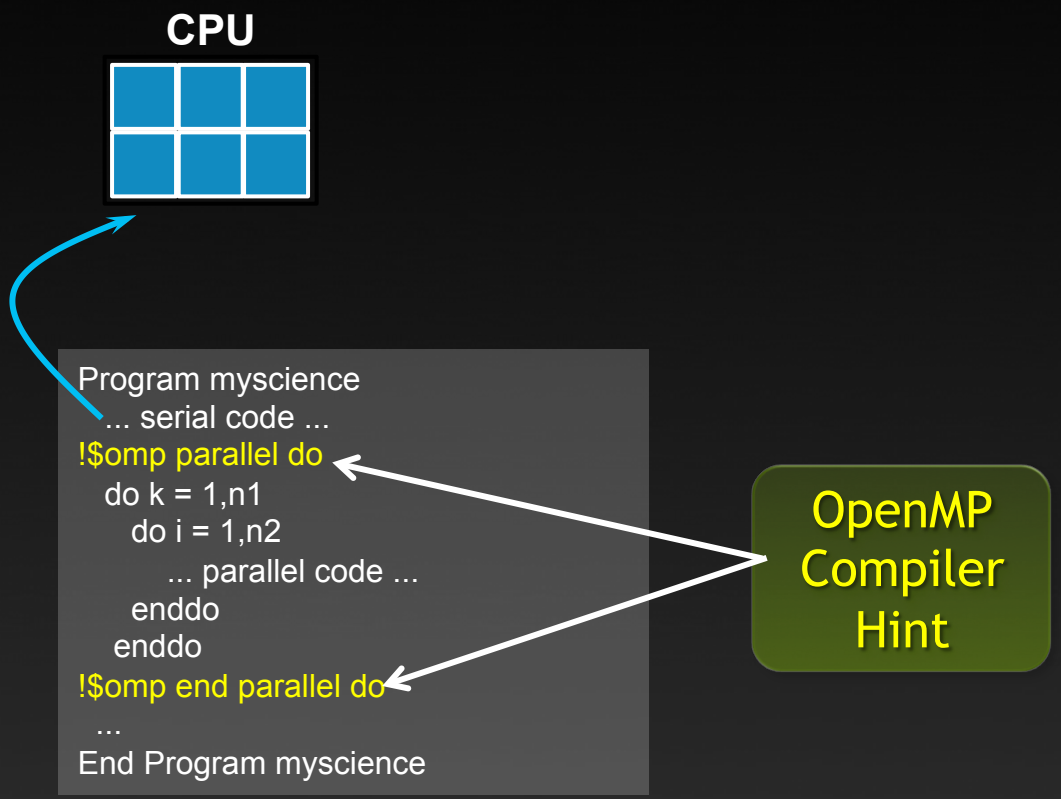
Introduction to OpenMP

John Urbanic
Parallel Computing Specialist
Pittsburgh Supercomputing Center

What is OpenMP?

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

Directives



Simple compiler hints from coder.

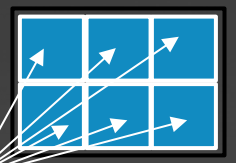
Compiler generates parallel threaded code.

Ignorant compiler just sees some comments.

Directives: an awesome idea whose time has arrived.

OpenMP

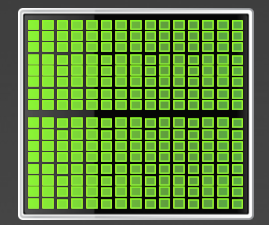
CPU



```
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);
}
```

OpenACC

GPU



```
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);
}
```

Key Advantages Of This Approach

- High-level. No involvement of pthreads or hardware specifics.
- Single source. No forking off a separate GPU 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)

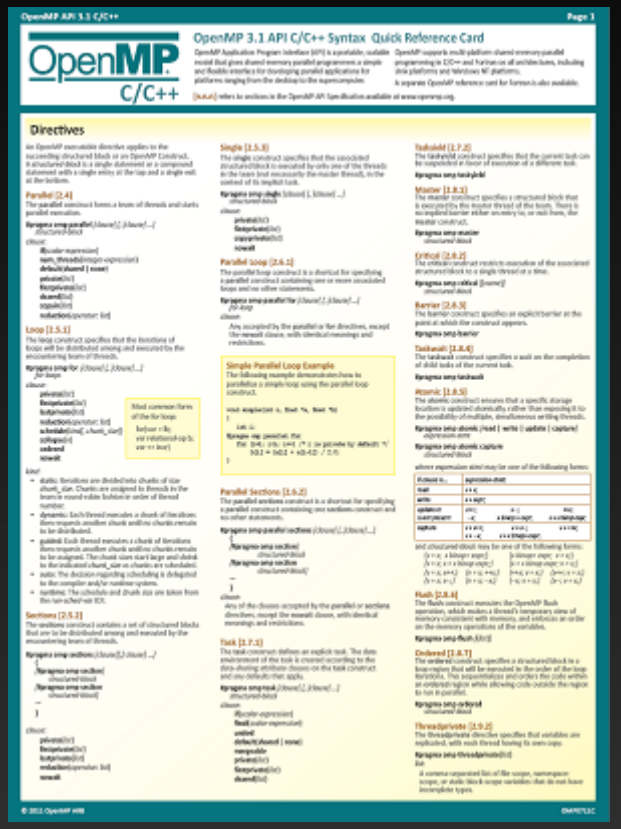
- GCC
- MS Visual Studio
- Intel
- IBM
- PGI

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)


OpenMP 3.1 API C/C++ Syntax Quick Reference Card

OpenMP C/C++

Directives

PITTSBURGH SUPERCOMPUTING CENTER

Hello World

Hello World in C

```
int main(int argc, char** argv){

  #pragma omp parallel
  {
    printf("Hello world.\n");
  }

}
```

Hello World in Fortran

```
 program hello

!$OMP PARALLEL

      print *,"Hello World."

!$OMP END PARALLEL

      stop
      end
```

```
Hello World.
Hello World.
Hello World.
Hello World.
```

Output with OMP_NUM_THREADS=4

General Directive Syntax and Scope

This is how these directives integrate into code:

Fortran

```
!$omp parallel [clause …]
    structured block
!$acc end parallel
```

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.

