

# Introduction to OpenMP

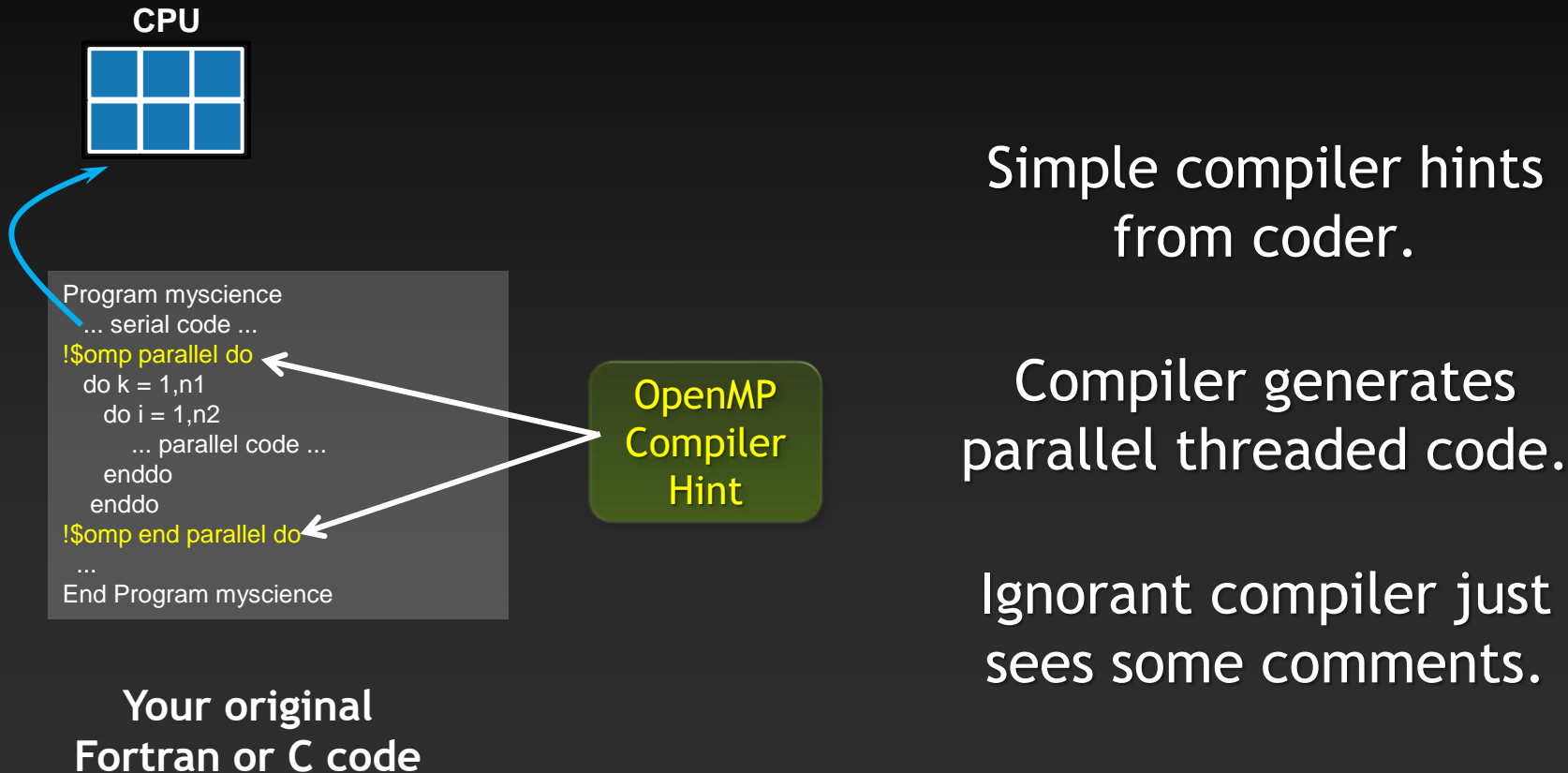
John Urbanic

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.*

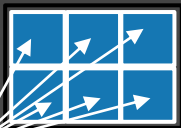
# Directives



# Directives: an awesome idea whose time has arrived.

## OpenMP

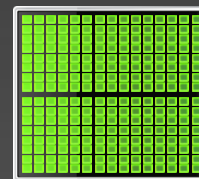
### CPU



```
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
- PGI
- Cray
- MS Visual Studio\*

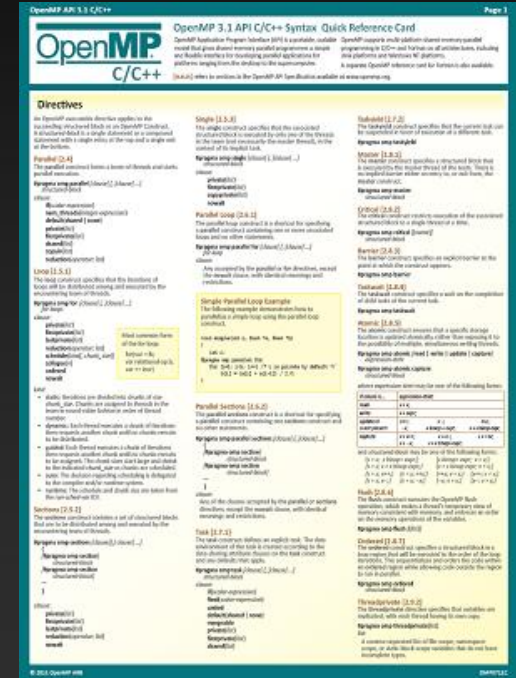
\*MS is missing some useful pieces.

# 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



# 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

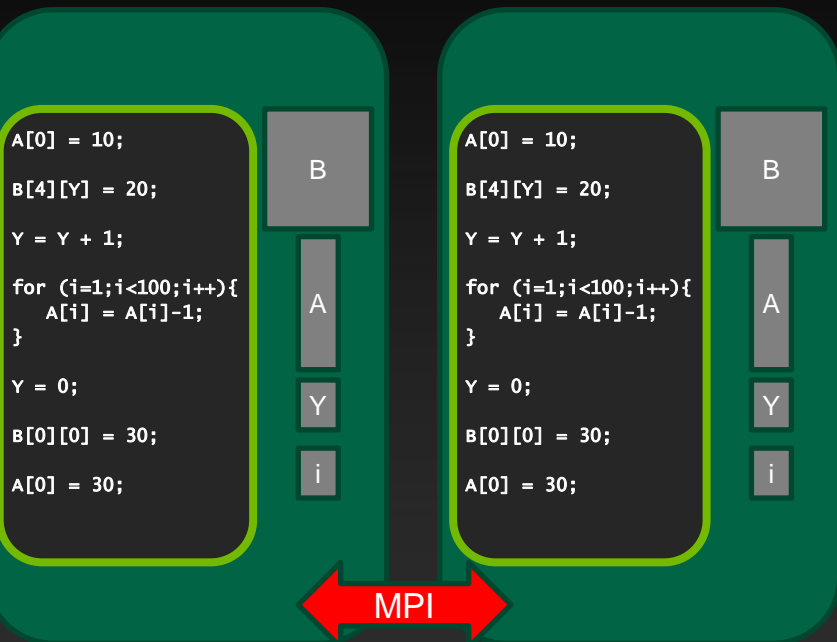
void *PrintHello(void *threadid)
{
    printf("Hello World.\n");
    pthread_exit(NULL);
}

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);
}
```

