OpenMP - 1

CSE 625

Fall 2022

Computer Engineering and Computer Science

University of Louisville

Sources and References

[1] Official OpenMP Website

http://www.openmp.org

[2] Microsoft OpenMP Functions Reference

https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-functions?view=vs-2019

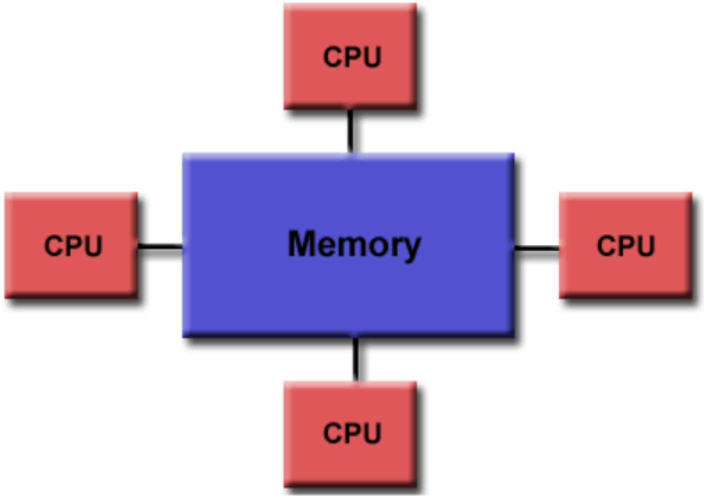
[3] VS 2019 OpenMPExamples Project

Contents

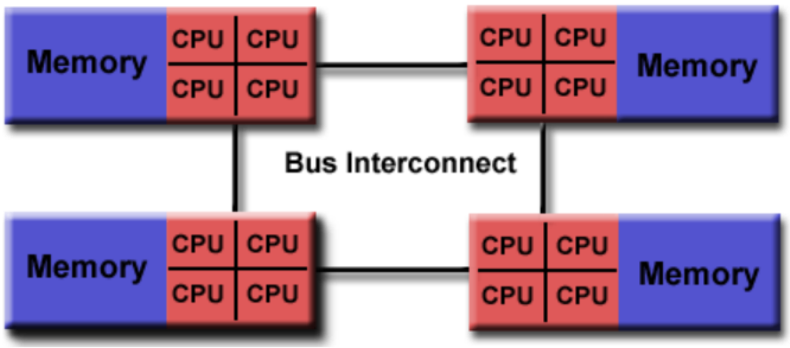
- OpenMP Concepts
- Parallel Construct
- Work-Sharing Constructs

What is OpenMP?

- An Application Program Interface (API) that may be used to explicitly direct *multi-threaded*, *shared memory* parallelism in Fortran and C/C++.
  (OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.)



Uniform Memory Access



Non-Uniform Memory Access

OpenMP History

- The OpenMP Architecture Review Board (ARB) published its first API specifications, OpenMP for Fortran 1.0, in October 1997. In October 1998, the OpenMP C/C++ specifications were released.

- OpenMP 2.0 for Fortran released in 2000
  OpenMP 2.0 for C/C++ released in 2002

OpenMP History - continued

- OpenMP 2.5 Combined C/C++/Fortran released in 2005
- OpenMP 3.0 released in May, 2008
- OpenMP 4.0 released in July, 2014
- OpenMP 4.5 released in November 15, 2015
- OpenMP 5.0 released on November 8, 2018

OpenMP API Summary

- Compiler Directives

  For example:

```
#pragma omp parallel
std::cout << "Hello, world.\n";
```

- Library Routines

  For example

```
omp_get_thread_num()     // get thread id
omp_set_num_threads (2)  // using 2 threads
```

- Environment Variables

  For example

```
setenv OMP_NUM_THREADS 8
```

Program OpenMP using VS 2019

1. Create a new VS 2019 C++ console project

2. Open (Configuration) Property of the project (debug or release).

3. Click C/C++ → Language
   and modify the OpenMP Support as needed.

4. Include OpenMP header

```
#include <omp.h>
```

Visual C++ (2015-2019) supports only the OpenMP 2.0 standard.

