

Introduction to Parallel Programming with OpenMP

By Shinjae Yoo and Dave Stampf

All slides/programs/handouts available at

<https://github.com/sjyoo/OpenMPTutorialBase.git>



U.S. DEPARTMENT OF
ENERGY

Office of
Science

Lets start with some motivation

```
#include <stdio.h>
#include <limits.h>

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

    for (int i = INT_MIN; i < INT_MAX; i++) ;
    printf("Done with first pass\n");

    for (int i = INT_MIN; i < INT_MAX; i++) ;
    printf("Done with second pass\n");

    ...

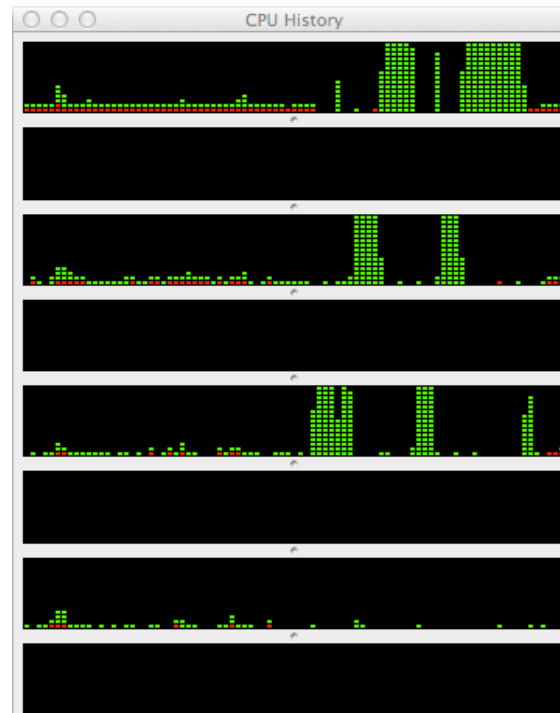
    for (int i = INT_MIN; i < INT_MAX; i++) ;
    printf("Done with eighth pass\n");

}
```

Serial Results (4-core laptop with hyperthreading...)

```
$ time ./Motivation
Done with first pass
Done with second pass
Done with third pass
Done with fourth pass
Done with fifth pass
Done with sixth pass
Done with seventh pass
Done with eighth pass
```

```
real 1m12.210s
user 1m12.017s
sys  0m0.020s$
```



This fancy computer seems 7/8 of it is on vacation - what gives?

Now, with some very simple mods

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

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

#pragma omp parallel num_threads(8)
{
    for (int i = INT_MIN; i < INT_MAX; i++) ;
    printf("Done with pass %d\n", omp_get_thread_num());
}
}
```

And running with OpenMP

```
$ time ./MotivationOMP
```

```
Done with pass 0
```

```
Done with pass 3
```

```
Done with pass 5
```

```
Done with pass 2
```

```
Done with pass 7
```

```
Done with pass 4
```

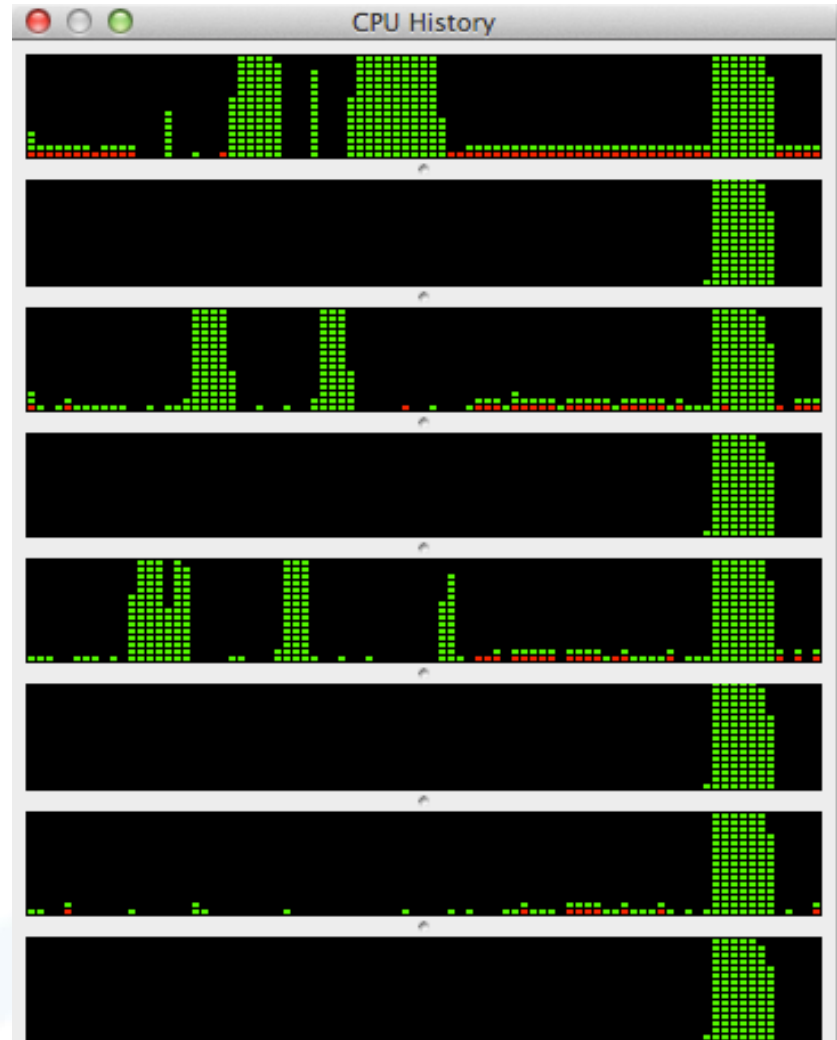
```
Done with pass 6
```

```
Done with pass 1
```

```
real 0m13.723s
```

```
user 1m43.803s
```

```
sys 0m0.114s
```



I don't know about you, but...

I WANT THAT!!!!!!

SCREEN basic command list

