Lecture 13: Parallel Programming with OpenMP

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Today's Class

Acknowledgements: Slides heavily borrowed from following three sources:

a) ECE563, Purdue University, Dr. Seung-Jai Min

b) COMP422, Rice University, Dr. Vivek Sarkar

c) The tutorial "Programming Irregular Applications with OpenMP" that was given at SC 2016 (Salt Lake City, Utah). Presenters of this tutorial were Dr. Tim Mattson, Dr. Alice Koniges, Dr. Clay Breshears, and Dr. Jeremy Kemp

Today's Class

- **Introduction to OpenMP**
- Work-sharing pragmas
- Tasking pragmas in OpenMP

What is OpenMP

- De-facto standard API to write shared memory parallel applications in C / C++ / Fortran
- Consists of compiler directives, runtime routines and environment variables

Hello World using OpenMP

```
int main() {



    printf("Hello World\n");

  printf("Must execute sequentially\n");
  return 0;
}
```

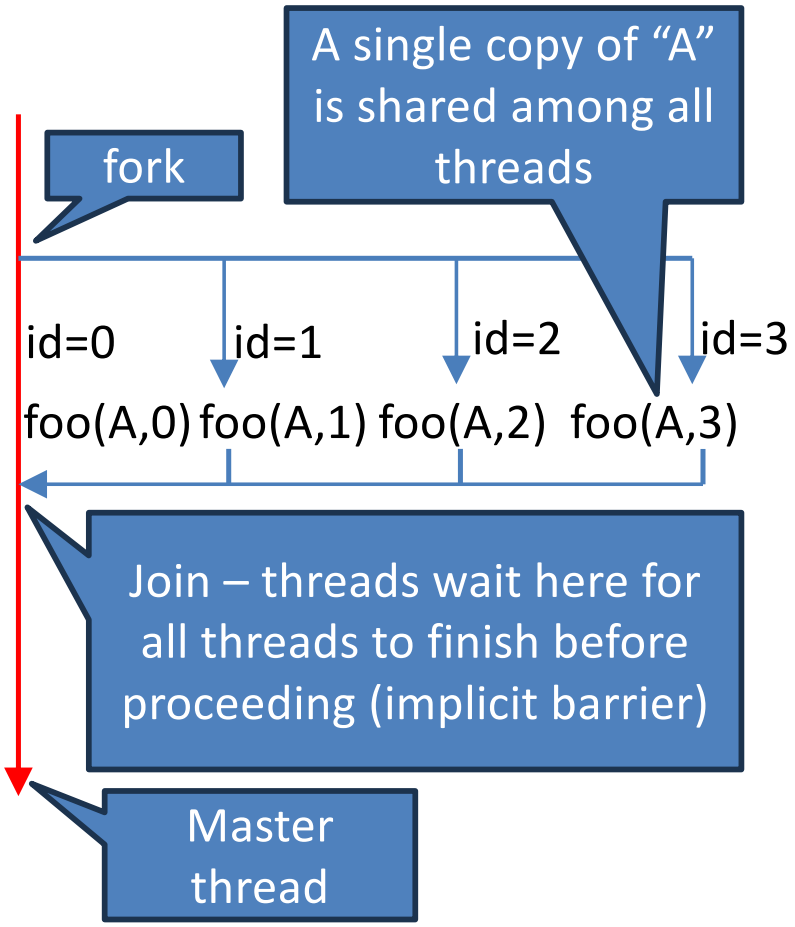
Hello World using OpenMP

```
#include<omp.h>
int main() {
  omp_set_num_threads(4);
  #pragma omp parallel
  {
    printf("Hello World\n");
  }
  printf("Must execute sequentially\n");
  return 0;
}
```

- All OpenMP directives begin with #pragma
- #pragma omp parallel
  - Forks a group of 4 threads where each thread executes the copy of the code within the structured block
- Compile
  - gcc prog.c -fopenmp

