Shared Memory Programming

OpenMP: private and shared variables

Adding matrices C=A+B

```
for (i = 0; i < n; i++)
{
     for (j = 0; j < n; j++)
     {
          C[i][j] = A[i][j] + B[i][j];
     }
}
```

Which loop to parallelize?
Which loops have dependencies?
The i and j loops have no dependencies at all.

Adding matrices C=A+B: OpenMP version 1

```
for (i = 0; i < n; i++)
{
      #pragma omp parallel for
       for (j = 0; j < n; j++)
       {
            C[i][j] = A[i][j] + B[i][j];
       }
}
```

Adding matrices C=A+B: OpenMP version 1

```
for (i = 0; i < n; i++)
{
      #pragma omp parallel for
       for (j = 0; j < n; j++)
       {
            C[i][j] = A[i][j] + B[i][j];
       }
}
```

Pay the fork/join overhead n times, once for each i.

Adding matrices C=A+B: OpenMP version 2: INCORRECT

```
#pragma omp parallel for
for (i = 0; i < n; i++)
{
      for (j = 0; j < n; j++)
      {
            C[i][j] = A[i][j] + B[i][j];
      }
}
```

Adding matrices C=A+B: OpenMP version 2: INCORRECT

```
#pragma omp parallel for
for (i = 0; i < n; i++)
{
     for (j = 0; j < n; j++)
     {
          C[i][j] = A[i][j] + B[i][j];
     }
}
```

By default, only i will be a private variable. Everything else, including j, will be a shared variable. Each thread will be initializing and incrementing the same j. Unlikely to get correct results.

Adding matrices C=A+B: OpenMP version 2: CORRECT

```
#pragma omp parallel for private(j)
for (i = 0; i < n; i++)
{
     for (j = 0; j < n; j++)
     {
          C[i][j] = A[i][j] + B[i][j];
     }
}
```

- A ***clause*** is an optional, additional component to a pragma.
- The **`private (<variable list>)`** clause directs the compiler to make listed variables private
- By default private variables are undefined at loop entry and loop exit.

Adding matrices C=A+B: OpenMP version 2: ALSO CORRECT

```
#pragma omp parallel for private(j)
for (i = 0; i < n; i++)
{
        int j;
        for (j = 0; j < n; j++)
        {
                C[i][j] = A[i][j] + B[i][j];
        }
}
```

