Introduction to OpenMP

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Copyright 2019
What is OpenMP?

It is a directive based standard to allow programmers to develop threaded parallel codes on shared memory computers.
Directives

CPU

Program myscience
... serial code ...
!$omp parallel do
do k = 1,n1
do i = 1,n2
... parallel code ...
enddo
enddo
!$omp end parallel do
...
End Program myscience

OpenMP Compiler Hint

Simple compiler hints from coder.

Compiler generates parallel threaded code.

Ignorant compiler just sees some comments.

Your original Fortran or C code
Directives: an awesome idea whose time has arrived.

```c
main() {
    double pi = 0.0; long i;

#pragma omp parallel for reduction(+:pi)
    for (i=0; i<N; i++) {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f\n", pi/N);
}
```

**OpenMP**

```c
main() {
    double pi = 0.0; long i;

    #pragma acc kernels
    for (i=0; i<N; i++) {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f\n", pi/N);
}
```

**OpenACC**
Key Advantages Of This Approach

- High-level. No involvement of pthreads or hardware specifics.
- Single source. No forking off a separate code. Compile the same program for multi-core or serial, non-parallel programmers can play along.
- Efficient. Very favorable comparison to pthreads.
- Performance portable. Easily scales to different configurations.
- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be quick.
Broad Compiler Support (For 3.x)

- Gnu
- Intel
- IBM
- PGI
- Cray
- MS Visual Studio*

*MS is missing some useful pieces.
A True Standard With A History

OpenMP.org: specs and forums and useful links

- POSIX threads
- 1997 OpenMP 1.0
- 1998 OpenMP 2.0
- 2005 OpenMP 2.5 (Combined C/C++/Fortran)
- 2008 OpenMP 3.0
- 2011 OpenMP 3.1
- 2013 OpenMP 4.0 (Accelerators)
- 2015 OpenMP 4.5
- 2018 OpenMP 5.0 (Draft)
Hello World

Hello World in C

```c
int main(int argc, char** argv){
    #pragma omp parallel
    {
        printf("Hello world.\n");
    }
}
```

Hello World in Fortran

```fortran
program hello
    !$OMP PARALLEL
    print *,"Hello World."
    !$OMP END PARALLEL
    stop
end
```

Output with OMP_NUM_THREADS=4

```
Hello World.
Hello World.
Hello World.
Hello World.
```
This is how these directives integrate into code:

**Fortran**

```fortran
!$omp parallel [clause ...]
  structured block
$omp end parallel
```

**C**

```c
#pragma omp parallel [clause ...]
{
  structured block
}
```

clause: optional modifiers
Which we shall discuss

I will indent the directives at the natural code indentation level for readability. It is a common practice to always start them in the first column (ala `#define/#ifdef`). Either is fine with C or Fortran 90 compilers.
```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 4

void *PrintHello(void *threadid)
{
    printf("Hello World.\n");
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```
With pthreads, we changed the structure of the original code. Non-threading programmers can’t understand new code.

We have separate sections for the original flow, and the threaded code. Serial path now gone forever.

This only gets worse as we do more with the code.

Exact same situation as assembly used to be. How much hand-assembled code is still being written in HPC now that compilers have gotten so efficient?
Thread vs. Process

**Two Processes**

- `A[0] = 10;`
- `B[4][Y] = 20;`
- `Y = Y + 1;`
- `for (i=1; i<100; i++){
   A[i] = A[i]-1;
}
- `Y = 0;`
- `B[0][0] = 30;`
- `A[0] = 30;`

**Two Threads**

- `A[0] = 10;`
- `B[4][Y] = 20;`
- `Y = Y + 1;`
- `for (i=1; i<100; i++){
   A[i] = A[i]-1;
}
- `Y = 0;`
- `B[0][0] = 30;`
- `A[0] = 30;`
General Thread Capability

Master

Thread

Spawned Threads

Thread Killed
Typical Desktop Application Threading

Open Browser Tabs (Spawn Thread)

Close Browser Tab (Kill Thread)
HPC Application Threading

for or do loop

works on big array

if (y=4) { 
  for( ){
    X[1000][10..]
  }
}

for or do loop

works on big array

print X

A[0] = 10;
i = i+1;
for( ){
  B[100][100]
}

for( ){
  if (y=4){
    
  }
}
This last fact means that we will emphasize the capabilities of OpenMP with a different focus than non-HPC programmers.

We will focus on getting our kernels to parallelize well.

We will be most concerned with dependencies, and not deadlocks and race conditions which confound other OpenMP applications.

This is very different from the generic approach you are likely to see elsewhere. The “encyclopedic” version can obscure how easy it is to get started with common loops.

But we will return to the most generic and flexible capabilities before we are done (OpenMP tasks).
This looks easy! Too easy...

- Why don’t we just throw *parallel for/do* (the OpenMP command for this purpose) in front of every loop?

- Better yet, why doesn’t the compiler do this for me?

The answer is that there are several general issues that would generate incorrect results or program hangs if we don’t recognize them:

- Data Dependencies

- Data Races
Data Dependencies

Most directive-based parallelization consists of splitting up big do/for loops into independent chunks that the many processors can work on simultaneously.

Take, for example, a simple for loop like this:

```java
for(index=0; index<10000; index++)
    Array[index] = 4 * Array[index];
```

When run on 10 cores, it will execute something like this...
No Data Dependency

```
for(index=0, index<999,index++)
    Array[index] = 4*Array[index];
```

```
for(index=1000, index<1999,index++)
    Array[index] = 4*Array[index];
```

```
for(index=2000, index<2999,index++)
    Array[index] = 4*Array[index];
```

```
for(index=3000, index<3999,index++)
    Array[index] = 4*Array[index];
```

```
for(index=4000, index<4999,index++)
    Array[index] = 4*Array[index];
```
Data Dependency

But what if the loops are not entirely independent?

Take, for example, a similar loop like this:

```c
for(index=1; index<10000; index++)
    Array[index] = 4 * Array[index] - Array[index-1];
```

This is perfectly valid serial code.
Data Dependency

Now core 1, in trying to calculate its first iteration,

```c
for(index=1000; index<1999; index++)
    Array[1000] = 4 * Array[1000] - Array[999];
```

needs the result of core 0’s last iteration. If we want the correct (“same as serial”) result, we need to wait until core 0 finishes. Likewise for cores 2, 3, ...
Recognizing and Eliminating Data Dependencies

- Recognize dependencies by looking for:
  - A dependence between iterations. Often visible due to use of differing indices.
  - Is the variable written and also read?
  - Any non-indexed (scaler) variables that are written to by index dependent variables.
  - You may get compiler warnings, and you may not.

- Can these be overcome?
  - Sometimes a simple rearrangement of the code will suffice. There is a common bag of tricks developed for this as this issue goes back 40 years in HPC (for vectorized computers). Many are quite trivial to apply.
  - We will now learn about OpenMP capabilities that will make some of these disappear.
  - Sometimes they are fundamental to the algorithm and there is no answer other than rewrite completely or leave as serial.

But you must catch these!
Plenty of Loops Don't Have Dependencies

If there aren't dependencies, we can go ahead and parallelize the loop. In the most straightforward case:

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

    int array[1000000];

    #pragma omp parallel for
    for (int i = 0; i <= 1000000; i++ ){
        array[i] = i;
    }
}
```

```fortran
program simple

    integer    array(1000000)

    !$omp parallel do
    do i = 1,1000000
        array(i)=i
    enddo
    !$omp end parallel do

end program
```

Standard C   Fortran
Compile and Run

We are using PGI compilers here. Others are very similar (-fopenmp, -omp). Likewise, if you are using a different command shell, you may do “setenv OMP_NUM_THREADS 8”.

Fortran:
pgf90 -mp simple.f90
export OMP_NUM_THREADS=8
a.out

C:
pgcc -mp simple.c
export OMP_NUM_THREADS=8
a.out

If you wonder if/how your directives are taking effect (a very valid question), the compilers always offer to be more verbose. With PGI, you can add the "-Minfo=mp" option. Give it a try.
Loops with Shared Variables

Most serious loops have other variables besides an array or two. The sharing of these variables introduces some potential issues. Here is a toy problem with a scalar that is written to.

```c
float height[1000], width[1000], cost_of_paint[1000];
float area, price_per_gallon = 20.00, coverage = 20.5;

for (index=0; index<1000; index++){
    area = height[index] * width[index];
    cost_of_paint[index] = area * price_per_gallon / coverage;
}
```

C Version

```fortran
real*8 height(1000), width(1000), cost_of_paint(1000)
real*8 area, price_per_gallon, coverage

do index=1,1000
    area = height(index) * width(index)
    cost_of_paint(index) = area * price_per_gallon / coverage
end do
```

Fortran Version
Applying Some OpenMP

A quick dab of OpenMP would start like this:

```c
#pragma omp parallel for
for (index=0; index<1000; index++){
    area = height[index] * width[index];
    cost_of_paint[index] = area * price_per_gallon / coverage;
}
```

We are requesting that this for/do loop be executed in parallel on the available processors.
If we ran this code we would find that sometimes our results differ from the serial code (and are simply wrong). The reason is that we have a shared variable that is getting overwritten by all of the threads.

```
#pragma omp parallel for
for (index=0; index<1000; index++){
    area = height[index] * width[index];
    cost_of_paint[index] = area * price_per_gallon / coverage;
}
```

Between its assignment and use by any one thread, there are other threads (7 here) potentially accessing and changing it. This is prone to error. Possibly the worst kind: the intermittent one.
By default variables are shared in OpenMP. Exceptions include index variables and variables declared inside parallel regions (C/C++). More later.
What We Want

With Two Threads

We can accomplish this with the **private** clause.
Apply the private clause and we have a working loop:

```c
#pragma omp parallel for private(area)
for (index=0; index<1000; index++){
    area = height[index] * width[index];
    cost_of_paint[index] = area * price_per_gallon / coverage;
}
```

```fortran
!$omp parallel do private(area)
do index=1,1000
    area = height(index) * width(index)
    cost_of_paint(index) = area * price_per_gallon / coverage
end do
!$omp end parallel do
```

There are several ways we might wish these controlled variables to behave. Let’s look at the related data-sharing clauses. **private** is the most common by far.
Other Data Sharing Clauses

**shared(list)**
This is the default (with the exception of index and locally declared variables. You might use this clause for clarification purposes.

**firstprivate(list)**
This will initialize the privates with the value from the master thread. _Otherwise, this does not happen!_

**lastprivate(list)**
This will copy out the last thread value into the master thread copy. _Otherwise, this does not happen!_ Available in for/do loop or section only, not available where “last iteration” isn’t clearly defined.

**default(list)**
You can change the default type to some of the others.

**threadprivate(list)**
Define at global level and these privates will be available in every parallel region. Use with `copyin()` to initialize values from master thread. Can think of these as on heap, while privates are on stack.
What is automatically private?

The default rules for sharing (which you should never be shy about redundantly designating with clauses) have a few subtleties.

Default is **shared**, except for things that *can not possibly be*:

- local variables in any called subroutine, unless using *static* (C) or *save* (Fortran)
- loop index variable
- inner loop index variables in Fortran, **but not in C**.
- variables declared within the block (for C).