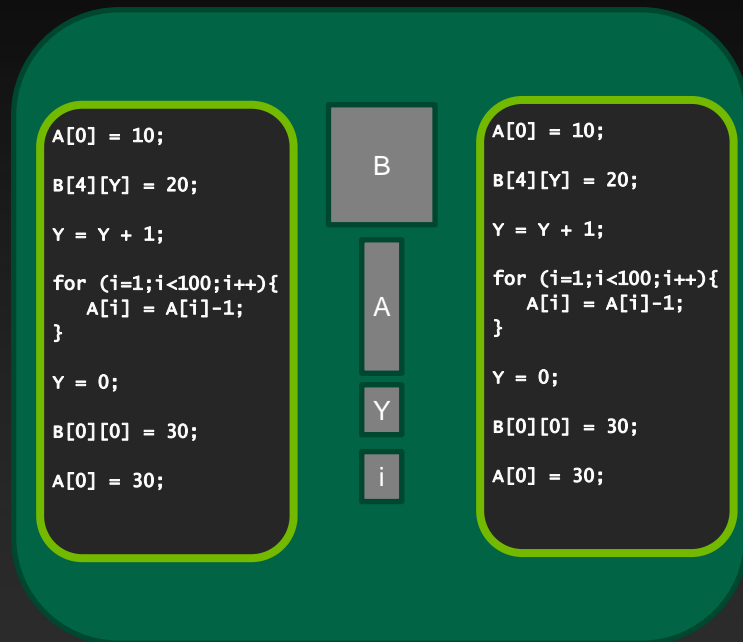
# Big Difference!