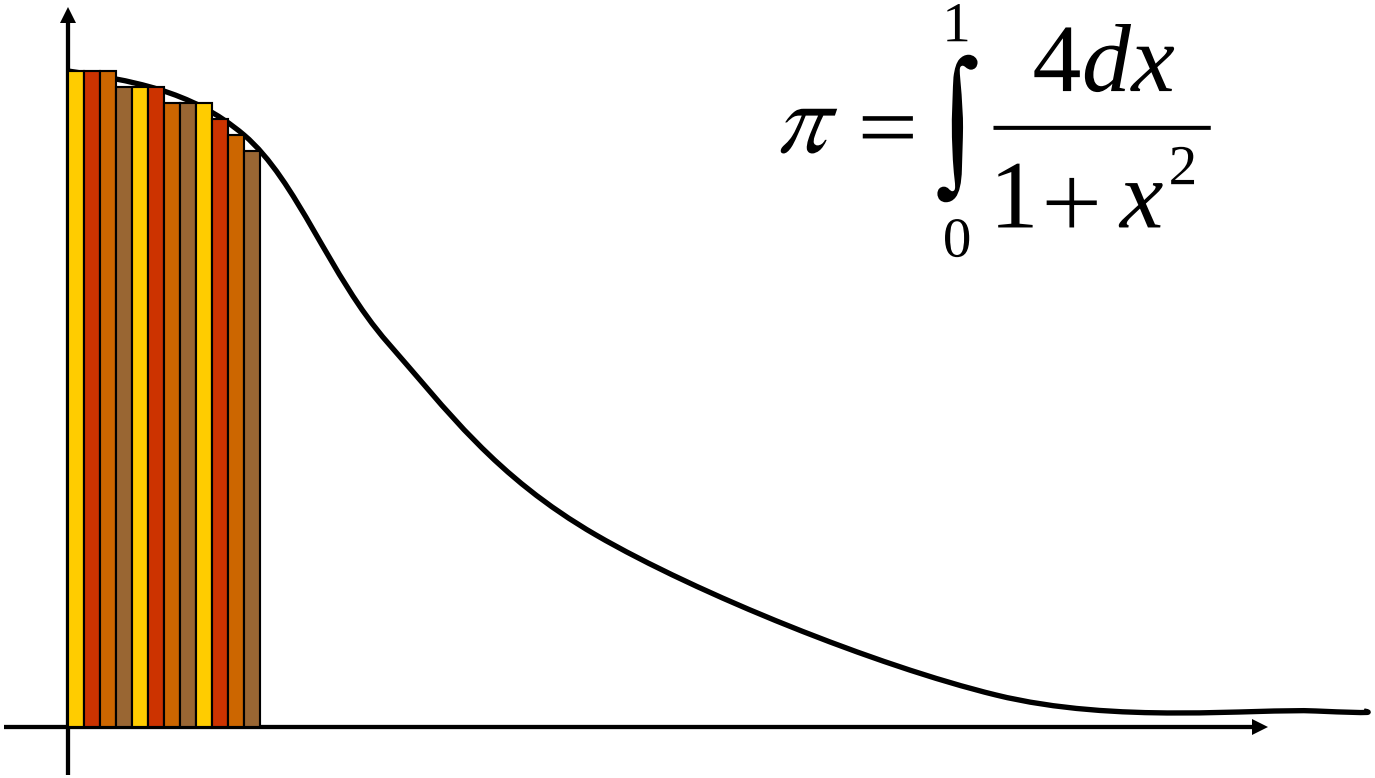
All variables declared inside the for loops are private.

OpenMP: private variables

- By default, private variables are undefined at loop entry and loop exit.
- The clause **`firstprivate (x)`** directs the compiler to make x a private variable whose initial value for each thread is the value of x in the master thread before the loop.
- The clause **`lastprivate (x)`** directs the compiler to make x a private variable whose value in the master thread after the loop will be whatever the value of x is in the thread that did the iteration that would come last sequentially.

Pi in OpenMP

Approximate the integral (area under curve) by area of rectangles



$$\pi = \int_0^1 \frac{4dx}{1+x^2}$$

“Code” fragment

/* compute width of the rectangles */

$$xDelta = \ldots$$

/* Loop over rectangles i=1 to n */

{

/* compute midpoint of rectangle */

$$xMid = \ldots$$

/* add area of rectangle: width times function height */

$$area = xDelta * f(xMid)$$

$$pi = pi + area$$

}

Assignment 2 done in OpenMP

- We'll mark the loop over rectangles to be done in parallel.
- What variables, if any, need to be private?