OpenMP Hello-World Example
VS 2019 Project - OpenMPExamples

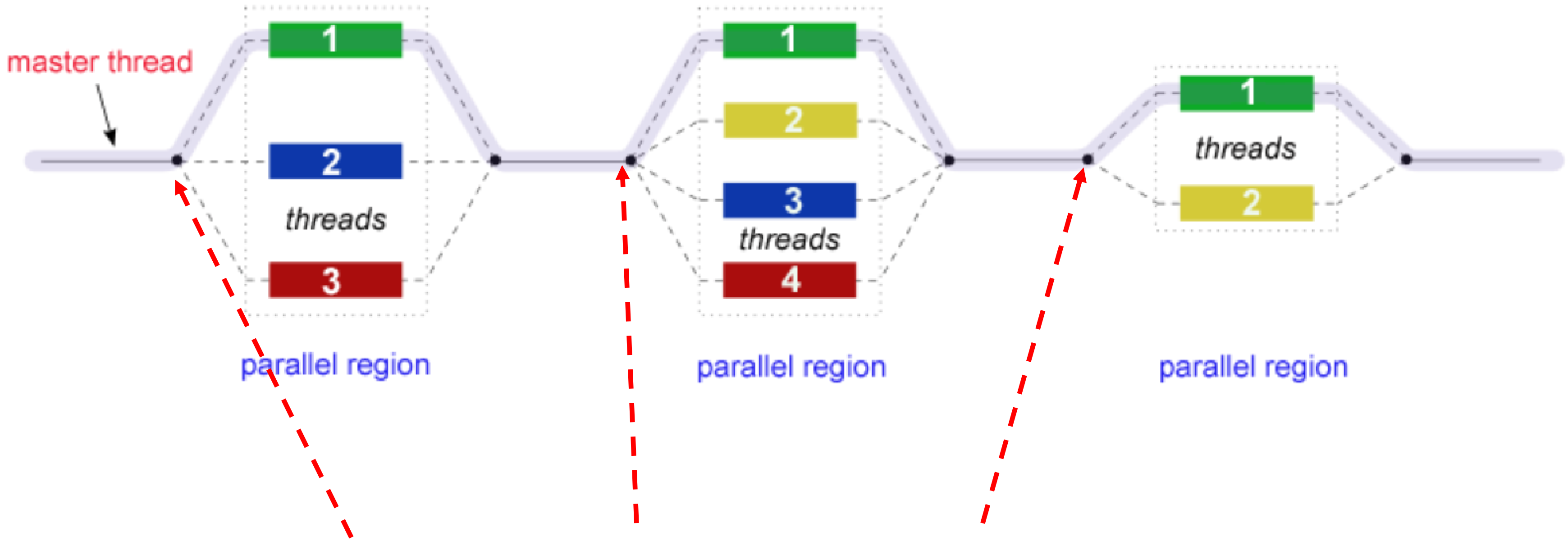
```
#include <omp.h>
#include <iostream>

void omp_hello_world()
{
	std::cout << "The CPU has " << omp_get_num_procs()
		  << " cores. \n\n";

	#pragma omp parallel
	{
		int id = omp_get_thread_num();
		std::cout << "Hello, world greeting from thread "
			  << id << std::endl;
	}
}
```

OpenMP Execution Model

Multi-Threading, fork-join model:

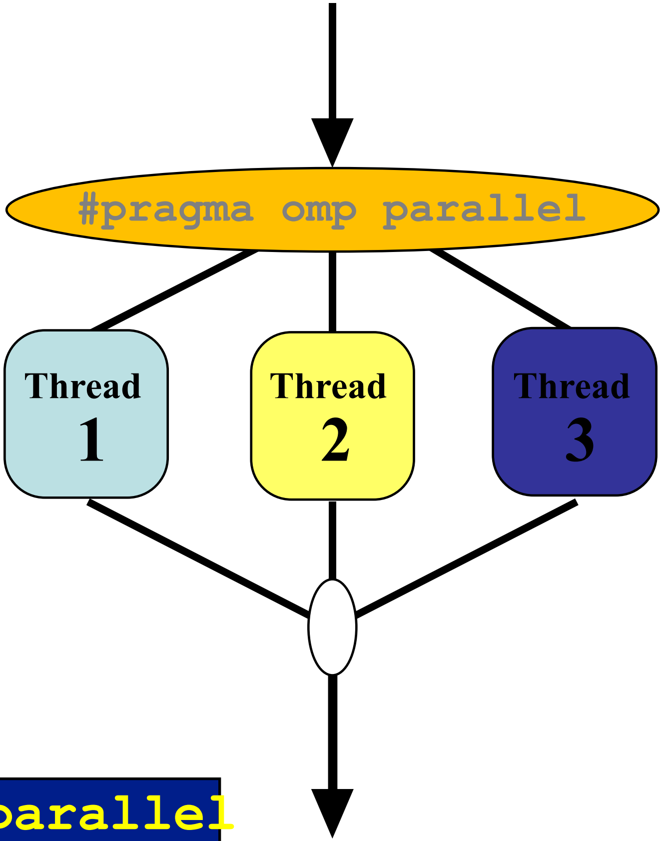


Using *#pragma omp parallel* directive

OpenMP Execution Model - continued

- Defines parallel region over structured block of code. Threads are created as ‘parallel’ pragma.

- Data is shared among threads unless specified otherwise (note variables define in the block are private).



C/C++ :

```
#pragma omp parallel
    {
            block
    }
```

OpenMP Execution Model - continued

- The master thread forks off a number of concurrent threads which execute structured blocks code in parallel.

- The number of concurrent threads:
  - default (number of cores, `omp_get_num_procs()`)
  - via a function call like this `omp_set_num_threads (4);`
  - via compiler directive like this

    ```
    #pragma omp parallel num_threads(4)
    ```
  - specified in an environment variable

- After the execution of the parallel structured block, the threads join into the master thread.

- Both data-parallel and task-parallel can be achieved.

Thread ID

- Each thread running in parallel region has a unique id assigned to it. The unique thread id can be obtained by

```
int id = omp_get_thread_num();
```

- The unique ids are assigned as 0,1,2,3,4 ….

- Consider this example

```
#pragma omp parallel
{
   int id = omp_get_thread_num();
   std::cout << id << std::endl;
}
```

- The master thread has an id of 0.

OpenMP Programming Ideas

- Create teams of threads for parallel execution.
- Specify how to share work among members of a thread team (load balancing).
- Declare *shared* and *private* variables as needed.
- Synchronize threads to enable them to perform certain operations exclusively without interferences from other threads (e.g. handling race conditions).

Contents

- OpenMP Concepts
- Parallel Construct
- Work-Sharing Constructs

Parallel Construct

Syntax

```
#pragma omp parallel [clause ...]  newline
                   if (scalar_expression)
                   private (list)
                   shared (list)
                   default (shared | none)
                   firstprivate (list)
                   reduction (operator: list)
                   copyin (list)
                   num_threads (integer-expression)


     structured_block
```

Parallel Construct - continued

- One entry and one exit point

- There is an implied *barrier* at the end of a parallel region. Only the master thread continues execution past this point.