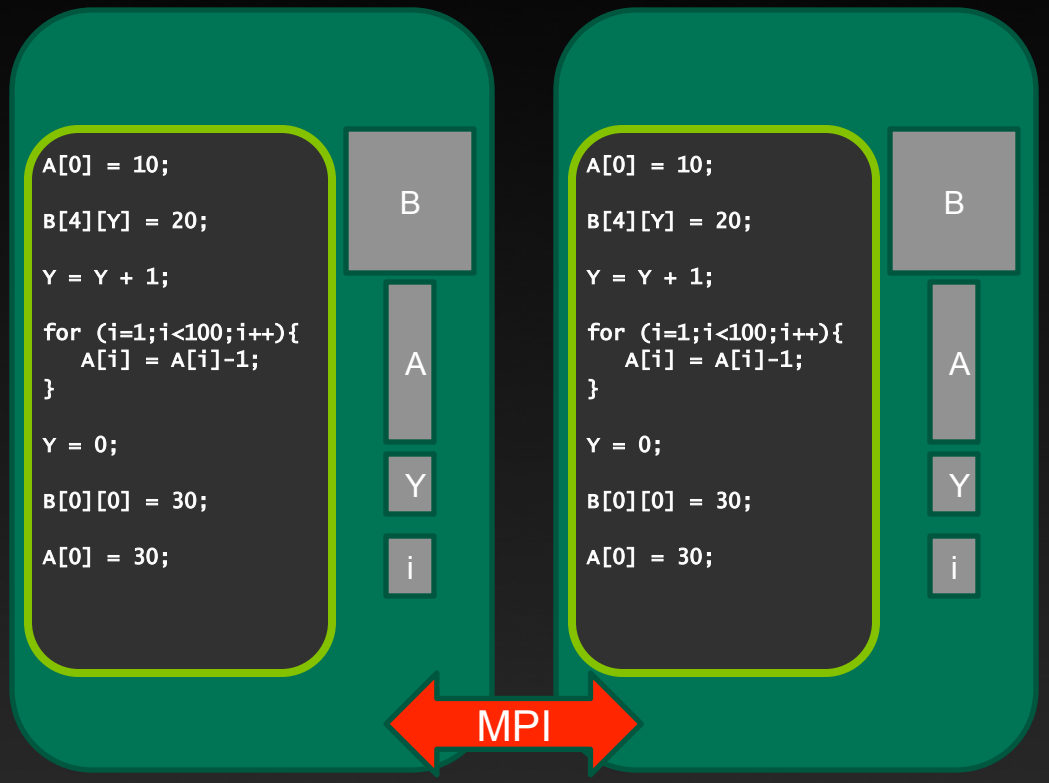
Pthreads

```
#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);
}
```

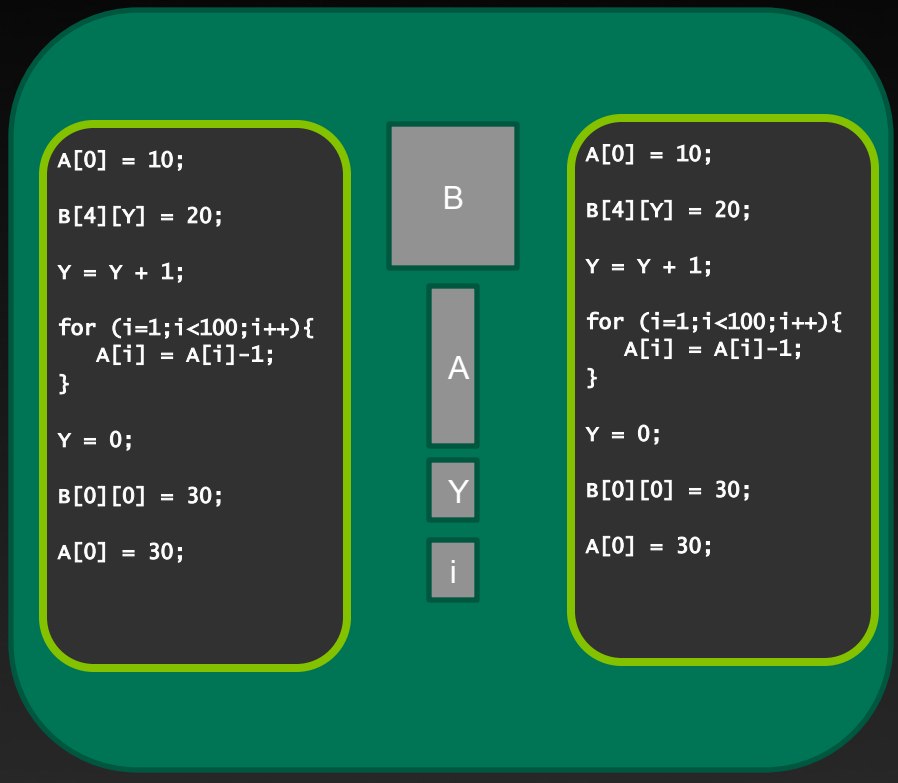
Big Difference!