- 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

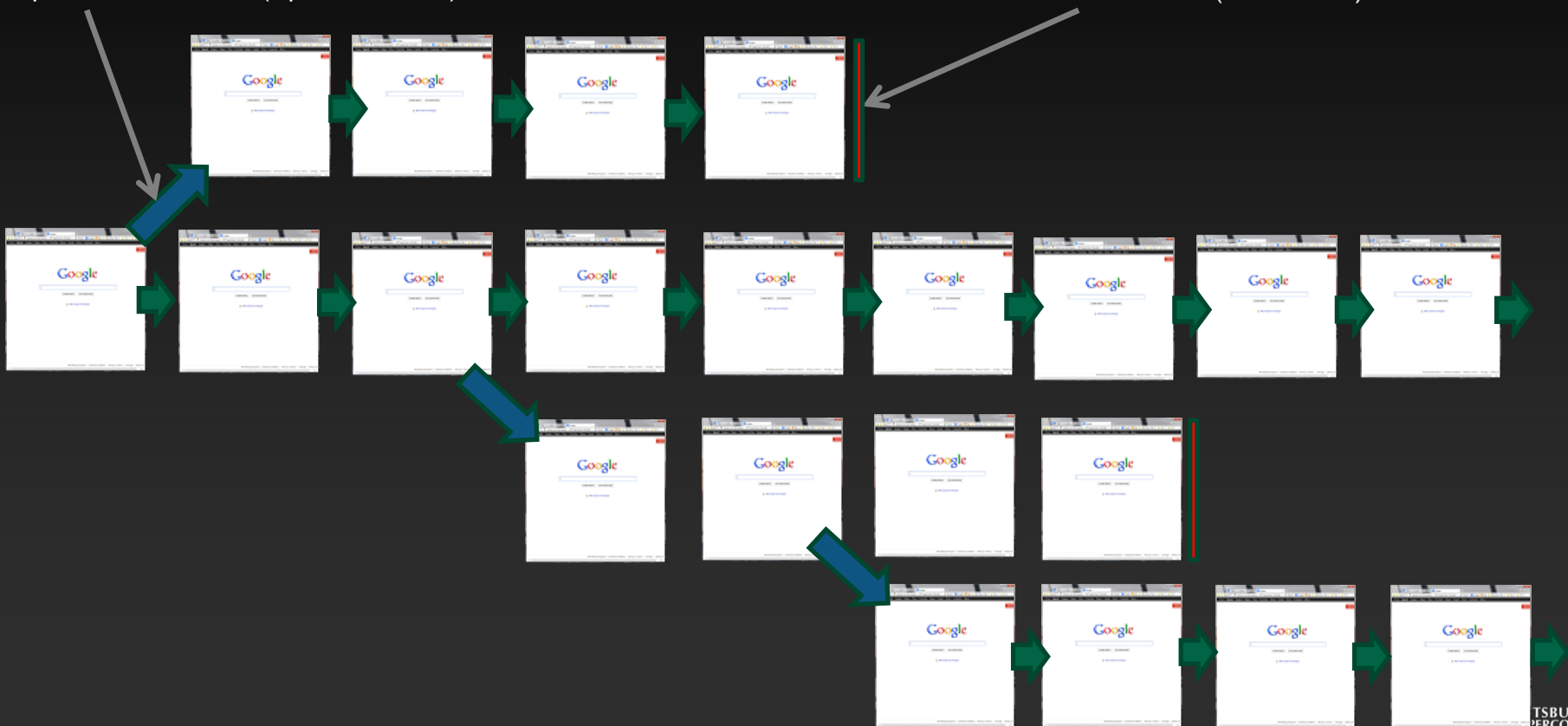
# General Thread Capability



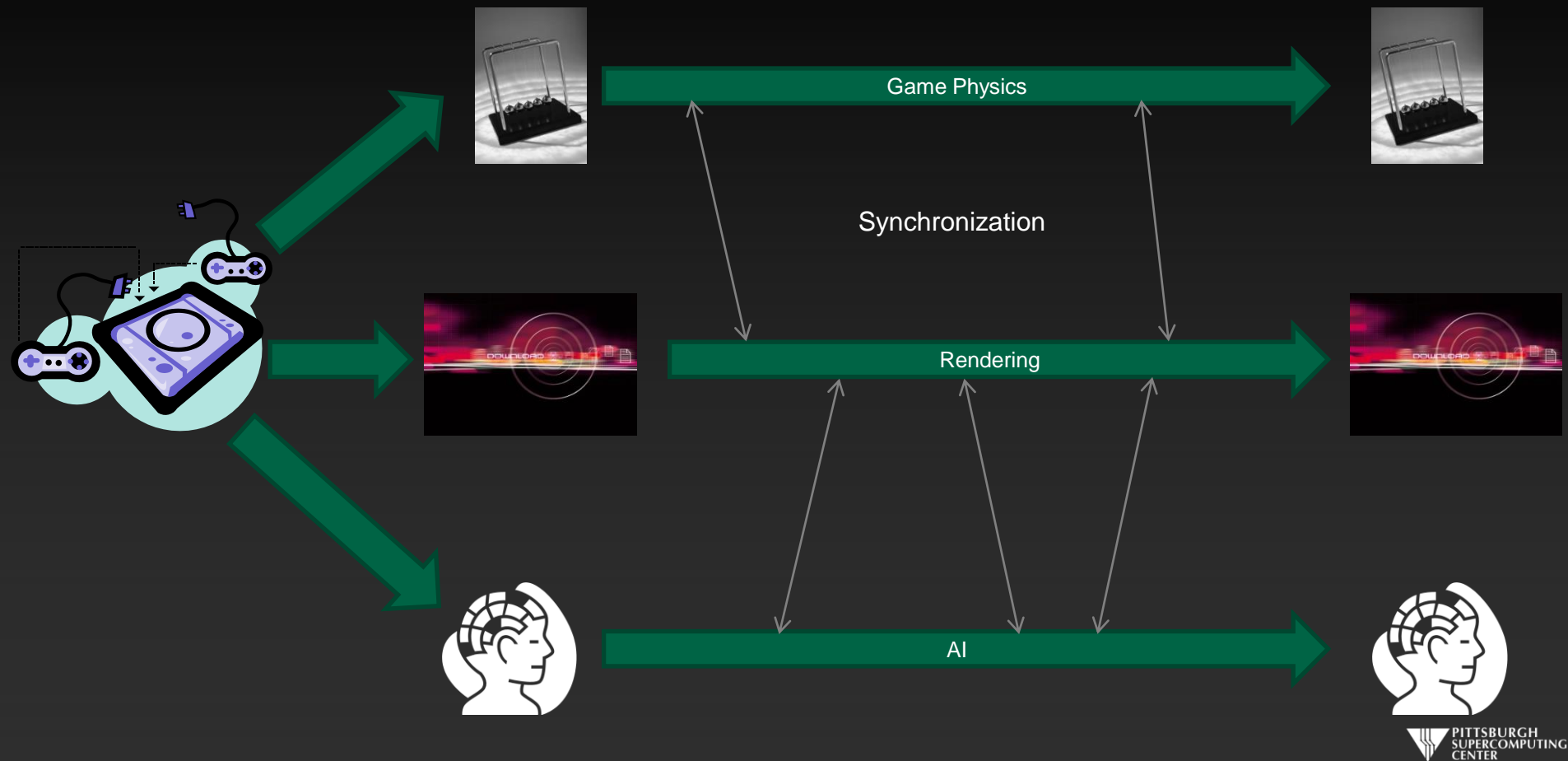
# Typical Desktop Application Threading

Open Browser Tabs (Spawn Thread)

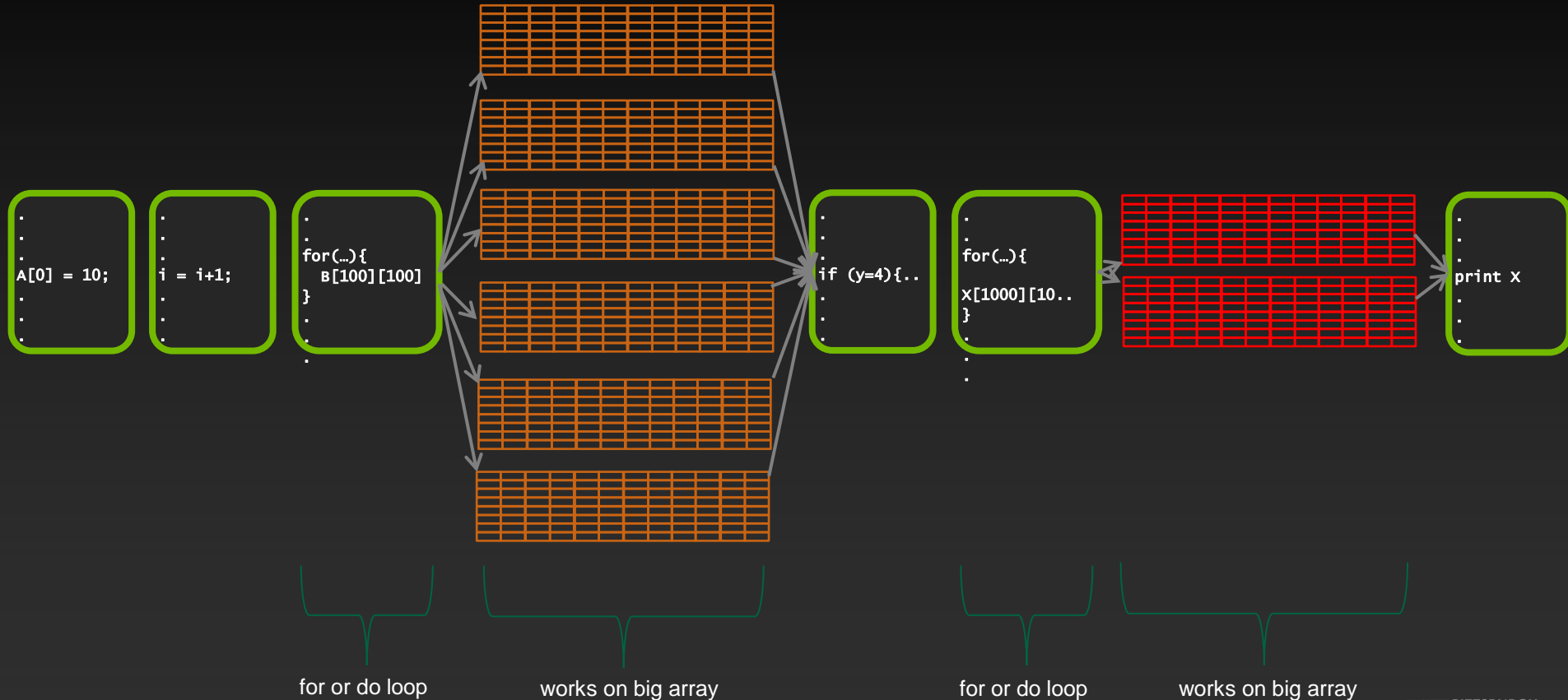
Close Browser Tab (Kill Thread)