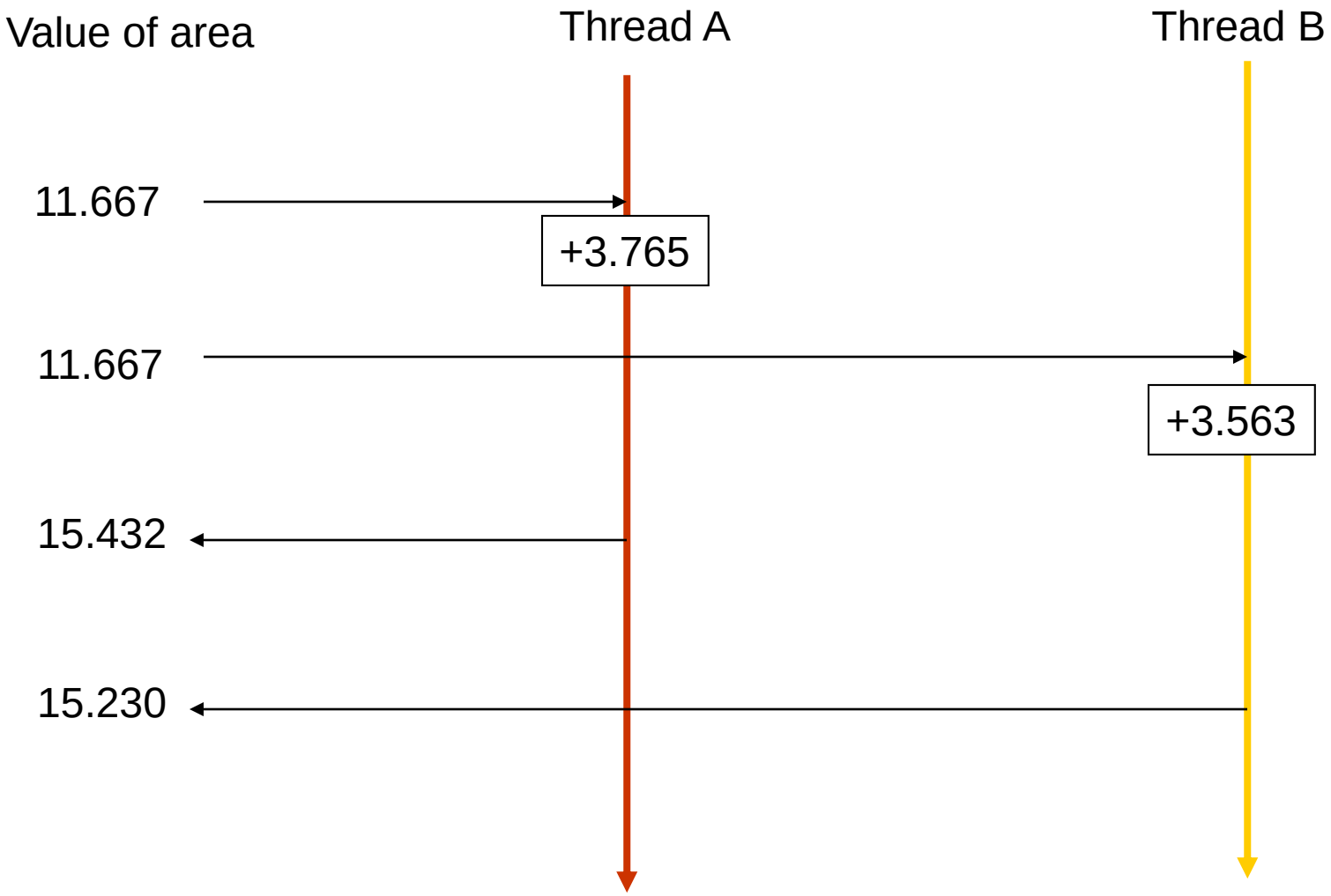
Pi code: serial

```
h   = 1.0 / (double) n;
area = 0.0;

for (i = 1; i <= n; i++)
{
   x = h * ((double)i - 0.5);
   area += (4.0/(1.0 + x*x));
}
pi = h*area;
```