Hello World using OpenMP

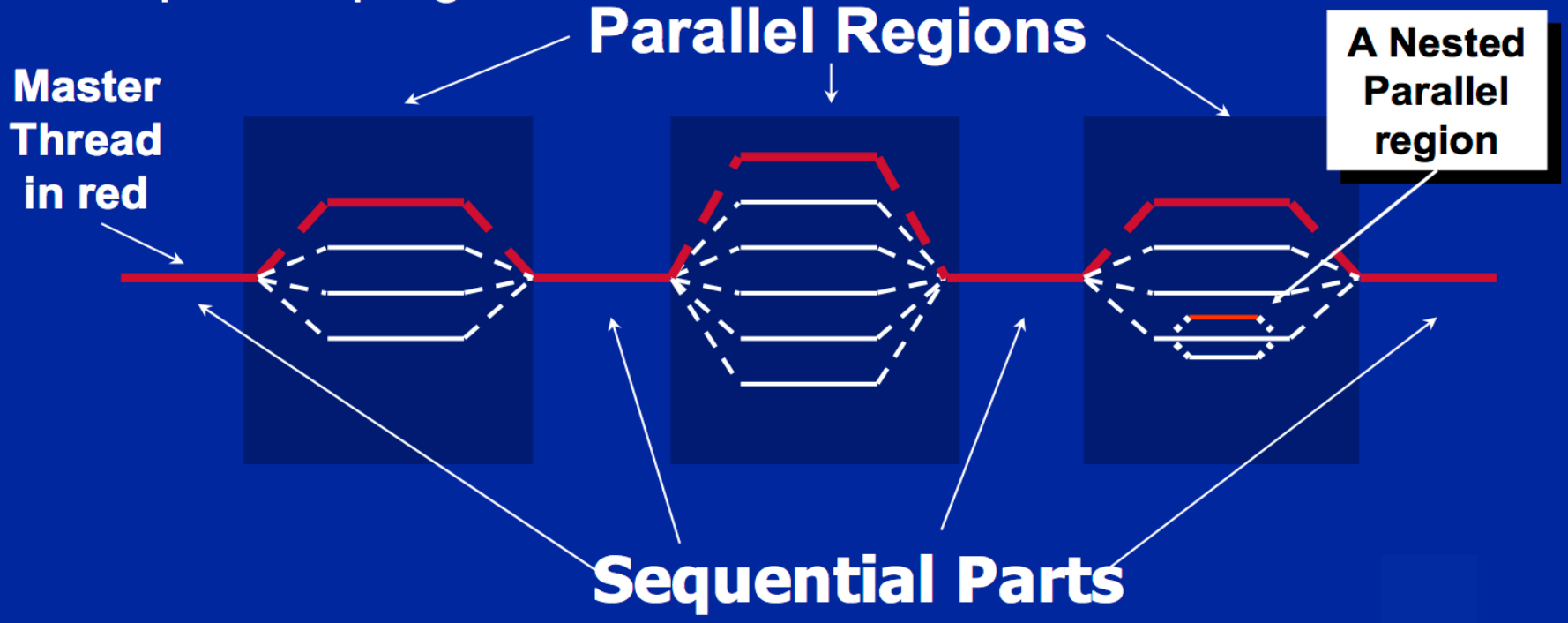
```
#include<omp.h>
int main() {
  int A[100];
  omp_set_num_threads(4);
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    foo(A, id);
  }
  printf("Must execute sequentially\n");
  return 0;
}

void foo(int* A, int id) { ..... }
```



OpenMP Programming Model:

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



Specifying Total Number of Threads

- Globally via command line during runtime
  - OMP_NUM_THREADS=N ./a.out
    - Lowest priority
- Inside the program using runtime API
  - omp_set_num_threads(N);
    - Higher priority than OMP_NUM_THREADS
- Inside the program but different count for each fork/join block
  - #pragma omp parallel num_threads(N)
    - Highest priority than the other two techniques above

Today's Class

- Introduction to OpenMP
- **Work-sharing pragmas**
- Tasking pragmas in OpenMP

How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Sequential Program

```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
}
```

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

How is OpenMP typically used? (Cont.)