```
ssh guestNN@129.49.83.41
```

```
# Open new screen Session
```

```
screen -S S1
```

```
# Get today's tutorial copy
```

```
git clone https://github.com/sjyoo/OpenMPTutorialBase
```

```
cd OpenMPTutorialBase; git pull # update to the latest copy
```

```
# Source environment
```

```
. OpenMPTutorialBase/setenv.bash
```

```
# detach current session
```

```
Ctrl+a d
```

```
# reattach to Session S1
```

```
screen -x S1 or screen -x # if there is only one session
```

```
# list all the Sessions
```

```
screen -ls
```

Parallel Programming IS Mainstream Programming!

Popular Parallel Programming Models

- CPU-Intensive
 - OpenMP
 - MPI (Message Passing Interface)
 - CUDA
- Data-Intensive
 - MapReduce (Hadoop)
 - Storm

Agenda

- ◆ Nomenclature & Core Concepts
 - ◆ cores, threads (vs processes), hyperthreads
- ◆ What is OpenMP
 - ◆ Execution Model
 - ◆ Memory Model
- ◆ Programming with OpenMP
 - ◆ Parallel Regions
 - ◆ Loop-level Parallelism
 - ◆ Synchronization & Cooperating Threads
 - ◆ Work Sharing

Plus! Hands On Code Examples & Exercises!

- Hello World/Parallel Independent Tasks
- Map & Saxpy
- Trapezoid Rule
- Monte Carlo
- Difference Eq.

We call these smallish toy programs,
Programming Katas - the simple forms we follow
to internalize a programming skill

Reference

- ◆ I found “Using OpenMP - Portable Shared Memory Parallel Programming” by David J. Kuck to be quite useful - covers both C and Fortran)
- ◆ “Patterns for Parallel Programming” by Matson, Sanders, Massingill - talks about parallel algorithms and has nice brief introductions to OpenMP and MPI.
- ◆ “Structured Parallel Programming/Patterns for Efficient Computation” by McCool, Robison, and Reinders - also good for parallel algorithms
- ◆ But don't forget omp.org & formal spec:
 - ◆ 160 pages of specifications (10 pages of glossary!)
 - ◆ 160 pages of examples

What just happened in our example?

- ◆ There are many ways to do 8 tasks - e.g., serially.
 - ◆ Do the 0th task, then the 1st, then the 2nd, and finally the 7th.
 - ◆ Nearly every loop you see in programs is an example of this way of thinking (and an opportunity to speed things up!)
- ◆ Or, if you have the resources in parallel
 - ◆ Get 4 processors that can work independently of each other and give each one a task.

◆ In our code, the line

`"#pragma omp parallel num_threads(8)"`
asks the Compiler to please write the code necessary to create 8 threads and then ask each of those threads to execute the next statement or block, and when they finish, to wait for all to complete.

OMP's "Big Idea" - Execution Model

Simple "fork/join"

```
// serial code
```

```
#pragma omp parallel ...
```

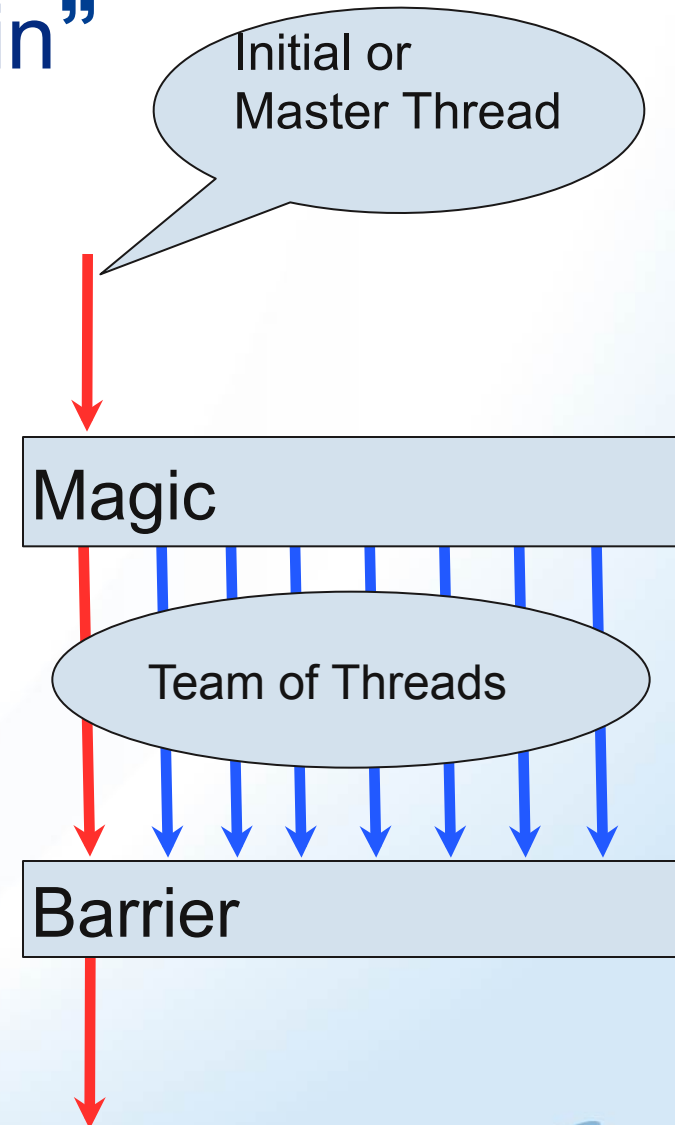
```
{
```