These last two points make the C99 loop syntax quite convenient:

```c
#pragma omp parallel for
for ( int i = 0; i <= n; i++ ){
    for ( int j = 0; j<= m; j++ ){
        Array[i][j] = Array[i][j]+1
    }
}
```
The parallel for/do loop is common enough that we want to make sure we really understand what is going on.

```c
#pragma omp parallel for private (i,j)
for ( i = 0; i <= n; i++ ){
    for ( j = 0; j<= m; j++ ){
        Array[i][j] = Array[i][j]+1
    }
}
```

```
!$omp parallel do private (i,j)
do i = 2,n
do j = 2,i-1
    Array(j,i) = Array(j,i)+1
end do
end do
!$omp end parallel do
```

In general (well beyond OpenMP reasons), you want your innermost loop to index over adjacent items in memory. This is opposite for Fortran and C. In C this last index changes fastest. We can collapse nested loops with a `collapse(n)` clause.
Let’s try a slightly more complicated loop. This counts prime numbers.

### C Version

```c
#include <stdlib.h>
#include <stdio.h>

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

    int n = 500000;
    int not_primes=0;
    int i,j;

    for ( i = 2; i <= n; i++ ){
        for ( j = 2; j < i; j++ ){
            if ( i % j == 0 ){
                not_primes++;
                break;
            }
        }
    }

    printf("Primes: %d\n", n - not_primes);
}
```

### Fortran Version

```fortran
program primes

    integer n, not_primes, i, j
    n = 500000
    not_primes=0

    do i = 2,n
        do j = 2,i-1
            if (mod(i,j) == 0) then
                not_primes = not_primes + 1
                exit
            end if
        end do
    end do

    print *, 'Primes: ', n - not_primes
end program
```
Parallel Prime Counter

The most obvious thing is to parallelize the main loop.

C Version

```c
#pragma omp parallel for private (j)
for ( i = 2; i <= n; i++ ){
    for ( j = 2; j < i; j++ ){
        if ( i % j == 0 ){
            not_primes++;
            break;
        }
    }
}
```

Fortran Version

```fortran
!$omp parallel do
    do i = 2,n
        do j = 2,i-1
            if (mod(i,j) == 0) then
                not_primes = not_primes + 1
                exit
            end if
        end do
    end do
!$omp end parallel do
```

If we run this code on multiple threads, we will find that we get inconsistent results. What is going on?
Data Races

The problem here is a shared variable (not_primes) that is being written to by many threads.

The statement `not_primes = not_primes + 1` may look “atomic”, but in reality it requires the processor to first read, then update, then write the variable into memory. While this is happening, another thread may be writing its own (now obsolete) update. In this case, some of the additions to not_primes may be overwritten and ignored.

This sounds similar to our paint calculator example earlier. So will private fix this? Almost. Private variables aren’t subject to data races, and we will end up with multiple valid not_prime subtotals. So far so good.

The question then becomes, how do we sum these up into the real total we are looking for?

It is common to have a private variable that has to live on after the loop. This requires us to reduce these private copies back to a single scaler.
Reductions are **private** variables that must be reduced to a single value eventually.

### C Version

```c
#pragma omp parallel for private (j) 
  reduction(+: not_primes)
for ( i = 2; i <= n; i++ ){
  for ( j = 2; j < i; j++ ){
    if ( i % j == 0 ){
      not_primes++;
      break;
    }
  }
}
```

### Fortran Version

```fortran
!$omp parallel do reduction(+:not_primes)
do i = 2,n
  do j = 2,i-1
    if (mod(i,j) == 0) then
      not_primes = not_primes + 1
      exit
    end if
  end do
end do
!$omp end parallel do
```

At the end of the parallel region (the do/for loop), the private reduction variables will get combined using the operation we specified. Here, it is sum (+).
In addition to sum, we have a number of other options. You will find sum, min and max to be the most common. Note that the private variable copies are all initialized to the values specified.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>max</td>
<td>least number possible</td>
</tr>
<tr>
<td>min</td>
<td>largest number possible</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Bit (&amp;,</td>
<td>, ^, iand, ior)</td>
</tr>
<tr>
<td>Logical (&amp;&amp;,</td>
<td></td>
</tr>
</tbody>
</table>

The 4.0 standard even allows you to define your own. You probably won't.
We shall return.

```c
#pragma omp parallel for private (j) \
    reduction(+:not_primes)
for ( i = 2; i <= n; i++ ){
    for ( j = 2; j < i; j++ ){
        if ( i % j == 0 ){
            not_primes++;
            break;
        }
    }
}

C Version

A few notes before we leave (for now):

• The OpenMP standard forbids branching out of parallel do/for loops, although you can now cancel. Since the outside loop is the threaded one (that is how it works), our break/exit statement for the inside loop are OK.

• You can verify the output at primes.utm.edu/nthprime/index.php#piofx Note that we count 1 as prime. They do not.
We will also use this for MPI and OpenACC. It is a great simulation problem, not rigged for OpenMP.

In this most basic form, it solves the Laplace equation: \( \nabla^2 f(x, y) = 0 \)

The Laplace Equation applies to many physical problems, including:
- Electrostatics
- Fluid Flow
- Temperature

For temperature, it is the Steady State Heat Equation:
Exercise Foundation: Jacobi Iteration

- The Laplace equation on a grid states that each grid point is the average of its neighbors.
- We can iteratively converge to that state by repeatedly computing new values at each point from the average of neighboring points.
- We just keep doing this until the difference from one pass to the next is small enough for us to tolerate.

\[
A_{k+1}(i, j) = \frac{A_k(i-1, j) + A_k(i+1, j) + A_k(i, j-1) + A_k(i, j+1)}{4}
\]
for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
        Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] + Temperature_last[i][j+1] + Temperature_last[i][j-1]);
    }
}

do j=1,columns
    do i=1,rows
        temperature(i,j)= 0.25 * (temperature_last(i+1,j)+temperature_last(i-1,j) + & temperature_last(i,j+1)+temperature_last(i,j-1) )
    enddo
enddo
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] + Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    dt = 0.0;
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    }

    if((iteration % 100) == 0) {
        track_progress(iteration);
    }

    iteration++;
}

Serial C Code (kernel)

Done?
Calculate
Update temp array and find max change
Output
void initialize() {
    int i,j;
    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }
    // these boundary conditions never change throughout run
    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++){
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }
    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++){
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}

void track_progress(int iteration) {
    int i;
    printf("-- Iteration: %d --\n", iteration);
    for(i = ROWS-5; i <= ROWS; i++) {
        printf("[%d,%d]: %5.2f ", i, i, Temperature[i][i]);
    }
    printf("\n");
}
```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