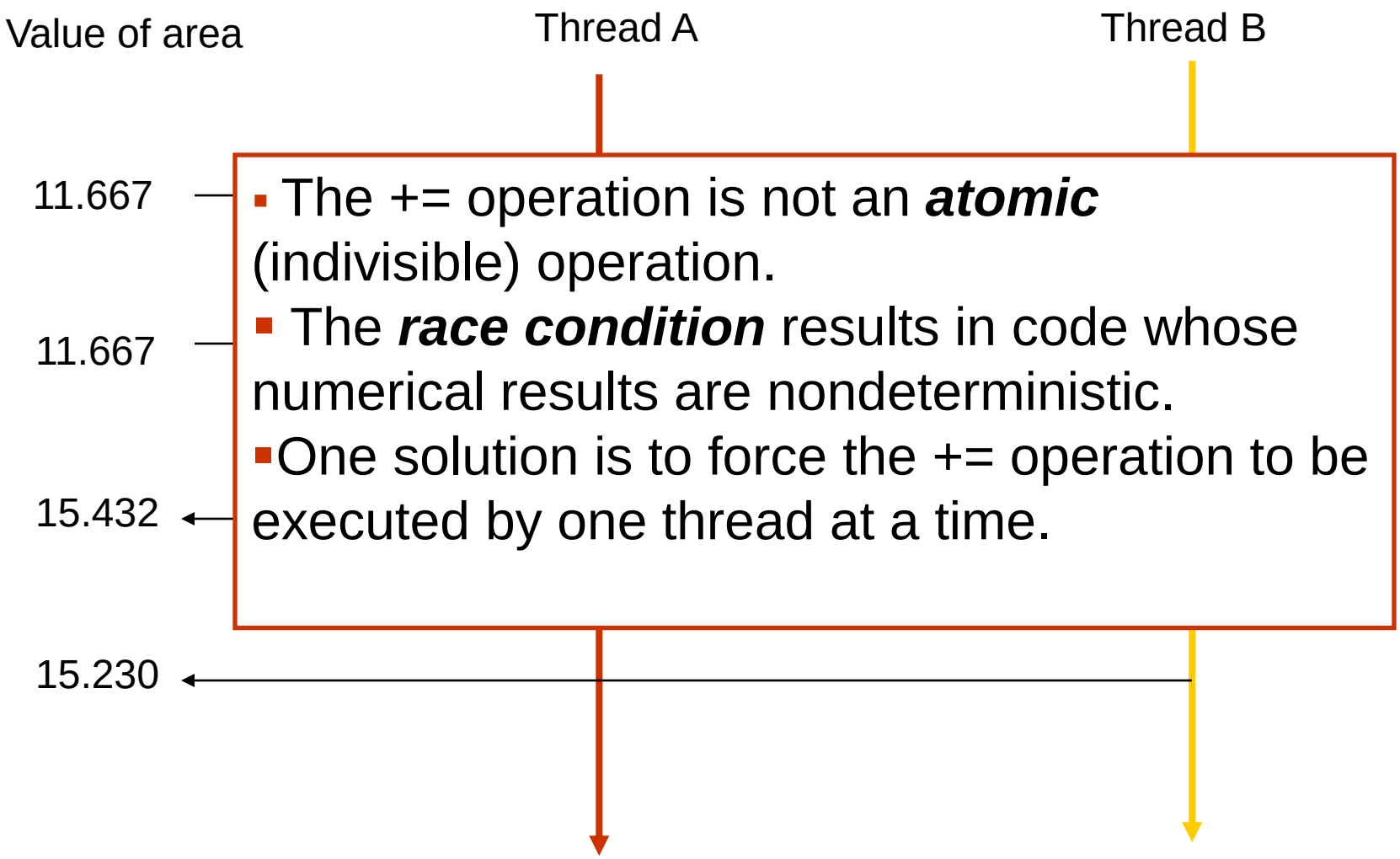
Pi code: OpenMP INCORRECT

```
h   = 1.0 / (double) n;
area = 0.0;

#pragma omp parallel for private(x)
for (i = 1; i <= n; i++)
{
   x = h * ((double)i - 0.5);
   area += (4.0/(1.0 + x*x));
}
pi = h*area;
```

Race condition



Race condition



- The += operation is not an ***atomic*** (indivisible) operation.
- The ***race condition*** results in code whose numerical results are nondeterministic.
- One solution is to force the += operation to be executed by one thread at a time.