- How many threads? Use the default or the following clause:

```
num_threads (integer-expression)
```

For example:

```
#pragma omp parallel num_threads(4)
{
   int id = omp_get_thread_num();
   std::cout << id << std::endl;
} // Implicit barrier here
```

Parallel Construct Example
VS 2019 Project - OpenMPExamples

```
#include <omp.h>
#include <iostream>

 void omp_hello_world_2()
 {
        int nthreads, tid;

        #pragma omp parallel private(tid)
        {
                /* Obtain and print thread id */
                tid = omp_get_thread_num();
                printf("Hello World from thread = %d\n", tid);

                /* Only master thread does this */
                if (tid == 0)
                {
                    nthreads = omp_get_num_threads();
                    printf("Number of threads = %d\n", nthreads);
                }

        }  /* All threads join master thread and terminate */

 }
```

Contents

- OpenMP Concepts
- Parallel Construct
- Work-Sharing Constructs

Sharing Work among Concurrent Threads

- Redundantly execute all of the structured block code (not very useful indeed).

- Execute on selected data items based on the thread id (like programming in POSIX pthreads or C++11 thread).

- Distribute the work in a Work-Sharing Construct, for example,

```
#pragma omp parallel
#pragma omp for
for (i = 0; i < _N; i++)
    sum += a[i];
```

How does OpenMP distribute for-loop work to running threads (job scheduling strategies)?

Work-Sharing Constructs

- Loop construct

```
#pragma omp for [clause ...]
```

- Section work unit

```
#pragma omp sections [clause ...]
```

- Single

```
#pragma omp single [clause ...]
```

Work-Sharing Constructs Concepts

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.

```
void worker (int jobId)
{
     int tid = omp_get_thread_num();
     //#pragma omp critical
     printf("Thread %d does job %d\n", tid, jobId);
}

#pragma omp parallel num_threads(2)
{
     #pragma omp sections
     {
            #pragma omp section
            worker(10);

            #pragma omp section
            worker(20);
     }
}
```

Work-Sharing Constructs Concepts -continued

- A work-sharing construct does not launch new threads, has an implicit barrier at the end, but does not have an implicit barrier on entry.

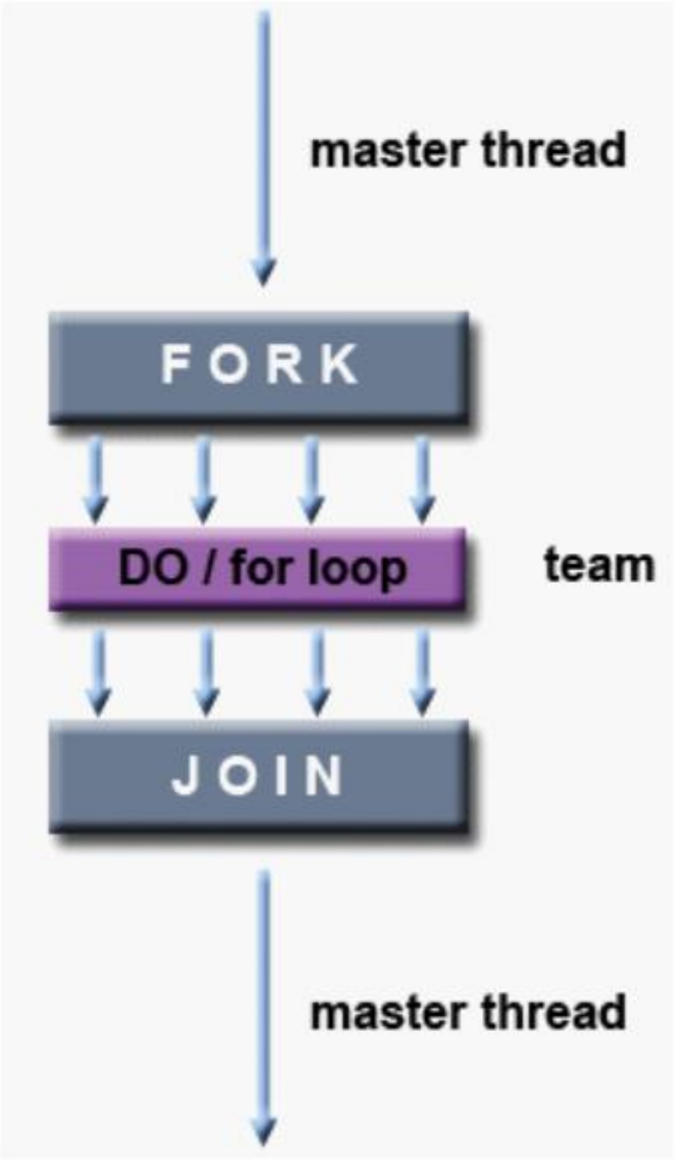
- Threads wait at a barrier until the last thread has reached the barrier.