// size of plate
#define COLUMNS 1000
#define ROWS 1000

// largest permitted change in temp (This value takes about 3400 steps)
#define MAX_TEMP_ERROR 0.01

double Temperature[ROWS+2][COLUMNS+2]; // temperature grid
double Temperature_last[ROWS+2][COLUMNS+2]; // temperature grid from last iteration

// helper routines
void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {
    int i, j;                                            // grid indexes
    int max_iterations;                                  // number of iterations
    int iteration=1;                                     // current iteration
    double dt=100;                                       // largest change in t
    struct timeval start_time, stop_time, elapsed_time;  // timers
    printf("Maximum iterations [100-4000]?\n");
    scanf("%d", &max_iterations);
    gettimeofday(&start_time,NULL); // Unix timer
    initialize();                   // initialize Temp_last including boundary conditions
    // do until error is minimal or until max steps
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {
        // main calculation: average my four neighbors
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] + Temperature_last[i][j+1] + Temperature_last[i][j-1]);
            }
        }
        dt = 0.0; // reset largest temperature change
        // copy grid to old grid for next iteration and find latest dt
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
                Temperature_last[i][j] = Temperature[i][j];
                Temperature_last[i][i] = Temperature[i][i];
            }
        }
        // periodically print test values
        if((iteration % 100) == 0) {
            track_progress(iteration);
        }
        iteration++;
    }
    gettimeofday(&stop_time,NULL);
timersub(&stop_time, &start_time, &elapsed_time); // Unix time subtract routine
    printf("Max error at iteration %d was %f\n", iteration-1, dt);
    printf("Total time was %f seconds.\n", elapsed_time.tv_sec +elapsed_time.tv_usec/1000000.0);
}

// initialize plate and boundary conditions
// Temp_last is used to to start first iteration
void initialize(){
    int i,j;
    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }
    // these boundary conditions never change throughout run
    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }
    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}

