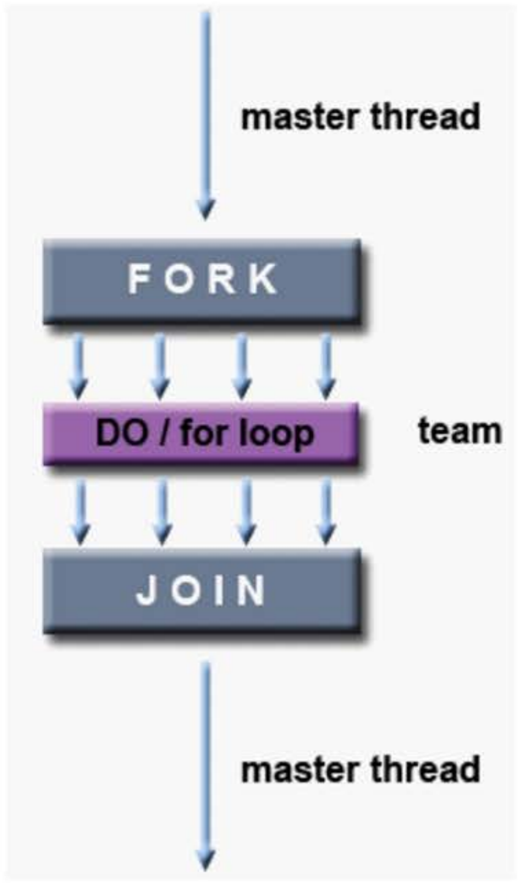
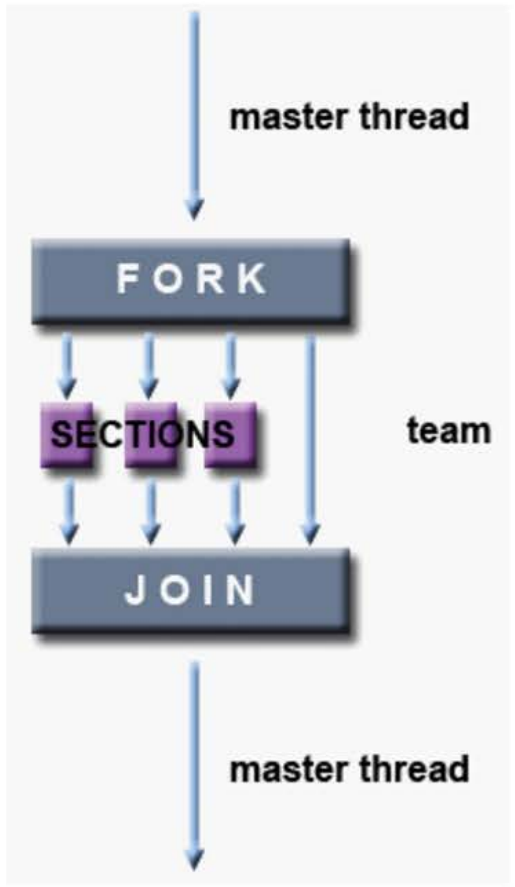
ECE 432/532
Programming for Parallel Processors

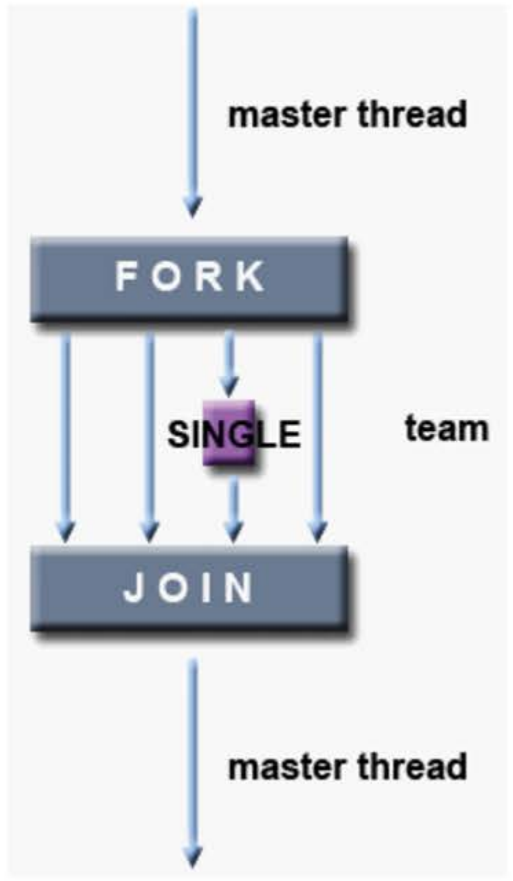
More on OpenMP directives

- **Work-Sharing Constructs**
  - A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
  - Work-sharing constructs do not launch new threads
  - There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