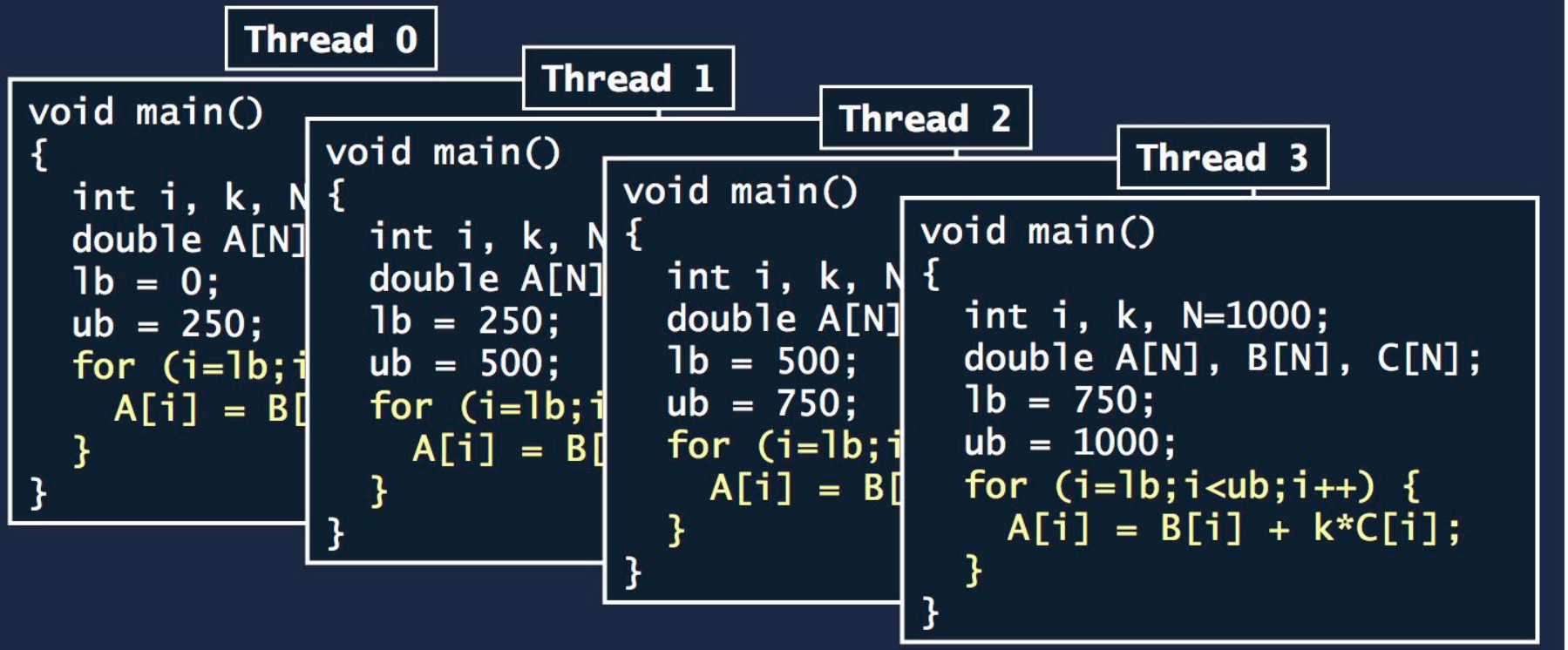
- Single Program Multiple Data (SPMD)

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

How is OpenMP typically used? (Cont.)

- Single Program Multiple Data (SPMD)

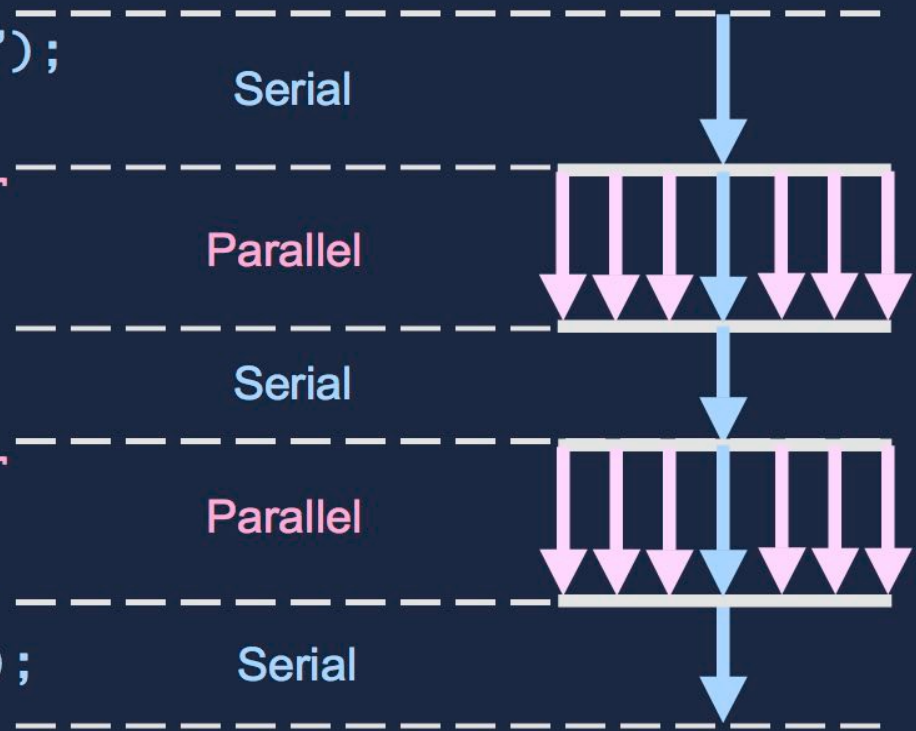


OpenMP Fork-and-Join model

```
printf("program begin\n");
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf("program done\n");
```