// print diagonal in bottom right corner where most action is
void track_progress(int iter){
    int i;
    printf("---------- Iteration number: %d -------------\n", iteration);
    for(i = ROWS-5; i <= ROWS; i++) {
        printf("[%d,%d]: %5.2f  ", i, i, Temperature[i][i]);
    }
    printf("\n");
}
```

do while ( dt > max_temp_error .and. iteration <= max_iterations)

    do j=1,columns
        do i=1,rows
            temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                              temperature_last(i,j+1)+temperature_last(i,j-1) )
        enddo
    enddo

    dt=0.0

    do j=1,columns
        do i=1,rows
            dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
            temperature_last(i,j) = temperature(i,j)
        enddo
    enddo

    if( mod(iteration,100).eq.0 ) then
        call track_progress(temperature, iteration)
    endif

    iteration = iteration+1

endo
subroutine initialize( temperature_last )
  implicit none
  integer, parameter :: columns=1000
  integer, parameter :: rows=1000
  integer :: i, j
  double precision, dimension(0:rows+1,0:columns+1) :: temperature_last
  !these boundary conditions never change throughout run
  !set left side to 0 and right to linear increase
  do i=0,rows+1
    temperature_last(i,0) = 0.0
    temperature_last(i,columns+1) = (100.0/rows) * i
  enddo
  !set top to 0 and bottom to linear increase
  do j=0,columns+1
    temperature_last(0,j) = 0.0
    temperature_last(rows+1,j) = ((100.0)/columns) * j
  enddo
end subroutine initialize

subroutine track_progress(temperature, iteration)
  implicit none
  integer, parameter :: columns=1000
  integer, parameter :: rows=1000
  integer :: i, iteration
  double precision, dimension(0:rows+1,0:columns+1) :: temperature
  !these boundary conditions never change throughout run
  !set left side to 0 and right to linear increase
  do i=0,rows+1
    temperature_last(i,0) = 0.0
    temperature_last(i,columns+1) = (100.0/rows) * i
  enddo
  !set top to 0 and bottom to linear increase
  do j=0,columns+1
    temperature_last(0,j) = 0.0
    temperature_last(rows+1,j) = ((100.0)/columns) * j
  enddo
end subroutine track_progress
program serial

implicit none

! Size of plate
integer, parameter :: columns=1000
integer, parameter :: rows=1000

! Set max iterations, iteration = 1
integer :: i, j, max_iterations, iteration=1

double precision ::
dt=100.0
real ::
start_time, stop_time

double precision, dimension(0:rows+1,0:columns+1) :: temperature,

temperature_last

double precision, dimension(0:columns+1) :: temperature_last

integer, parameter :: columns=1000
integer, parameter :: rows=1000
integer :: i, j

double precision, dimension(0:columns+1) :: temperature_last

integer, parameter :: columns=1000
integer, parameter :: rows=1000
integer :: i, j

double precision, dimension(0:columns+1) :: temperature_last

print*, 'Maximum iterations [100-4000]?'
read*, max_iterations

call cpu_time(start_time) ! Fortran timer

call initialize(temperature_last)

! Do until error is minimal or until max iterations
do while ( dt > max_temp_error .and. iteration <= max_iterations)

do j=1,columns

doit j=1,rows

temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                             temperature_last(i,j+1)+temperature_last(i,j-1) )

endo
door

dt=0.0

! Copy grid to old grid for next iteration and find max change
do j=1,columns

doit j=1,rows

dt = max( abs(temperature_last(i,j) - temperature_last(i,j)), dt )

temperature_last(i,j) = temperature(i,j)

endo
endo

! Periodically print test values
if ( mod(iteration,100).eq.0 ) then

call track_progress(temperature, iteration)
endif

iteration = iteration+1
endo

call cpu_time(stop_time)

print*, 'Max error at iteration ', iteration-1, ' was ',dt
print*, 'Total time was ',stop_time-start_time, ' seconds.'
end program serial
Exercise 1: Use OpenMP to parallelize the Jacobi loops
(About 45 minutes)

1) Log onto a node requesting all the cores (28 on a regular Bridges node).

   > interact -n 28

2) Edit laplace_serial.c or laplace_serial.f90 (your choice) and add directives where it helps. Try adding "-Minfo=mp" to verify what you are doing.

3) Run your code on various numbers of cores (such as 8, per below) and see what kind of speedup you achieve.

   > pgcc -mp laplace_omp.c or pgf90 -mp laplace_omp.f90
   > export OMP_NUM_THREADS=8
   > a.out
Fortran Timing Note

On some platforms the universal Fortran cpu_time() function will report aggregate CPU time. You can divide your answer by the number of threads to get an effective answer. Or, you can take this opportunity to start using some of the useful OpenMP run time library - namely omp_get_time().

C:
#include <omp.h>
double start_time = omp_get_wtime();
...
double end_time = omp_get_wtime();

Fortran:
use omp_lib
double precision :: start_time, stop_time
start_time = omp_get_wtime()
...
end_time = omp_get_wtime()
while (dt > MAX_TEMP_ERROR && iteration <= max_iterations) {
    #pragma omp parallel for private(i,j)
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] + Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }
    dt = 0.0; // reset largest temperature change

    #pragma omp parallel for reduction(max:dt) private(i,j)
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    }

    if((iteration % 100) == 0) {
        track_progress(iteration);
    }
    iteration++;
}
do while ( dt > max_temp_error .and. iteration <= max_iterations)

!$omp parallel do
do j=1,columns
    do i=1,rows
        temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
        temperature_last(i,j+1)+temperature_last(i,j-1) )
    enddo
enddo
!$omp end parallel do

dt=0.0

!$omp parallel do reduction(max:dt)
do j=1,columns
    do i=1,rows
        dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
        temperature_last(i,j) = temperature(i,j)
    enddo
enddo
!$omp end parallel do

if( mod(iteration,100).eq.0 ) then
    call track_progress(temperature, iteration)
endif

iteration = iteration+1
enddo
Scaling?

For the solution in the Laplace directory, we found this kind of scaling when running to convergence at 3372 iterations.

<table>
<thead>
<tr>
<th>Threads</th>
<th>C (s)</th>
<th>Fortran (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>14.4</td>
<td>14.7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7.2</td>
<td>7.1</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>3.7</td>
<td>3.5</td>
<td>3.9</td>
</tr>
<tr>
<td>8</td>
<td>2.1</td>
<td>2.4</td>
<td>6.9</td>
</tr>
<tr>
<td>16</td>
<td>1.2</td>
<td>1.3</td>
<td>12</td>
</tr>
<tr>
<td>28</td>
<td>0.8</td>
<td>0.9</td>
<td>18</td>
</tr>
<tr>
<td>32</td>
<td>1.8</td>
<td>2.4</td>
<td>0.8</td>
</tr>
</tbody>
</table>

The larger version of this problem that we use for the hybrid programming example (10K x 10K) continues to scale nicely on Bridges 12TB memory nodes to hundreds of cores!
Time for a breather.

Congratulations, you have now learned the OpenMP parallel for/do loop. That is a pretty solid basis for using OpenMP. To recap, you just have to keep an eye out for:

- Dependencies
- Data races

and know how to deal with them using:

- Private variables
- Reductions
Different Work Sharing Constructs

What we have been doing

What we could do (*less overhead, no idle cores, finer control, more flexible algorithms*)
Number of Threads in a Parallel Region

In order of precedence:

**IF clause**
Logical value determines if this region is parallel or serial.

**NUM_THREADS clause**
Set this to specify how many threads in this region.

**omp_set_num_threads()**
A library API to set the threads.

**OMP_NUM_THREADS**
The environment variable we have been using.

**Default**
Often the number of cores on the node.

There is also, depending on the compute environment, the possibility of dynamic thread counts. There are a few library APIs to deal with that.
Fortran 90 has data parallel constructs that map very well to threads. You can declare a workshare region and OpenMP will do the right thing for:

- **FORALL**
- **WHERE**
- Array assignments

```fortran
PROGRAM WORKSHARE

INTEGER N, I, J
PARAMETER (N=100)
REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
.
.
.
!$OMP PARALLEL SHARED(AA, BB, CC, DD, FIRST, LAST)

!$OMP WORKSHARE
  CC = AA * BB
  DD = AA + BB
  FIRST = CC(1,1) + DD(1,1)
  LAST = CC(N,N) + DD(N,N)

!$OMP END WORKSHARE

!$OMP END PARALLEL

END
```
Another Work Sharing Construct

Sections

Each section will be processed by one thread. The number of sections can be greater or less than the number of threads available - in which case threads will do more than one section or skip, respectively.
Sections

Both for/do loops run concurrently. Still same results as serial here.
And for ultimate flexibility: Tasks

Actually, any thread can spin off tasks. And any thread can pick up a task. They will all wait for completion at the end of the region.
Let's take the simple task of summing an array.

```c
float array_sum(float *a, int length){
    float total=0;
    for (int i = 0; i < length; i++) {
        total += a[i];
    }
    return total;
}
```

Serial Code

```c
float array_sum(float *a, int length){
    float total=0;
    #pragma omp parallel for reduction(+:total)
    for (int i = 0; i < length; i++) {
        total += a[i];
    }
    return total;
}
```

Easy OpenMP Version
Recursively Summing An Array

But maybe we are handed a recursive version of this same code. This represents a large class of algorithms.