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

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

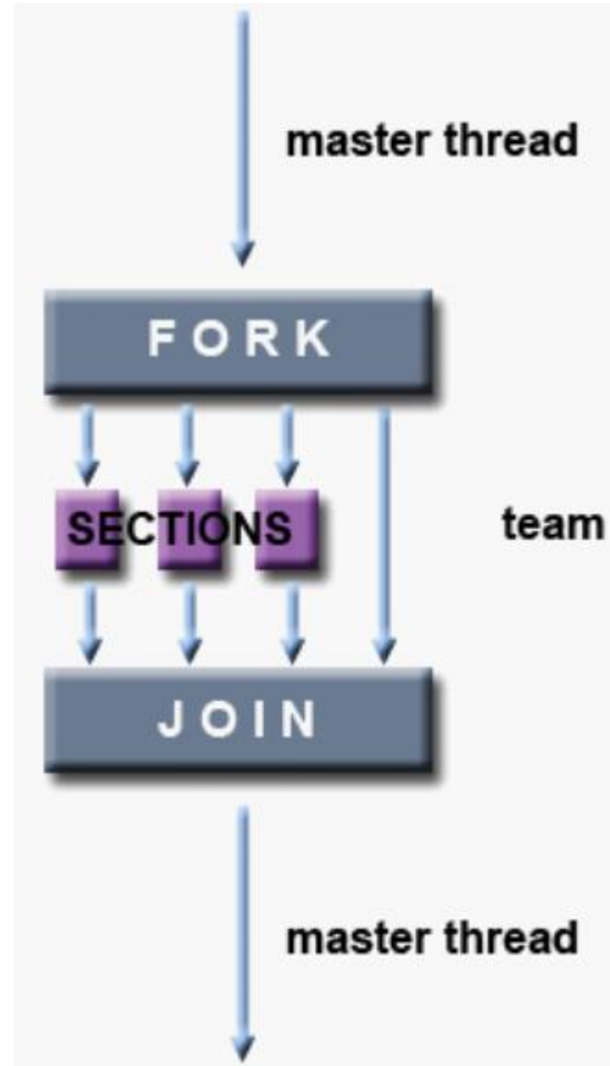
Work-Sharing Constructs Concepts - continued

- DO/for - shares iterations of a loop across the team. Represents a type of "data parallelism".