```
    // a block with regular C
```

```
    // and perhaps some
```

```
    // calls to omp functions
```

```
}
```



This is the “Big Idea” of OMP

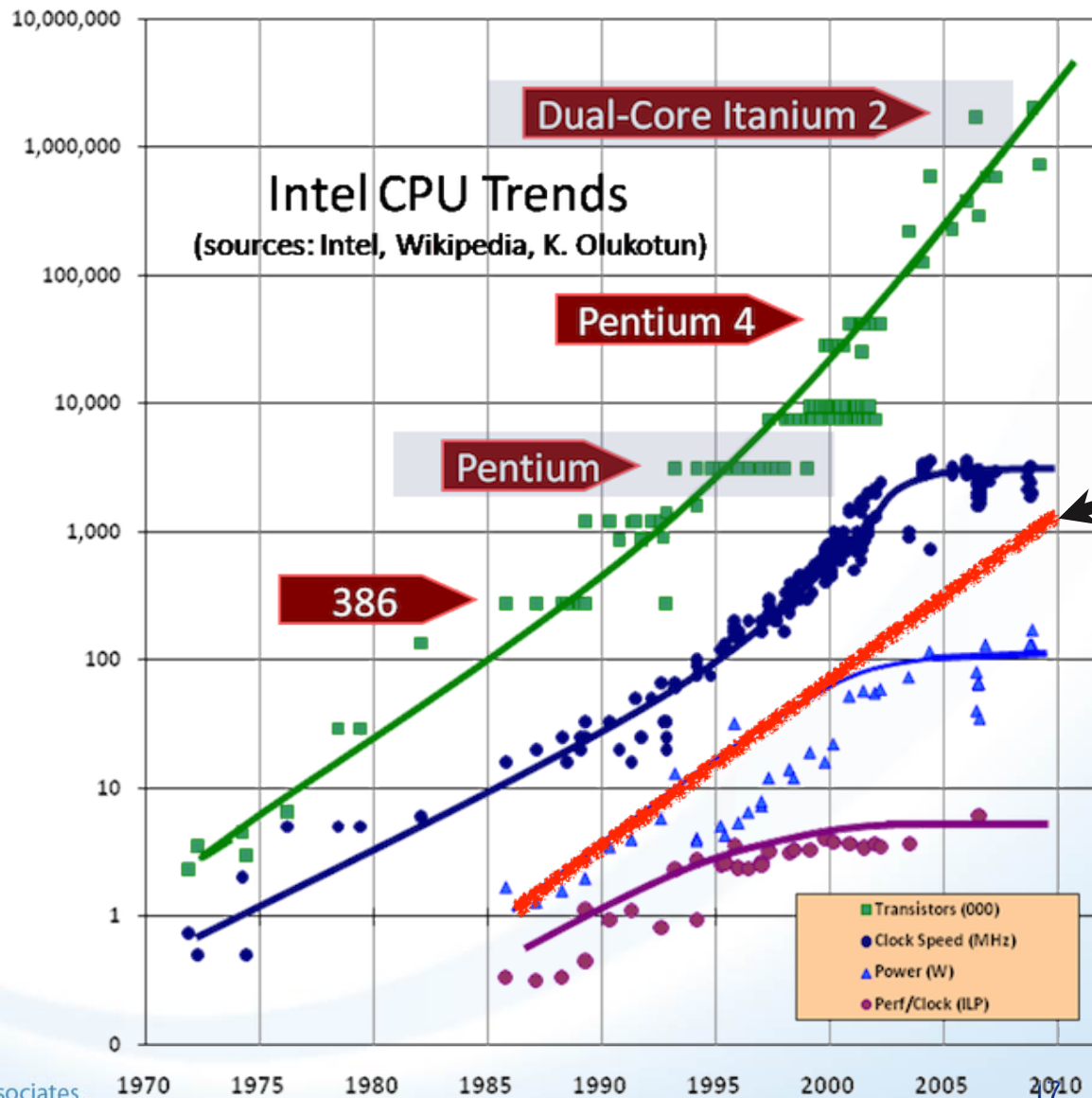
- The Fork/Join model is the only way to create threads
- By default, the threads all independently execute the next statement/block
- Everything else in OpenMP is focussed on getting some control of these threads in order to allow them to cooperate to accomplish some useful tasks.

Some History & Nomenclature Cores and Threads

- ◆About 10 years ago, CPU manufacturers hit a brick wall that strictly limited how much faster they could drive their CPUs.
- ◆Moore's law (transistor density doubling every 2 years) was still going strong, but they could no longer translate that into faster CPUs.
- ◆And so was born the Core and a whole class of new software technologies.

Classic (by now) Performance Graph

(Source: "The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software" by Herb Sutter (DDJ 3/2005))

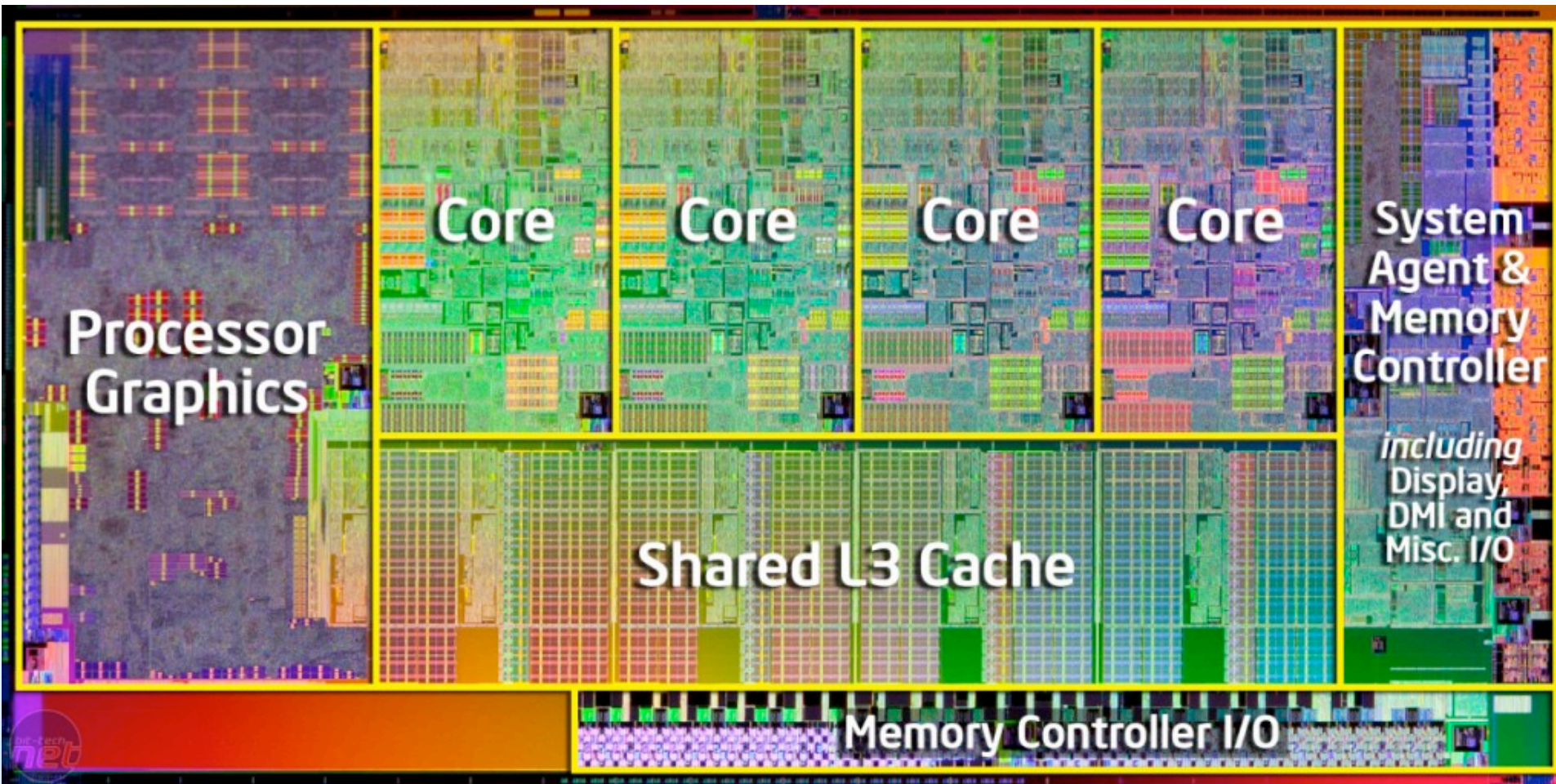


Enough to
toast a bagel
in 2010.

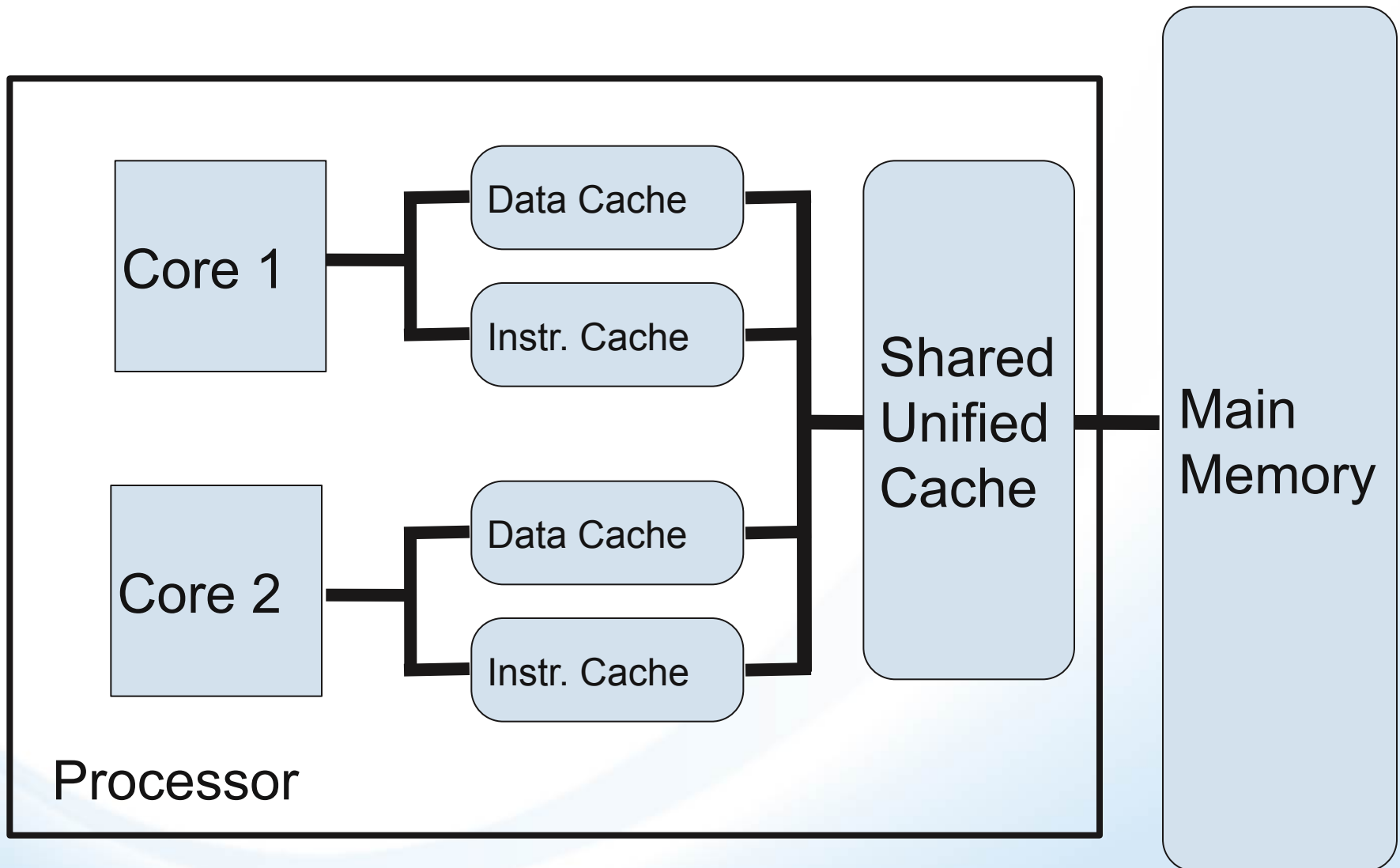
Student Exercise 1

- Work with the motivation example.
 - Make sure you can
 - edit
 - compile or make
 - time & run the programs on your computer. Does it indeed run faster? If not, why not?
 - Remove or comment out the `num_threads(8)` and re-run. How many threads are created?
 - Using my makefile, edit the line “`#pragma omp parallel`” so that you misspell `omp` (say, `opm`). What happens when you compile and/or run the code?
 - Modify the makefile to remove the `-Wall` compiler option. Re-make the program. What happens when you compile and/or run the code?

