CSE502: Foundations of Parallel Programming

Lecture 18: OpenMP Work-Sharing Pragmas

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Class: Intro. to OpenMP Programming

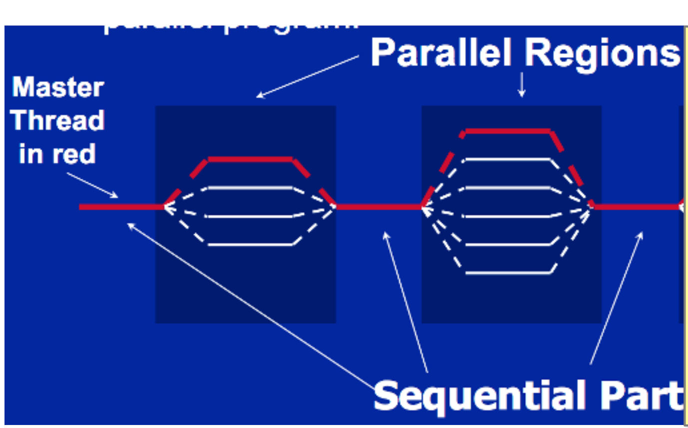
```
#include "hclib_cpp.h"
main() {
  launch([&]() {
    finish ([=]() {
        async([=](){ printf("Hello"\n");});
        async([=](){ printf("Hello"\n");});
    });
  });           HClib
}
```

```
#include <omp.h>
main() {
  #pragma omp parallel num_threads(2)
  {
    printf("Hello"\n");        OpenMP
  }
} /* OMP_NUM_THREADS=2 ./a.out */
```

```
#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];
  }
}
```

Single Program Multiple Data
(SPMD)

num_threads has higher precedence than OMP_NUM_THREADS / omp_set_num_threads()




```
#include <omp.h>
main() {
  do_sequential();
  #pragma omp parallel num_threads(4)
    printf("Hello"\n");
  do_sequential();
  #pragma omp parallel num_threads(6)
    printf("Hello"\n");
} /* OMP_NUM_THREADS=2 ./a.out */
```

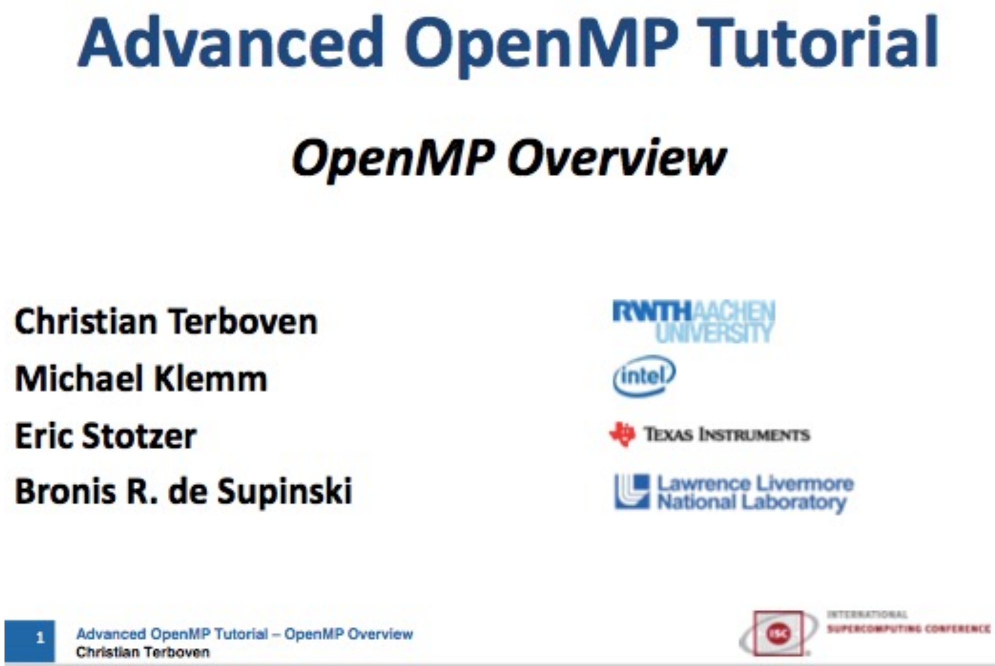
Today’s Class

- Work-sharing constructs in OpenMP (contd.)
  - Data sharing modes
- Synchronization in OpenMP

Acknowledgements: Slides heavily borrowed from following two sources:

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

b) COMP422, Rice University, Dr. Vivek Sarkar



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

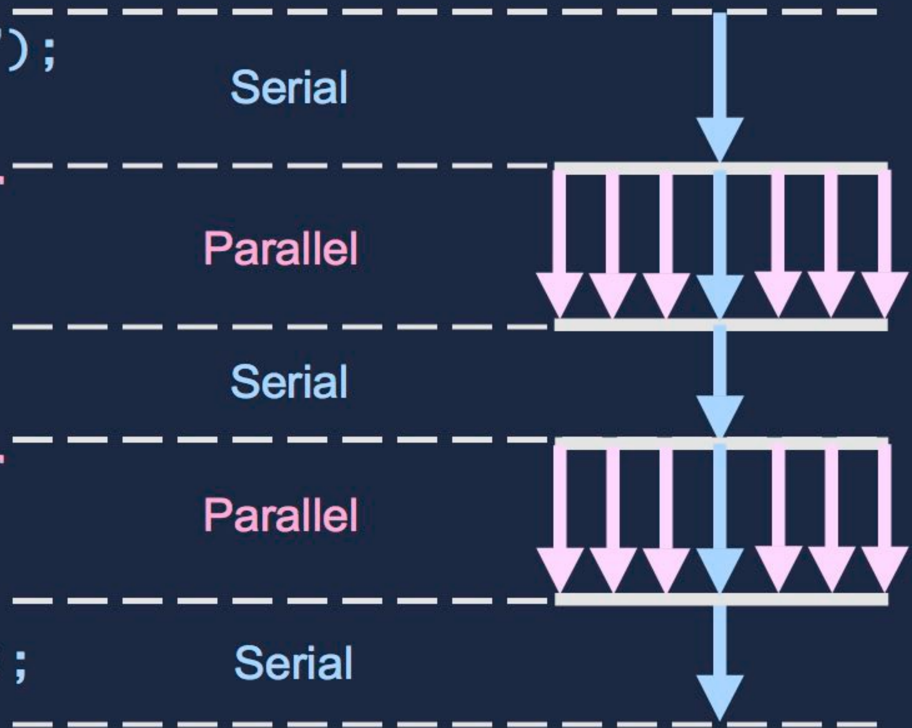
```
printf("program begin\n");
N = 1000;

#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();
}
```