- 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

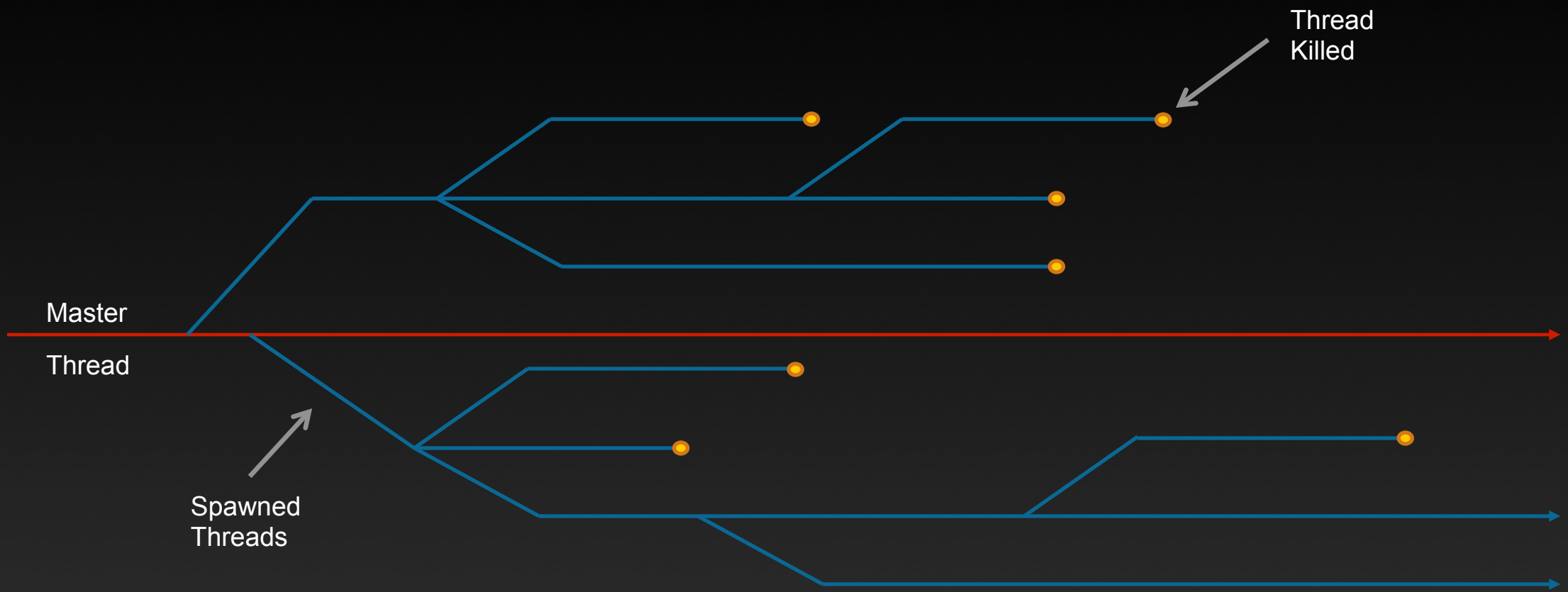


Two Processes

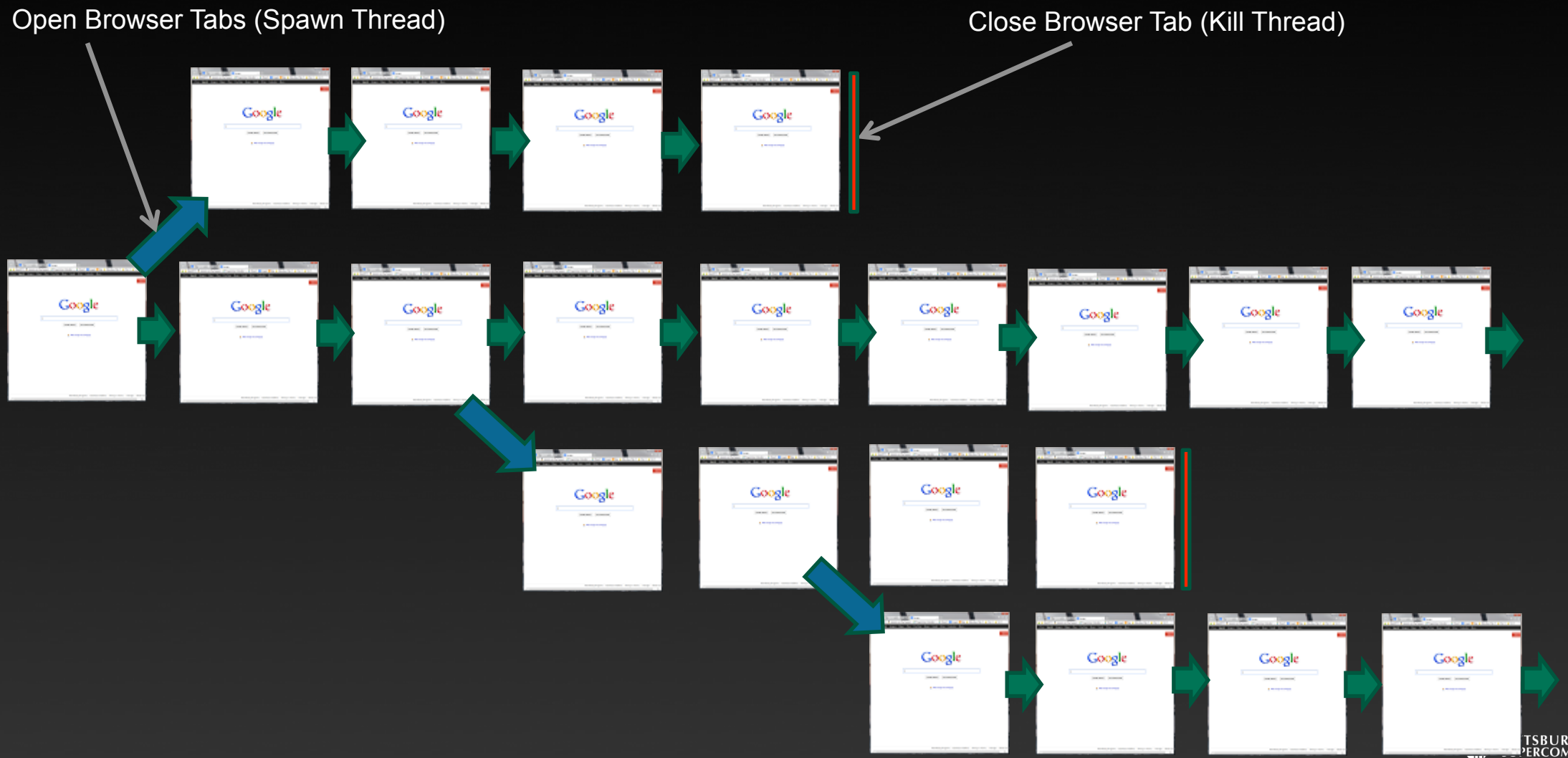


Two Threads

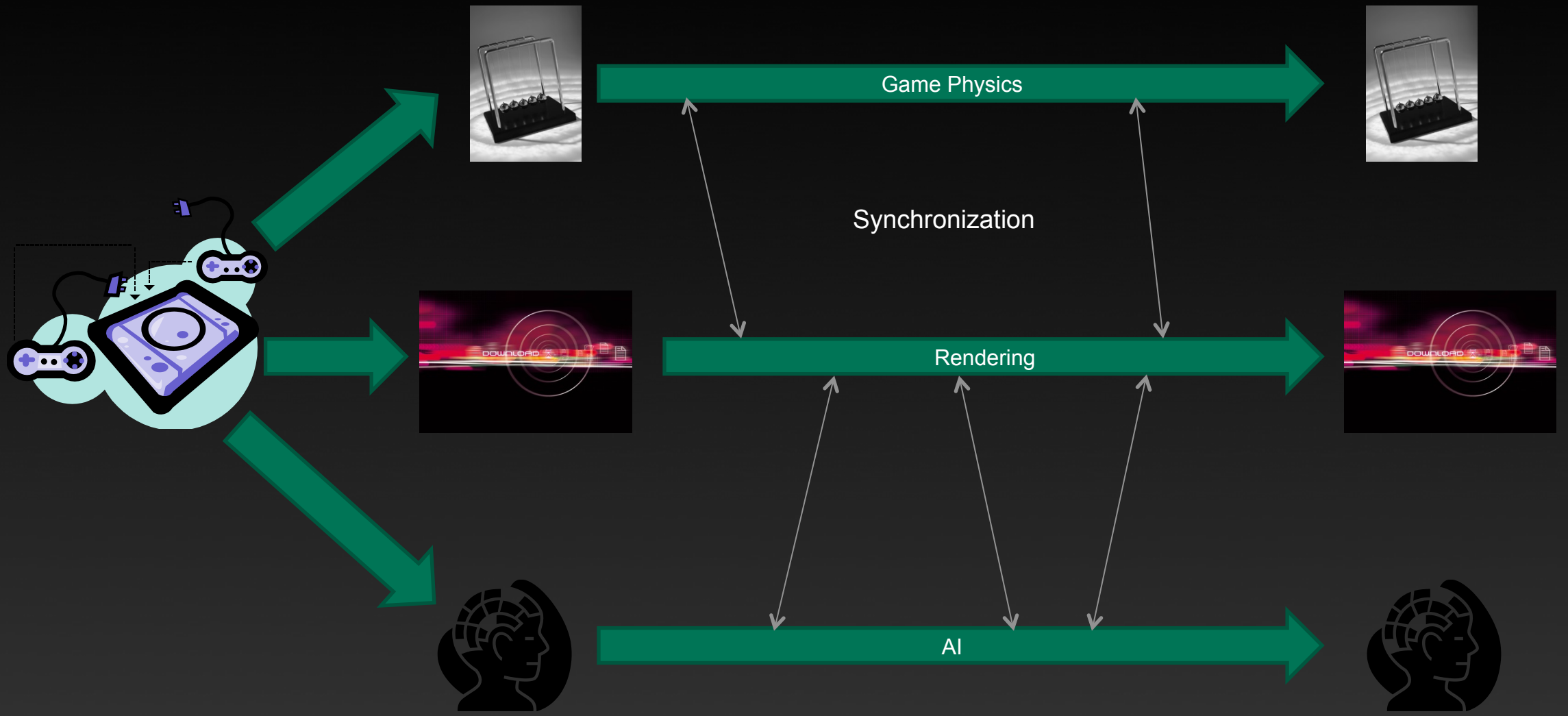
General Thread Capability



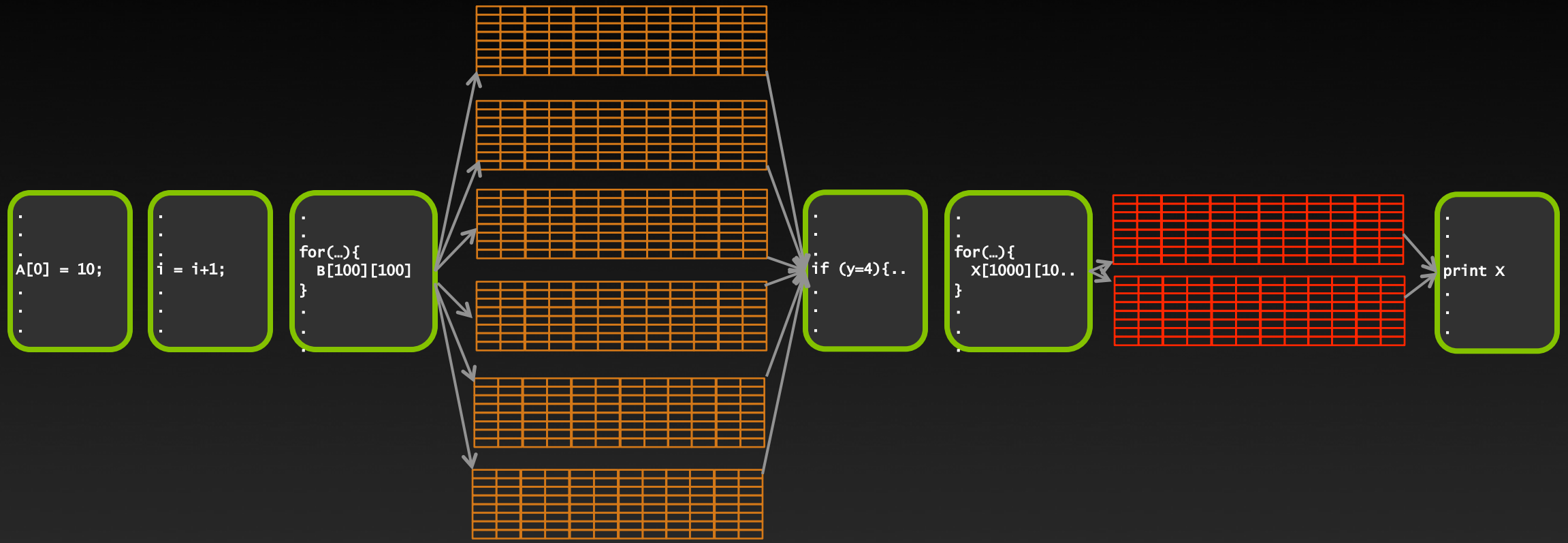
Typical Desktop Application Threading



Typical Game Threading



HPC Application Threading



HPC Use of OpenMP

- 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.

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 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:

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

When run on 10 processors, it will execute something like this…

No Data Dependency

Processor 1

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

Processor 2

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

Processor 3

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

Processor 4

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

Processor 5

```
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:

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

This is perfectly valid serial code.

Data Dependency

Now Processor 2, in trying to calculate its first iteration,

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

needs the result of Processor 1's last iteration. If we want the correct ("same as serial") result, we need to wait until processor 1 finishes. Likewise for processors 3, 4, …

Output Dependency

How about this spread out on those same 10 processors?

```
for (index=1; index<10000; index++){
     Array[index] = Array[index]+1
     X = Array[index];
}
```

