INTRODUCTION TO OPENMP & OPENACC

Kadin Tseng

Boston University

Scientific Computing and Visualization

Outline

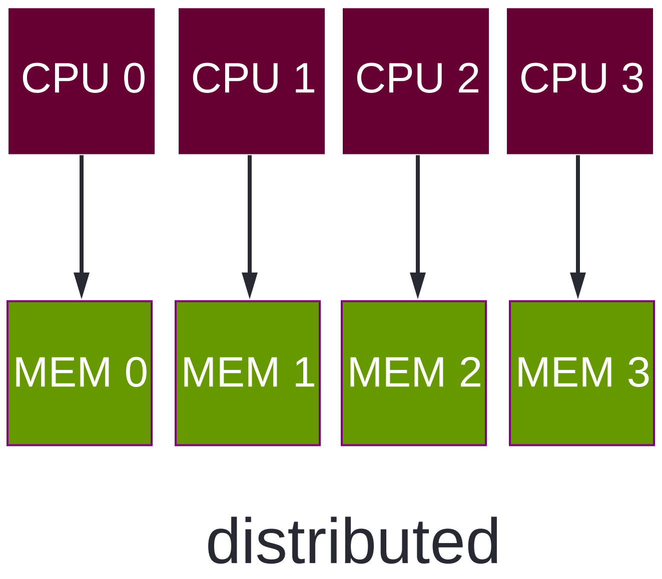
- Introduction to OpenMP
- Introduction to OpenACC

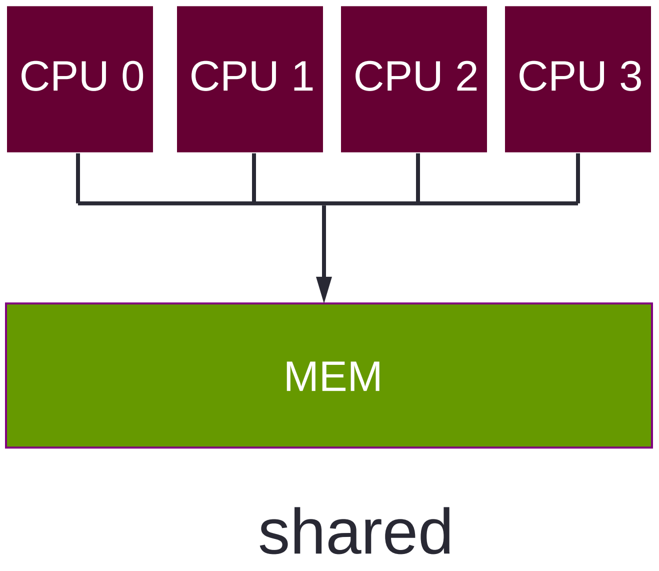
Introduction to OpenMP (for CPUs)

- Types of parallel machines
  - distributed memory
    - each processor has its own memory address space
    - variable values are independent