```c
float array_sum(float *a, int length){
    // terminal case
    if (length == 0) {
        return 0;
    }
    else if (length == 1) {
        return a[0];
    }
    // recursive case
    int half = length / 2;
    return array_sum(a, half) + sum(a + half, length - half);
}
```
Recursively Summing An Array With Tasks

OpenMP tasks allow us to naturally spin off threads of work.

```c
float array_sum(float *a, int length){
    if (length == 0) {
        return 0;
    } else if (length == 1) {
        return a[0];
    }
    int half = length / 2;
    float x, y;
    #pragma omp parallel
    #pragma omp single nowait
    {
        #pragma omp task shared(x)
        x = array_sum(a, half);
        #pragma omp task shared(y)
        y = array_sum(a + half, length - half);
        #pragma omp taskwait
        x += y;
    }
    return x;
}
```
float array_sum(float *a, int length) {
    float total;
    #pragma omp parallel
    #pragma omp single nowait
    total = parallel_sum(a, n);
    return total;
}

float parallel_sum(float *a, int length) {
    if (length <= CUTOFF) {
        return serial_sum(a, length);
    }
    int half = length / 2;
    float x, y;
    #pragma omp task shared(x)
    x = parallel_sum(a, half);
    #pragma omp task shared(y)
    y = parallel_sum(a + half, length - half);
    #pragma omp taskwait
    x += y;
    return x;
}

float serial_sum(float *a, int length) {
    if (length == 0) {
        return 0;
    } else if (length == 1) {
        return a[0];
    }
    size_t half = n / 2;
    return serial_sum(a, half) + serial_sum(a + half, length - half);
}

BTW, we have essentially reproduced the functionality here of the newish taskloop directive.
Fibonacci Tasks

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

int main()
{
    int n = 10;
    #pragma omp parallel shared(n)
    {
        #pragma omp single
        printf("fib(%d) = %d\n", n, fib(n));
    }
}

int fib(int n)
{
    int i, j;
    if (n<2)
        return n;
    else {
        #pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);
        #pragma omp task shared(j) firstprivate(n)
        j=fib(n-2);
        #pragma omp taskwait
        return i+j;
    }
}
```

Here is one that is almost always presented as a recursive algorithm.
Task Capability

Tasks have additional directives and clauses. The most important are:

- **taskwait** (wait for completion of child tasks, should almost always use)
- **taskgroup** (can wait on child & descendants)
- **taskyield** (can suspend for another task, avoid deadlock)
- **final** (no more task creation after this level)
- **untied** (can change thread dynamically)
- **mergable** (can merge data with enclosing region)
- **depend** (list variable dependencies between tasks [in/out/inout] This provides a way to order workflow.)

This last one gives us some very powerful capabilities to efficiently manage order dependencies, and has been an active area of OpenMP development in versions 3.0 through the latest 5.0.
int x = 0, y = 0, res = 0;

#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(out: res) //T0
    res = 0;

    #pragma omp task depend(out: x) //T1
    long_computation(x);

    #pragma omp task depend(out: y) //T2
    short_computation(y);

    #pragma omp task depend(in: x) depend(inout: res) //T3
    res += x;

    #pragma omp task depend(in: y) depend(inout: res) //T4
    res += y;

    #pragma omp task depend(in: res) //T5
    std::cout << res << std::endl;
}
New Task Dependencies

```c
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
    #pragma omp task depend(out: res) //T0
    res = 0;

    #pragma omp task depend(out: x) //T1
    long_computation(x);

    #pragma omp task depend(out: y) //T2
    short_computation(y);

    #pragma omp task depend(in: x) depend(mutexinoutset: res) //T3
    res += x;

    #pragma omp task depend(in: y) depend(mutexinoutset: res) //T4
    res += y;

    #pragma omp task depend(in: res) //T5
    std::cout << res << std::endl;
}
```
Dynamic Dependencies

We can also now (as of OpenMP 5.0) deal with dynamically defined dependencies.

```cpp
#pragma omp parallel
#pragma omp single
{
    for (int i = 0; i < n; ++i)
        #pragma omp task depend(out: array[i])
        compute_element(array[i]);

    #pragma omp task depend(iterator(k=0:n),in: array[k])
    use_elements(array);
}
```
```c
#pragma omp parallel shared(t, t_old) private(i,j, iter) firstprivate(niter)
for(iter = 1; iter <= niter; iter++) {

#pragma omp for
for(i = 1; i <= NR; i++) {
    for(j = 1; j <= NC; j++) {
        t[i][j] = 0.25 * (t_old[i+1][j] + t_old[i-1][j] +
                          t_old[i][j+1] + t_old[i][j-1]);
    }
}

dt = 0.0;

#pragma omp for reduction(max:dt)
for(i = 1; i <= NR; i++){
    for(j = 1; j <= NC; j++){
        dt = fmax(fabs(t[i][j]-t_old[i][j]), dt);
        t_old[i][j] = t[i][j];
    }
}
if((iter % 100) == 0) {
    print_trace(iter);
}
Parallel Region Loops with Fortran