There is no obvious dependence between iterations, but X may not get set to Array[999999] as it would in the serial execution. Any one of the PE's may get the "final word". Versions of this crop up and are called Output Dependencies.

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 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!

Some applied OpenMP

Now that you know the general pitfalls and the general idea of how we accelerate large loops, let's look at how we apply these to some actual code with some actual OpenMP.

How about a simple loop that does some basic math. Most scientific codes have more sophisticated versions of something like this:

```
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

```
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;
}
```

C Version

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

Fortran Version

We are requesting that this for/do loop be executed in parallel on the available processors. This might be considered the most basic OpenMP construct.

Compile and Run

We may as well follow through and see how we would compile and run this. We are using Intel compilers here. Others are very similar (-fopenmp, -omp). Likewise, if you are using a different command shell, you may do “setenv MP_NUM_THREADS 32”.

Fortran:

```
ifort -openmp paintcost.f
export OMP_NUM_THREADS=32
a.out
```

C:

```
icc -openmp paintcost.c
export OMP_NUM_THREADS=32
a.out
```

This has the effect of running with 32 threads. It is as simple as that to change our request.

Something is wrong.

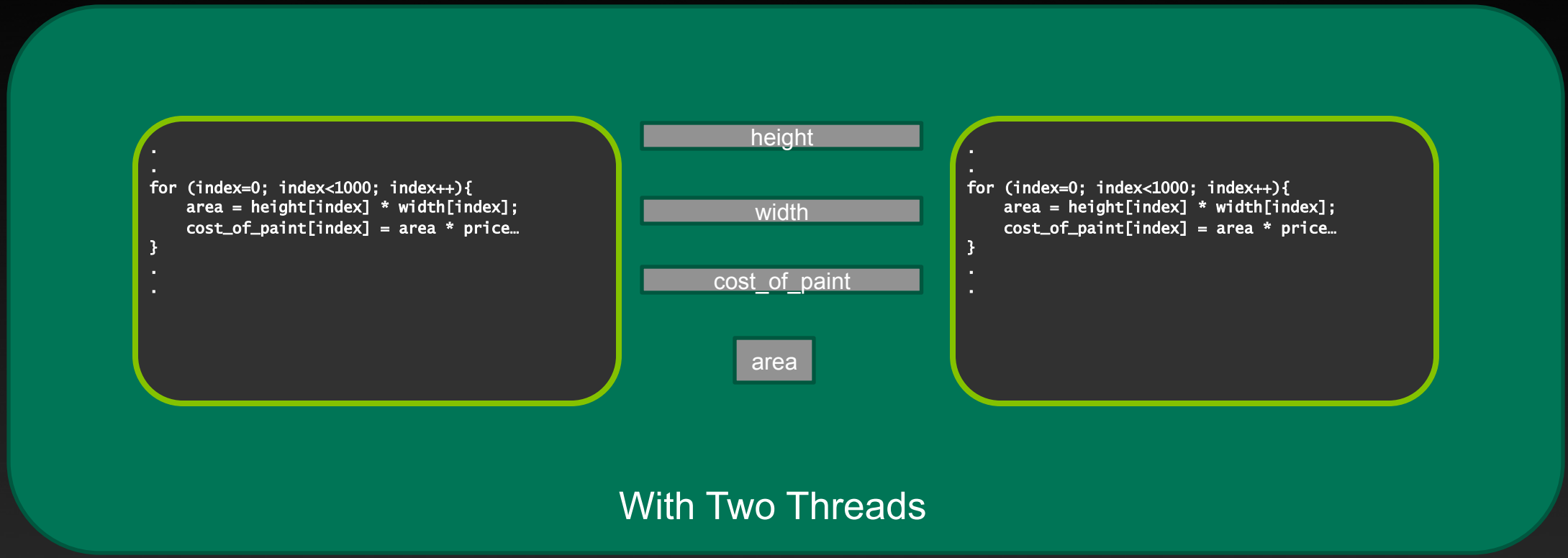
If we ran this code we would find that our results differ from the serial code (and are simply wrong). The reason is that we have a shared variable that is getting overwritten by all of the threads.