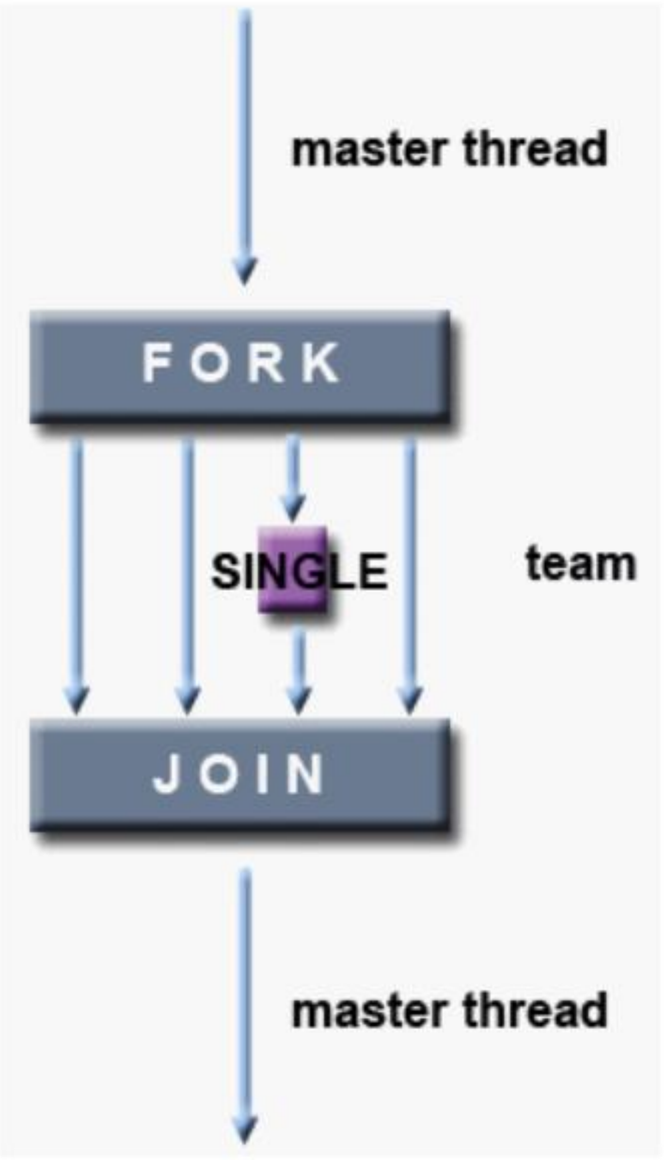
Work-Sharing Constructs Concepts - continued

- SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional or task parallelism".



Work-Sharing Constructs - continued

- SINGLE - serializes a section of code.



Work-Sharing Constructs Concepts - continued

Restrictions:

• A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.

• Work-sharing constructs must be encountered by all members of a team or none at all.

• Successive work-sharing constructs must be encountered in the same order by all members of a team.

For Work-Sharing Construct Syntax

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

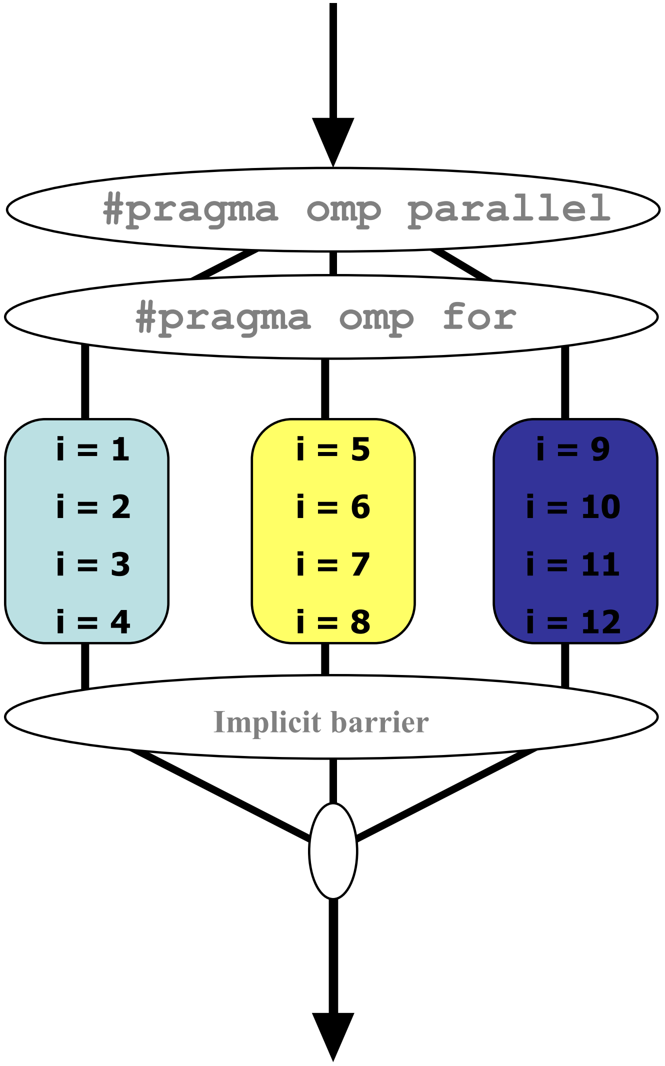
For Work Sharing Concepts

```
#pragma omp parallel
#pragma omp for
      for (I=0;I<N;I++){
            Do_Work(I);
      }
```