More on OpenMP directives

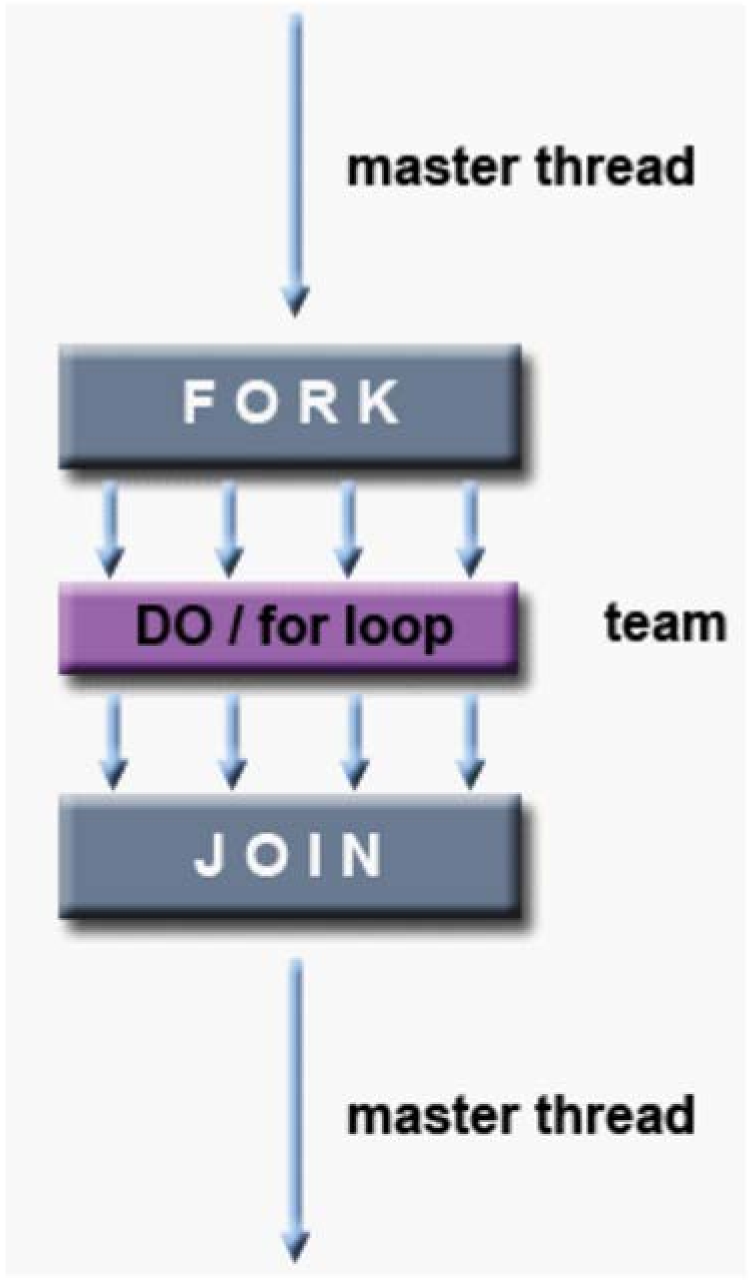






Work-Sharing Constructs

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

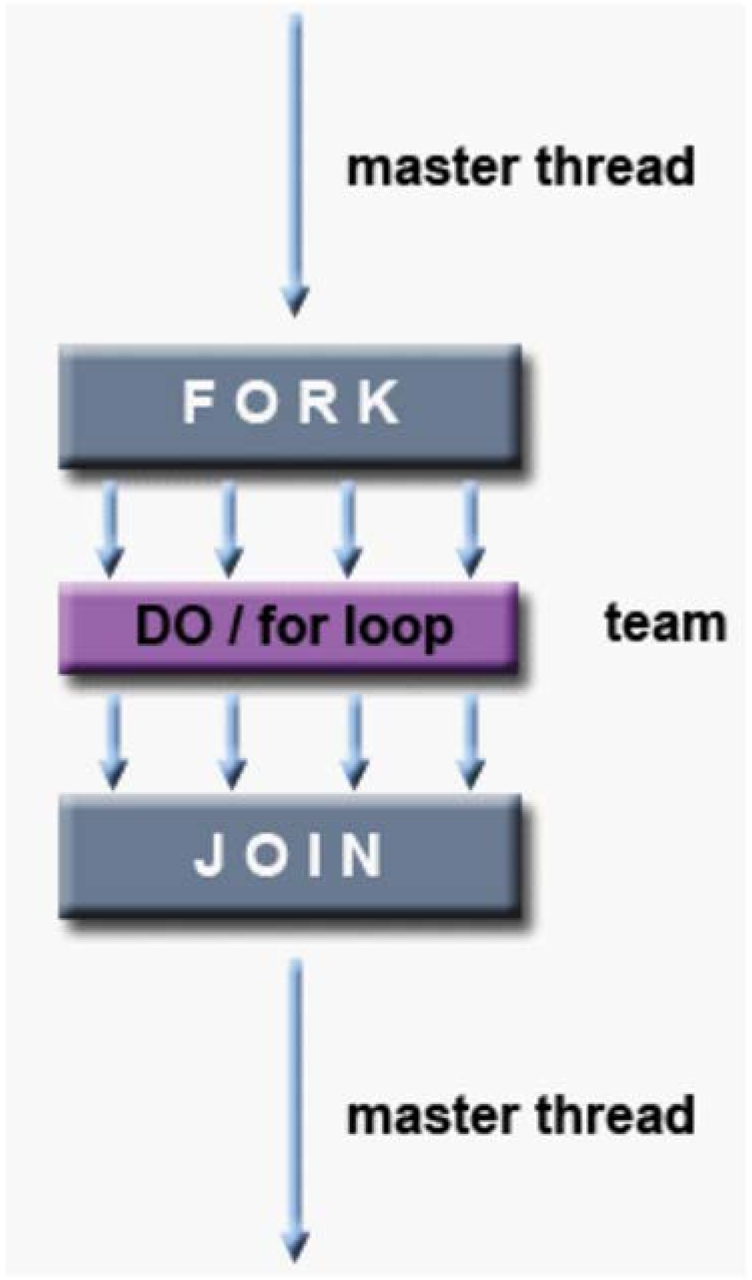


```
#include <omp.h>
 #define N 1000
 #define CHUNKSIZE 100

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