Intel Die Map



Cores with their Caches



How to know what your computer has?

- Linux systems
 - Try `lscpu`
- Mac
 - Try `sysctl -n hw.ncpu` (also includes “hyper” threaded cores)
 - Try `system_profiler SPHardwareDataType` to get the actual number of physical cores
- PC
 - ?

More on Cores

- Now, each of these cores may operate independently
 - separate ALU
 - separate registers
 - separate program counter
 - separate stacks
 - and maybe(!) even separate memory spaces etc.
- Each core is running a “thread” (think program for now) within a process
 - each process has an id
 - processes are by default protected from each other (can’t see/modify each others data)
 - If two cores are running processes with different IDs, they are protected from each other
 - If two cores are running processes that the same ID then, ...

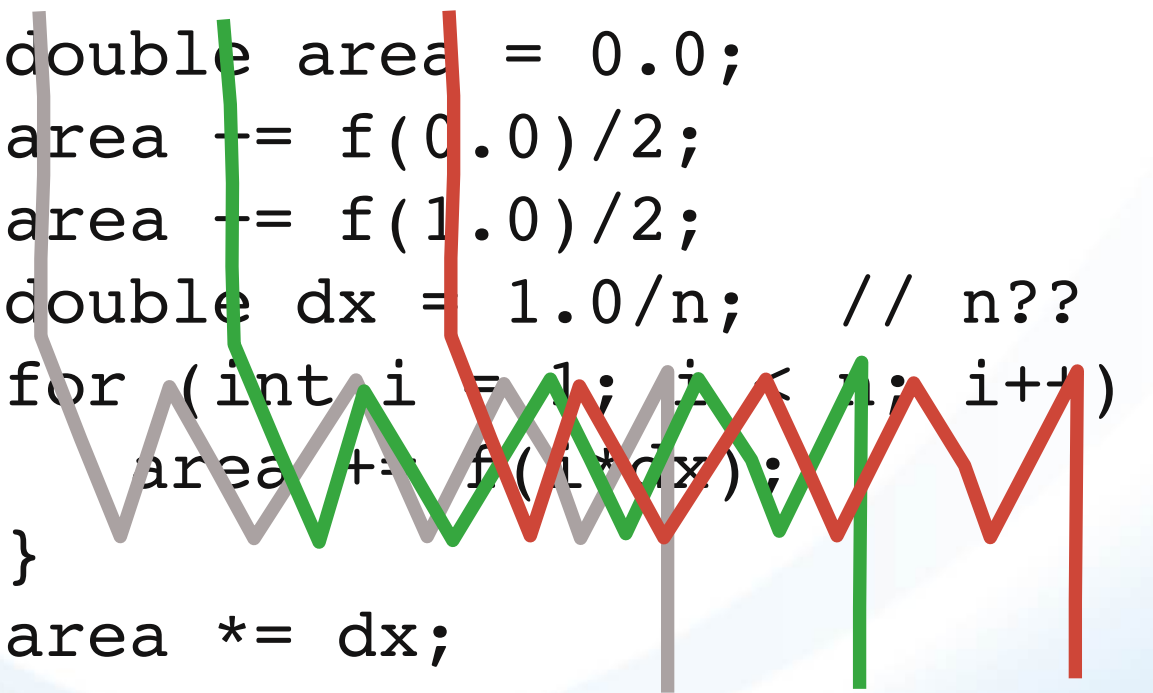
And so the concept of Threads

- A thread is a run time entity that is able to independently execute a stream of instructions (think Robot)
- Processes need a lot of information in order to protect themselves and behave rationally.
 - Contents of all of the registers
 - Contents of the memory map
 - Data structures for external resources (files, network connections etc.)
 - And so when you switch processes, there is a lot of stuff to save and restore.
- A process can have any number of threads.
- But threads (nee Light Weight Processes) share everything, except the register stack, with every other thread in the process..

One Definition

- A thread is a run time entity that is able to independently execute a stream of instructions (think Robot)

