - b(i)
enddo
!$OMP BARRIER
!$OMP SINGLE
write(21) a
!$OMP END SINGLE
call do_work(istart,iend)
!$OMP END PARALLEL
```
Single Example - C

```c
#pragma omp parallel private(myid, istart, iend)
    myrange(myid, istart, iend);
    for (i = istart; i <= iend; i++) {
        a[i] = a[i] - b[i];
    }
#pragma omp barrier
#pragma omp single
    fwrite(fid, sizeof(float), nvals, a);
#pragma omp end single
    do_work(istart, iend);
```
Critical

If you have code section that:

1. must be executed by every thread
2. threads may execute in any order
3. threads must not execute simultaneously

This does not have a barrier.
Critical Example - Fortran

```fortran
!$OMP PARALLEL PRIVATE(myid,istart,iend)
call myrange(myid,istart,iend)
do i = istart, iend
   a(i) = a(i) - b(i)
enddo
!$OMP CRITICAL
call mycrit(myid,a)
!$OMP END CRITICAL
!$OMP END PARALLEL
```
Critical Example - C

```c
#include <iostream>

int main() {
    int i, istart, iend;
    std::cin >> istart >> iend;
    #pragma omp parallel private(myid, istart, iend)
    myrange(myid, istart, iend);
    for (i = istart; i <= iend; i++) {
        a[i] = a[i] - b[i];
    }
    #pragma omp critical
    mycrit(myid, a);
    #pragma omp end critical
    do_work(istart, iend);
    #pragma omp end parallel
    return 0;
}
```

Introduction to OpenMP & OpenACC
Ordered

- Suppose you want to write values in a loop:

```
do i = 1, nproc
    call do_lots_of_work(result(i))
    write(21,*) i, result(i)
enddo
```

```
for(i = 0; i < nproc; i++){
    do_lots_of_work(result[i]);
    fprintf(fid,"%d %f\n","i,result[i]");
}
```

- If loop were parallelized, could write out of order
- **ordered** directive forces serial order
Ordered (cont’d)

```c
!$omp parallel do
do i = 1, nproc
    call do_lots_of_work(result(i))
    !$omp ordered
    write(21,*), i, result(i)
    !$omp end ordered
enddo
```