 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 */
 }
```

Work-Sharing Constructs

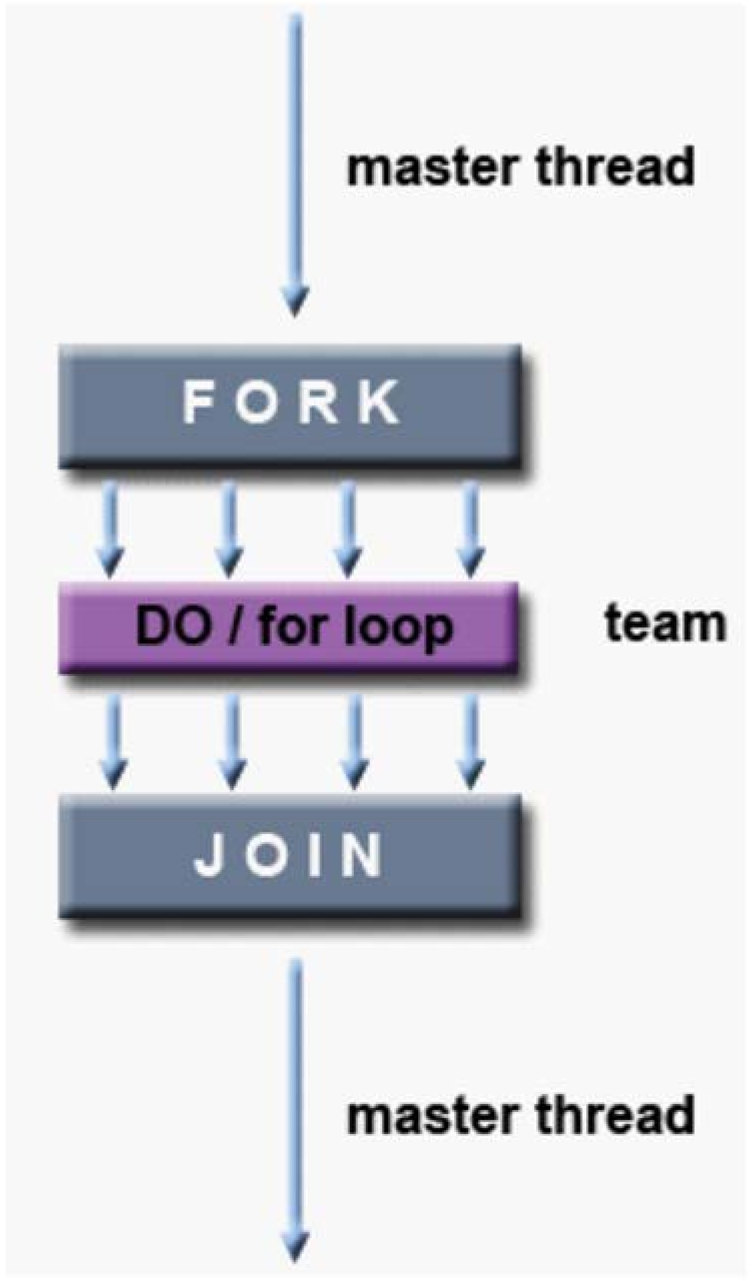
- The *for* directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team

- This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.