- Splits loop iterations into threads.
- Must be in the parallel region,
- Must precede the loop

For Work Sharing Concepts - continued

```
#pragma omp parallel num_threads(3)
#pragma omp for
   for(i = 1, i < 13, i++)
      c[i] = a[i] + b[i];
```



- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

For Work Sharing Concepts - continued

These two code segments are equivalent:

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

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

Matrix-Vector Multiplication
VS 2019 Project - OpenMPExamples

```
#include <omp.h>
void omp_matrix_vector(
    std::vector<float>& A,
    std::vector<float>& x,
    std::vector<float>& b,
    int m,
    int n)
{
       #pragma omp parallel for
       // notice in OpenMP 2.0 array index must be int type
       for (int row = 0; row < m; row++)
       {
              float accum = float(0);
                     for (int col = 0; col < n; col++)
                        accum += A[row * n + col] * x[col];

              b[row] = accum;
       }
}
```

For Work Sharing Example

```
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main(int argc, char *argv[])
{
        int i, chunk;
        float a[N], b[N], c[N];

        /* Some initializations */
        for (i=0; i < N; i++)
                a[i] = b[i] = i * 1.0;
        chunk = CHUNKSIZE;

        #pragma omp parallel shared(a,b,c,chunk) private(i)
        {

           #pragma omp for schedule(dynamic,chunk) nowait
           for (i=0; i < N; i++)
                c[i] = a[i] + b[i];
  }   /* end of parallel region */

}
```

For Work Sharing Example - nowait

```
#pragma omp parallel default(none) shared(n,a,b) private(i)
{
   #pragma omp for nowait
   for (i = 0; i < n; i++)
   {
      printf("Thread %d executes first loop iteration %d\n",
             omp_get_thread_num(),i);
      a[i] = i;
   }

   #pragma omp for
   for (i = 0; i < n; i++)
   {
      printf("Thread %d executes second loop iteration %d\n",
             omp_get_thread_num(),i);
      b[i] = 2 * a[i];
      //b[i] = a[n - i - 1];
   }
}
```

Is the `notwait` clause correctly used?

For Work Sharing schedule Clause

- schedule: describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

- schedule clause syntax

```
schedule (<kind> [, <chunk_size>])
```

- schedule <kind> (i.e. algorithms to distribute work to threads):
  `static` (default)
  `dynamic`
  `guide`
  `runtime` (specified by the OMP_SCHEDULE environment variable.)

For Work Sharing Schedule - static

- For iterations are divided into equal size (specified by the <chunk_size>, or default to #number_intertions/number_threads).

- The chunks are assigned to the threads statically in a round-robin manners in the order the thread id.

```
#define N 1000
int i;
float a[N], b[N], c[N];
for (i=0; i < N; i++)
a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i)
{
    #pragma omp for schedule(static, 10) nowait
    for (i = 0; i < N; i++)
          c[i] = a[i] + b[i];
}
```

For Work Sharing Schedule - dynamic

- Each thread executes a chunk of iterations, then requests another chunk until the job (chunk) queue is empty.