OpenMP Fork-and-Join model

```
printf("program begin\n");
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

printf("program done\n");
```



The OpenMP API
Combined parallel work-share

- OpenMP shortcut: Put the "parallel" and the work-share on the same line

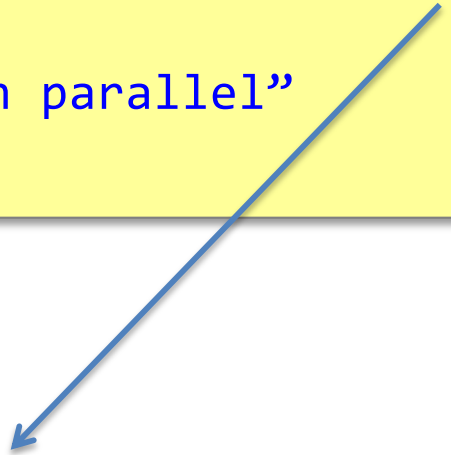
```
int i;
double  res[MAX];
#pragma omp parallel
{
  #pragma omp for
  for (i=0;i< MAX; i++) {
    res[i] = huge();
  }
}
```

```
int i;
double  res[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
  res[i] = huge();
}
```

the same OpenMP

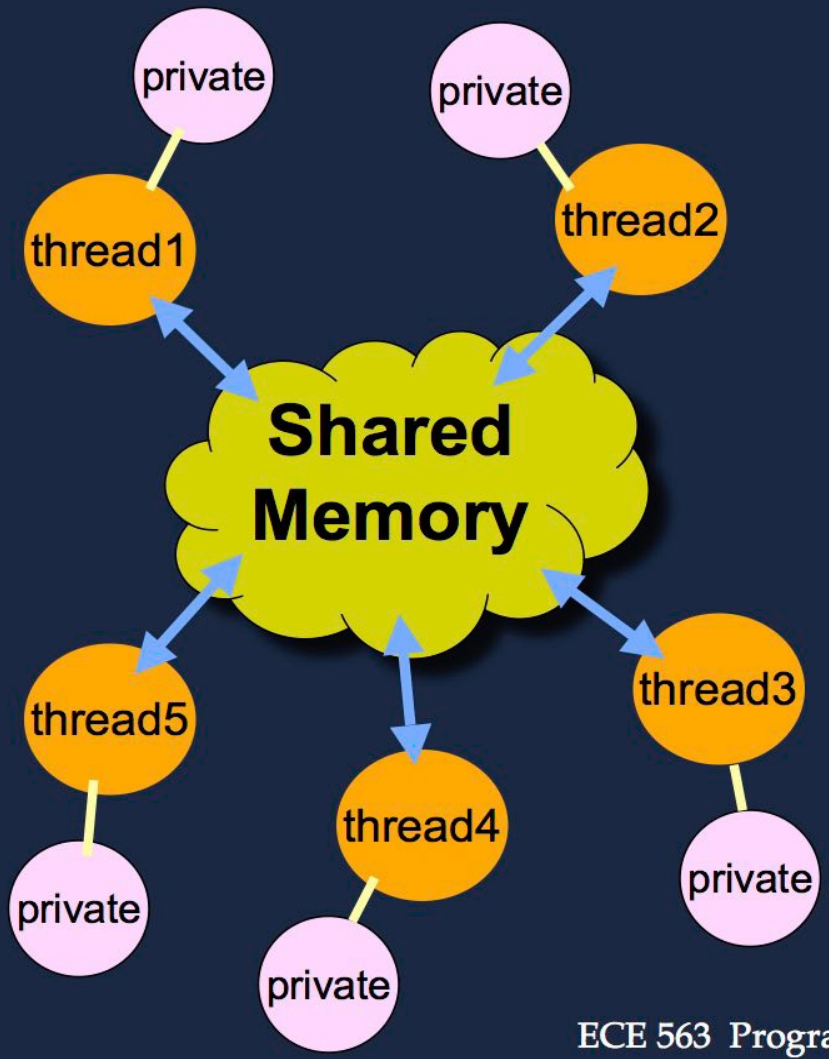
OpenMP Clauses

```
#pragma omp parallel [clauses[[,] clauses] ...]
{
    “this is executed in parallel”
} (implied barrier)
```



```
if                  (scalar expression)
private             (list)
shared              (list)
default             (none | shared)
reduction           (operator : list)
firstprivate        (list)
num_threads         (scalar_integer_expression)
```