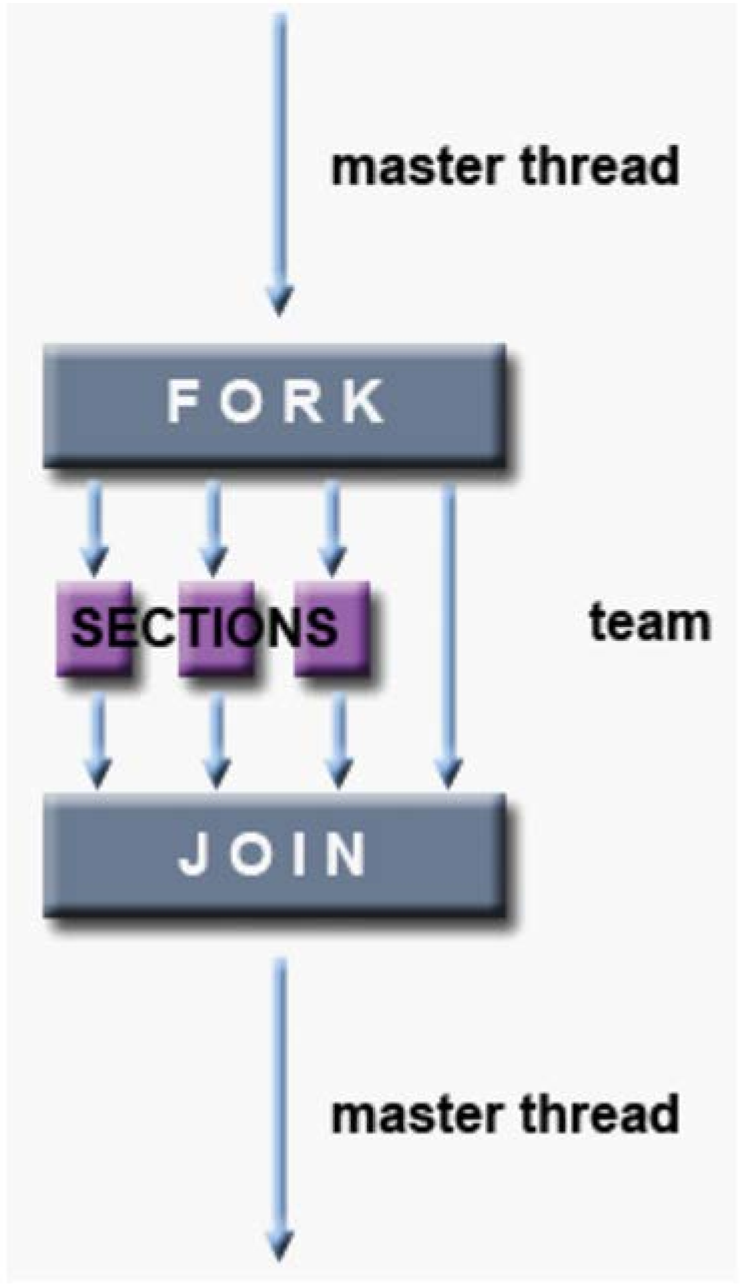
Work-Sharing Constructs

- **Restrictions:**
  - The loop has to be a for loop
  - The loop iteration variable must be an integer and the loop control parameters must be the same for all threads
  - Program correctness must not depend upon which thread executes a particular iteration
  - It is illegal to branch out of a loop associated with a *for* directive
  - The chunk size must be specified as a loop invarient integer expression, as there is no synchronization during its evaluation by different threads