```
double area = 0.0;
area += f(0.0)/2;
area += f(1.0)/2;
double dx = 1.0/n; // n??
for (int i = 1; i < n; i++) {
    area += f(i*dx);
}
area *= dx;
```

The image shows three colored lines (grey, green, and red) representing different threads interleaving their execution of the code blocks. The grey line starts at the first line, goes to the second, then the third, then the loop body, and finally the last line. The green line starts at the first line, goes to the second, then the loop body, then the third, and finally the last line. The red line starts at the first line, goes to the second, then the loop body, then the third, and finally the last line. The lines are jagged, indicating the interleaving of execution.

But, imagine...

- Imagine that you have a program that has 3 functions to perform - the functions are independent of each other and so, could be computed at the same time.
 - So a process could create 3 threads and have each compute 1 integral using the same algorithm (function).
 - If you are on a machine with 1 core, then they interleave their execution as if they were just “light-weight” processes
 - But if you are on a machine with 2 or more cores, the computations will overlap and if the time to create the extra thread is less than the time to compute the integral - YOU WIN!

Furthermore

- The big question with threads
 - How do you create a new thread?
 - What data do threads you create share?
 - What data do thread your create hold private?
 - How can the threads you create synchronize themselves?
- Note - OpenMP doesn't bring anything new to the threading table
 - It uses the threading capabilities that it finds on the system
 - It presents to the programmer a “simplified” model of thread execution and data sharing
 - It brings a high degree of portability to threads

Processes and Threads

- On Unix-like systems, processes are created with the “fork” system call (usually invoked as fork/exec).
 - 2 entities are created (processes) each of which are protected from each other
- On Unix-like systems, threads are created ...
 - Well, its complicated... not because their creation is hard, but working with them is complicated.

Final word on Threads

- What a thread is, depends upon who is using the term
 - To a Hardware designer, a thread is something that runs on a core. Intel says with a straight face that their processors run 4, 8, 16, ... threads. Yet one can write a program that uses hundreds of threads
 - To a Software Engineer, a thread is what I've been talking about - a path through a program with an associated memory (some of which is shared)
 - To a Systems Programmer - a thread is a unit of scheduling. If there are threads available to run and cores that aren't doing anything, they connect the thread to the core. If all of the cores are busy with their own threads, then a running thread will eventually be swapped out for one ready to run.

Some programming!

- In addition to directives (pragmas) OpenMP is also a collection of environment variables and library functions. Lets look at the most important of these. (These are on the cheat sheet)
- Environment
 - OMP_NUM_THREADS
 - is used to set the number of threads that the parallel directive uses lacking any other direction
- Library
 - void omp_set_num_threads(int n)
 - set number of threads for subsequent parallel regions
 - int omp_get_num_threads()
 - number of threads in the current team
 - continued...

Some programming! (continued)

■ Library

- `int omp_get_thread_num();`
 - the id of this thread (0 based)
- `int omp_get_num_procs();`
 - how many processors (cores) are available
- `double omp_get_wtime();`
 - returns elapsed wall time in seconds
- `double omp_get_wtick();`
 - returns precision of timer used by `omp_get_wtime`

Serial Hello World

In C:

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