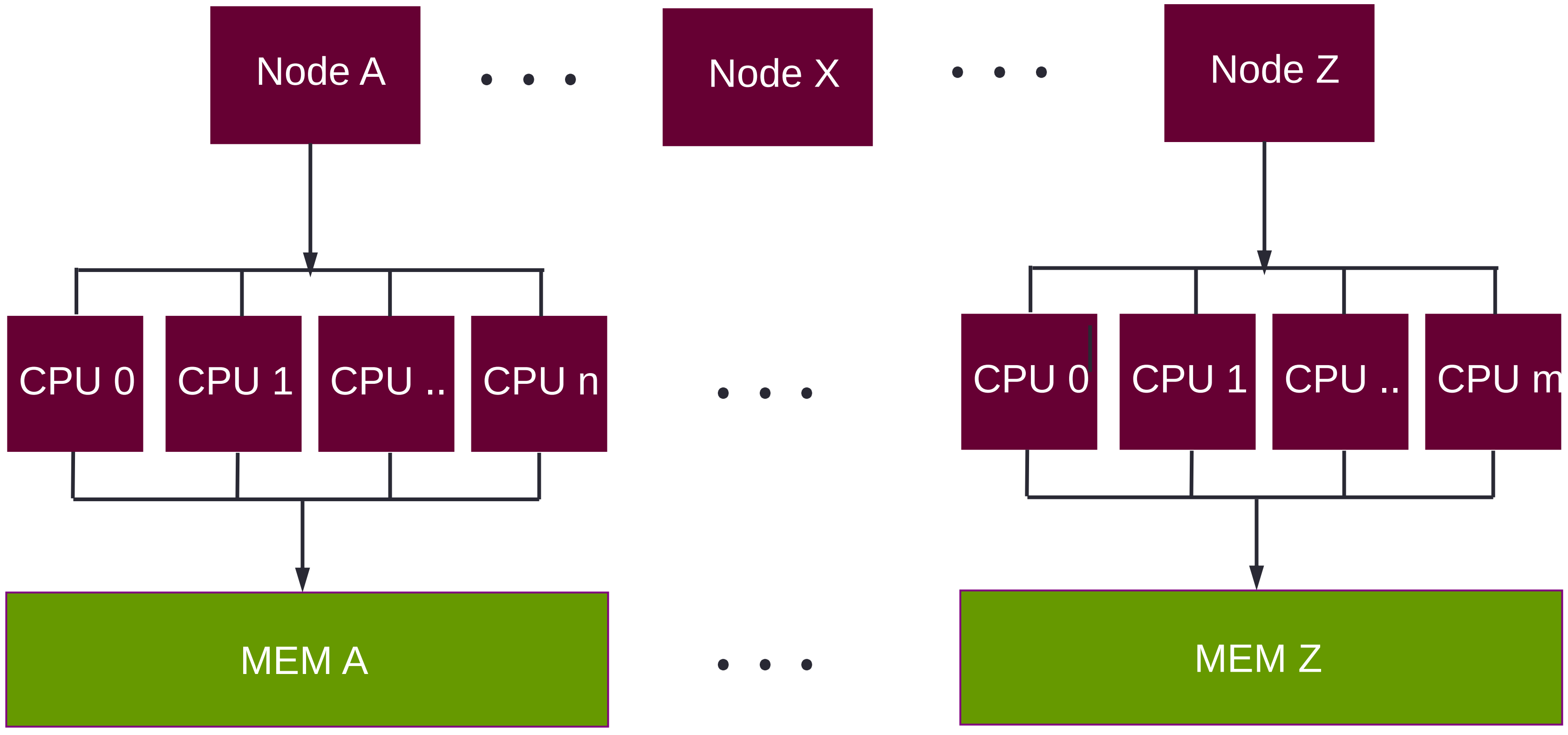
      x = 2 on one processor, x = 3 on a different processor
    - example: the nodes of a Linux cluster, like SCC's nodes
  - shared memory
    - also called Symmetric Multiprocessing (SMP)
    - single address space for all processors
      - If one processor sets x = 2 , x will also equal 2 on other processors (unless specified otherwise)
    - example: cores within each SCC node

Shared vs. Distributed Memory





Shared Computing Cluster (SCC)



shared

distributed among nodes

shared

The word "shared" in SCC is not for "shared memory."

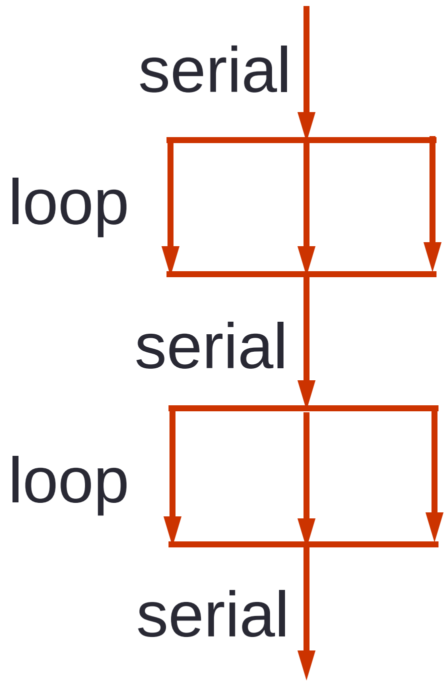
What is OpenMP ?