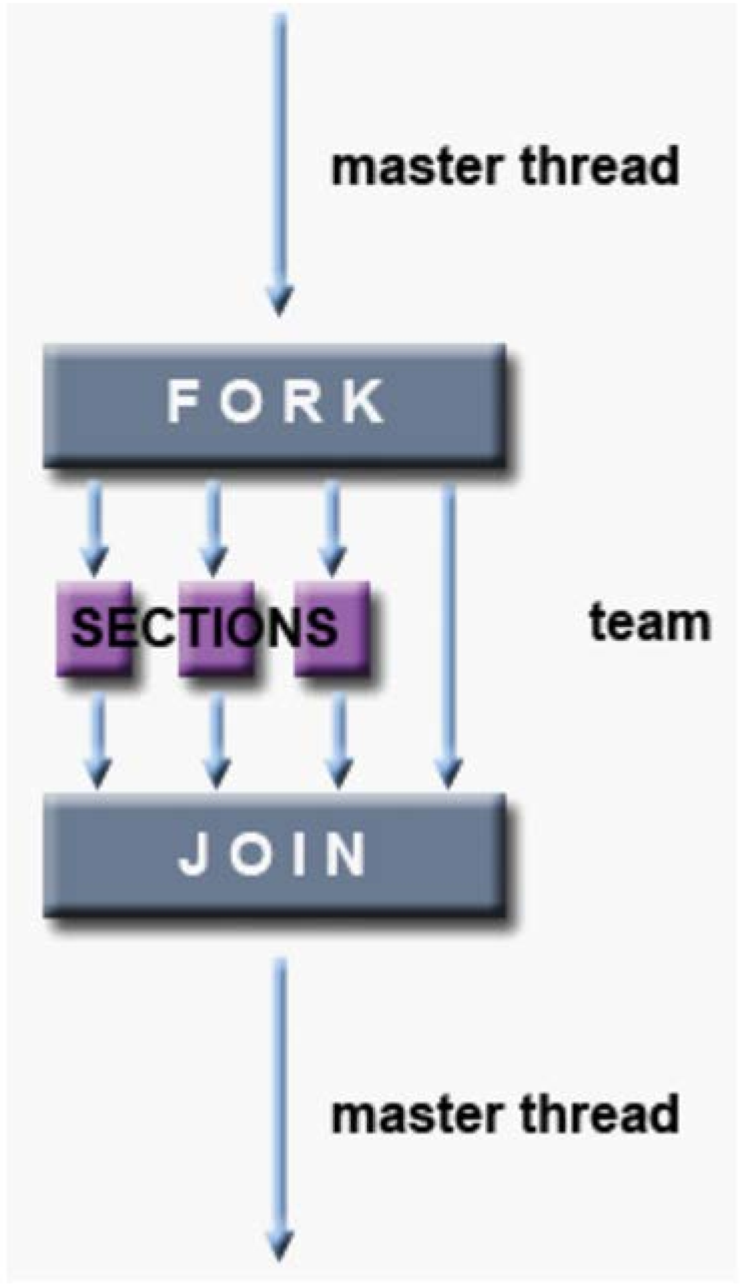
Work-Sharing Constructs

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



Work-Sharing Constructs

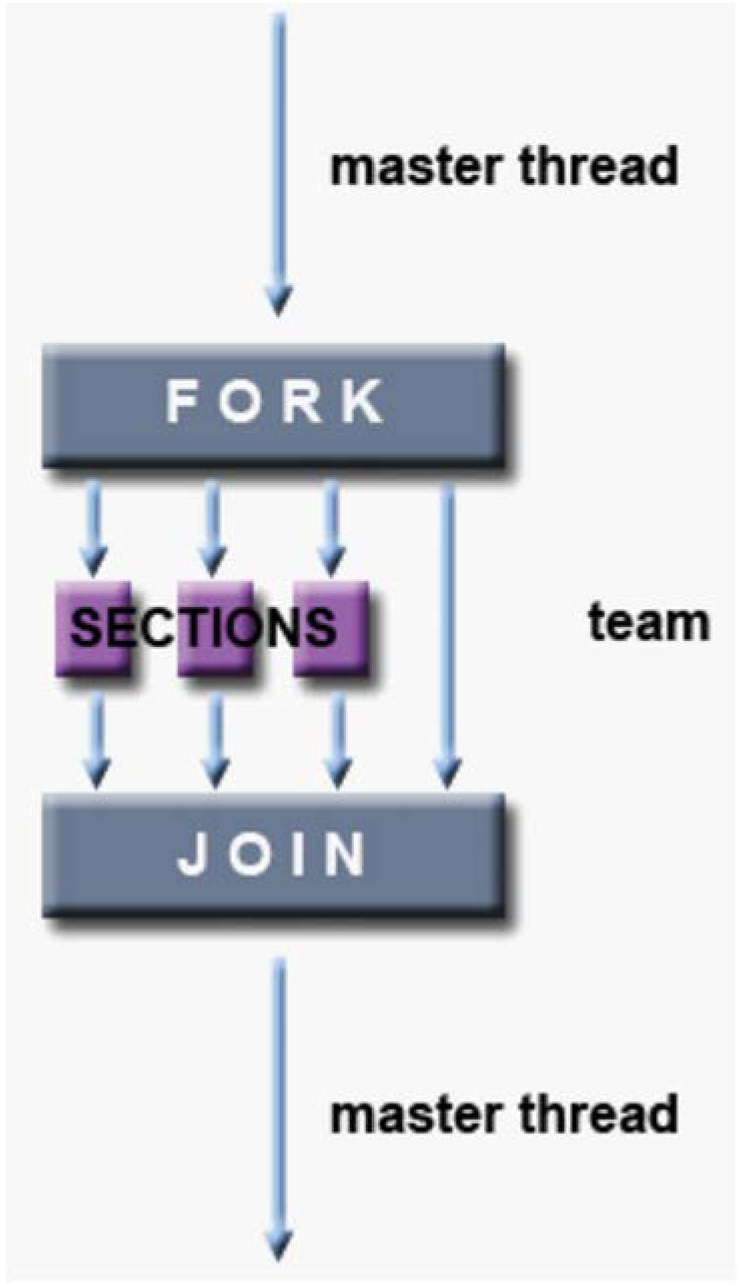
- The **SECTIONS** directive is a non-iterative work-sharing construct.
  - It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
- Independent **SECTION** directives are nested within a **SECTIONS** directive
  - Each SECTION is executed once by a thread in the team.
  - Different sections may be executed by different threads.
  - It is possible for a thread to execute more than one section