```c
#pragma omp parallel for
for(i = 0; i < nproc; i++){
do_lots_of_work(result[i]);
    #pragma omp ordered
    fprintf(fid,"%d %f\n" ,i,result[i]);
    #pragma omp end ordered
}
```

- Since `do_lots_of_work` takes a lot of time, most parallel benefit will be realized
Schedule

- **schedule** refers to the way in which loop indices are distributed among threads

- ([`static[, chunk]`])
  - static is the default
  - each thread is assigned a contiguous chunk of indices in thread number order
  - number of indices assigned to each thread is as equal as possible
  - Chunk size may be specified

- (dynamic[, chunk])
  - Good way for varying work load among loop iterations
Integration Example

• An integration of the cosine function between 0 and $\pi/2$
• Integral $\approx$ sum of areas of rectangles (height $x$ width)
• Several parallel methods will be demonstrated.

\[
\int_a^b \cos(x)dx = \sum_{i=1}^{p} \sum_{j=1}^{n} \cos(a_{ij} + h) \approx \sum_{i=1}^{p} \left[ \sum_{j=1}^{n} \cos(a_{ij} + \frac{h}{2})h \right]
\]
Introduction to OpenACC

- OpenMP is for CPUs, OpenACC is for GPUs
- Has runtime library like OpenMP
- Can mix OpenMP with OpenACC
Laplace Equation

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \]

Boundary Conditions:

\[
\begin{align*}
  u(x,0) &= 0 & 0 \leq x \leq 1 \\
  u(x,1) &= 0 & 0 \leq x \leq 1 \\
  u(0, y) &= u(1, y) = 0 & 0 \leq y \leq 1
\end{align*}
\]
Finite Difference Numerical Discretization

Discretize equation by centered-difference yields:

\[ u_{i,j}^{n+1} \approx \frac{u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n}{4} \quad i = 1,2,\ldots,m; \quad j = 1,2,\ldots,m \]

where \( n \) and \( n+1 \) denote the current and the next time step, respectively, while

\[ u_{i,j}^n = u^n(x_i, y_j) \quad i = 0,1,2,\ldots,m+1; \quad j = 0,1,2,\ldots,m+1 \]

\[ = u^n(i\Delta x, j\Delta y) \]

For simplicity, we take

\[ \Delta x = \Delta y = \frac{1}{m+1} \]
Computational Domain

\[ u(x,1) = 0 \]

\[ u(0,y) = 0 \]

\[ u(x,0) = 0 \]

\[ u_{i,j}^{n+1} \approx \frac{u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n}{4} \quad i = 1,2,\ldots,m; \quad j = 1,2,\ldots,m \]
Five-point Finite-difference Stencil

- **Interior cells.** Where solution of the Laplace equation is sought.

- **Exterior cells.** Green cells denote cells where homogeneous boundary conditions are imposed while non-homogeneous boundary conditions are colored in blue.
Laplace Solver with OpenMP

!$omp parallel do shared(m, n, up, u)   reduction( max:error )
   do j=1,m
      do i=1,n
         up(i,j) = ( u(i+1, j) + u(i-1,j ) + u(i, j-1) + u(i, j+1) ) * 0.25
         error = max( error, abs(up(i, j)-u(i, j)) )
      end do
   end do
end do
!$omp end parallel do

Corresponding C parallel directive is:
#pragma parallel for shared(m,n,up,u)   reduction( max:error )
Laplace Solver with OpenACC

ViewState $\text{acc kernels}$

do $j=1,m$

do $i=1,n$

$$up(i, j) = \left( u(i+1, j) + u(i-1, j) + u(i, j-1) + u(i, j+1) \right) \times 0.25$$

$$error = \max(error, \abs{up(i, j) - u(i, j)})$$

$end do$

$end do$

$\text{acc end kernels}$

• `#pragma acc kernels` for C
• Alternatively, `$acc parallel loop`, `$acc parallel` and `$acc loop` are available. Good to start with kernels . . .
OpenACC data clause

#pragma acc data copy(u), create(up)

while ( error > tol && iter < iter_max ) { error = 0.0;
#pragma acc kernels
for (int i = 1;  i <= n;  i++) {
    for (int j = 1;  j <= m;  j++ ) {
        up[i][j] = ( u[i][j+1] + u[i][j-1] + u[i-1][j] + u[i+1][j]) * 0.25;
        error = fmax( error, fabs(up[i][j] - u[i][j])); } }
#pragma acc kernels
for (int i = 1;  i <= n;  i++) {
    for (int j = 1;  j <= m;  j++ ) {
        u[i][j] = up[i][j]; } }
iter++; }

• copy into and out of region
• copyin only on in
• copyout only on out
• create within region
• Default is copy without data
OpenACC on SCC

• Hardware (GPU)
  • Each node has 3 Nvidia Tesla M2050 GPUs – Nehalem class buy-in 12-core nodes
    • 3 GB memory/gpu, 448 cores/gpu
  • Each node has 8 Nvidia Tesla M2070 GPUs – Nehalem class public 12-core nodes
    • 6 GB memory/gpu, 448 cores/gpu

• Compiler
  • On the SCC, only Portland Group compilers support OpenACC
  • Current (default) version is 13.5

• How to compile codes with OpenACC directives
  • `scc1% pgfortran -o prog prog.f90 -tp=nehalem -acc -ta=nvidia,time -Minfo=accel`
  • `scc1% pgcc -o myprog myprog.c -tp=nehalem -acc -ta=nvidia,time -Minfo=accel`
  • `-tp=nehalem` below creates executable for Intel Nehalem class
  • `-acc` engages the OpenACC API
  • `-ta=nvidia,time` links with Nvidia library for timing data in accelerator region
  • `-Minfo=accel` instructs compiler to display warning and error messages

• Tips from PGI
  • [http://www.pgroup.com/resources/openacc_tips_fortran.htm](http://www.pgroup.com/resources/openacc_tips_fortran.htm)
OpenACC on SCC (cont’d)

- How to run jobs
  Login nodes have no GPUs. Must run via batch scheduler
  - **Interactive batch** -- for program development and debugging
    Example: 1 gpu, 1 cpu, 4 hours of estimated runtime
    - `scc1% qsh -l gpus=1 -l h_rt=04:00:00`
    - `-l gpus=G/C; G = number of GPUs, C = number of CPU cores`
  - **Background Batch** -- for production runs
    Example: 8 GPUs, 12 CPUs, 4 hours of runtime
    - `scc1% qsub -l gpus=0.667 -pe omp 12 -l h_rt=04:00:00`
    - `-l gpus = G/C = 8/12 = 0.667`
    - `scc1% qsub myscript (myscript includes above parameters)`