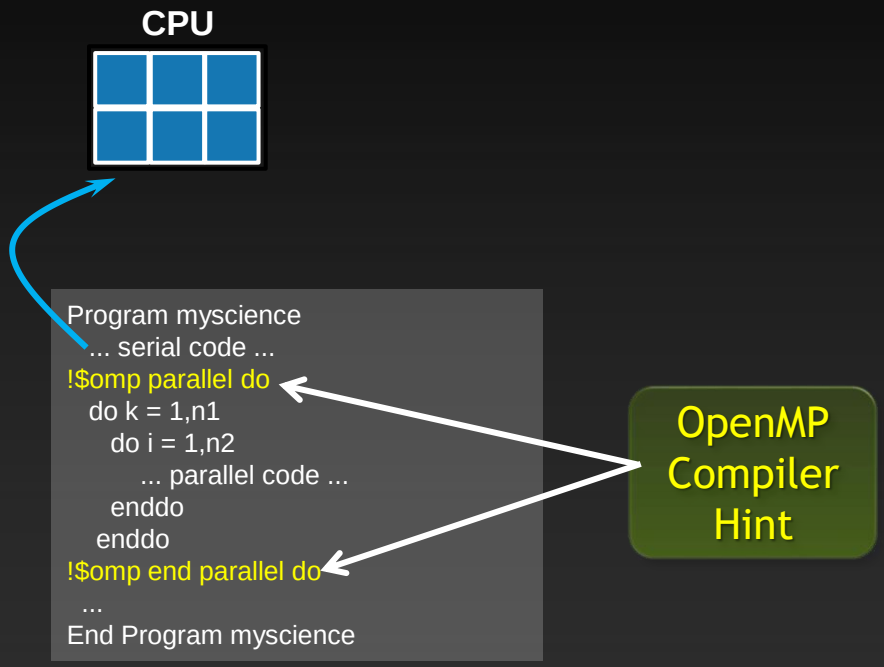
# Introduction to OpenMP

John Urbanic
Parallel Computing Scientist
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.

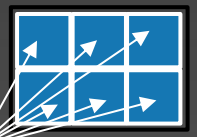
Compiler generates parallel threaded code.

Ignorant compiler just sees some comments.

# Directives: an awesome idea whose time has arrived.

OpenMP

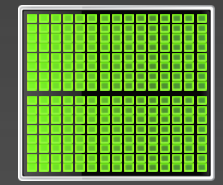
CPU



```
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 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
- NVIDIA
- Clang/Flang/LLVM
- AMD
- ARM
- MS Visual Studio*

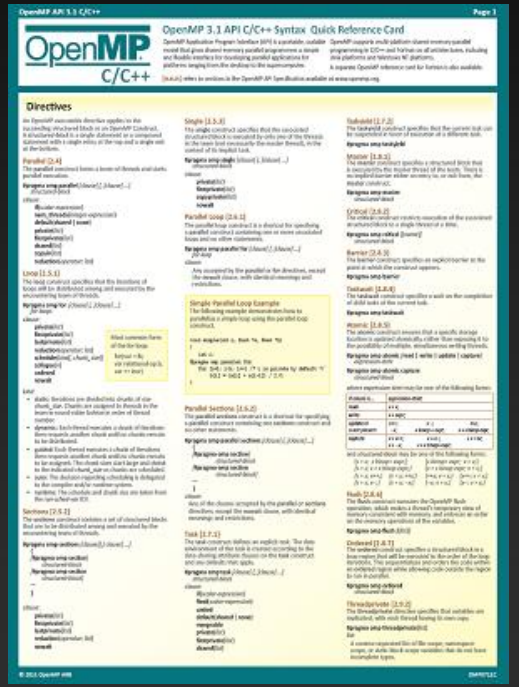
*MS is missing some useful pieces.

# 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
- 2021 OpenMP 5.2
- 2024 OpenMP 6.0



# 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

```
!$omp parallel [clause …]
    structured block
!$omp 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.