# 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 “encyclopedia” 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

Core  
0

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

Core  
1

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

Core  
2

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

Core  
3

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

Core  
4

```
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 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:

```
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

```
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

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

C:

Run with 8  
threads

```
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.

```
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:

```
#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

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

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

height

width

cost\_of\_paint

area

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

With Two Threads

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

# What We Want

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

area

height

width

cost\_of\_paint

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

area

With Two Threads

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 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*:

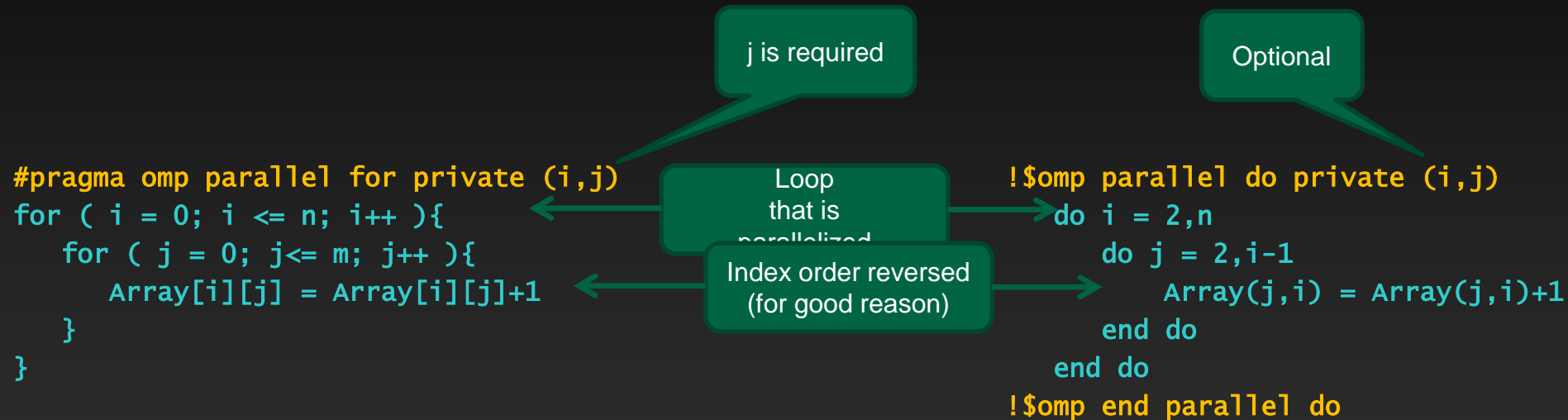
- outer loop index variable
- inner loop index variables in Fortran, but not in C.
- local variables in any called subroutine, unless using **static** (C) or **save** (Fortran)
- variables declared within the block (for C).

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

```
# 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

```
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 scalar.

# Reductions

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

## C Version

```
#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;  
        }  
    }  
}
```

Line  
Continuation

## 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

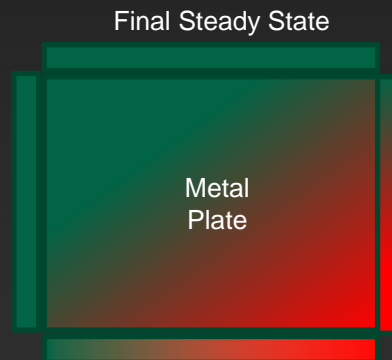
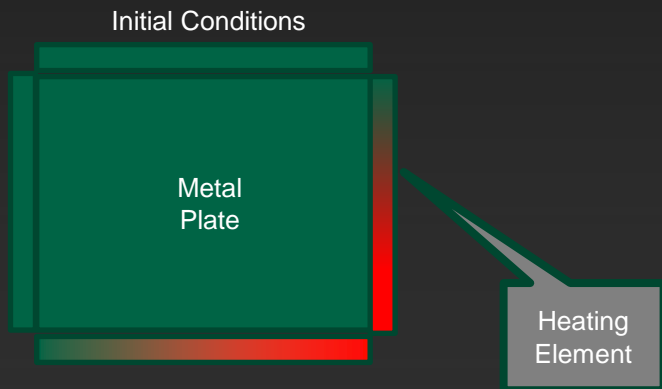
A few notes before we leave (for now):

- 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](https://primes.utm.edu/nthprime/index.php#piofx) Note that we count 1 as prime. They do not.



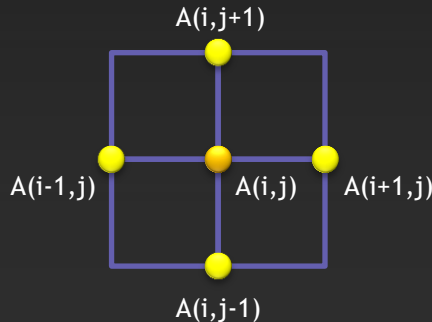
# Our Foundation Exercise: Laplace Solver

- 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.



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

# 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;  
    }  
}
```

```
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");  
}
```

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

## 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 boundary 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	0.72	0.72	28.6

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

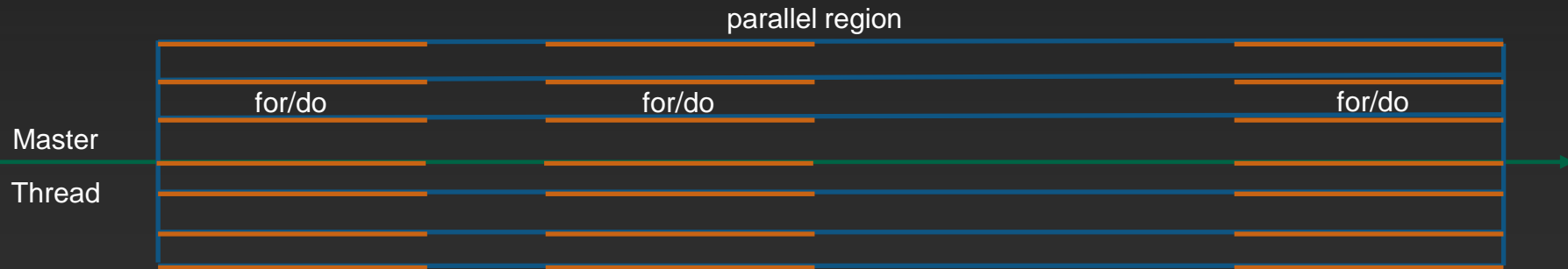
and know how to deal with them using:

- Private variables
- Reductions

# Different Work Sharing Constructs



What we have been doing



What we could do (*less overhead, no idle cores, finer control, more flexible algorithms*)

# The parallel Construct

This sets the stage for most of the more advanced or flexible directives we are going to use. It tells the system to grab the specified number of threads and set them loose.