Shared Memory Model



- Data can be shared or private
- Shared data is accessible by all threads
- Private data can be accessed only by the thread that owns it
- Data transfer is transparent to the programmer

Data Environment

```
int foo(int x)
{
  /* PRIVATE */
  int count=0;
  return x*count;
}
```

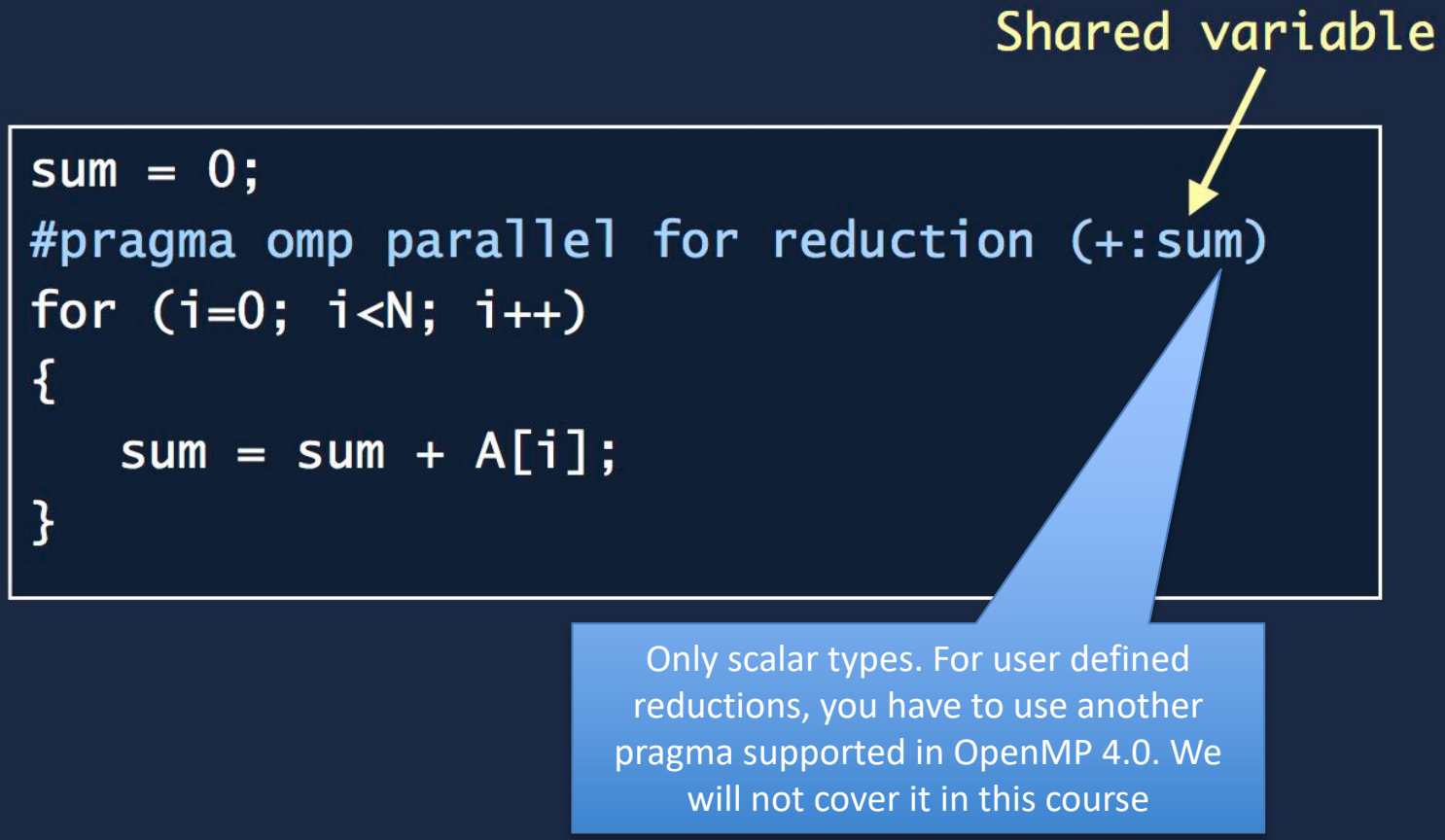
```
int A[100]; /* (Global) SHARED */

int main()
{
  int ii, jj;     /* PRIVATE */
  int B[100];  /* SHARED */
#pragma omp parallel private(jj)
  {
     int kk = 1;      /* PRIVATE */
#pragma omp for
     for (ii=0; ii<N; ii++)
        for (jj=0; jj<N; jj++)
           A[ii][jj] = foo(B[ii][jj]);
  }
}
```