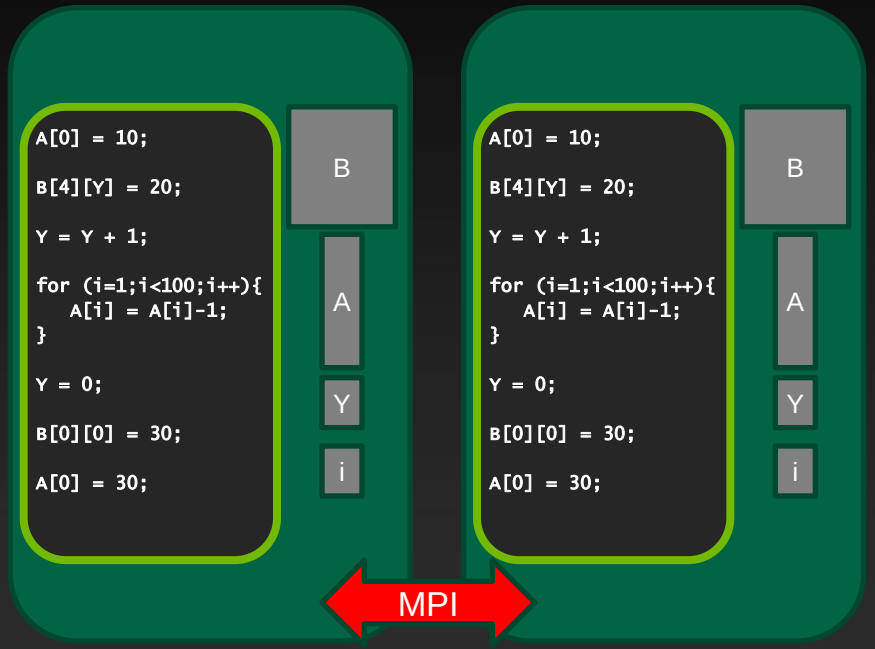
# Pthreads

```
#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);
}
```

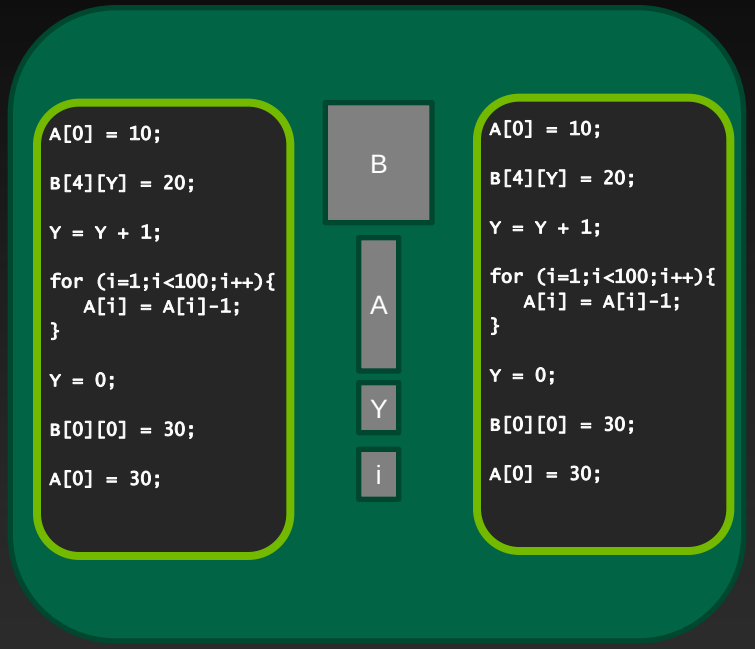
# Big Difference!