- Application Programming Interface (API) for *multithreaded* parallelization consisting of
  - Source code directives
  - Functions
  - Environment variables
- Advantage
  - Easy to use
  - Incremental parallelization
  - Flexible -- Loop-level or coarse-grain
  - Portable
    - Work on any SMP machine (*e.g.*, each individual SCC node)
- Disadvantage
  - Shared-memory systems only (*i.e.*, not across SCC's nodes)

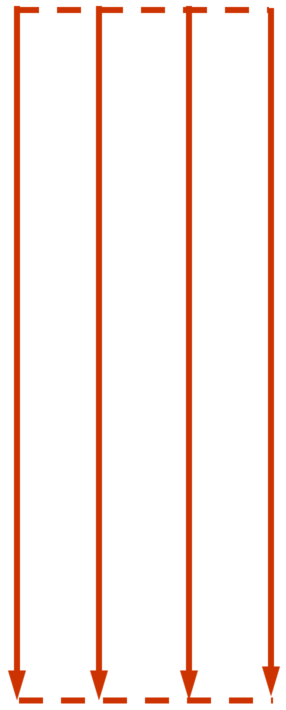
Basics

- Goal – distribute work among threads
- Two methods to be discussed
  - Loop-level
    - Specified loops are parallelized
    - Used in automatic parallelization tools, like MATLAB PCT
  - Parallel regions
    - Also called “coarse-grained parallelism”
    - Usually used in message-passing (MPI)

Basics (cont'd)



Loop-level



Parallel regions

parallel do & parallel for

- parallel do (Fortran) and parallel for (C) directives

```
!$omp parallel do
do i = 1, maxi
    c(i) = a(i) + b(i)
enddo
!$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, all variables are *shared* by default
- Loop indices are *private* by default
- What is wrong with the following code segment ?

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

```
ifirst = 10;     // shared by all threads
#pragma omp parallel for
for(i = 0; i < imax; 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 i2 with a private clause

```
    ifirst = 10    !shared by all threads
    !$omp parallel do private(i2)
    do i = 1, imax   ! i is private
       i2 = 2*i      ! i2 different on each thread
       j(i) = ifirst + i2
    enddo
    !$omp end parallel do
```

```
    ifirst = 10;
    #pragma omp parallel for private(i2)
    for(i = 0; i < imax; i++){   // i is private
       i2 = 2*i;     // i2 different on each thread
       j[i] = ifirst + i2;
    }
```

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, …

```
    a(1) = 0
    a(2) = 1
    do i = 3, 100
       a(i) = a(i-1) + a(i-2)
    enddo
```

Lets 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 ?

```
    a[1] = 0;
    a[2] = 1;
    for(i = 3; i <= 100; i++){
       a[i] = a[i-1] + a[i-2];
    }
```

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

```
ifirst = 10
!$omp parallel do        &
!$omp default(private)  &
!$omp shared(ifirst,imax,j)
do i = 1, imax
   i2 = 2*i
   j(i) = ifirst + i2
Enddo
!$omp end parallel do
```

More clauses (cont'd)

- Can use default none
  - declare all variables (except loop variables) as shared or private
  - If you don't declare any variables, you get a handy list of all variables in loop

More clauses (3)

```
   ifirst = 10
   !$omp parallel do        &
   !$omp default(none)      &
   !$omp shared(ifirst,imax,j) private(i2)
   do i = 1, imax
      i2 = 2*i
      j(i) = ifirst + i2
   enddo
   !$omp end parallel do
```