- The size of chunks is specified by chunk_size, which is default to 1.

```
#define N 1000
int i;
float a[N], b[N], c[N];
for (i=0; i < N; i++)
a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i)
{
    #pragma omp for schedule(dynamic) nowait
    for (i = 0; i < N; i++)
          c[i] = a[i] + b[i];
}
```

For Work Sharing Schedule - guide

- Similar to dynamic, but the chunk_size decreases in time, for example:

  `chunk_size ∝ (number_unsigned_iterations / number_threads)`

- For chunk_size, of *k*, *k* > 1, the size of chunks is not fewer than *k*.

- The chunk_size is default to 1.

For Work Sharing reduction Clause

- Syntax

```
reduction (operator: list)
```

Operation	Fortran	C/C++	Initialization
Addition	+	+	0
Multiplication	*	*	1
Subtraction	-	-	0
Logical AND	.and.	&&	0
Logical OR	.or.	||	.false. / 0
AND bitwise	iand	&	all bits on / 1
OR bitwise	ior	|	0
Exclusive OR bitwise	ieor	^	0
Equivalent	.eqv.		.true.
Not Equivalent	.neqv.		.false.
Maximum	max	max	Most negative #
Minimum	min	min	Largest positive #

For Work Sharing reduction Clause Example

```
#include <omp.h>
void main(int argc, char *argv[])
{
      int   i, n, chunk;
      float a[100], b[100], result;

      n = 100;
      chunk = 10;
      result = 0.0;
      for (i=0; i < n; i++) {
             a[i] = i * 1.0;
             b[i] = i * 2.0;
      }
      #pragma omp parallel for private(i) schedule(static,chunk) \
             reduction(+:result)
      for (i=0; i < n; i++)
             result = result + (a[i] * b[i]);
}
```

Sections Work-Sharing Construct Syntax

```
 #pragma omp sections [clause ...]  newline
              private (list)
              firstprivate (list)
              lastprivate (list)
              reduction (operator: list)
              nowait
{

  #pragma omp section   newline

  structured_block

  #pragma omp section   newline

  structured_block

}
```

Sections Work-Sharing Construct Example

```
#include <omp.h>
#define N 1000

void main(int argc, char *argv[])
{

        int i;
        float a[N], b[N], c[N], d[N];

        /* Some initializations */
        for (i = 0; i < N; i++)
        {
                a[i] = i * 1.5;
                b[i] = i + 22.35;
        }
```

Sections Work-Sharing Construct Example - continued

```
    #pragma omp parallel shared(a,b,c,d) private(i)
    {

        #pragma omp sections nowait
        {
                #pragma omp section
                for (i = 0; i < N; i++)
                        c[i] = a[i] + b[i];

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

        }  /* end of sections */

    }  /* end of parallel region */

} /* end of main */
```

Single Work-Sharing Construct Syntax

```
#pragma omp single [clause ...]  newline
                  private (list)
                  firstprivate (list)
                  nowait

    structured_block
```

Single Work-Sharing Example

```
#include <omp.h>
#include <iostream>

#define _N (10)
void singleDemo()
{
      int a, b[_N], i;

      #pragma omp parallel shared(a, b) private(i)
      {
            #pragma omp single
            {
                  a = 10;
                  std::cout << "Single construct “
                        " executed by thread "
                        << omp_get_thread_num()
                  << std::endl;
            }
```

Single Work-Sharing Example - continued

```
            // end of #pragma omp single

            // implicit barrier here

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

      } // end of #pragma omp parallel

      for (i = 0; i < _N; i++)
            printf ("b[%d] = %d\n", i, b[i]);

} // end of singleDemo function
```

One More Work-Sharing Example

```
#include <omp.h>
#define N       1000
#define CHUNKSIZE   100

void  main(int argc, char *argv[])
{
       int i, chunk;
       float a[N], b[N], c[N];

       /* Some initializations */
       for (i=0; i < N; i++)
       a[i] = b[i] = i * 1.0;
       chunk = CHUNKSIZE;

       #pragma omp parallel for \
              shared(a,b,c,chunk) private(i) \
              schedule(static,chunk)
       for (i=0; i < N; i++)
       c[i] = a[i] + b[i];
 }
```