```
!$omp parallel shared(T, Told) private(i,j,iter) firstprivate(niter)
  do iter=1,niter
    !$omp do
    do j=1,NC
      do i=1,NR
        T(i,j) = 0.25 * ( Told(i+1,j)+Told(i-1,j)+
                         Told(i,j+1)+Told(i,j-1) )
      enddo
    enddo
    !$omp end do
    dt = 0
    !$omp do reduction(max:dt)
    do j=1,NC
      do i=1,NR
        dt = max( abs(t(i,j) - told(i,j)), dt )
        Told(i,j) = T(i,j)
      enddo
    enddo
    !$omp end do
    if( mod(iter,100).eq.0 ) then
      call print_trace(t, iter)
    endif
  enddo
!$omp end parallel
```
Thread control.

If we did this, we would get correct results, but we would also find that our output is a mess.

How many iterations [100-1000]? 1000

---------- Iteration number: 100 ----------

[995,995]: 63.33  [996,996]: 72.67  [997,997]: 81.40  [998,998]: 88.97  [999,999]: 94.86  [1000,1000]: 98.67

---------- Iteration number: 100 ----------

[995,995]: 63.33  [996,996]: 72.67  [997,997]: 81.40  [998,998]: 88.97

---------- Iteration number: 100 ----------

[995,995]: 63.33  [996,996]: 72.67  [997,997]: 81.40  [998,998]: 88.97  [999,999]: 94.86  [1000,1000]: 98.67

---------- Iteration number: 100 ----------

[995,995]: 63.33  [996,996]: 72.67

[999,999]: 94.86  [1000,1000]: 98.67

All of our threads are doing output. We only want the master thread to do this. This is where we find the rich set of thread control tools available to us in OpenMP.
### Solution with Master

```c
if( mod(iter,100).eq.0 ) then
    call print_trace(t, iter)
endif
!
```  

The Master directive will only allow the region to be executed by the master thread. Other threads skip. By skip we mean race ahead - to the next iteration. We really should have an "omp barrier" after this or threads could already be altering $t$ as we are writing it out. Life in parallel regions can get tricky!
A barrier is executed by all threads only at:

- A **barrier** command
- Entry to and exit from a parallel region
- Exit only from a worksharing command (like `do`/`for`)
- Except if we use the `nowait` clause

There are no barriers for any other constructs including `master` and `critical`!
Solution with thread IDs

```
... tid = omp_get_thread_num();
if (tid == 0) {
    if((iter % 100) == 0) {
        print_trace(iter);
    }
}
... tid = OMP_GET_THREAD_NUM();
if( tid .eq. 0 ) then
    if( mod(iter,100).eq.0 ) then
        call print_trace(t, iter)
    endif
endif
...```

Now we are using OpenMP runtime library routines, and not directives. We would have to use ifdef if we wanted to preserve the serial version. Also, we should include a barrier somewhere here as well.
<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>Like Master, but any thread will do. Has a <code>copyprivate</code> clause that can be used to copy its private values to all other threads.</td>
</tr>
<tr>
<td>critical</td>
<td>Only one thread at a time can go through this section. Can be named or unnamed (only one thread in all unnamed regions).</td>
</tr>
<tr>
<td>atomic</td>
<td>Eliminates data race on this one specific, simple statement. More efficient than <code>critical</code>.</td>
</tr>
<tr>
<td>ordered</td>
<td>Forces serial order on loops.</td>
</tr>
<tr>
<td>nowait</td>
<td>This clause will eliminate implied barriers on certain directives.</td>
</tr>
<tr>
<td>flush</td>
<td>Even cache coherent architectures need this to eliminate possibility of register storage issues. Tricky, but important <em>iff</em> you get tricky. We will return to this.</td>
</tr>
</tbody>
</table>
Is this starting to seem tricky?

As we have started to get away from the simplicity of the do/for loop and pursue the freedom of parallel regions and individual thread control, we have started to encounter subtle pitfalls.

So, you may be relieved to know that we have covered almost all of the OpenMP directives at this point. However, there are a few more run-time library routines to mention...
Run-time Library Routines

- **OMP_SET_NUM_THREADS**: Sets the number of threads that will be used in the next parallel region.
- **OMP_GET_NUM_THREADS**: Returns the number of threads that are currently in the team executing the parallel region from which it is called.
- **OMP_GET_MAX_THREADS**: Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function.
- **OMP_GET_THREAD_NUM**: Returns the thread number of the thread, within the team, making this call.
- **OMP_GET_THREAD_LIMIT**: Returns the maximum number of OpenMP threads available to a program.
- **OMP_GET_NUM_PROCS**: Returns the number of processors that are available to the program.
- **OMP_IN_PARALLEL**: Used to determine if the section of code which is executing is parallel or not.
- **OMP_SET_DYNAMIC**: Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
- **OMP_GET_DYNAMIC**: Used to determine if dynamic thread adjustment is enabled or not.
- **OMP_SET_NESTED**: Used to enable or disable nested parallelism.
- **OMP_GET_NESTED**: Used to determine if nested parallelism is enabled or not.
- **OMP_SET_SCHEDULE**: Sets the loop scheduling policy when “runtime” is used as the schedule kind in the OpenMP directive.
- **OMP_GET_SCHEDULE**: Returns the loop scheduling policy when “runtime” is used as the schedule kind in the OpenMP directive.
- **OMP_SET_MAX_ACTIVE_LEVELS**: Sets the maximum number of nested parallel regions.
- **OMP_GET_MAX_ACTIVE_LEVELS**: Returns the maximum number of nested parallel regions.
- **OMP_GET_LEVEL**: Returns the current level of nested parallel regions.
- **OMP_GET_ANCESTOR_THREAD_NUM**: Returns, for a given nested level of the current thread, the thread number of ancestor thread.
- **OMP_GET_TEAM_SIZE**: Returns, for a given nested level of the current thread, the size of the thread team.
- **OMP_GET_ACTIVE_LEVEL**: Returns the number of nested, active parallel regions enclosing the task that contains the call.
- **OMP_IN_FINAL**: Returns true if the routine is executed in the final task region; otherwise it returns false.
- **OMP_INIT_LOCK**: Initializes a lock associated with the lock variable.
- **OMP_DESTROY_LOCK**: Disassociates the given lock variable from any locks.
- **OMP_SET_LOCK**: Acquires ownership of a lock.
- **OMP_UNSET_LOCK**: Releases a lock.
- **OMP_TEST_LOCK**: Attempts to set a lock, but does not block if the lock is unavailable.
- **OMP_INIT_NEST_LOCK**: Initializes a nested lock associated with the lock variable.
- **OMP_DESTROY_NEST_LOCK**: Disassociates the given nested lock variable from any locks.
- **OMP_SET_NEST_LOCK**: Acquires ownership of a nested lock.
- **OMP_UNSET_NEST_LOCK**: Releases a nested lock.
- **OMP_TEST_NEST_LOCK**: Attempts to set a nested lock, but does not block if the lock is unavailable.
#include <stdio.h>
#include <omp.h>

omp_lock_t my_lock;

int main() {
    omp_init_lock(&my_lock);