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

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

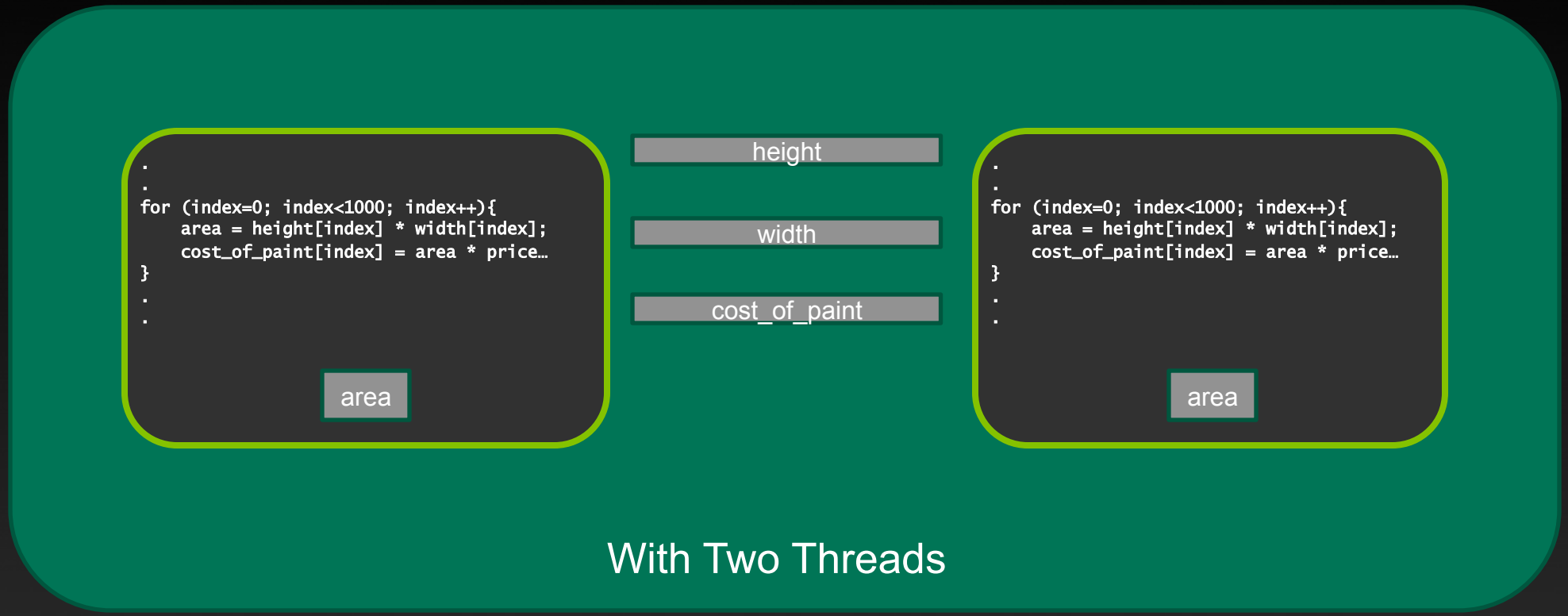
Between it's assignment and use there are (31 here) other threads accessing and changing it. This is obviously not what we want.

Shared Variables



By default variables are shared in OpenMP. Exceptions include index variables and variables declared inside parallel regions (C/C++). More later.

What We Want



We can accomplish this with the private clause.

Private Clause At Work

Apply the private clause and we have a working loop:

```
#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;
}
```

C Version

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

Fortran Version

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 only use this clause for documentation purposes, but it is an excellent idea to designate all of your variables and leave no confusion.
`default(`*`list`*`)`	You can change the default type to some of the others.
`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 (iteration or section) thread value into the master thread copy. *Otherwise, this does not happen!*

There are also similar `threadprivate` directives which allow us to manage these at the global or common block level. Convenient, but not worth repeating here.

What is automatically private?

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

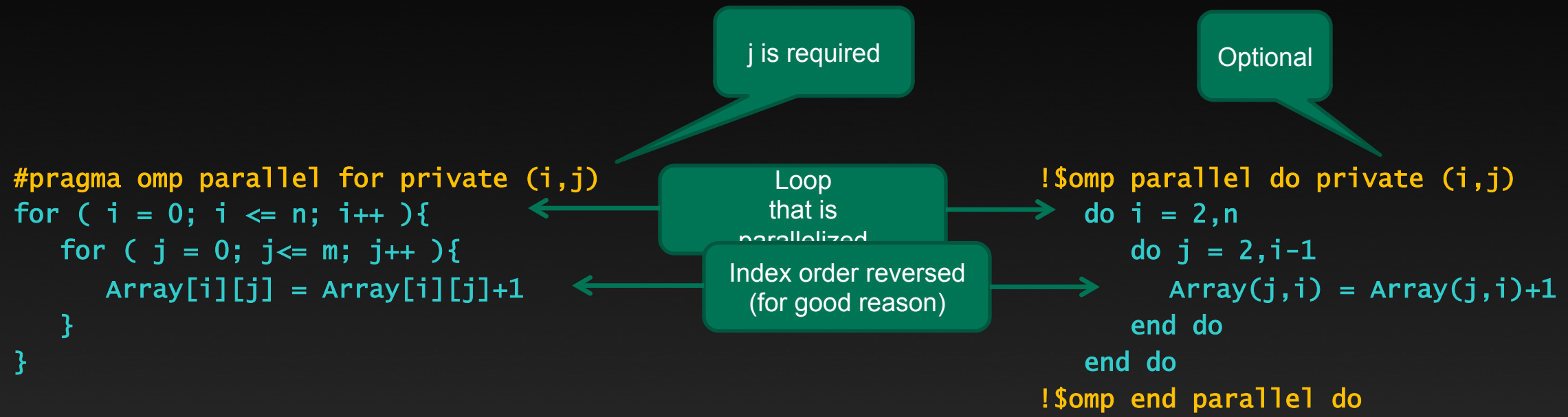
- Default is shared, except for…
- local variables in any called subroutine, unless using static (C) or save (Fortran)
- loop index variable
- inner loop index variable in Fortran, but not in C.
- variables declared within the block (for C).

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

```
#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
   }
}
```

Loop Order and Depth

The parallel for/do loop is in common and enough that we want to make sure we really understand what is going on.



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.

Prime Accelerator

Let's see what we can do with a simple program that 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
```

Prime Accelerator

The most obvious thing is to parallelize the main loop.

C Version

```
#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

```
!$omp parallel do
      do i = 2,n
         do j = 2,i-1
            if (mod(i,j) == 0) then
               not_primes = not_primes + 1
               exit
            end if
         end do
      end do
!$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 it’s own (now obsolete) update. In this case, some of the additions to not_primes may be overwritten and ignored.

Will private fix this? Private variables aren’t subject to data races, and we will end up with multiple valid not_prime subtotals. The question then becomes, how do we sum these up into the real total we are looking for?

Reductions

The answer is to use the data reduction data clause designed for just this common case.

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 (+).

Reductions

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.

Operation	Initialization
+	0
max	least number possible
min	largest number possible
-	0
Bit (&, |, ^, iand, ior)	~0, 0
Logical (&&, ||, .and., .or.)	1,0, .true., .false.

We shall return.