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

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

 #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 */
 }
```

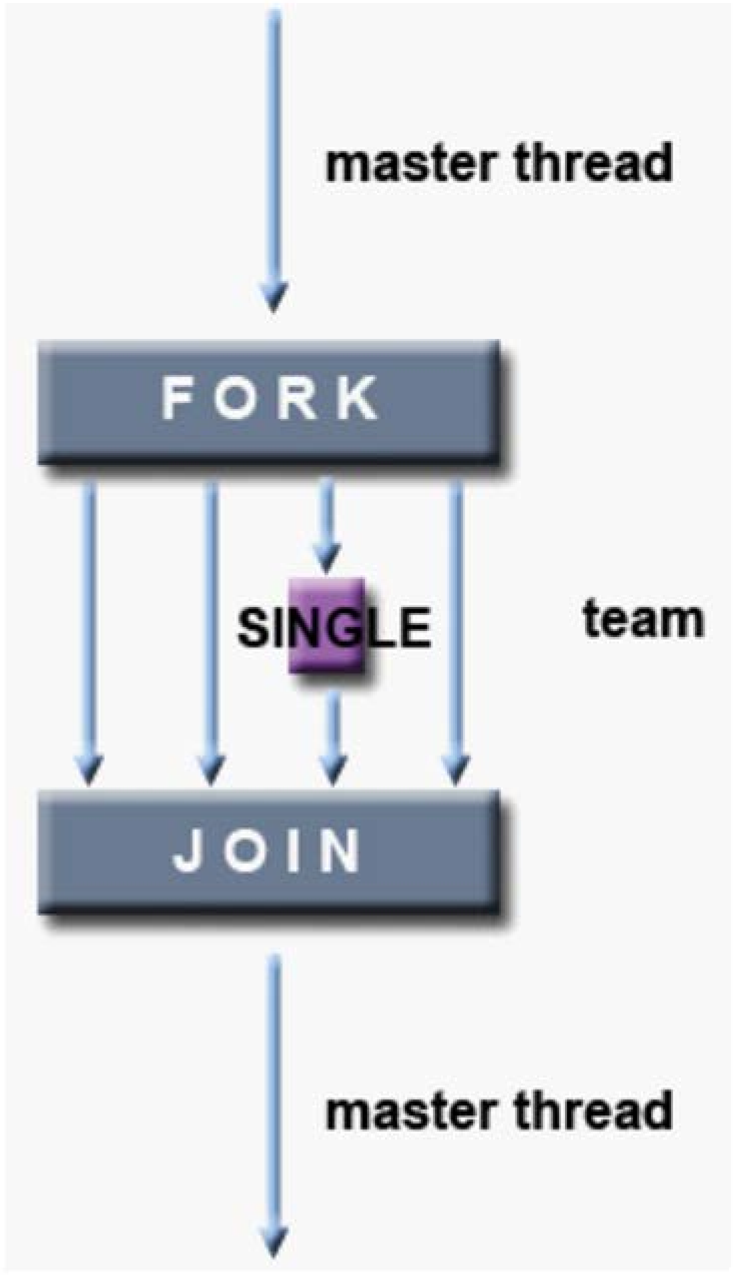
Work-Sharing Constructs

- There is an implied barrier at the end of a **SECTIONS** directive, unless the **nowait** clause is used
- Q1: What happens if the number of threads and the number of SECTIONs are different? More threads than SECTIONs? Less threads than SECTIONs?
- Q2: Which thread executes which SECTION?



Work-Sharing Constructs

- **SINGLE** - serializes a section of code
- The **SINGLE** directive specifies that the enclosed code is to be executed by only one thread in the team
- May be useful when dealing with sections of code that are not thread safe (such as I/O)
- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a NOWAIT/nowait clause is specified



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

int main() {
   #pragma omp parallel num_threads(2)
   {
      #pragma omp single
      // Only a single thread can read the input.
      printf("read input\n");

      // Multiple threads in the team compute the results.
      printf("compute results\n");

      #pragma omp single
      // Only a single thread can write the output.
      printf("write output\n");
    }
}
```