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int i;

        omp_set_lock(&my_lock);

        for (i = 0; i < 5; ++i) {
            printf("Thread %d - in locked region\n", tid);
        }

        printf("Thread %d - in locked region\n", tid);

        omp_unset_lock(&my_lock);
    }

    omp_destroy_lock(&my_lock);
}

This could have been done with just an omp critical!
Pthreads like flexibility

We now have the ability to start coding just about any kind of thread flow we can imagine. And we can start creating all kinds of subtle and non-repeatable bugs. This is normally where we start the fun of cataloging all of the ways we can get into trouble:

- Race conditions
- Deadlocks
- Livelocks
- Missing flush

So, what are the benefits of these paradigms? Efficiency
flush

If you start delving into these capabilities, you need to understand the flush command. Even shared memory machines have cache issues and compiler instruction reordering that can cause shared values to get out of sync *if you insist on reading and writing shared variables from different threads* (like rolling your own locks or mutexes). You can rectify these problems with:

- implicit barriers (as mentioned previously)
- **barrier** (incurs synchronization penalty)
- **flush** (no sync)

If you think you are wandering into this territory, the best reference for examples and warnings is:

OpenMP Application Program Interface
http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf
Complexity vs. Efficiency

How much you will gain in efficiency by using these more flexible (dangerous) routines depends upon your algorithm. How asynchronous can it be?

OpenMP Library API
- OMP_SET_NUM_THREADS
- OMP_SET_LOCK
- flush

OpenMP Directives
- omp parallel for
- omp parallel do

Password cracking (Using work farming)
Prime Number Finding
Matrix Multiply

The general question is, how much time are threads spending at barriers? If you can’t tell, profiling will.
We do have a way of greatly affecting the thread scheduling while still using do/for loops. That is to use the `schedule` clause.

Let’s think about what happens with our prime number program if the loop iterations are just evenly distributed across our processors. Some of our iterations/threads will finish much earlier than others.
Scheduling Options

**static, n**  
Divides iterations evenly amongst threads. You can optionally specify the chunk size to use.

**dynamic, n**  
As a thread finishes, it is assigned another. Default chunk size is 1.

**guided, n**  
Block size will decrease with each new assignment to account for remaining iterations at that time. Chunk size specifies minimum (and defaults to 1).

**runtime**  
Decided at runtime by OMP_SCHEDULE variable.

**auto**  
Let the compiler/runtime decide.
Exercise 2: Improving Prime Number
(About 30 minutes)

Speed up the prime number count just using the scheduling options you have available.

1) Start with the prime_serial.c/f version in the OpenMP/Prime folder and then add the parallel directives as per the previous lecture slides. See how much it speeds up on various thread counts. Then...

2) Try various scheduling options to see if anything is effective at optimizing further. This “empirical” approach is a perfectly reasonable, and safe, way to find some low-hanging fruit.
One Scheduling Solution

### C Version

```c
#pragma omp parallel for private (j) 
    reduction(+:not_primes) 
    schedule(dynamic)
for ( i = 2; i <= n; i++ ){
    for ( j = 2; j < i; j++ ){
        if ( i % j == 0 ){
            not_primes++;
            break;
        }
    }
}
```

### Fortran Version

```fortran
!$omp parallel do reduction(+:not_primes) schedule(dynamic)
do i = 2,n
    do j = 2,i-1
        if (mod(i,j) == 0) then
            not_primes = not_primes + 1
            exit
        end if
    end do
end do
!$omp end parallel do
```

Dynamic scheduling with a default chunksize (of 1).
Results

We get a pretty big win for little work and even less danger. The Fortran and C times are almost exactly the same for this code.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Default (s)</th>
<th>Dynamic</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>30.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>22.3</td>
<td>15.2</td>
<td>1.5</td>
</tr>
<tr>
<td>4</td>
<td>13.0</td>
<td>8.1</td>
<td>1.6</td>
</tr>
<tr>
<td>8</td>
<td>7.6</td>
<td>4.2</td>
<td>1.8</td>
</tr>
<tr>
<td>16</td>
<td>4.2</td>
<td>2.2</td>
<td>1.9</td>
</tr>
<tr>
<td>28</td>
<td>2.4</td>
<td>1.2</td>
<td>2</td>
</tr>
</tbody>
</table>

500,000 iterations.
C++

- private /shared, etc. work with objects
  constructors/destructor are called for private
  things can get complicated with firstprivate, threadprivate, etc.

- Probably biggest question is std::vector
  Safe if no reallocation: No push_back(), pop_back(), insert()
  Iterators are even allowed in for loop here

- Other containers less likely to just work
  For example, std::list (a doubly linked list) updated by multiple threads would be a nightmare

- Note: MPI 3 and newer have dropped C++, so be aware if aiming for larger scalability
We have now covered everything up to (but not completely including) OpenMP 4.0. I hope you still recall how much we accomplished with just a parallel for/do. Let’s recap:

- Look at your large, time-consuming for/do loops first
  - Deal with dependencies and reductions
  - Using private and reductions
  - Consider scheduling

- If you find a lot of barrier time (via inspection or profiler) then:
  - Sections
  - Tasks
  - Run-time library
  - Locks
  - Barriers/nowaits