```
#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

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

Fortran Version

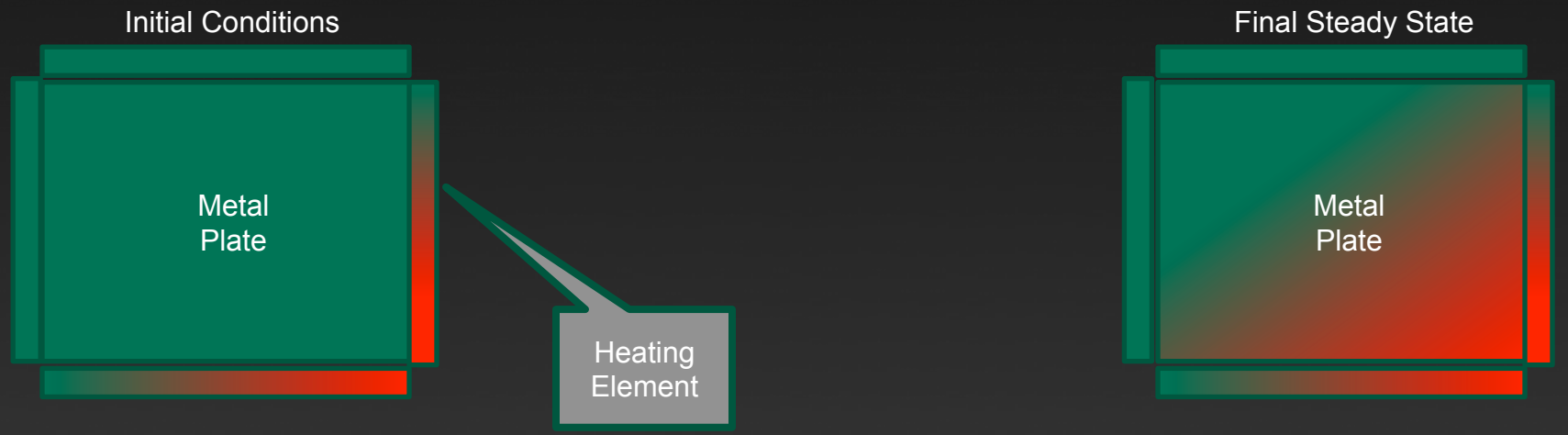
A few notes before we leave (for now):

- The OpenMP standard forbids branching out of parallel do/for loops. 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.

Our Foundation Exercise: Laplace Solver

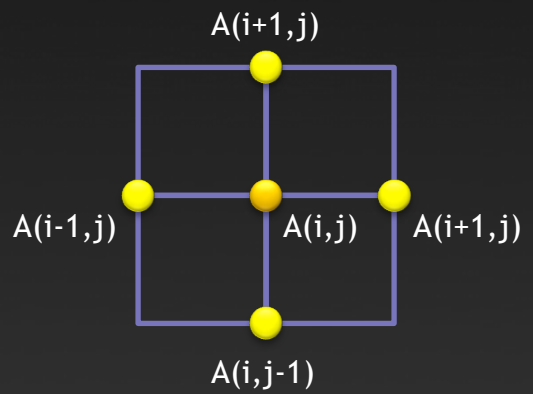
- I've been using this for MPI, OpenACC and now OpenMP. It is a great simulation problem, not rigged for OpenMP.
- In this most basic form, it solves the Laplace equation: $\nabla^2 f(x,y)=0$
- The Laplace Equation applies to many physical problems, including:
  - Electrostatics
  - Fluid Flow
  - Temperature

- 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 it's 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.



$$A{\downarrow}k{+}1\ (i,j){=}A{\downarrow}k\ (i{-}1,j){+}A{\downarrow}k\ (i{+}1,j){+}A{\downarrow}k\ (i,j{-}1){+}A{\downarrow}k\ (i,j{+}1)\ /4$$

Serial Code Implementation

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

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

Serial C Code (kernel)

```
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_old[i+1][j] + Temperature_old[i-1][j] +
                                        Temperature_old[i][j+1] + Temperature_old[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_old[i][j]), dt);
            Temperature_old[i][j] = Temperature[i][j];
        }
    }

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

    iteration++;

}
```

Done?

Calculate

Update temp array and find max change

Output

Serial C Code Subroutines

```
// initialize plate to 0
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature[i][j] = 0.0;
        }
    }
}


// print diagonal in bottom right corner action
void track_progress(int iteration) {

    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");
}
```

```
// set the values at the boundary
void set_boundary_conditions(){

    int i,j;

    // set left side to 0 and right to increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature[i][0] = 0.0;
        Temperature[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature[0][j] = 0.0;
        Temperature[ROWS+1][j] = (100.0/ROWS)*j;
    }
}
```

Whole C Code

```
#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_old[ROWS+2][COLUMNS+2];  // last iteration temperature grid

//   helper routines
void initialize();
void set_boundary_conditions();
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 [100-1000]? ");
    scanf("%d", &max_iterations);

    gettimeofday(&start_time,NULL); // Unix timer

    initialize();                   // initialize grid
    set_boundary_conditions();      // set boundary conditions

    // initialize Temperature_old
    for(i = 0; i <= ROWS+1; i++){
        for(j=0; j<=COLUMNS+1; j++){
            Temperature_old[i][j] = Temperature[i][j];
        }
    }

    // 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_old[i+1][j] + Temperature_old[i-1][j] +
                                            Temperature_old[i][j+1] + Temperature_old[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++){
                        dt = fmax( fabs(Temperature[i][j]-Temperature_old[i][j]), dt);
                        Temperature_old[i][j] = Temperature[i][j];
            }
        }

        // 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("\nMax 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 to 0
void initialize(){

  int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature[i][j] = 0.0;
        }
    }
}


// set the values at the boundary; these boundary values do not change
void set_boundary_conditions(){

  int i,j;

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature[i][0] = 0.0;
        Temperature[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature[0][j] = 0.0;
        Temperature[ROWS+1][j] = (100.0/ROWS)*j;
    }
}


// print diagonal in bottom right corner where most action is
void track_progress(int iteration) {

    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");
}
```

Serial Fortran Code (kernel)

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

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

  dt=0.0

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

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

  iteration = iteration+1

enddo
```

Done?

Calculate

Update temp array and find max change

Output

Serial Fortran Code Subroutines

```
!initialize plate to 0
subroutine initialize( temperature )
      implicit none

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

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

      do i=0, rows+1
         do j=0, columns+1
            temperature(i,j) = 0
         enddo
      enddo
end subroutine initialize


!print diagonal in bottom corner where most action is
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

      print *, '---------- Iteration number: ', iteration, ' ---------------'
      do i=5,0,-1
         write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no'), &
                    rows-i,columns-i,temperature(rows-i,columns-i)
      enddo
end subroutine track_progress
```

```
!set the values at the boundary.  These stay constant.
subroutine set_boundary_conditions( temperature )
      implicit none

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

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

      !set left side to 0 and right to linear increase
      do i=0,rows+1
         temperature(i,0) = 0.0
         temperature(i,columns+1) = (100.0/rows) * i
      enddo

      !set top to 0 and bottom to linear increase
      do j=0,columns+1
         temperature(0,j) = 0.0
         temperature(rows+1,j) = ((100.0)/columns) * j
      enddo
end subroutine set_boundary_conditions
```

Whole Fortran Code

```
program serial
      implicit none

      !Size of plate
      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      double precision, parameter    :: max_temp_error=0.01

      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_old

      print*, 'Maximum iterations [100-1000]?'
      read*,   max_iterations

      call cpu_time(start_time)      !Fortran timer

      call initialize( temperature )
      call set_boundary_conditions( temperature )

      !initialize temperature_old
      do i=0, rows+1
         do j=0, columns+1
            temperature_old(i,j) = temperature(i,j)
         enddo
      enddo

      !do until error is minimal or until maximum steps
      do while ( dt > max_temp_error .and. iteration <= max_iterations)

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

         dt=0.0

         !copy grid to old grid for next iteration and find max change
         do j=1,columns
            do i=1,rows
               dt = max( abs(temperature(i,j) - temperature_old(i,j)), dt )
               temperature_old(i,j) = temperature(i,j)
            enddo
         enddo

         !periodically print test values
         if( mod(iteration,100).eq.0 ) then
            call track_progress(temperature, iteration)
         endif

         iteration = iteration+1

      enddo

      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
```

```
!initialize plate to 0
subroutine initialize( temperature )
      implicit none

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

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

      do i=0, rows+1
         do j=0, columns+1
            temperature(i,j) = 0
         enddo
      enddo
end subroutine initialize