Pi code: OpenMP correct, but inefficient: critical clause

```
h   = 1.0 / (double) n;
area = 0.0;

#pragma omp parallel for private(x)
for (i = 1; i <= n; i++)
{
   x = h * ((double)i - 0.5);
#pragma omp critical
   area += (4.0/(1.0 + x*x));
}
pi = h * area;
```

Critical section is executed by one thread at a time.
Limits attainable speedup.

Pi code: OpenMP correct and maybe better: atomic clause

```
h   = 1.0 / (double) n;
area = 0.0;

#pragma omp parallel for private(x)
for (i = 1; i <= n; i++)
{
   x = h * ((double)i - 0.5);
#pragma omp atomic
   area += (4.0/(1.0 + x*x));
}
pi = h * area;
```

atomic section forces += to be executed together.
May limit attainable speedup.

Pi code: Open MP better solution: reduction clause

```
h   = 1.0 / (double) n;
area = 0.0;

#pragma omp parallel for private(x) reduction(+:area)
for (i = 1; i <= n; i++)
{
   x = h * ((double)i - 0.5);
   area += (4.0/(1.0 + x*x));
}
pi = h * area;
```

Note reduction clause on the parallel for pragma

- Compiler handles setting up private variables for partial sums
- syntax **`reduction (<op>:<variable>)`**

Loop reordering and scheduling

The fork/join cost may be reduced by reordering loops.

```
for ( i=1; i<m; i++)
   for ( j=0; j<n; j++)
      a[i][j] = 2 * a[i-1][j];
```

Loop over rows outside

```
for ( i=1; i<m; i++)
   for ( j=0; j<n; j++)
      a[i][j] = 2 * a[i-1][j];
```

2	7	9
4	*	*
*	*	*

→

2	7	9
4	14	*
*	*	*

→

2	7	9
4	14	18
*	*	*

→

2	7	9
4	14	18
8	*	*

→

2	7	9
4	14	18
8	28	*

→

2	7	9
4	14	18
8	28	36

Loop over rows outside

```
for ( i=1; i<m; i++)
   for ( j=0; j<n; j++)
      a[i][j] = 2 * a[i-1][j];
```

$$\begin{matrix} 2 & 7 & 9 \\ 4 & * & * \\ * & * & * \end{matrix} \longrightarrow \begin{matrix} 2 & 7 & 9 \\ 4 & 14 & * \\ * & * & * \end{matrix} \longrightarrow \begin{matrix} 2 & 7 & 9 \\ 4 & 14 & 18 \\ * & * & * \end{matrix}$$

$$\begin{matrix} 2 & 7 & 9 \\ 4 & 14 & 18 \\ 8 & * & * \end{matrix} \longrightarrow \begin{matrix} 2 & 7 & 9 \\ 4 & 14 & 18 \\ 8 & 28 & * \end{matrix} \longrightarrow \begin{matrix} 2 & 7 & 9 \\ 4 & 14 & 18 \\ 8 & 28 & 36 \end{matrix}$$

Can't parallelize outer i loop.
Row 3 depends on Row 2

The fork/join cost may be reduced by reordering loops.

```
for ( i=1; i<m; i++)
   for ( j=0; j<n; j++)
      a[i][j] = 2 * a[i-1][j];
```

```
for ( i=1; i<m; i++)
#pragma omp parallel for
   for ( j=0; j<n; j++)
      a[i][j] = 2 * a[i-1][j];
```

Fork/Join m-1 times

Loop over columns outside

```
for ( j=0; j<n; j++)
   for ( i=1; i<m; i++)
      a[i][j] = 2 * a[i-1][j];
```