- 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

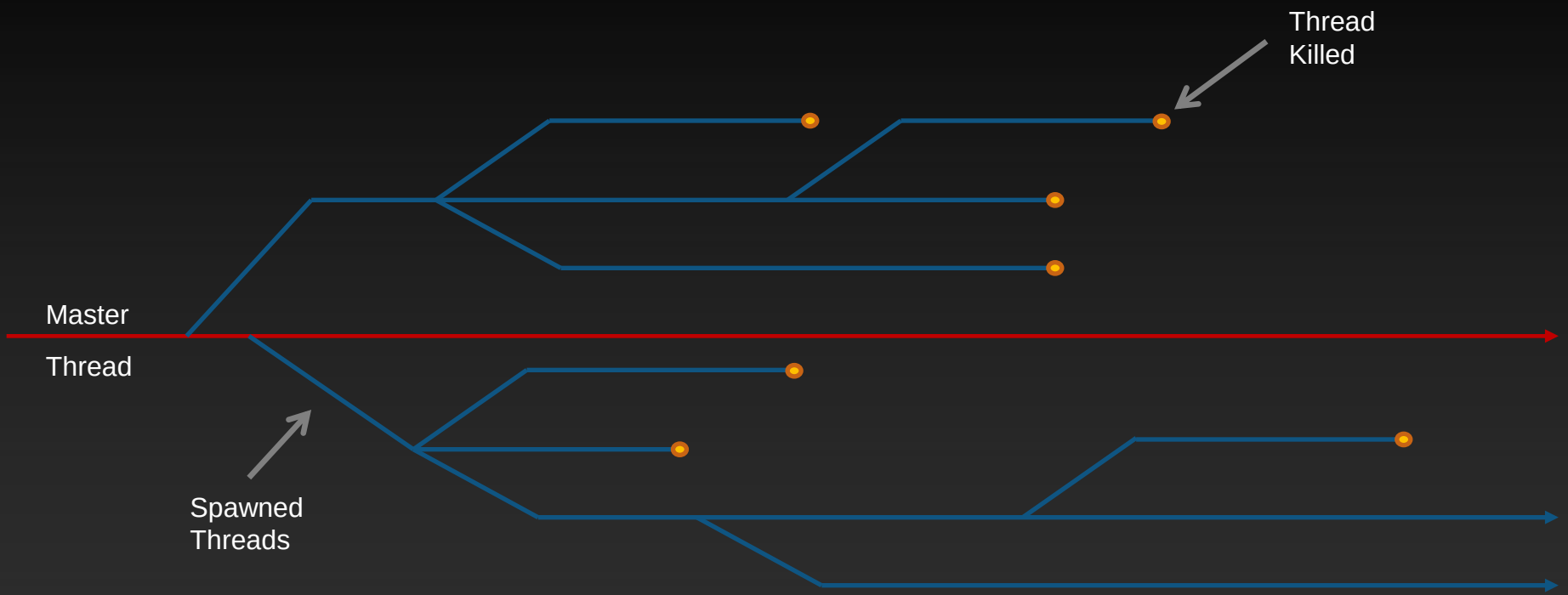



Two Processes




Two Threads

# General Thread Capability

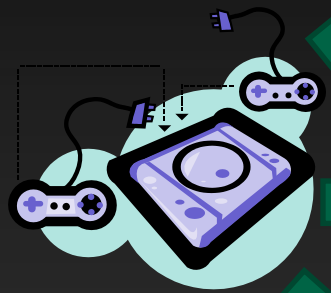


# Typical Desktop Application Threading

# Typical Game Threading

Game Physics

Synchronization

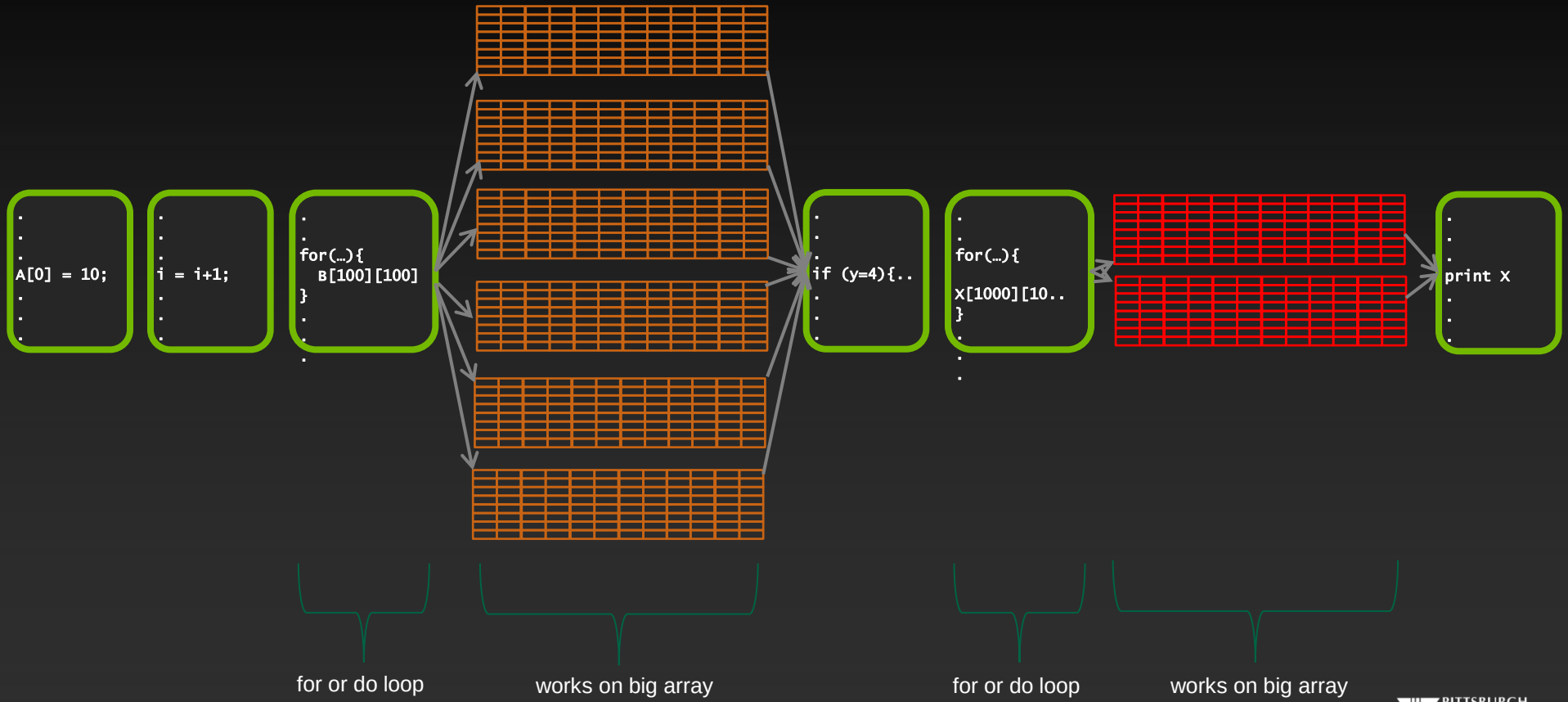


Rendering

AI

PITTSBURGH SUPERCOMPUTING CENTER

# 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.
- 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:

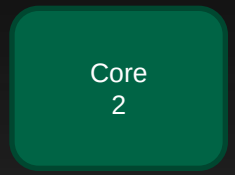
```
for(index=0; index<10000; index++)
      Array[index] = 4 * Array[index];
```

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:

```
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,

```
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 depend… dices.
  - Is the var…
  - Any non-… variables.
  - You may …
- Can these be …
  - Sometim… mmon bag of tricks de… rized compute…
  - We will n… disappear.
  - Sometim… other than rewrite …

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

For example, one possible fix here could be to:

1) Do the multiply
2) Shift the array by 1
3) Do the subtraction

(2) is non-trivial as we have to make a complete copy of the array (probably want to have one pre-allocated). But, this can also be done in parallel.

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;
  }

}
```

Standard c

```fortran
  program simple

      integer    array(1000000)

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

  end program
```

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:**

Activate OpenMP directives

```
nvfortran -mp simple.f90
export OMP_NUM_THREADS=8
a.out
```

**C:**

Run with 8 threads