"Private" as these are used as work-sharing loop iterator variable, else shared scope

Data Environment

- “default(none)”
  - Best programming practice
  - All local variables (including loop iterators) declared outside the parallel region cannot be accessed inside the parallel region without explicitly declaring the sharing mode
    - Compilation error otherwise
- “default(shared)”
  - All local variables declared outside the parallel region will be shared among all the threads inside the parallel region
- “shared(var_a, var_b)”
  - Local variables “var_a” and “var_b” are being shared among all the threads inside the parallel region
- “firstprivate(var_a)”
  - Same as “private” except that threads get a private copy of “var_a” initialized with the last known value for this variable just before the start of parallel region

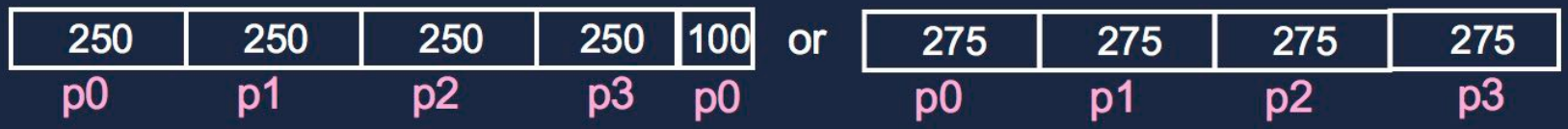
Reduction Clause



Schedule

```
for (i=0; i<1100; i++)
   A[i] = … ;
```

#pragma omp parallel for schedule (static, 250) or (static)



#pragma omp parallel for schedule (dynamic, 200)

No fixed mapping between threads and chunks

#pragma omp parallel for schedule (guided, 100)

No fixed mapping between threads and chunks

#pragma omp parallel for schedule (auto)

Schedule (Summary)

- **static**
  - Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads
- **dynamic**
  - Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1
- **guided**
  - Similar to “dynamic” except that the block size decreases each time a parcel of work is given to a thread. The size of the initial block is proportional to “number_of_iterations/numThreads”. Subsequent blocks are proportional to “number_of_iterations_remaining/numWorkers”
- **auto**
  - The scheduling decision is delegated to the compiler and/or runtime system

Out-of-line ("orphaned") directives

- *The OpenMP standard does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be orphaned*
- *That is, they can appear outside the lexical extent of a parallel region*

```
    (void) dowork(); !- Sequential FOR

#pragma omp parallel
{
    (void) dowork(); !- Parallel FOR
}
```

```
void dowork()
{
#pragma omp for
    for (i=0;....)
    {
        :
    }
}
```

- *When an orphaned worksharing or synchronization directive is encountered in the sequential part of the program (outside the dynamic extent of any parallel region), it is executed by the master thread only. In effect, the directive will be ignored*

Synchronization in OpenMP

- Implicit barriers
- #pragma omp barrier
- #pragma omp critical
- *Few more techniques that are out of scope of this course (locks, atomic, flush, and ordered)*

Critical Construct

```
sum = 0;
#pragma omp parallel private (lsum)
{
    lsum = 0;
    #pragma omp for
    for (i=0; i<N; i++) {
        lsum = lsum + A[i];
    }
    #pragma omp critical
    { sum += lsum; }
}
```

Threads wait their turn;
only one thread at a time
executes the critical section

Today's Class

- Introduction to OpenMP
- Work-sharing pragmas
- **Tasking pragmas in OpenMP**

Task Directive

`#pragma omp task` *[clauses]*

structured-block

```
#pragma omp parallel num_threads(NP)
{

   #pragma omp task
      fred();
   #pragma omp task
      daisy();
   #pragma omp task
      billy();

}
```

Total NP tasks created at each pragma (task duplication

Tasks executed by some thread in some order

All tasks complete before this barrier is released

Task Directive

`#pragma omp task` *[clauses]*

structured-block

```
#pragma omp parallel num_threads(NP)
{
   #pragma omp master
    {
    #pragma omp task
       fred();
    #pragma omp task
       daisy();
    #pragma omp task
       billy();
    }
}
```

Thread 0 packages tasks

Tasks executed by some thread in some order

All tasks complete before this barrier is released

When/where are tasks complete?