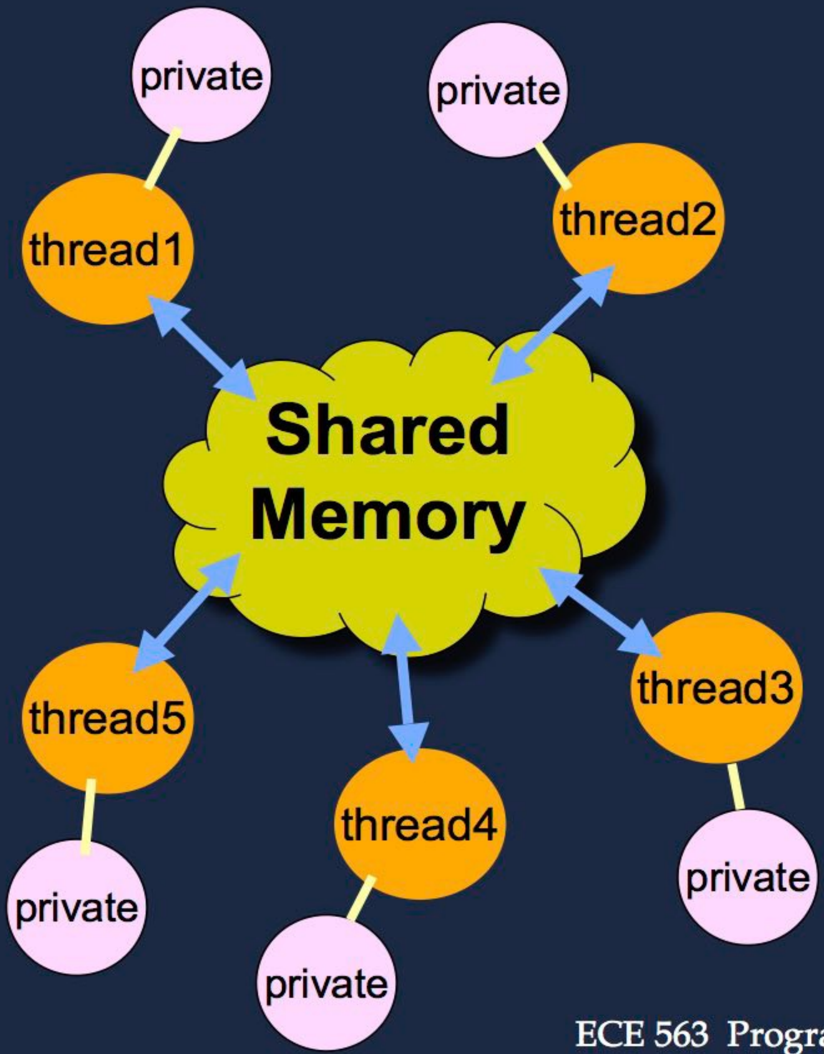
the same OpenMP

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

ECE 563 Programming Parallel Machines

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

Work-sharing constructs in a Parallel Region

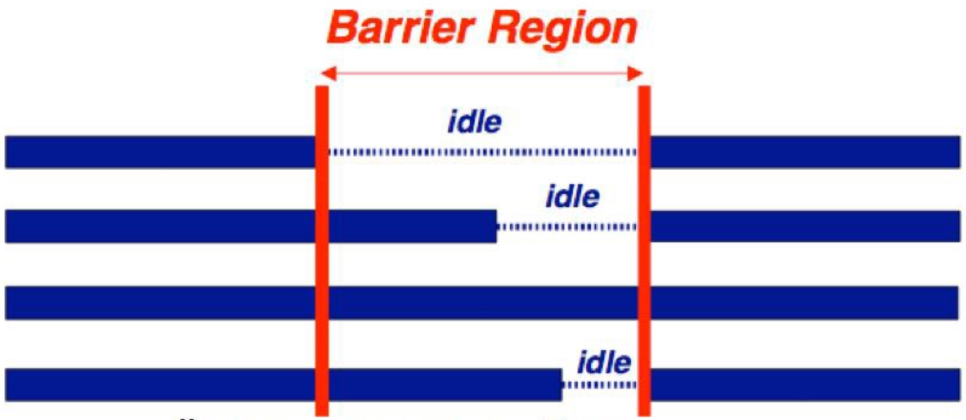
```
#pragma omp for
{
    ....
}
```

```
#pragma omp sections
{
        ....
}
```

```
#pragma omp single
{
        ....
}
```

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless nowait is specified)
- A work-sharing construct does not launch any new threads

Implicit barrier



Do you see any issues here?

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

Implicit barrier — wait ! — barrier

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

NOTE: barrier is redundant if there is a guarantee that the mapping of iterations onto threads is identical in both loops

nowait clause & explicit barrier

```
#pragma omp for nowait
{
        :
}
```

```
#pragma omp barrier
```

- To minimize synchronization, some OpenMP directives/pragmas support the optional *nowait* clause
- If present, threads do not synchronize/wait at the end of that particular construct
- An explicit barrier can then be inserted at only the desired program points

A more elaborate example

```c
#pragma omp parallel if (n>limit) default(none) \
        shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
#pragma omp for nowait

    for (i=0; i<n; i++)
       z[i] = x[i] + y[i];

#pragma omp for nowait

    for (i=0; i<n; i++)
       a[i] = b[i] + c[i];

#pragma omp barrier
          ....
    scale = sum(a,0,n) + sum(z,0,n) + f;
          ....
} /*-- End of parallel region --*/
```

Statement is executed by all threads

parallel loop (work is distributed)

parallel loop (work is distributed)

synchronization

Statement is executed by all threads

parallel region

“single” and “master” constructs in a parallel region

Only one thread in the team executes the code enclosed

```
#pragma omp single [clause[[,] clause] ...]
{
        <code-block>
}
```

Only the master thread executes the code block,

```
#pragma omp master
{<code-block>}
```

- Single and master are useful for computations that are intended for single-processor execution e.g., I/O and initializations
- There is no implied barrier on entry or exit of a master construct

OpenMP Sections

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
       for (i=0; i<n-1; i++)
           b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
       for (i=0; i<n; i++)
           d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```

Reduction Clause

Shared variable

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

Only scalar types. For user defined reductions, you have to use another pragma supported in OpenMP 4.0. We will not cover it in this course

Schedule

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

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

250	250	250	250	100
p0	p1	p2	p3	p0

or

275	275	275	275
p0	p1	p2	p3

#pragma omp parallel for schedule (dynamic, 200)

200	200	200	200	200	100

No fixed mapping between threads and chunks

#pragma omp parallel for schedule (guided, 100)

275	206	109	100	100	100	100	100	10

No fixed mapping between threads and chunks

#pragma omp parallel for schedule (auto)

Schedule

- **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*

OpenMP Library Functions

- In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs

```
/* thread and processor count */
void omp_set_num_threads(int num_threads);
int omp_get_num_threads();
int omp_get_thread_num();
int omp_get_num_procs();
int omp_in_parallel();
```

Synchronization in OpenMP

- Implicit barriers
- #pragma omp barrier
- #pragma omp critical
- Simple locks and nested locks
- *Few more techniques that are out of scope of this course (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

OpenMP Locks

- *Simple locks: may not be locked if already in a locked state*
- *Nestable locks: may be locked multiple times by the same thread before being unlocked*
- *In the remainder, we discuss simple locks only*
- *The interface for functions dealing with nested locks is similar (but using nestable lock variables):*

Simple locks	Nestable locks
omp_init_lock	omp_init_nest_lock
omp_destroy_lock	omp_destroy_nest_lock
omp_set_lock	omp_set_nest_lock
omp_unset_lock	omp_unset_nest_lock
omp_test_lock	omp_test_nest_lock

Next Class

- Tasks based parallelism in OpenMP
- Lab-5 next week (Monday)
  – Syllabus: Lectures 17-18
- Quiz-4 on Thursday in lecture slot
  – Syllabus: Lectures 17-19 (OpenMP)

Reading Material

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