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

Firstprivate

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

```
    iper = 0
    do i = 1, imax
        iper = iper + 1
        j(i) = iper
    enddo
```

```
    iper = 0;
    for(i = 0; i < imax; i++){
        iper = iper + 1;
        j[i] = iper;
    }
```

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

Firstprivate (cont'd)

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

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

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

Lastprivate

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

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

```
#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

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

```
    s = 0.0;
    #pragma omp parallel for
    for(i = 0; i < imaxi; i++){
        s = s + a[i];
    }
```

Reduction (cont'd)

- Solution is to use the reduction clause

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

```
    s = 0;
    #pragma omp parallel for  reduction(+:s)
    for(i = 0; i < imaxi; 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
  … do stuff …
#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 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 parallelize a loop with a dependency, it will go ahead and do it!
- Do not parallelize small loops
  - Overhead will be greater than speedup
  - How small is “small”?
    - Answer depends on processor speed and other system-dependent parameters
- Maximize number of operations performed in parallel
  - parallelize outer loops where possible
- For fortran, add “use omp_lib” to include header

Compile and Run on SCC

- Portland Group compilers:
  - Compile with -mp flag to turn on OpenMP
  - Depending on the node, can use up to 16 threads
  - scc1% pgfortran –o myprog myprog.f90 –mp –O3
  - scc1% pgcc –o myprog myprog.c –mp –O3
  - scc1% setenv OMP_NUM_THREADS 4
  - scc1% myprog
- GNU compilers:
  - Compile with -fopenmp flag toturn on OpenMP
  - Depending on the node, can use up to 16 threads
  - scc1% gfortran –o myprog myprog.f90 –fopenmp –O3
  - scc1% gcc –o myprog myprog.c –fopenmp –O3

Parallel

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

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

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

is the same as

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

```
#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.
- 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:

```
!$omp parallel
!$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
```

```
#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-Grained

- OpenMP is not restricted to loop-level, or fine-grained, parallelism.
- The !$omp parallel or #pragma omp parallel directive duplicates subsequent code on all threads until a !$omp end parallel or #pragma omp end parallel directive is encountered.
- Allows parallelization similar to “MPI paradigm.”

Coarse-Grained (cont'd)

```
!$omp parallel &
!$omp private(myid,istart,iend,nthreads,nper)
nthreads = omp_get_num_threads()
nper = imax/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
```

```
#pragma omp parallel \
#pragma omp private(myid,istart,iend,nthreads,nper)
nthreads = OMP_GET_NUM_THREADS();
nper = imax/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

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

```
#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

```
!$OMP PARALLEL PRIVATE(myid,istart,iend)
 call myrange(myid,istart,iend)
 do i = istart, iend
    a(i) = a(i) - b(i)
 enddo
 !$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

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

Single Example - C

```
#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

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

Critical 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 critical
 mycrit(myid,a);
 #pragma omp end critical
 do_work(istart,iend);
 #pragma omp end parallel
```

Ordered

- Suppose you want to write values in a loop:

```
do i = 1, nproc
  call do_lots_of_work(result(i))
  write(21,101) 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)

```
   !$omp parallel do
   do i = 1, nproc
     call do_lots_of_work(result(i))
     !$omp ordered
     write(21,101) i, result(i)
     !$omp end ordered
   enddo
```

```
   #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

Hands On Exercise

- Parallelize a serial C or Fortran code with OpenMP
- The code invokes a function multiple times via a for/do loop. Parallelize the loop with an OpenMP directive
- The function, *mywork*, also has a loop (no need to parallelize) whose iteration count is randomly chosen each time it is invoked. Hence, *mywork*'s workload varies with each iteration. This causes load imbalance. Use an appropriate OpenMP clause to address this problem
- A serial myprog.c and myprog.f90 are available for you to start with.

  Look in /scratch/kadin

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:

$$u(x,0) = 0 \qquad 0 \le x \le 1$$

$$u(x,1) = 0 \qquad 0 \le x \le 1$$

$$u(0,y) = u(1,y) = 0 \qquad 0 \le y \le 1$$

Finite Difference Numerical Discretization

Discretize equation by centered-difference yields:

$$u_{i,j}^{n+1} \cong \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4} \qquad i = 1,2,\ldots,m; \;\; j = 1,2,\ldots,m$$

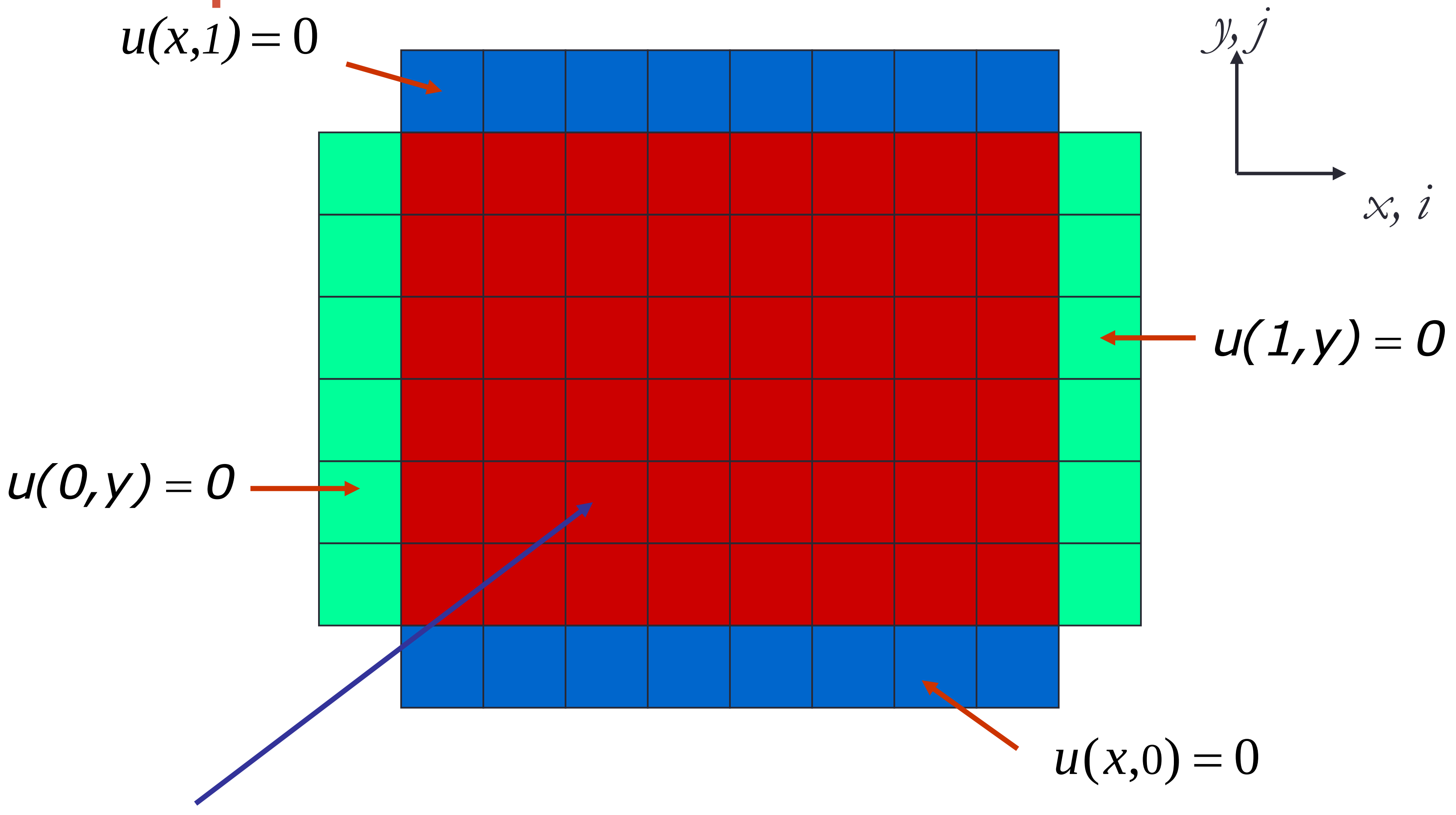
where *n* and *n+1* denote the current and the next time step, respectively, while

$$\begin{aligned} u_{i,j}^{n} &= u^{n}(x_i, y_j) \qquad i = 0,1,2,\ldots,m+1; \;\; j = 0,1,2,\ldots,m+1 \\ &= u^{n}(i\Delta x, j\Delta y) \end{aligned}$$

For simplicity, we take

$$\Delta x = \Delta y = \frac{1}{m+1}$$

Computational Domain



$$u_{i,j}^{n+1} \cong \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4}$$

$i = 1,2,\ldots,m; \quad j = 1,2,\ldots,m$

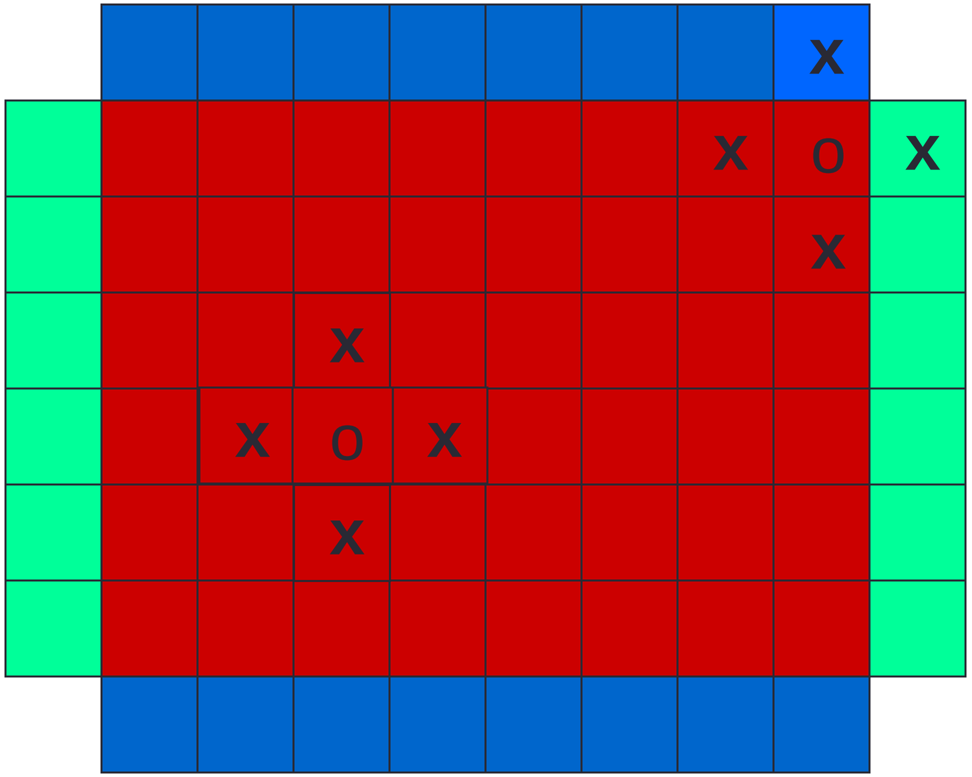
Five-point Finite-difference Stencil

🟥 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

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

Corresponding C parallel directive is:

```
#pragma parallel  for  shared(m,n,up,u)   reduction( max:error )
```

Laplace Solver with OpenACC

```
!$acc kernels
   do j=1,m
      do i=1,n
         up(i, j) = ( u(i+1, j)  + u(i-1, j) +  u(i, j-1) + u(i, j+1) ) * 0.25
         error = max( error, abs(up(i, j) - u(i, j)) )
      end do
   end do
!$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

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)