!set the values at the boundary.  These stay constant.
subroutine set_boundary_conditions( temperature )
      implicit none

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

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

      !set left side to 0 and right to linear increase
      do i=0,rows+1
         temperature(i,0) = 0.0
         temperature(i,columns+1) = (100.0/rows) * i
      enddo

      !set top to 0 and bottom to linear increase
      do j=0,columns+1
         temperature(0,j) = 0.0
         temperature(rows+1,j) = ((100.0)/columns) * j
      enddo
end subroutine set_boundary_conditions


!print diagonal in bottom corner where most action is
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

      print *, '---------- Iteration number: ', iteration, ' ---------------'
      do i=5,0,-1
         write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no'), &
                     rows-i,columns-i,temperature(rows-i,columns-i)
      enddo
end subroutine track_progress
```

Exercise 1: Use OpenMP to parallelize the Jacobi loops

(About 30 minutes)

1) Edit laplace_serial.c or laplace_serial.f90 (your choice) and add directives where it helps

Fortran Note: The universal 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()
```

2) Run your code on various numbers of cores and see what kind of speedup you achieve.

```
> export OMP_NUM_THREADS=8
> a.out
```

Exercise 1 C Solution

```
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_old[i+1][j] + Temperature_old[i-1][j] +
                                        Temperature_old[i][j+1] + Temperature_old[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_old[i][j]), dt);
                        Temperature_old[i][j] = Temperature[i][j];
        }
    }

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

    iteration++;
}
```

Thread this loop

And this one with a reduction

Exercise 1 Fortran Solution

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

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

    dt=0.0

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

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

    iteration = iteration+1

enddo
```

Thread this loop

And this one with a reduction

Scaling?

For the solution in the Laplace directory, we found this kind of scaling when running for 1000 iterations.

Threads	C (s)	Fortran (s)	Speedup
1	7.5	3.0	
2	3.8	1.5	2.0 / 2.0
4	1.9	0.75	3.9 / 4.0
8	0.96	0.39	7.8 / 7.7
16	0.50	0.22	15 / 14
32	1.2	0.79	6.3 / 3.8

Codes were compiled with no extra flags, and there was some minor variability.

Time for a breather.

Congratulations, you have now mastered 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

Fortran 90

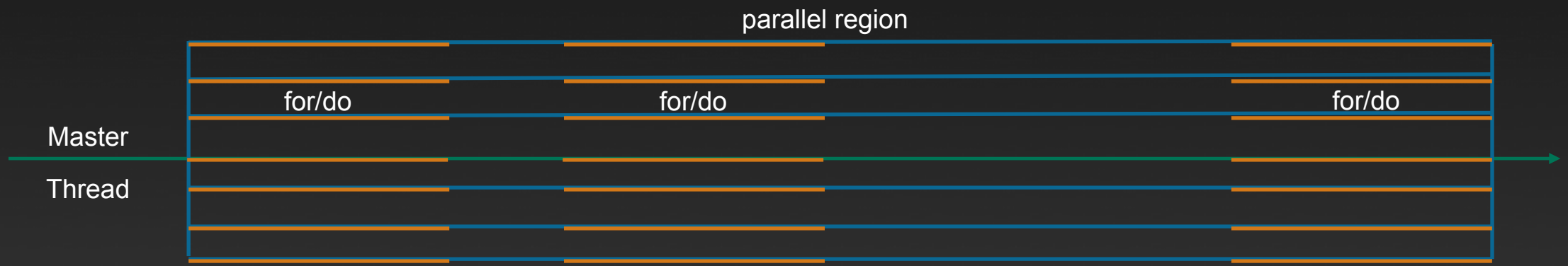
Fortran 90 has data parallel constructs that map very well to threads. You can declare a workshare region and OpenMP will do the right thing for:

- FORALL
- WHERE
- Array assignments

```
      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
```

Different Work Sharing Constructs

What we have been doing



What we could do (Less overhead?)

Parallel Region with C

```
#pragma omp parallel shared(t, t_old) private(i,j, iter) firstprivate(niter) reduction
(max:dt)
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
   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);
   }
}
```

Note that this is a simpler loop than our actual exercise. Adding max dt as a loop condition complicates this a good bit.

Parallel Region with Fortran

```
!$omp parallel shared(T, Told) private(i,j,iter)
!$omp&         firstprivate(niter) reduction(max:dt)
       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

          !$omp do
          do j=1,NC
             do i=1,NR
                dt = max( abs(t(i,j) - told(i,j)), dt )
                Told(i,j) = T(i,j)
             enddo
          enddo
         !$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 out 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

```
.
.
.
#pragma omp master
if((iter % 100) == 0) {
        print_trace(iter);
}
.
.
```

```
.
.
!$omp master
          if( mod(iter,100).eq.0 ) then
             call print_trace(t, iter)
          endif
!$omp end master
.
.
```

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 a "omp barrier" after this or threads could already be altering *t* as we are writing it out. Life in parallel regions can get tricky!

Barrier

```
.
.
.
#pragma omp master
if((iter % 100) == 0) {
       print_trace(iter);
}
#pragma omp barrier
.
.
```

```
.
.
!$omp master
         if( mod(iter,100).eq.0 ) then
            call print_trace(t, iter)
         endif
!$omp end master

!$omp barrier
.
.
```

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 and 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
.
.
```

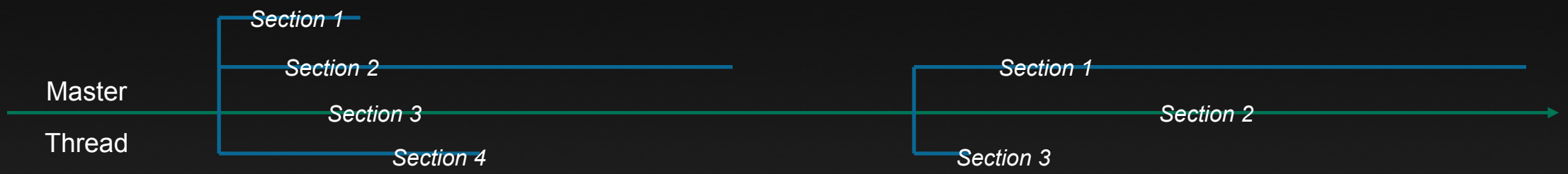
Now we are using OpenMP runtime library routines, and not directives. We would have to use ifdef if we wanted to preserve the serial version. Also, we should really include a barrier somewhere here as well.

Other Synchronization Directives

`atomic`	Eliminates data race on this one specific location.
`critical`	Only one thread at a time can go through this section.
`flush`	Even coherent cache architecture need this to eliminate possibility of register storage issues. Tricky, but important if you get tricky.
`nowait`	This clause will eliminate implied barriers on certain directives.
`ordered`	Forces serial order on loops.
`single`	Like Master, but any thread will do.

Another Work Sharing Construct

Sections



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

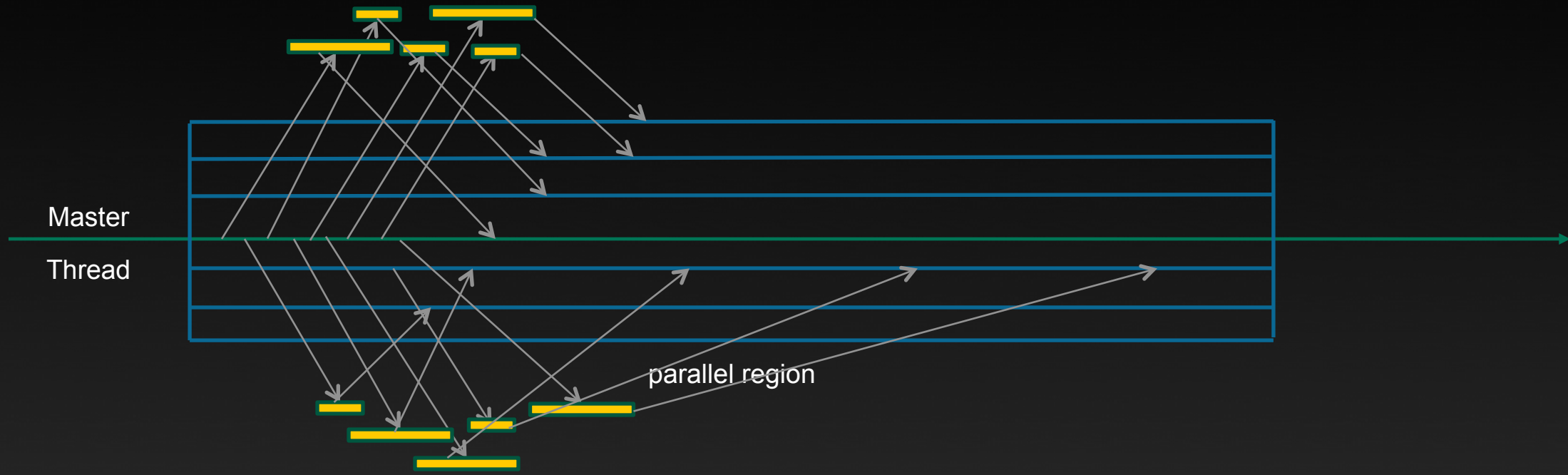
Sections