```
#pragma omp parallel [clause, clause, ... ]  
    structured-block
```

The clauses are

```
if([ parallel :] scalar-expression)  
num_threads(integer-expression)  
default(data-sharing-attribute)  
private(list)  
firstprivate(list)  
shared(list)  
copyin(list)  
reduction([reduction-modifier ,] reduction-identifier : list)  
proc_bind(affinity-policy) One of primary, close, spread  
allocate([allocator :] list)
```



# Multiple ways of specifying threads.

In order of precedence:

<code>if</code> clause	Logical value determines if this region is parallel or serial.
<code>num_threads</code> clause	Set this to specify how many threads in this region.
<code>omp_set_num_threads()</code>	A library API to set the threads.
<code>OMP_NUM_THREADS</code>	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

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)
```

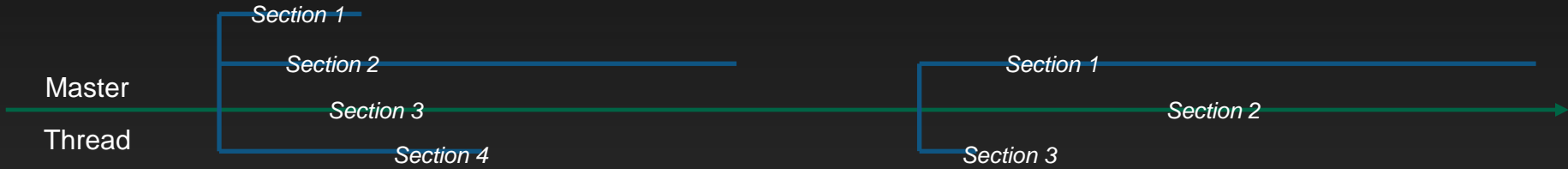
```
!$OMP END WORKSHARE
```

```
!$OMP END PARALLEL
```

```
END
```

# 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

```
.
.
.
#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

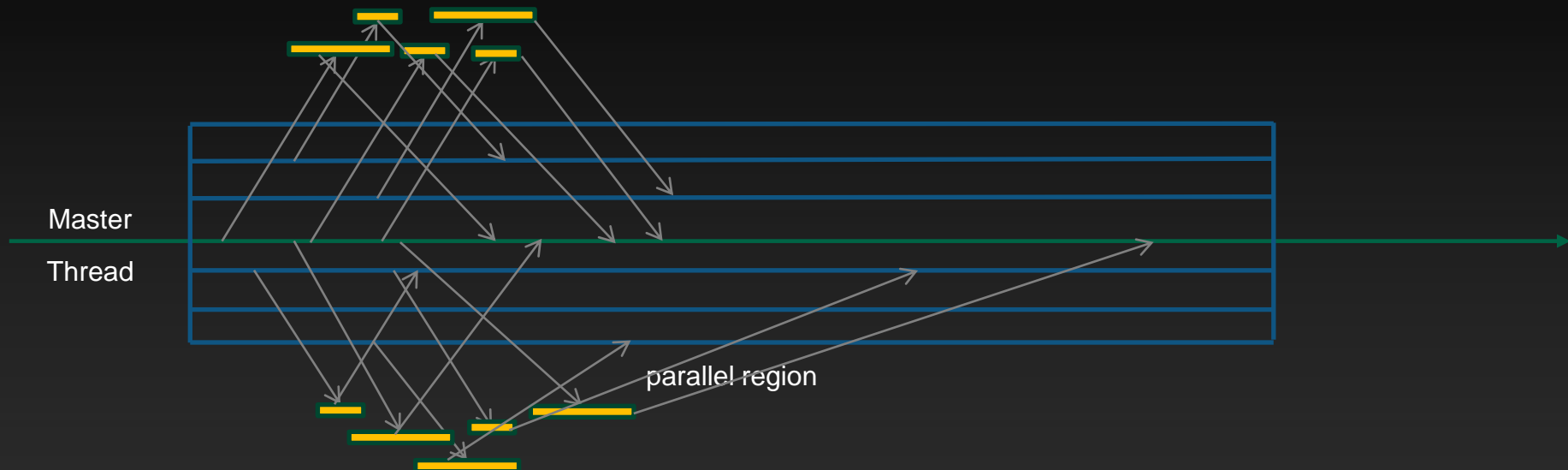
!$OMP END SECTIONS

!$OMP END PARALLEL

.
.
```

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

# And for ultimate flexibility: Tasks



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.

# Summing An Array

Let's take the simple task of summing an array.

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

Serial Code

```
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.

```
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.

```
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;  
}
```



# Optimized Recursively Summing An Array With Tasks

```
float array_sum(float *a, int length) {  
    float total;  
  
    #pragma omp parallel  
    #pragma omp single nowait  
    total = parallel_sum(a, n);  
  
    return total;  
}  
  
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);  
}
```

```
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;  
}
```

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

# 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;
    }
}
```

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.

# New Task Dependencies

```
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);

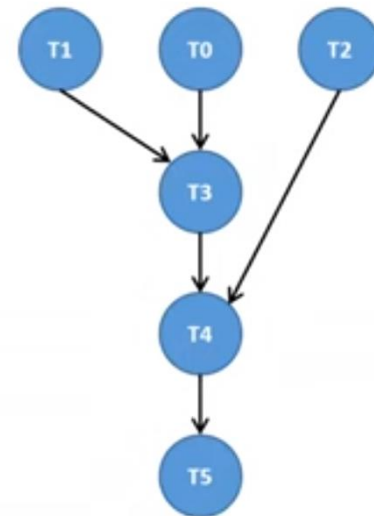
    #pragma omp task depend(in: x) depend(inout: res) //T3
    res += x;

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

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

Use the dependencies to describe what is happening to the data, not to force some execution order.

The execution order will depend up upon the actual order of the source code, with the dependencies limiting when tasks may be executed.