```
nvc -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.

```
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.

# Something is wrong.

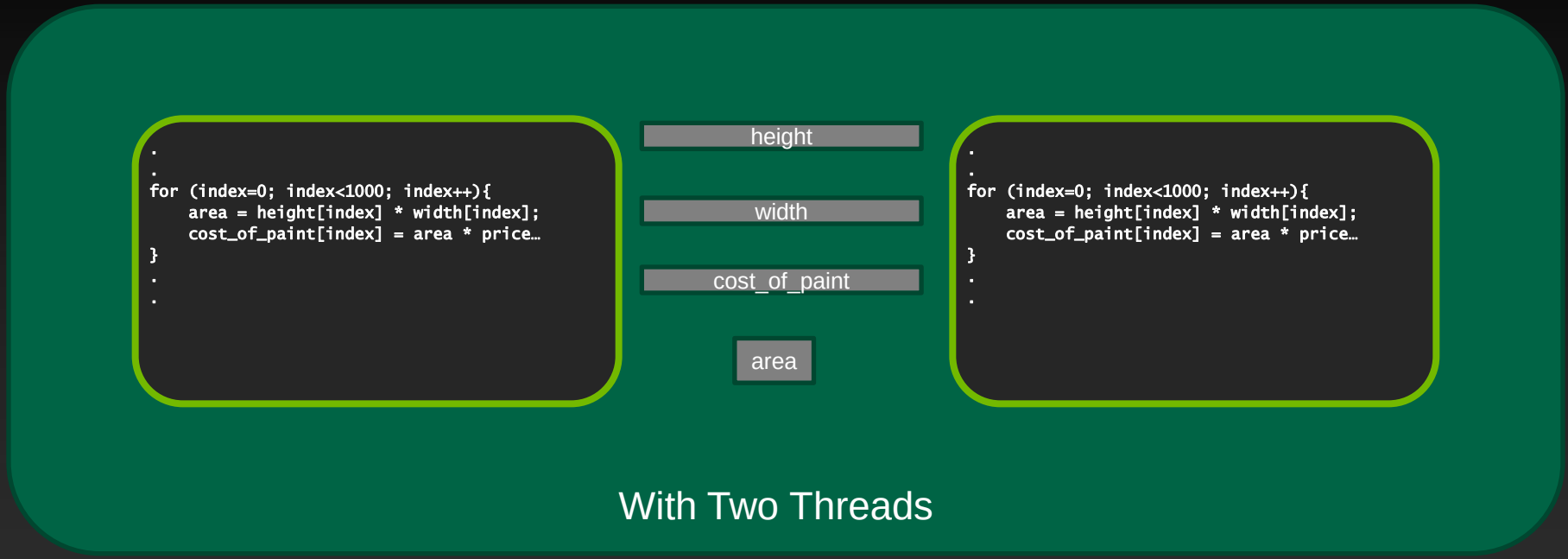
If we ran this code we would find that sometimes our results differ from the serial code (and are simply wrong). The reason is that we have a shared variable that is getting overwritten by all of the threads.

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

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

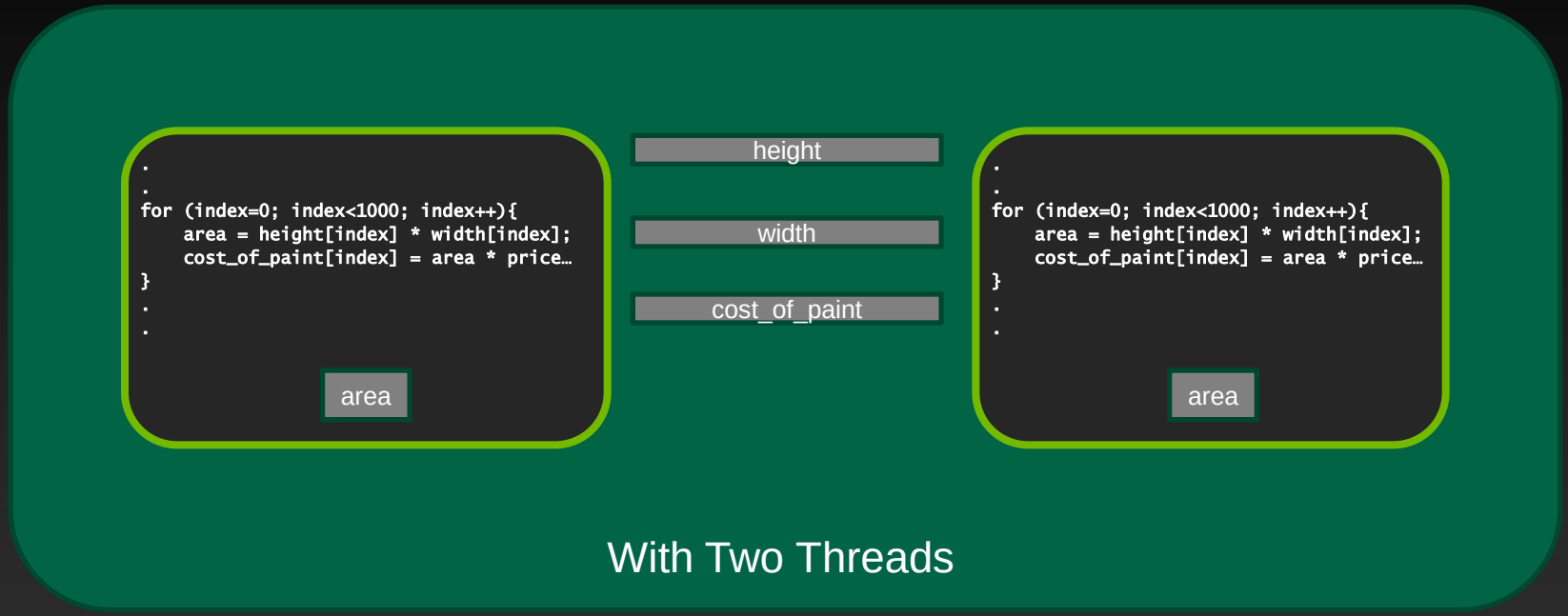
Between its assignment and use by any one thread, there are other threads (7 here) potentially accessing and changing it. This is prone to error. *Possibly the worst kind: the intermittent one.*

# 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:

```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;
}
```

C Version

```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
```

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 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*:

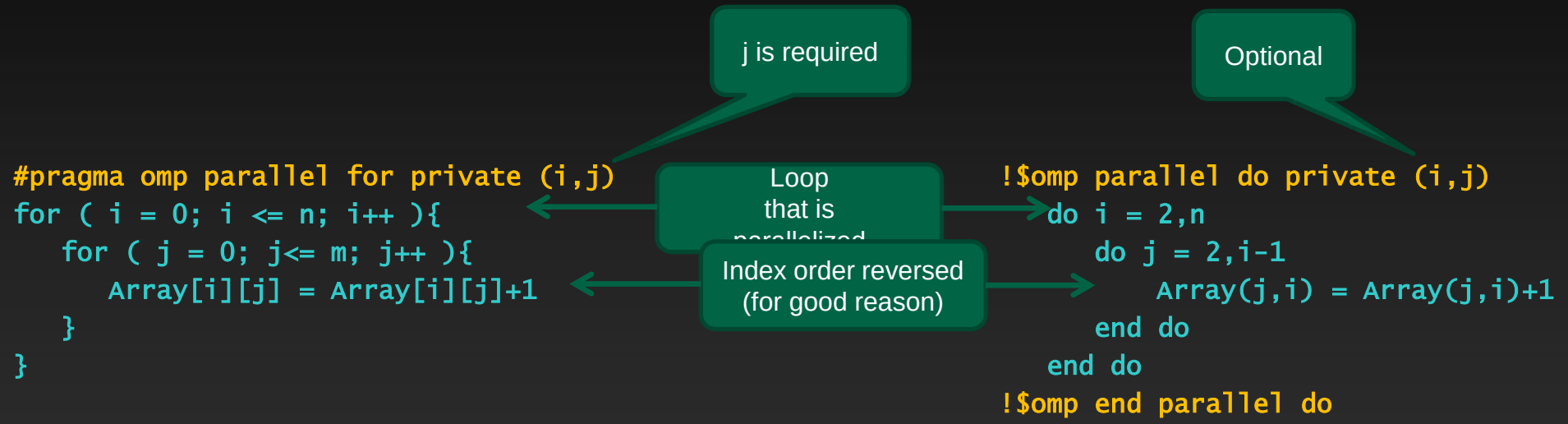
- outer loop index variable
- inner loop index variables in Fortran, but not in C.
- local variables in any called subroutine, unless using static (C) or save (Fortran)
- variables declared within the block (for C).

This last makes the C99 loop syntax quite convenient for nested loops:

```
#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 common 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 Counter

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

```
#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 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

Reductions are private variables that must be reduced to a single value eventually.

C Version

Line Continuation