2	7	9
4	*	*
*	*	*

→

2	7	9
4	*	*
8	*	*

→

2	7	9
4	14	*
8	*	*

2	7	9
4	14	*
8	28	*

→

2	7	9
4	14	18
8	28	*

→

2	7	9
4	14	18
8	28	36

The fork/join cost may be reduced by reordering loops.

```
for ( i=1; i<m; i++)
   for ( j=0; j<n; j++)
      a[i][j] = 2 * a[i-1][j];
```

```
for ( i=1; i<m; i++)
#pragma omp parallel for
   for ( j=0; j<n; j++)
      a[i][j] = 2 * a[i-1][j];
```

Fork/Join m-1 times

```
#pragma omp parallel for private(i)
for ( j=0; j<n; j++)
   for ( i=1; i<m; i++)
      a[i][j] = 2 * a[i-1][j];
```

Fork/Join once

The schedule clause

- Can control how the loop iterations are distributed among threads
- Specify a kind of distribution
  - Static, dynamic, guided, auto and runtime
  - Static is often the default, has lowest overhead, best if run time is independent of loop index.
  - Dynamic (and guided) help for variable and unpredictable run time.
- Optionally specify a chunk size, the base number of loop iterations dealt out to threads

Static scheduling is default

```
#pragma omp parallel for
for (i = 0; i < n; i++)
{
        x[i]=foo(i);
}
```

OpenMP compiler will generate code like below:

```
int this_thread = omp_get_thread_num()
int num_threads = omp_get_num_threads();
int my_start = (this_thread  ) * n / num_threads;
int my_end   = (this_thread+1) * n / num_threads;
for (i = my_start; i < my_end; i++)

{
        x[i]=foo(i);
}
```

Static scheduling is default

```
#pragma omp parallel for
for (i = 0; i < n; i++)
{
        x[i]=foo(i);
}
```

OpenMP compiler will generate code like below:

```
int this_thread = omp_get_thread_num()
int num_threads = o
int my_start = (thi
int my_end   = (this_thread+1) * n / num_threads;
for (i = my_start; i < my_end; i++)

{
        x[i]=foo(i);
}
```

If n is not divisible by num_threads,
Last thread may get fewer loop iterations

Dynamic scheduling is like worker/manager

- OpenMP runtime library manages, gives threads one iteration. When they finish, they get another.

```
#pragma omp parallel for schedule(dynamic)
for (i = 0; i < n; i++){
      x[i]=foo(i);
}
```

- chunk_size=3, gives threads three iterations at a time. When they finish, they get three more.

```
#pragma omp parallel for schedule(dynamic,3)
for (i = 0; i < n; i++){
      x[i]=foo(i);
}
```

Nested loops: the collapse clause

- If no dependencies in nested loops, can mark with collapse
- Will form a single big nxm loop on distribute it

```
#pragma omp parallel for collapse(2)
for (i = 0; i < n; i++)
{
     for (j = 0; j < m; j++)
     {
          C[i][j] = A[i][j] + B[i][j];
     }
}
```

Force specific ordering on execution: ordered clause/construct

- In general, the execution ordering within a parallel for loop is unspecified.
- Can force an ordering for certain statements. May have performance hit.

```
#pragma omp parallel for ordered
for (i = 0; i < n; i++)
{
      A[i]=foo(i);
      #pragma omp ordered
      printf(A= %f\n,a[i]);
}
```

Ordered clause alerts compiler of an ordered construct inside loop

All statements outside ordered construct executed in any order

Prints will come i=0,1,…

Parallel regions in OMP

Parallel pragma alone

- The parallel pragma starts a parallel region.
- This starts (forks) a team of threads all of which execute the region. Each thread assigned an id: 0,1, …, num_threads-1
- Implicit barrier at end of parallel region, threads wait until all finish
- After the region, threads join back to one.

```
#pragma omp parallel
{
        printf("Hello!\n");
}
```

Output:
One print from each thread

Parallel pragma alone

- Cant branch into or out of parallel region
- Does not distribute work
- Unless work sharing construct is inside the parallel region, each thread will exectute all of the region (won’t speed things up at all).