Example - Is it correct?

```
#pragma omp parallel shared(n,a,b,c,d,sum) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n; i++)
        a[i] += b[i];

    #pragma omp for nowait
    for (i=0; i<n; i++)
        c[i] += d[i];
    #pragma omp barrier

    #pragma omp for nowait reduction(+:sum)
    for (i=0; i<n; i++)
        sum += a[i] + c[i];
} /*-- End of parallel region --*/
```

Other directives

- ORDERED
- FIRSTRPIVATE
- LASTPRIVATE
- MASTER

ORDERED

- Specifies that code under a parallelized for loop should be executed like a sequential loop
    - The ordered directive must be within the dynamic extent of a for or parallel for construct with an ordered clause
    - The ordered directive supports no OpenMP clauses.

```
#pragma omp ordered
  structured-block
```

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

static float a[1000], b[1000], c[1000];

void test(int first, int last){
    #pragma omp for schedule(static) ordered
    for (int i = first; i <= last; ++i) {
        // Do something here.
        if (i % 2) {
            #pragma omp ordered
            printf("test() iteration %d\n", i);
        }
    }
}
void test2(int iter) {
    #pragma omp ordered
    printf("test2() iteration %d\n", iter);
}
```

```
int main( ) {
    int i;
    #pragma omp parallel
    {
        test(1, 8);
        #pragma omp for ordered
        for (i = 0 ; i < 5 ; i++)
            test2(i);
    }
}
```

```
int main( ) {
    int i;
    #pragma omp parallel
    {
        test(1, 8);
        #pragma omp for ordered
        for (i = 0 ; i < 5 ; i++)
            test2(i);
    }
}
```

```
test() iteration 1
test() iteration 3
test() iteration 5
test() iteration 7
test2() iteration 0
test2() iteration 1
test2() iteration 2
test2() iteration 3
test2() iteration 4
```

FIRSTPRIVATE

- Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.

```
firstprivate(var)
```

LASTPRIVATE

- Specifies that the enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section (#pragma sections).

lastprivate(var)

```
j = jstart;
#pragma omp parallel for firstprivate(j)
{
   for(i=1; i<=n; i++){
      if(i == 1 || i == n)
         j = j + 1;
      a[i] = a[i] + j;
   }
}
```

```
#pragma omp parallel for lastprivate(x)
{
   for(i=1; i<=n; i++){
      x = sin( pi * dx * (float)i );
      a[i] = exp(x);
   }
}
lastx = x;
```

MASTER

- Specifies that only the master threadshould execute a section of the program

```
#pragma omp master
  structured-block
```

```c
int main( )
{
    int a[5], i;
    #pragma omp parallel
    {
        #pragma omp for
         for (i = 0; i < 5; i++)
             a[i] = i * i;

         #pragma omp master
             for (i = 0; i < 5; i++)
                 printf("a[%d] = %d\n", i, a[i]);

         #pragma omp barrier

         #pragma omp for
         for (i = 0; i < 5; i++)
             a[i] += i;
    }
}
```

Nested parallelsim

```
#pragma omp parallel for
for(j=0; j<jmax; j++){
   #pragma omp parallel for
   for(i=0; i<imax; i++){
      do_work(i,j);
}
```

- OpenMP standard allows but does not require nested parallelism
- Logical function omp_get_nested() returns “true” or “false” (1 or 0) to indicate whether or not nested parallelism is enabled in the current region

```
#pragma omp parallel for
for(int y=0; y<25; ++y) {
  #pragma omp parallel for
  for(int x=0; x<80; ++x)  {
    tick(x,y);
  }
}
```

- How many ticks will be triggered?