- At thread barriers (explicit or implicit)
  - C/C++: **#pragma omp barrier**
  - All tasks created by any thread are guranteed to be completed at barrier exit

- At taskwait directive
  - i.e. Wait until all tasks defined in the current task have completed.
  - C/C++: **`#pragma omp taskwait`**

  - The code executed by a thread in a parallel region is considered a task here
  - Note: applies only to tasks generated in the current task
    - Waits for children of current task
    - Does not wait for grand children
      - Different than Hclib::finish that waits for all tasks created under the finish scope

Example

```
#pragma omp parallel num_threads(NP)
{
   #pragma omp master
    {
    #pragma omp task
       fred();
    #pragma omp task
       daisy();
     #pragma omp taskwait
    #pragma omp task
       billy();
     }
}
```

Can we have?

`#pragma omp single`

Yes, but "single" has an implicit barrier unlike "master"

`fred()` and **`daisy()`** must complete before **`billy()`** starts

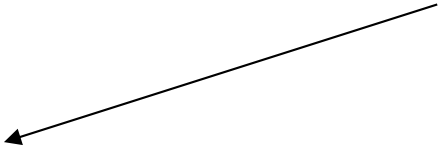
Data Scoping with Tasks

Clause	omp for	omp task
shared	Shared between all the **threads**	Shared between all **tasks** generated within the current task
private	Each **thread** gets it own **local** copy of the variable which is uninitialized	Each **task** gets it own local copy of the variable which is uninitialized
firstprivate	Each thread gets its own local copy, which is initialized with the value the variable held immediately before the omp for construct was encountered by the thread	Each task gets its own local copy, initialized with the value the variable held immediately before the omp task construct was encountered in the generating task region

Data Scoping Defaults

- The behavior you want for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are private when the task construct is encountered are firstprivate by default
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
    ...
#pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```



A is shared
B is firstprivate
C is private

Data scoping defaults (1/6)

Best Practice to avoid unexpected results !!

```
int a=1, b=2;
#pragma omp parallel default(none)
{
    int c=3;
    #pragma omp task
    {
    printf("IN: a=%d, b=%d, c=%d\n",a++,b++,c++);
    }
    #pragma omp taskwait
    printf("OUT: c=%d\n",c);
}
printf("OUT: a=%d, b=%d\n",a,b);
```

What will be output ? (OMP_NUM_THREADS=1)
>>compilation error !!

Data scoping defaults (2/6)

Best Practice to avoid unexpected results !!

```
int a=1, b=2;
#pragma omp parallel default(none) shared(a,b)
{
    int c=3;
    #pragma omp task
    {
    printf("IN: a=%d, b=%d, c=%d\n",a++,b++,c++);
    }
    #pragma omp taskwait
    printf("OUT: c=%d\n",c);
}
printf("OUT: a=%d, b=%d\n",a,b);
```

What will be output ? (OMP_NUM_THREADS=1)

>> IN: a=1,b=2,c=3

>> OUT: c=3

>> OUT: a=2,b=3

Data scoping defaults (3/6)

Best Practice to avoid unexpected results !!

```
int a=1, b=2;
#pragma omp parallel default(none) private(a,b)
{
    int c=3;
    #pragma omp task
    {
    printf("IN: a=%d, b=%d, c=%d\n",a++,b++,c++);
    }
    #pragma omp taskwait
    printf("OUT: c=%d\n",c);
}
printf("OUT: a=%d, b=%d\n",a,b);
```

What will be output ? (OMP_NUM_THREADS=1)

>> IN: a=0,b=0,c=3

>> OUT: c=3

>> OUT: a=1,b=2

Data scoping defaults (4/6)

Best Practice to avoid unexpected results !!

```
int a=1, b=2;
#pragma omp parallel default(none) firstprivate(a,b)
{
    int c=3;
    #pragma omp task
    {
    printf("IN: a=%d, b=%d, c=%d\n",a++,b++,c++);
    }
    #pragma omp taskwait
    printf("OUT: c=%d\n",c);
}
printf("OUT: a=%d, b=%d\n",a,b);
```

What will be output ? (OMP_NUM_THREADS=1)

>> IN: a=1,b=2,c=3

>> OUT: c=3

>> OUT: a=1,b=2

Data scoping defaults (5/6)