```
int main()
```

```
{
```

```
    printf("hello world\n");
```

```
    return 0;
```

```
}
```

Student Exercise 2 - Parallel Hello World

Prerequisites

- ◆Environment:

 - ◆export OMP_NUM_THREADS=4

- ◆Includes

 - ◆#include <omp.h>

- ◆Directive:

 - ◆#pragma omp parallel

- ◆Function

 - ◆omp_get_thread_num() - returns int id for the thread in the team (0 is the master thread and the id increases by 1 for each additional thread in the team.) You must also include <omp.h> if you use OpenMP functions.

- ◆Compiler & compiler flags

 - ◆-fopenmp -Wall

Parallel Hello World: (don't peek!)

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

int main()
{

    return 0;
}
```

Student Exercise 3

- Start again with the motivational problem and using ONLY what we have discussed so far...
 - Imagine that rather than having to perform the same task 8 times, we had to do 8 different and independent tasks - perhaps as part of initializing a larger program. They may include reading an input file, connecting to a remote database, etc. Since these tasks are independent of each other, they could be performed in parallel saving a lot of time. Write the code to do so. (create `doTaskA()`, `doTaskB()` etc. and call them so they run in parallel on different threads)
 - Find out what `omp_get_num_procs()` returns on your computer. Can you justify that number?
 - There are 3 ways to set the number of threads - env. var `OMP_NUM_THREADS`, `omp_set_num_threads(int)` and in the parallel directive itself.
 - What happens when these numbers disagree. Try 16, 8, 4 and 2 for test values.

Other Directives - Sections/Section

- In Example 3 - we had N tasks to perform and T threads available with which to perform them. You can be forgiven if you assumed that $T \geq N$, but it may not be so! Enter the sections/section directive.

- From inside a parallel region where a team of threads run:

```
#pragma omp sections
{
    #pragma omp section
        // structured block
    #pragma omp section
        // structured block
    /// ...
}
```

Other Directives - Sections/Section

- These directives have the compiler write the code necessary to have each individual segment to be given to an available thread for execution.
 - If there are more threads than tasks, then some threads may go idle
 - If there are more tasks than threads, then as threads complete some tasks, they will be assigned to another.

- Alternative syntax - if the sections directive is the entire contents of the omp parallel directive, you can combine as:

```
#pragma omp parallel sections
{
    #pragma omp section
    structured block
```

```
...
```

```
}
```

Student Exercise 4

- Redo Exercise 2 using the sections/section directives. Make sure you have more tasks to perform than threads available to accomplish them simultaneously.
 - Which thread does which task?
 - Is it always the same or does it seem to differ?
 - Try to make the tasks take different times to execute.

Recapping some OpenMP Concepts

- ◆ It is NOT a separate language
- ◆ It is implemented inside the compiler from hints you give it via `#pragma`
- ◆ Goals for the designers of OpenMP (not necessarily met nor religious about it...)
 - ◆ Sequential Equivalence - the SAME program should work and produce the same answer with 1 thread as it does with N threads
 - ◆ Incremental Parallelism - One should be able to start with a serial program and gradually parallelize it.
- ◆ These goals often conflict with the goal of maximizing performance.

High Performance Computing Concepts

- ◆ If you have to perform some task (e.g., think about digging a hole), then to do it faster you can
 - ◆ work faster - this is a non-starter - see the slide on power
 - ◆ do more with every step (i.e., get a bigger shovel) - this may work for a while, but think about how heavy that shovel gets
 - ◆ have lots of friends help you out - but this offer requires some sort of cooperation
- ◆ OpenMP tries to do more with every step (utilizing extra cores running thread) and provide the means for the threads to cooperate with each other.