```
#pragma omp parallel for
for(int y=0; y<25; ++y) {
  #pragma omp parallel for
  for(int x=0; x<80; ++x)  {
    tick(x,y);
  }
}
```

- How many ticks will be triggered?
- The inner loop is not actually parallelized. Only the outer loop is.
- The inner loop runs in a pure sequence, as if the whole inner #pragma was omitted.
- The OpenMP detects that there already exists a team, and instead of a new team of N threads, it will create a team consisting of only the calling thread

```
#pragma omp parallel for
for(int y=0; y<25; ++y) {
  #pragma omp parallel for
  for(int x=0; x<80; ++x)  {
    tick(x,y);
  }
}
```

```
#pragma omp parallel for
for(int y=0; y<25; ++y) {
  #pragma omp for
  for(int x=0; x<80; ++x) {
    tick(x,y);
  }
}
```

- How many ticks will be triggered?
- The inner loop is not actually parallelized. Only the outer loop is.
- The inner loop runs in a pure sequence, as if the whole inner #pragma was omitted.
- The OpenMP detects that there already exists a team, and instead of a new team of N threads, it will create a team consisting of only the calling thread

```
#pragma omp parallel for
for(int y=0; y<25; ++y) {
  #pragma omp parallel for
  for(int x=0; x<80; ++x)  {
    tick(x,y);
  }
}
```

```
#pragma omp parallel for
for(int y=0; y<25; ++y) {
  #pragma omp for
  for(int x=0; x<80; ++x) {
    tick(x,y);
  }
}
```

- How many ticks will be triggered?
- The inner loop is not actually parallelized. Only the outer loop is.
- The inner loop runs in a pure sequence, as if the whole inner #pragma was omitted.
- The OpenMP detects that there already exists a team, and instead of a new team of N threads, it will create a team consisting of only the calling thread

```
#pragma omp parallel for collapse(2)
 for(int y=0; y<25; ++y)
   for(int x=0; x<80; ++x)
   {
     tick(x,y);
   }
```

- The number specified in the *collapse* clause is the number of nested loops that are subject to the work-sharing semantics of the OpenMP for construct.

What will this program print if mpirun –n 2 ./hybrid

```
#include <stdio.h>
#include "mpi.h"
#include <omp.h>

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int iam = 0, np = 1;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  #pragma omp parallel default(shared) private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d from process %d out of %d\n",iam, np, rank,
numprocs);
  }
  MPI_Finalize(); }
```

```c
#include <stdio.h>
#include "mpi.h"
#include <omp.h>

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int iam = 0, np = 1;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  #pragma omp parallel default(shared) private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d from process %d out of %d\n",iam, np, rank,
numprocs);
  }
  MPI_Finalize(); }
```

```
Hello from thread 0 out of 4 from process 0 out of 2
Hello from thread 1 out of 4 from process 0 out of 2
Hello from thread 2 out of 4 from process 0 out of 2
Hello from thread 3 out of 4 from process 0 out of 2
Hello from thread 0 out of 4 from process 1 out of 2
Hello from thread 3 out of 4 from process 1 out of 2
Hello from thread 1 out of 4 from process 1 out of 2
Hello from thread 2 out of 4 from process 1 out of 2
```

```
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4

int main (int argc, char *argv[]) {
 int p,my_rank,c;
 omp_set_num_threads(_NUM_THREADS); /* set number of threads to spawn */
 MPI_Init(&argc, &argv); /* initialize MPI stuff */
 MPI_Comm_size(MPI_COMM_WORLD,&p);
 MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 #pragma omp parallel reduction(+:c) {
    #pragma omp master {
       if ( 0 == my_rank)
          c = 1;
       else
          c = 2;    } }
 printf("%d\n",c);
 MPI_Finalize();
 return 0;
}
```

What will this program print if mpirun –n 5 ./hybrid

```
#include <omp.h>
#include "mpi.h"
#include <stdio.h>
#define _NUM_THREADS 4

int main (int argc, char *argv[]) {
 int p,my_rank,c;
 omp_set_num_threads(_NUM_THREADS); /* set number of threads to spawn */
 MPI_Init(&argc, &argv); /* initialize MPI stuff */
 MPI_Comm_size(MPI_COMM_WORLD,&p);
 MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 #pragma omp parallel reduction(+:c) {
    #pragma omp master {
       if ( 0 == my_rank)
          c = 1;
       else
          c = 2;    } }
 printf("%d\n",c);
 MPI_Finalize();
 return 0;
}
```

```
2
2
2
2
1
```

Concluding Remarks (1)

- OpenMP is a standard for programming shared-memory systems.
- OpenMP uses both special functions and preprocessor directives called pragmas.
- OpenMP programs start multiple threads rather than multiple processes.
- Many OpenMP directives can be modified by clauses.

Concluding Remarks (2)

- A major problem in the development of shared memory programs is the possibility of race conditions.
- OpenMP provides several mechanisms for insuring mutual exclusion in critical sections.
  - Critical directives
  - Named critical directives
  - Atomic directives
  - Simple locks

Concluding Remarks (3)

- By default most systems use a block-partitioning of the iterations in a parallelized for loop.
- OpenMP offers a variety of scheduling options.
- In OpenMP the scope of a variable is the collection of threads to which the variable is accessible.

Concluding Remarks (4)

- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.