```
#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

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

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

# 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

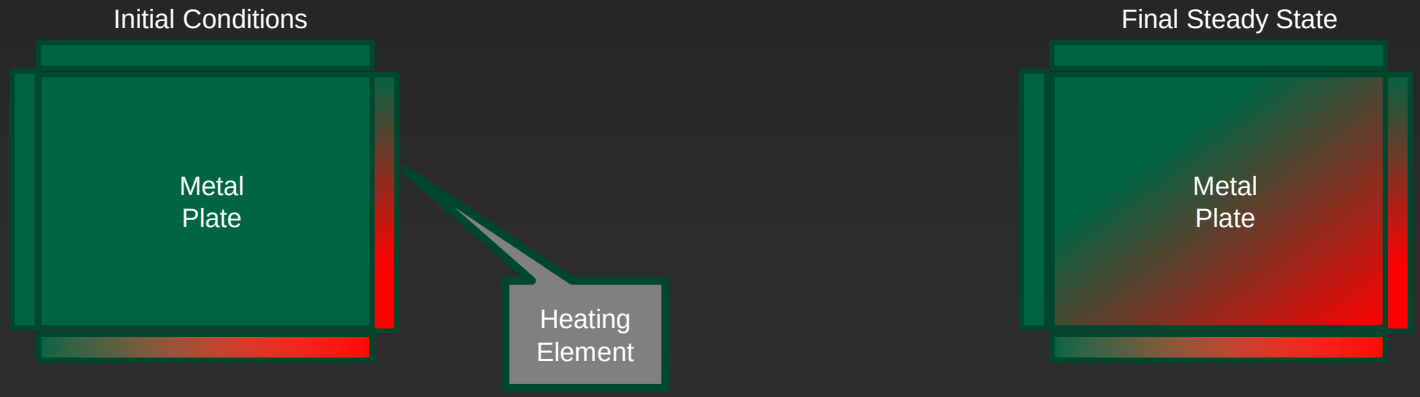
A few notes before we leave (for now):

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

# Our Exercise: Laplace Solver

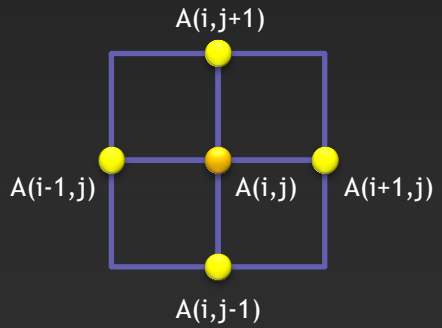
- We also use this for MPI and OpenACC. It is a great simulation problem, not rigged for OpenMP.
- In this most basic form, it solves the Laplace equation: $\nabla^2 f(x, y) = 0$
- The Laplace Equation applies to many physical problems, including:
  - Electrostatics
  - Fluid Flow
  - Temperature

- 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.




$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

# Serial Code Implementation

```
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
```

# 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_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++;

}
```

Done?

Calculate

Update temp array and find max change

Output

# Serial C Code Subroutines

```
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;
    }
}
```

BCs could run from 0 to ROWS+1 or from 1 to ROWS. We chose the former.

```
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");
}
```

# 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_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++){
              dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
              Temperature_last[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 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 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_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

enddo
```

Done?

Calculate

Update temp array and find max change

Output

# Serial Fortran Code Subroutines

```
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

      temperature_last = 0.0

      !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

      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
      print *
```

# 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_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 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_last(i+1,j)+temperature_last(i-1,j)+ &
                                      temperature_last(i,j+1)+temperature_last(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_last(i,j)), dt )
               temperature_last(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 and boundery conditions
! temp_last is used to to start first iteration
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

      temperature_last = 0.0

      !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

!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
      print *
end subroutine track_progress
```

# Exercise 1: Use OpenMP to parallelize the Jacobi loops

(About 45 minutes)

1) Log onto a node requesting all the 32 cores.

```
> interact -n 32
```

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.

```
> nvc -mp laplace_omp.c   or   nvfortran -mp laplace_omp.f90
> 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_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++;
}
```

Thread this loop

Also this one, with a reduction

# Exercise 1 Fortran Solution

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

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

Thread this loop

Also here, plus a reduction

# Scaling?

For the solution in the Laplace directory, we found this kind of scaling when running to convergence at 3372 iterations. This is on a clean 128 core node.

| Threads | C (s) | Fortran (s) | Speedup |
|---|---|---|---|
| Serial | 21.4 | 20.6 | |
| 2 | 10.8 | 10.3 | 2.0 |
| 4 | 5.4 | 5.2 | 4.0 |
| 8 | 2.7 | 2.6 | 7.9 |
| 16 | 1.4 | 1.4 | 14.7 |
| 32 | 0.80 | 0.80 | 25.7 |
| 64 | 0.59 | 0.59 | 34.9 |

The larger version of this problem that we use for the hybrid programming example (10K x 10K) continues to scale nicely on Bridges EM large memory nodes to 96 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