A Quick Reality Check

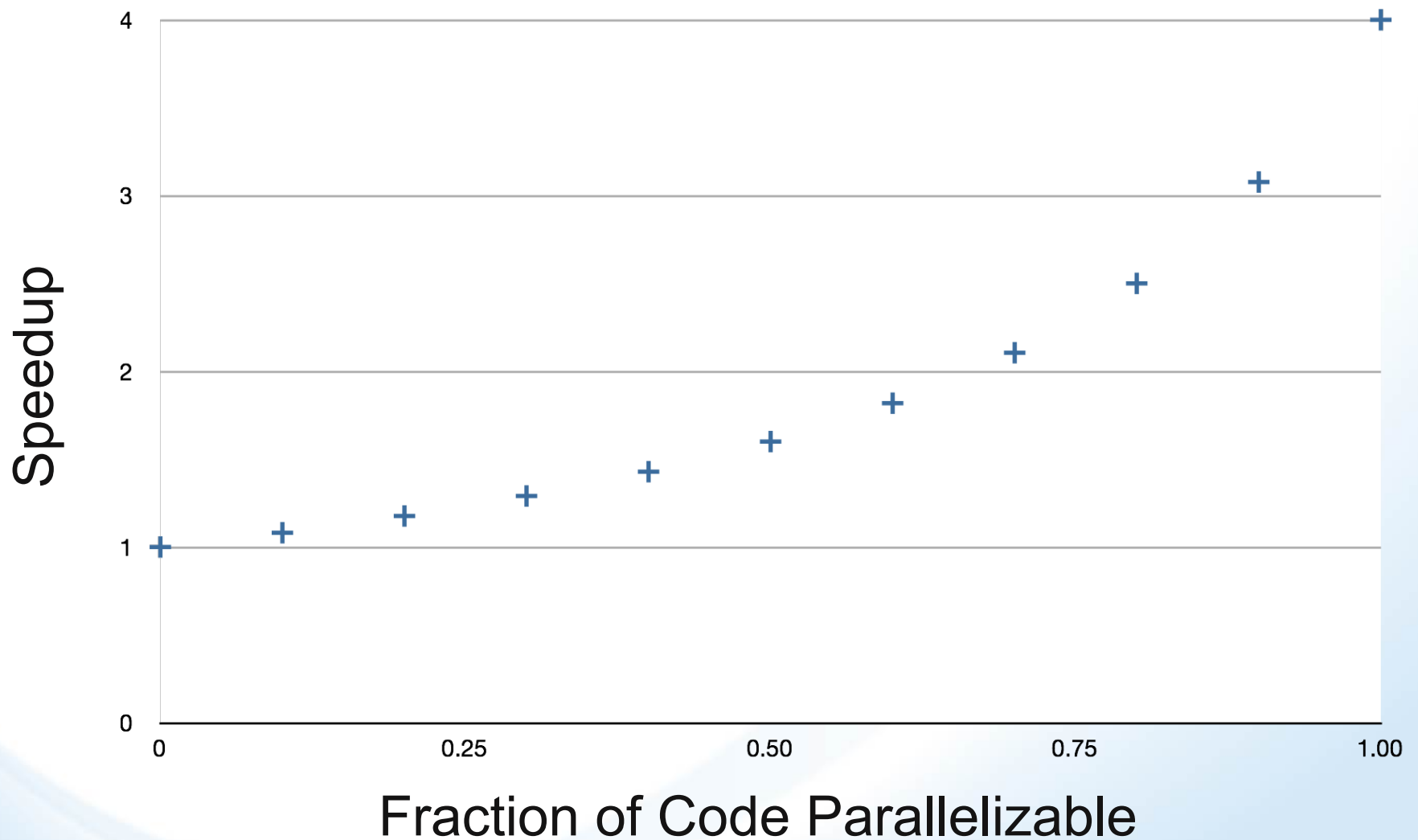
- Amdahl's Law:

- Let n be the number of threads
- Let f be the fraction of the program that can be parallelized
- $T(n)$ be the time an algorithm takes to complete using n threads.
- Let $S(n)$ be the speed-up one gets using n threads

$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{(1-f) + \frac{f}{n}}$$

Bottom line, no matter how many threads you have, you will never run more than $1/(1-f)$ times faster.

Amdahl's Law - For 4 Threads



Porting Code

- You want to make sure you get the same answer!
 - Preserve an initial copy of the code with test cases (input and output)
 - Any changes you make, should produce the same output for the same input. (modulo ordering of operations)
- You want it to be faster.
 - time your runs
- You want it to be WAAAAAY faster
 - find a good profiler on your system, locate the hot spots and focus there.

OMP Directives

- These are commands/suggestions to the compiler that may cause the compiler to generate some code or take some action.
 - `#pragma omp <keywords> <options>`
- If the compiler does not recognize a `#pragma`, it treats it like a comment - i.e., it silently IGNORES it.
 - This is a problem - e.g., `#pragma OMP`, `#pragma openmp`, `#pragma opm` are all misspellings, and are simply ignored
 - USE `-Wall` option of `gcc` to get warning about this. It is not an error - just a warning.
- Programming tip
 - If your compiler is handling OMP, it defines the macro `_OPENMP` to be the integer `yyyymm`. In your parallel programs, make sure it is defined!

Dependent vs Independent Iterations

```
for (int i = 0; i < n; i++) {  
    printf("%d\n", i);  
}
```

← Order dependent

```
for (int i = 1; i < n; i++) {  
    x[i] = x[i-1] + v[i]*dt;  
    v[i] = v[i-1] + g*dt/2;  
}
```

← Data Dependent

```
for (int i = 0; i < n; i++) {  
    // map!  
    y[i] = f(x[i]);  
}
```

← Independent
(maybe!!!)

Sometimes you can easily convert

```
for (int i = 0; i < n; i++) {  
    double res;  
    res = lotsOfIndependentWork(i);    // costly  
    dependentWork(res);                // quick  
}
```

```
double temp = malloc(n*sizeof(double));  
for (int i = 0; i < n; i++) {  
    temp[i] = lotsOfIndependentWork(i);    // costly  
}  
for (int i = 0; i < n; i++) {  
    dependentWork(temp[i]);                // quick  
}
```

So - how to get control of loops?

- ◆ `#pragma omp parallel for`

- ◆ This also starts “n” threads (independent of the size of the for loop).

- ◆ Each thread is given a subset of the indices to run over

- ◆ At the end of the for loop is a barrier where the threads wait.

- ◆ In OpenMP-speak, this is called “work sharing”, that is the work of the loop is shared among the threads.

Note - “`#pragma omp parallel for`” is short for `#pragma omp parallel`

```
{  
    #pragma omp for  
    for (...) {  
    }  
}
```

Simple Loop Parallelization

Parallel for/do directives

```
/* serial code */  
  
#pragma omp parallel for  
  for(i = 0; i < N; i++)  
    !compute stuff
```

The Basic OMP for pragma

```
// serial code

#pragma omp parallel for

for (...) { // very standard for loop
    /// time consuming computation
}

// serial code
```

Caveats

- No breaks
- No long jumps
- * No changing loop index/bounds inside the loop

Mapping Code Example

Take a vector of real numbers and map them to $\exp(x^2)$ using `omp parallel for/do` directive.

$[x_0, x_1, \dots, x_{N-1}]$

to

$[\exp(x_0^2), \exp(x_1^2), \dots, \exp(x_{N-1}^2)]$

Perform this mapping 1000 times (to get a reasonable execution time). Use made up values for vector x , $N=1,000,000$ and print the sum of the mapping on the screen. Write a serial (or use my serial version) and parallel version and compare. Time your runs and make sure the answers agree!