# New Task Dependencies

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

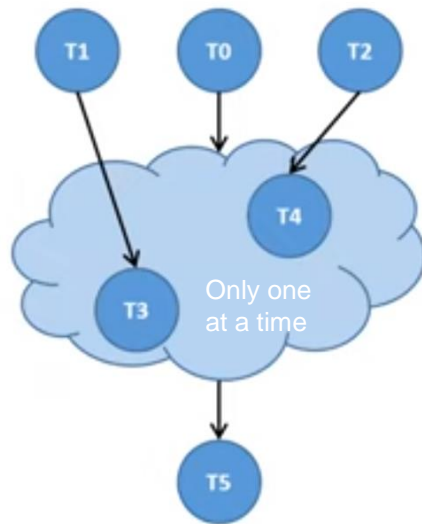
    #pragma omp task depend(out: x) //T1
    long_computation(x);

    #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, so a list of items may include array sections.

```
#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);
}
```

Here **n** is evaluated at runtime, and is the equivalent of creating n different in dependency clauses ( *depend (in: array[0], array[1], array[2],...)* ).

# Tasks Are Very Powerful

If you really embrace this task paradigm, there is now even a *taskloop* directive that allows you to decompose for/do loops into tasks in a very controlled manner. We won't go into it here.

However before we leave these elegant heights and descend into some much grittier low-level detail, I want to emphasize that this task approach provides a powerful, and robust (as in, not error prone) framework that would have been a dream for any pthreads programmer of yesteryear. You are getting all the scheduling that they have to do at no cost.

Now, let's go back to our original parallel for/do loops and see what happens if we want to manage them at a low level ourselves...

# Parallel Region Loops with 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);
    }
}
```

This is a simpler loop than our actual exercise two's condition while loop.

Working example in slide notes below is not that complicated, but we will skip it for the nonce.



# 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

```
.  
.   
.   
#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 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!

# 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 **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.

# Other Synchronization Directives & Clauses

<code>single</code>	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.
<code>critical</code>	Only one thread at a time can go unnamed (only one thread in all
<code>atomic</code>	Eliminates data race on this one : efficient than <code>critical</code> .
<code>ordered</code>	Forces serial order on loops.
<code>nowait</code>	This clause will eliminate implied barriers on certain directives.
<code>flush</code>	Even cache coherent architectures need this to eliminate possibility of register storage issues. Tricky, but important <u>iff</u> you get tricky. We will return to this.

## Hints

These two directives now have `hint` clauses. We will wait discuss those with locks in a few slides. But they have great potential to allow your code to automagically avoid unnecessary waits to enter these regions.

# Run-time Library Routines

OMP\_SET\_NUM\_THREADS  
OMP\_GET\_NUM\_THREADS  
OMP\_GET\_MAX\_THREADS  
OMP\_GET\_THREAD\_NUM  
OMP\_GET\_THREAD\_LIMIT  
OMP\_GET\_NUM\_PROCS  
OMP\_IN\_PARALLEL  
OMP\_SET\_DYNAMIC  
OMP\_GET\_DYNAMIC  
OMP\_SET\_NESTED  
OMP\_GET\_NESTED  
OMP\_SET\_SCHEDULE  
OMP\_GET\_SCHEDULE  
OMP\_SET\_MAX\_ACTIVE\_LEVELS  
OMP\_GET\_MAX\_ACTIVE\_LEVELS  
OMP\_GET\_LEVEL  
OMP\_GET\_ANCESTOR\_THREAD\_NUM  
OMP\_GET\_TEAM\_SIZE  
OMP\_GET\_ACTIVE\_LEVEL  
OMP\_IN\_FINAL  
OMP\_INIT\_LOCK  
OMP\_DESTROY\_LOCK  
OMP\_SET\_LOCK  
OMP\_UNSET\_LOCK  
OMP\_TEST\_LOCK  
OMP\_INIT\_NEST\_LOCK  
OMP\_DESTROY\_NEST\_LOCK  
OMP\_SET\_NEST\_LOCK  
OMP\_UNSET\_NEST\_LOCK  
OMP\_TEST\_NEST\_LOCK  
....

Sets the number of threads that will be used in the next parallel region

Returns the number of threads that are currently in the team executing the parallel region from which it is called

Returns the maximum value that can be returned by a call to the OMP\_GET\_NUM\_THREADS function

Returns the thread number of the thread, within the team, making this call.

Returns the maximum number of OpenMP threads available to a program

Returns the number of processors that are available to the program

Used to determine if the section of code which is executing is parallel or not

Enables or disables dynamic scheduling

Used to determine if dynamic scheduling is enabled

Used to enable or disable nested scheduling

Used to determine if nested scheduling is enabled

Sets the loop scheduling policy

Returns the loop scheduling policy

Sets the maximum number of active levels

Returns the maximum number of active levels

Returns the current level of nesting

Returns, for a given nested region, the number of threads in the team

Returns, for a given nested region, the number of processors

Returns the number of nested regions

Returns true if the routine is executing in a parallel region

Initializes a lock associated with the lock variable

Disassociates the given lock variable from any locks

Acquires ownership of a lock

Releases a lock

Attempts to set a lock, but does not block if the lock is unavailable

Initializes a nested lock associated with the lock variable

Disassociates the given nested lock variable from any locks

Acquires ownership of a nested lock

Releases a nested lock

Attempts to set a nested lock, but does not block if the lock is unavailable

Don't be intimidated.

These are either the equivalent of directives, or complementary.

They can easily be mixed and matched with directives.