```
.
.
.
#pragma omp parallel shared(a,b,x,y) private(index)
  {

  #pragma omp sections
    {

    #pragma omp section
    for (index=0; index <n; index++)
      x[i] = a[i] + b[i];

    #pragma omp section
    for (index=0; index <n; index++)
      y[i] = a[i] * b[i];

    }

  }
.
.
```

```
.
.
!$OMP PARALLEL SHARED(A,B,X,Y), PRIVATE
(INDEX)

!$OMP SECTIONS

!$OMP SECTION
      DO INDEX = 1, N
         X(INDEX) = A(INDEX) + B(INDEX)
      ENDDO

!$OMP SECTION
      DO INDEX = 1, N
         Y(INDEX) = A(INDEX) * B(INDEX)
      ENDDO

!$OMP END SECTIONS

!$OMP END PARALLEL
.
.
```

Both for/do loops run concurrently. Still same results as serial here.

And for ultimate flexibility: Tasks



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.

Fibonacci Tasks

```
#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;
  }
}
```

Our tasks are spinning off tasks recursively! The threads will eventually pick them all off.

Task Capability

Tasks have some additional directives and clauses.

- taskwait
- taskyield
- final, untied, mergable clauses

We won't go into them here, because you only need to know they exist in case you are one of the minority HPC applications that needs this. This capability is useful for:

- Graphs
- Any kind of pointer chasing

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

Locks

```
#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 - ending locked region\n", tid);

    omp_unset_lock(&my_lock);

  }

  omp_destroy_lock(&my_lock);
}
```

Output

```
Thread 2 - in locked region
Thread 2 - in locked region
Thread 2 - in locked region
Thread 2 - in locked region
Thread 2 - in locked region
Thread 2 - ending locked region
Thread 0 - in locked region
Thread 0 - in locked region
Thread 0 - in locked region
Thread 0 - in locked region
Thread 0 - in locked region
Thread 0 - ending locked region
Thread 1 - in locked region
Thread 1 - in locked region
Thread 1 - in locked region
Thread 1 - in locked region
Thread 1 - in locked region
Thread 1 - ending locked region
Thread 3 - in locked region
Thread 3 - in locked region
Thread 3 - in locked region
Thread 3 - in locked region
Thread 3 - in locked region
Thread 3 - ending locked region
```

This could have been done with just an omp critical!

flush

If you start delving into these capabilities, you really 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 synch when you start using complex threading. You can assure these problems don't occur with:

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

Pthreads like flexibility

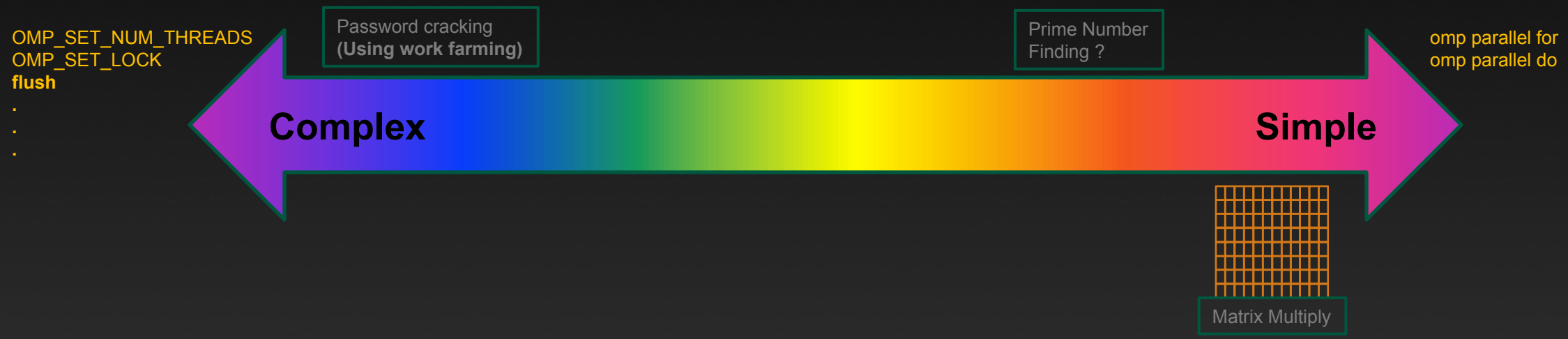
We now have the ability to start coding just about any kind of thread flow we can imagine. And, we can start creating all kinds of subtle and non-repeatable bugs. This is normally where we start the fun of cataloging all of the ways we can get into trouble:

- Race conditions
- Deadlocks
- Livelocks
- Missing flush

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

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?



The general question is, how much time are threads spending at barriers?

If you can't tell, profiling will.

Scheduling

```
#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

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

Fortran Version

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)

This one is a competitive exercise! We are going to see who can do best in two categories of improving our prime number code.

1) Speed up the prime number count just using the scheduling options you have available. No touching the serial code.

2) Speed up the prime number count by making the serial code smarter. Although our brute force method lends itself to some obvious improvements, you could also spend the next year working on this. You have 30 minutes for both.

We will use a reduce operation to find our winners. Let your TA know your best time, and they will chat it back to us. I will pick the lowest time from that. Basically a reduction(min:time)!

One Scheduling Solution

```
#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;
    }
  }
}
```

C Version

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

Fortran Version

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.

Threads	Default (s)	dynamic	Speedup
1	48	48	
2	35	24	1.5
4	20	12	1.7
8	11	6.2	1.8
16	5.8	3.1	1.9
32	4.7*	3.1	1.5

500,000 iterations.

** Note an interesting little effect with 32 threads on the default case. We basically forced a smaller chunk and more dynamic scheduling for a small win. We don't get that when we already have dynamic scheduling.*

Information Overload?

We have now covered everything up to (but definitely not including) OpenMP 4.0. I hope you still recall how much we accomplished with just a parallel for/do. Lets 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