Parallel pragma plus work sharing construct

- Example of work sharing construct is “for”

```
#pragma omp parallel
{
  # pragma omp for
  for (i = 0; i < n; i++)
  {
       C[i] = A[i] + B[i];
  }
}
```

Start up team of threads

Distribute loop iterations to current team of threads

Parallel pragma plus work sharing construct

- Example of work sharing construct is "for"

```
#pragma omp parallel
{
  # pragma omp for
  for (i = 0; i < n; i++)
  {
       C[i] = A[i] + B[i];
  }
  MORE CODE HERE
}
```

Start up team of threads

Distribute loop iterations to current team of threads

Implicit barrier at end of for, all threads wait.

Parallel pragma plus work sharing construct

- Example of work sharing construct is "for"

```
#pragma omp parallel
{
  # pragma omp for
  for (i = 0; i < n; i++)
  {
       C[i] = A[i] + B[i];
  }
}
```

Start up team of threads

Distribute loop iterations to current team of threads

- If pragma omp for is outside a parallel region, loop runs serially on master thread

parallel for is a combined parallel region and work sharing construct

```
#pragma omp parallel for
for (i = 0; i < n; i++)
{
        C[i] = A[i] + B[i];
}
```

=

```
#pragma omp parallel
{
  # pragma omp for
  for (i = 0; i < n; i++)
  {
        C[i] = A[i] + B[i];
  }
}
```

Jacobi Open MP: with combined parallel for

```
# include <omp.h>

for (step=0; step<num_steps; step++) {
#pragma omp parallel for
 for (i=1; i<=N; i++) }
  /* compute new temp from formula */
       u_new[i]= 0.5*(h2+u_old[i+1]+u_old[i-1]);
}
#pragma omp parallel for
 for (i=1; i<=N;  i++) {
  u_old[i]=u_new[i];
 }
}
```

Jacobi Open MP: with separate parallel & for

```
# include <omp.h>

for (step=0; step<num_steps; step++) {
  #pragma omp parallel
  {
   #pragma omp for
   for (i=1; i<=N; i++) {
    /* compute new temp from formula */
       u_new[i]= 0.5*(h2+u_old[i+1]+u_old[i-1]);
  }
   #pragma omp for
   for (i=1; i<=N;  i++) {
    u_old[i]=u_new[i];
   }
  } // end parallel region
} // end time step
```

Default: barrier after a work sharing construct

- Example of work sharing construct is "for"

```
#pragma omp parallel
{
  # pragma omp for
  for (i = 0; i < n; i++)
  {
       C[i] = A[i] + B[i];
  }
  MORE CODE HERE
}
```

Start up team of threads

Distribute loop iterations to current team of threads

No threads move past for loop until all have finished for loop.

nowait clause suppresses barrier

- Example of work sharing construct is "for"

```
#pragma omp parallel
{
  # pragma omp for nowait
  for (i = 0; i < n; i++)
  {
       C[i] = A[i] + B[i];
  }
  MORE CODE HERE
}
```

Start up team of threads

Distribute loop iterations to current team of threads

threads move past for loop without waiting on all to finish

Jacobi Open MP: what's wrong?

```
# include <omp.h>

for (step=0; step<num_steps; step++) {
  #pragma omp parallel
  {
   #pragma omp for nowait
   for (i=1; i<=N; i++) {
    /* compute new temp from formula */
       u_new[i]= 0.5*(h2+u_old[i+1]+u_old[i-1]);
  }
   #pragma omp for
   for (i=1; i<=N;  i++) {
    u_old[i]=u_new[i];
   }
  } // end parallel region
} // end time step
```

One thread only – the single construct

```
#pragma omp parallel shared (b,A,n) private (i)
{
   /* allow one thread to set base */
   #pragma omp single
   {
       b=SetBase();
   }
   # pragma omp for
   for (i = 0; i < n; i++)
   {
       A[i] = pow(b,i);
   }

}
```

Only one thread executes single block. Any of team can, typically first to reach this section will.