regions

ive

# 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 - 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 - 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 - ending locked region
```

*This could have been done with just an omp critical!*

# Pthreads like flexibility, and pitfalls.

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

Thread A	Thread B
Lock(USB Drive)	Lock(File)
Lock(File)	Lock(USB Drive)
Copy(File)	Copy(File)
Unlock(File)	Unlock(USB Drive)
Unlock(USB Drive)	Unlock(File)

Deadlock

For most applications you are more likely to have multiple data structures that are updated by multiple threads. You will need to protect them with locks and critical regions. Picture a hash map with all threads allowed to insert/delete/lookup.

# But more advanced than pthreads

Pthreads were standardized well before modern issues like thread affinity and transactional memory become important (we'll discuss those next).

The solution for the pthreads approach is a bunch of non-standard extensions and a lot of very ugly boiler-plate code.

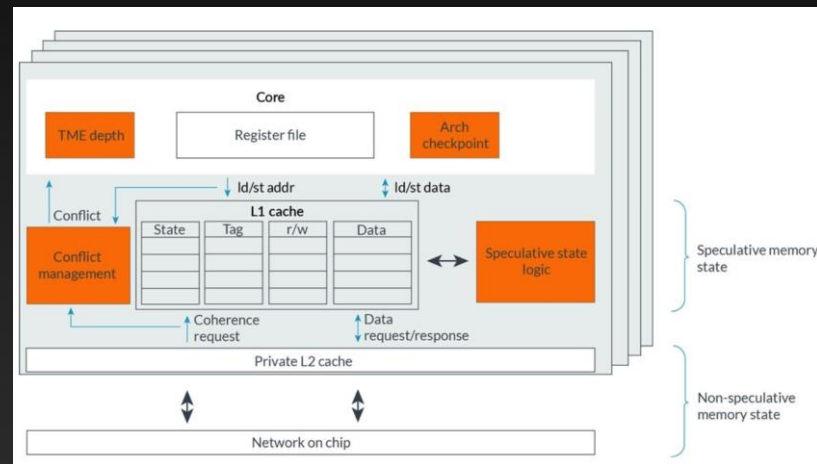
We are about to see how powerful OpenMP is. We are going to get the some very powerful capabilities with:

- Not much effort
- No performance overhead
- Portability

# Transactional Memory

As multi-core threading became dominant, the hardware vendors saw the need to help increase the efficiency of access to contended data structures. The answer we find on modern processors is transactional memory.

Transactional memory is hardware support to capture the full state of the memory access code and data, such that it can be done speculatively and rolled back if there is a conflict. If contention is low, this allows the thread to behave as though it is lock-free.



**Arm Transactional Memory Implementation**  
From their latest online guide.

This is tricky stuff. It is one of the things that bit Intel with security problems, and AMD and Arm took a long time to deploy it themselves.

# Hints

OpenMP gives us an easy way to let our *atomic* or *critical* regions, and our *omp\_init\_lock\_with\_hint* and *omp\_init\_nest\_lock\_with\_hint* to use this underlying hardware to our benefit. Just add one of the following hint clauses (or parameter to the lock).

- *omp\_sync\_hint\_uncontended*: low contention is expected in this operation, that is, few threads are expected to perform the operation simultaneously in a manner that requires synchronization.
- *omp\_sync\_hint\_contended*: high contention is expected in this operation, that is, many threads are expected to perform the operation simultaneously in a manner that requires synchronization.
- *omp\_sync\_hint\_speculative*: the programmer suggests that the operation should be implemented using speculative techniques such as transactional memory.
- *omp\_sync\_hint\_nonspeculative*: the programmer suggests that the operation should not be implemented using speculative techniques such as transactional memory.

\* *Nested locks are locks that can be set multiple times, and keep a count.*

# Affinity

Memory affinity has been a non-portable pain for decades. It has steadily grown to be a very important performance consideration. Thanks to OpenMP, there is finally a portable way to deal with it.

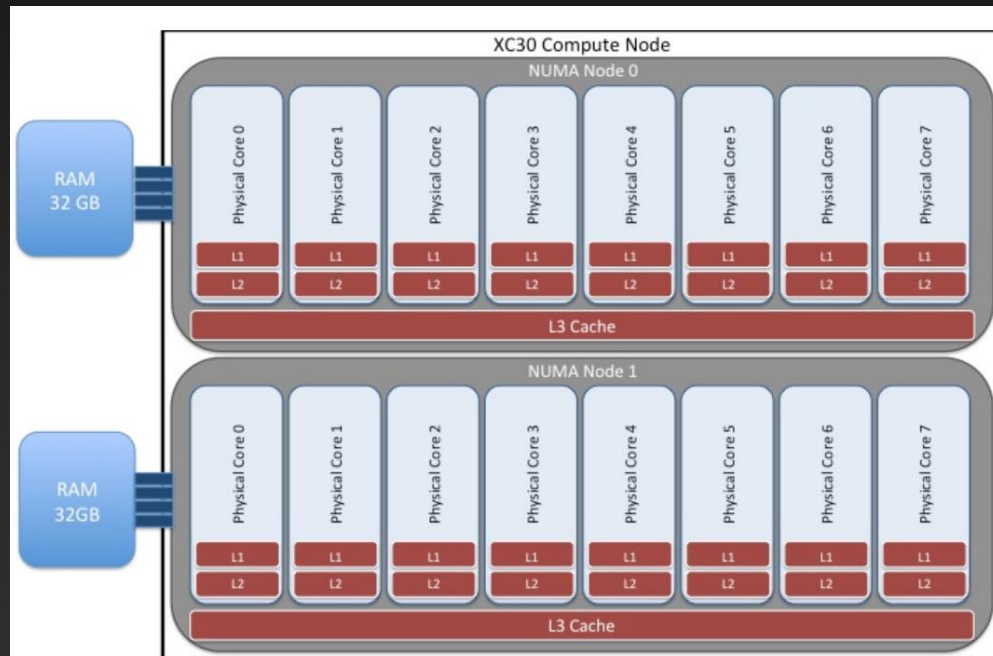
Just on a single node (our concern for OpenMP) we have:

- Registers (including vector registers)
- Caches (multiple levels)
- RAM (processor local or NUMA memory)
- HBM?
- Accelerators?
- NVM?

These are being accessed in various patterns by:

- Loops (hopefully vectorized)
- Threads
- Processes
- Cores
- Processors

*ORNL Cray XC30 Node*



# Easy Data Affinity

Here is a good example of how easy it can be to request data/thread affinity for a couple of tasks that we know share data.

```
void related_tasks( float* A, int n ){  
    float* B;  
  
    #pragma omp task shared(B) depend(out:B) affinity(A[0:n])  
    {  
        B = compute_B(A,n);  
    }  
    #pragma omp task firstprivate(B) depend(in:B) affinity (A[0:n])  
    {  
        update_B(B);  
    }  
    #pragma omp taskwait  
}
```



# Thread Placement and Memory Allocation

We can also manage these issues with explicit control of our thread placement or closely controlled management of our memory allocation. These approaches have also lacked any standard methods. We only have time to present the basics here. The documentation is comprehensive:

## Thread placement:

`OMP_PLACES` environment variable. It has lots of options and fine control mapping.

Clauses on *parallel* directive: *primary*, *close*, *spread*

## Memory Allocation:

*allocate* clause on all data sharing directives

*allocate* directive

*omp\_alloc()* and associated functions

The specifiers on these follow, and give you some idea of the kinds of hints/suggestions you can provide:

# OpenMP 5.0 Memory Hierarchy Awareness

The specifiers in the new spec give you some idea of how many ways we can characterize this.

**distance**  $\approx$  **near, far** Specifies the relative physical distance of the memory space with respect to the task the request binds to.

**bandwidth**  $\approx$  **highest, lowest** Specifies the relative bandwidth of the memory space with respect to other memories in the system

**latency**  $\approx$  **highest, lowest** Specifies the relative latency of the memory space with respect to other memories in the system.

**location** = Specifies the physical location of the memory space.

**optimized** = **bandwidth, latency, capacity, none** Specifies if the memory space underlying technology is optimized to maximize a certain characteristic. The exact mapping of these values to actual technologies is implementation defined.

**pagesize** = **positive integer** Specifies the size of the pages used by the memory space.

**permission** = **r, w, rw** Specifies if read operations (r), write operations (w) or both (rw) are supported by the memory space.

**capacity**  $\geq$  **positive integer** Specifies the physical capacity in bytes of the memory space. **available**  $\geq$  **positive integer** Specifies the current available capacity for new allocations in the memory space.

# OpenMP SIMD Extension

Much earlier I mentioned that vector instructions fall into the realm of “things you hope the compiler addresses”. However as they have become so critical achieving available performance on newer devices, the OpenMP 4.0 standard has included a `simd` directive to help you help the compiler. There are two main calls for it.

1) Indicate a simple loop that should be vectorized. It may be an inner loop on a parallel for, or it could be standalone.