Best Practice to avoid unexpected results !!

```
int a=1, b=2;
#pragma omp parallel default(none) shared(a) private(b)
{
    int c=3;
    #pragma omp task
    {
    printf("IN: a=%d, b=%d, c=%d\n",a++,b++,c++);
    }
    #pragma omp taskwait
    printf("OUT: c=%d\n",c);
}
printf("OUT: a=%d, b=%d\n",a,b);
```

What will be output ? (OMP_NUM_THREADS=1)

>> IN: a=1,b=0,c=3

>> OUT: c=3

>> OUT: a=2,b=2

Data scoping defaults (6/6)

Best Practice to avoid unexpected results !!

```
int a=1, b=2;
#pragma omp parallel default(none) shared(a) private(b)
{
    int c=3;
    b = 1;
    #pragma omp task
    {
    printf("IN: a=%d, b=%d, c=%d\n",a++,b++,c++);
    }
    #pragma omp taskwait
    printf("OUT: c=%d,b=%d\n",c,b);
}
printf("OUT: a=%d, b=%d\n",a,b);
```

What will be output ? (OMP_NUM_THREADS=1)

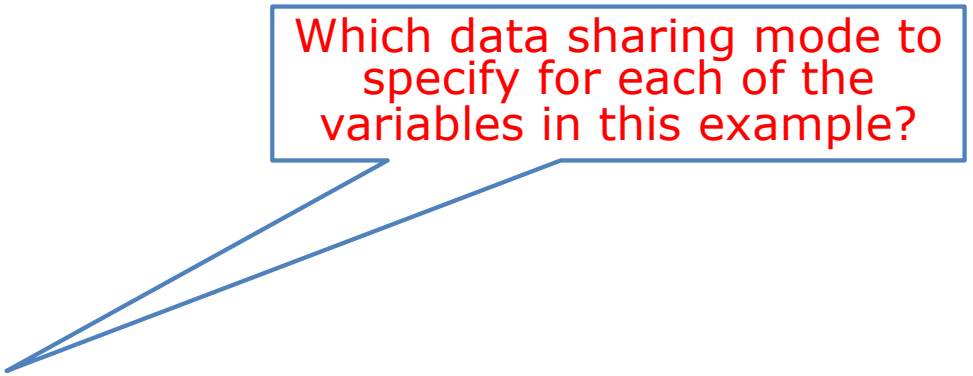
>> IN: a=1,b=1,c=3

>> OUT: c=3,b=1

>> OUT: a=2,b=2

Example: Fibonacci numbers

```
int fib (int n)
{
  int x,y;
  if (n < 2) return n;

  x = fib(n-1);
  y = fib (n-2);
  return (x+y);
}

Int main()
{
  int NW = 40;
  fib(NW);
}
```



Parallel Fibonacci

```
int fib (int n)
{   int x,y;
   if (n < 2) return n;

#pragma omp task shared(x)
   x = fib(n-1);
#pragma omp task shared(y)
   y = fib (n-2);
#pragma omp taskwait
   return (x+y);
}

Int main()
{  int NW = 40;
   #pragma omp parallel
   {
      #pragma omp master
           fib(NW);
   }
}
```

You must specify “shared” for “x” and “y”, as otherwise they will become “fristprivate” to tasks

Task switching

- Consider the following example … Where the program may generate so many tasks that the internal data structures managing tasks overflow.

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

Task switching

- Consider the following example … Where the program may generate so many tasks that the internal data structures managing tasks overflow.

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task untied
      process(item[i]);
}
```

- Solution … Task switching; Threads can switch to other tasks at certain points called ***thread scheduling*** points.
- With Task switching, a thread can
  - Execute an already generated task … to "drain the task pool"
  - Execute the encountered task immediately (instead of deferring task execution for later)

if Clause

#pragma omp task if(expr)

- If the expression of an if clause on a task evaluates to false
  - The encountering task is suspended
  - The new task is executed immediately
  - The parent task resumes when new tasks finishes
  - Used for optimization, e.g. avoid creation of small tasks

Parallel Fibonacci (**Improved**)

```
int fib (int n)
{   int x,y;
   if (n < 2) return n;

#pragma omp task untied shared(x) if(n>20)
   x = fib(n-1);
#pragma omp task untied shared(y) if(n>20)
   y = fib (n-2);
#pragma omp taskwait
   return (x+y);
}

int main()
{  int NW = 40;
   #pragma omp parallel
   {
      #pragma omp master
           fib(NW);
   }
}
```

Next Class

- Mid semester review

Reading Material

- OpenMP tutorial from LLNL
  – https://computing.llnl.gov/tutorials/openMP/