Implicit barrier at end of single, all threads wait.

thread 0 only – the master construct

```
#pragma omp parallel shared (b,A,n) private (i)
{
    /* master thread to set base */
    #pragma omp master
    {
        b=SetBase();
    }
    # pragma omp for
    for (i = 0; i < n; i++)
    {
        A[i] = pow(b,i);
    }

}
```

Only thread 0 executes master block.

WARNING:
No barrier at end of master, all other threads move on.

thread 0 only – the master construct

```
#pragma omp parallel shared (b,A,n) private (i)
{
    /* master thread to set base */
    #pragma omp master
    {
        b=SetBase();
    }
    #pragma omp barrier
    # pragma omp for
    for (i = 0; i < n; i++)
    {
        A[i] = pow(b,i);
    }

}
```

Only thread 0 executes master block. Any of team can, typically first to reach this section will.

WARNING:
No barrier at end of master, all other threads move on threads wait.

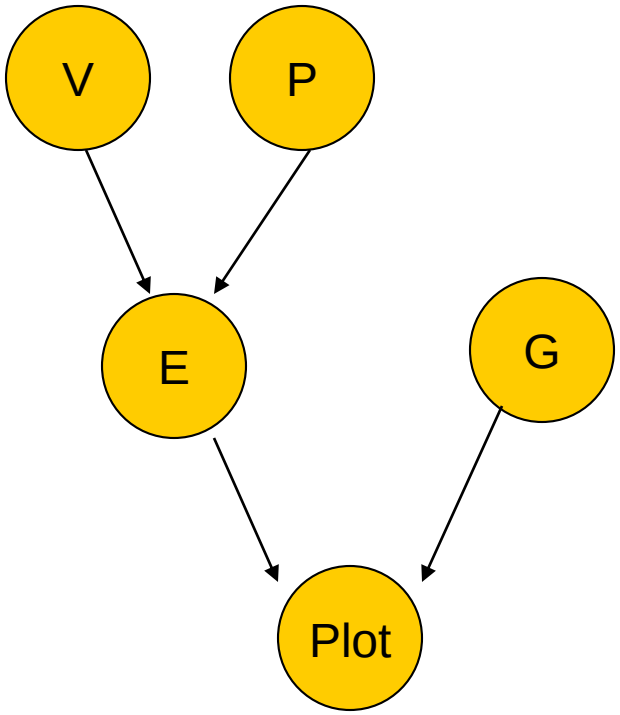
To get correct results here, need to add explicit barrier.

OpenMP and functional parallelism, beyond loops

```
v = velocity_solve( );
p = pressure_solve( );
e = energy(v,p);
g = grids( );
Plot(e,g);
```

OpenMP and functional parallelism

```
v = velocity_solve( );
p = pressure_solve( );
e = energy(v,p);
g = grids( );
Plot(e,g);
```



Work sharing by sections

```
#pragma omp parallel sections
   {
#pragma omp section
      v = velocity_solve( );
#pragma omp section
      p = pressure_solve( );
#pragma omp section
      g = grids( );
   }
   e = energy(v,p);
   Plot(e,g);
```

=

```
#pragma omp parallel
{
   #pragma omp sections
   {

#pragma omp section
      v = velocity_solve();
#pragma omp section
      p = pressure_solve();
#pragma omp section
      g = grids( );
   }
}
e = energy(v,p);
Plot(e,g);
```

Work sharing by sections

```
#pragma omp parallel
{
   #pragma omp sections
   {

#pragma omp section
      v = velocity_solve();
#pragma omp section
      p = pressure_solve();
#pragma omp section
      g = grids( );
   }
}
e = energy(v,p);
Plot(e,g);
```

If number of sections > number of threads, some threads will do multiple sections

If number of sections < number of threads, some threads will be idle

Implicit barrier at end of sections, all threads wait.