```
#pragma omp parallel
{
    #pragma omp for
    for (int i=0; i<N; i++) {
        #pragma omp simd safelen(18)
        for (int j=18; j<N-18; j++) {
            A[i][j] = A[i][j-18] + sinf(B[i][j]);
            B[i][j] = B[i][j+18] + cosf(A[i][j]);
        }
    }
}
```

There is dependency that prohibits vectorization. However, the code can be vectorized for any given vector length for array B and for vectors shorter than 18 elements for array A.

# OpenMP SIMD Extension

2) Indicate that a function is vectorizable.

```
#pragma omp declare simd
float some_func(float x) {
    ...
    ...
}

#pragma omp declare simd
extern float some_func(float);

void other_func(float *restrict a, float *restrict x, int n) {
    for (int i=0; i<n; i++) a[i] = some_func(x[i]);
}
```

There are a ton of clauses (**private**, **reduction**, **linear**, **reduction**, etc.) that help you to assure safe conditions for vectorization. They won't get our attention today.

We won't hype these any further. Suffice it to say that if the compiler report indicates that you are missing vectorization opportunities, this adds a portable tool.

# flush - a step too far?

An example of the kind of low-level control you can achieve is the flush directive. An experienced concurrent programmer may want to do risky stuff like *reading and writing shared variables from different threads* (perhaps for rolling your own locks or mutexes). As shared memory machines have cache issues and compiler instruction reordering that can cause shared values to get out of sync, this is tricky business.

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

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

OpenMP Application Program Interface

[http://openmp.org/mp-documents/OpenMP\\_Examples\\_4.0.1.pdf](http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf)

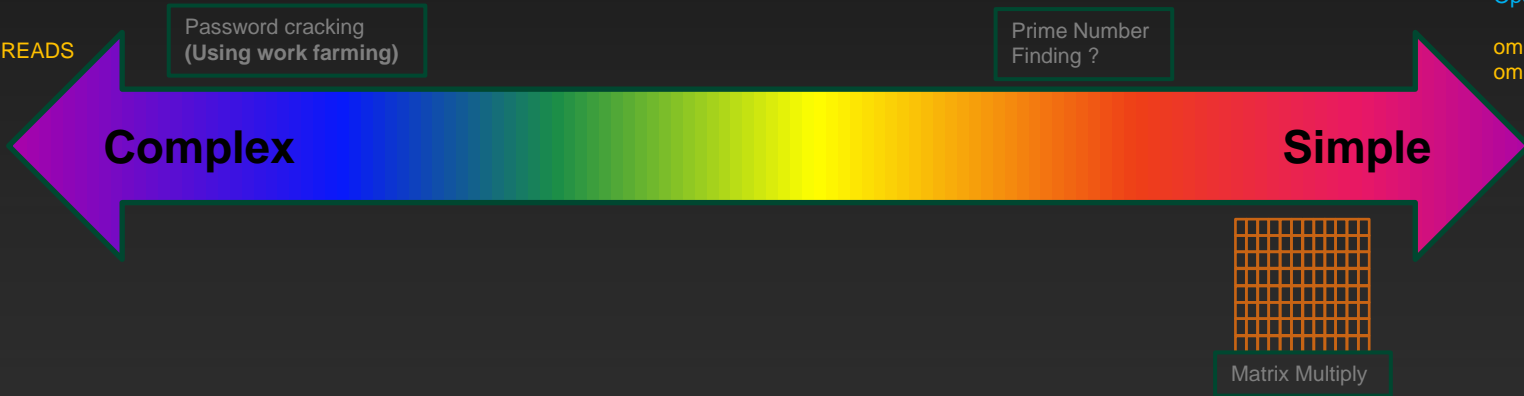
Most likely none of you will find this level of control advantageous.

# 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

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.

OpenMP 5 has now added modifiers (`monotonic`, `nonmonotonic`, `simd`) for use with the above, but they seem not to be widely implemented yet.



# Exercise 2: Improving Prime Number

(About 10 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

```
#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
Serial	43.6		
2	32.0	21.8	1.5
4	18.5	10.9	1.7
8	9.8	5.4	1.8
16	5.1	2.7	2.0
32	2.5	1.3	1.9
64	1.3	0.69	1.9
128	2.8	0.70	4

63X Serial!

500,000 iterations.

# OpenMP Environment

We've talked about a lot of tweakable configuration, and many of those parameters have multiple ways to set them (which is helpful). One convenient way I like to get a snapshot of the system is to use the `OMP_DISPLAY_ENV` variable to display most of the parameters. Just export `OMP_DISPLAY_ENV=TRUE`, or set it to `VERBOSE` for even more info.

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP='201611'
[host] OMP_CANCELLATION='FALSE'
[host] OMP_DEFAULT_DEVICE='0'
[host] OMP_DISPLAY_ENV='TRUE'
[host] OMP_DYNAMIC='FALSE'
[host] OMP_MAX_ACTIVE_LEVELS='2147483647'
[host] OMP_MAX_TASK_PRIORITY='0'
[host] OMP_NESTED='FALSE'
[host] OMP_NUM_THREADS: value is not defined
[host] OMP_PLACES: value is not defined
[host] OMP_PROC_BIND='false'
[host] OMP_SCHEDULE='static'
[host] OMP_STACKSIZE='4M'
[host] OMP_THREAD_LIMIT='2147483647'
[host] OMP_WAIT_POLICY='PASSIVE'
OPENMP DISPLAY ENVIRONMENT END
```

# 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

# Information Overload?

We have now covered just about everything with the exception of the GPU oriented stuff. I hope you recall how much we accomplished with just a parallel for/do. Let's recap. In HPC the most common approach is to:

- 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

There will be projects, such as graph oriented algorithms, where it will be more natural to just start with tasks, or another paradigm.