Serial Version (1 of 2)

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

int main()
{
    const int N = 1000000;
    float sum = 0.0f;
    float *x, *z; // range and domain
    x = (float*) malloc(N*sizeof(float));
    z = (float*) malloc(N*sizeof(float));

    int i;

    /* populate x */
    for(i = 0; i < N; i++)
        x[i] = (i+1)*.000001;
```

Serial Version (2 of 2)

```
/* map code here */

for(i = 0; i < N; i++) {
    z[i] = exp(x[i]*x[i]);
}

/* compute the sum */

for(i = 0; i < N; i++) {
    sum += z[i];
}

printf("%f\n", sum);

return 0;
}
```

Simple Loop Parallelization MAPPING in C

```
#pragma omp parallel for  
  for(i = 0; i < N; i++)  
    x[i] = (i+1)*.000001;
```

```
/* map code here */
```

```
#pragma omp parallel for  
  for(i = 0; i < N; i++) {  
    z[i] = exp(x[i]*x[i]);  
  }
```

NOTE - we cannot parallelize the summation loop the same way! Try it! Why??? We will come back to this shortly.

Simple Loop Parallelization (saxpy)

Single Precision $a*x+y$ or saxpy

$$z(i) = a*x(i) + y(i), \text{ (for } i=1, n\text{)}$$

This loop has no dependences. The result of one loop iteration does not depend on the result of any other iteration. Iterations may be run simultaneously.

Practice your newly learned skill. Write a code that implements SAXPY in serial and then parallel using the **parallel for/do** directive. Use made up values to populate your vectors with $a=.5$ and $N=1000$. Sum over the vector z and print the final sum on the screen. Make sure that you are running OpenMP and make sure the answers match the serial version.

Simple Loop Parallelization (saxpy) in C

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

int main()
{
    const int N = 1000;
    const float a = .5f;
    float sum = 0.0f;
    float z[N], x[N], y[N];
    int i;

    for(i = 0; i < N; i++)
    {
        x[i] = (i+1)*.15;
        y[i] = (i+1)*.1;
    }
```

```
#pragma omp parallel for
for(i = 0; i < N; i++)
{
    z[i] = a*x[i] + y[i];
}

for(i = 0; i < N; i++)
{
    sum += z[i];
}

printf("%f\n", sum);

return 0;
}
```

Back to the summation - Reduction

- ◆ Using a loop to find the sum, product, max, min, ... most any associative operation is so common that there is a special syntax for the parallel for pragma

- ◆ `#pragma omp parallel for reduction(op:list)`

- ◆ will generate code after the loop to combine the values

- appearing in list with the given operator for the various loops.

```
/* compute the sum */
```

```
#pragma omp parallel for reduction(+:sum)
for(i = 0; i < N; i++) {
    sum += z[i];
}
```

```
printf("%f\n", sum);
```

Quick Exercise

- ◆ Modify your best parallel version of the mapping code to compute the sum of the values using reduction.
- ◆ Time the serial, non-reduction and reduction version. Pleased?

Lets talk about Data

- ◆ In modern C and C++, one can declare variables inside a block (`{...}`). (Old fashioned C, like Fortran required all data to be declared before any executable per function).
- ◆ When a thread is running through the code in the block, it creates a local/private copy of that data that is NOT seen nor accessible by any other thread.
- ◆ However, any data defined outside of the block is visible (hence shared) by all of the threads

How to modify the default behavior?

- ◆omp parallel pragmas have optional clauses
 - ◆private(list) - each thread executing the block will have a private copy of each variable in the list. (Note, parallel for loop indices are always private).
 - ◆shared(list) - each thread executing the block will use a single shared copy of each variable in the list.
 - ◆default(none) - does not use any default rules. Every variable should be in a private or shared list.
- ◆Two schools of thought here
 - ◆Just use the default rules
 - ◆Never use the default rules and always list the variables
- ◆My school of thought - always err on the side of clarity

Shared Data - the Big Problem

- ◆ If two threads share data, and the data is always read, there is no problem. (And if this appeals to you, start studying languages with immutable data - e.g., Lisp, Clojure, Scala, F#, etc.)
- ◆ However, when threads share data that they modify, the big question is who modified it last, and do each of the threads (remember the diagram with cache memories) agree on the value in any variable at a point in time?

Shared Data - The Big Advantage

- ◆ Threads can communicate only if they have shared data - communication is a good thing!
- ◆ This can be safely done if access to shared data is controlled so that only one thread at a time can modify the data. This is the purpose behind the “critical” directive.
- ◆ Also threads can cooperate by waiting for one another to complete a task. This has been handled so far by an implicit “barrier”, but we can also use explicit barriers.

Synchronization in Simple Loop

```
int i;
#pragma omp parallel
for
  for(i = 0; i < N; i++)
  {
    z[i] = exp(x[i]*x[i]);
  }

/* omp implied barrier

for(i = 0; i < N; i++)
{
  Sum += z[i];
}
```

Sum depends on all z values having completed writing at the end of the parallel loop.

OpenMP has an implied **barrier** call at the end of the **parallel for** directive.

At the end of the first loop, the parent thread waits for all child threads to complete. Parent thread resumes serial execution after the implied **barrier**.

Shared and Private Clauses

Directives may have clauses to define data scope of variables.

Shared scope clause specifies that the named variables are shared by all threads in the parallel construct. Variables are shared by default.

Private scope clause specifies that the named variables are private to each thread in the parallel construct. Private variables are undefined upon entry and exit from parallel construct.

```
#pragma omp parallel private (private_sum)
{
    private_sum = 0.0;

#pragma omp for
    for(i = 0; i < N; i++)
    {
        private_sum += z[i];
    }
}
```

In example above, `private_sum` is a private variable and `z` is shared.

Shared and Private Clauses cont. and the Critical Directive

```
float sum = 0.0;
#pragma omp parallel private (private_sum) shared (sum)
{
    private_sum = 0.0;

#pragma omp for
    for(i = 0; i < N; i++)
    {
        private_sum += z[i];
    }

#pragma critical
    {
        sum = sum + private_sum;
    }
}
```

Parallel Reduction
example.

critical directive
restricts execution
of block to one
thread at a time.

Firstprivate and Lastprivate Clauses

```
float private_sum = 0.0;

#pragma omp parallel for firstprivate (private_sum) lastprivate (private_sum)
for(i = 0; i < N; i++)
{
    private_sum += z[i];
}
```

firstprivate clause initializes the private variable with the value of the master thread's copy upon entry.

lastprivate clause saves the last iteration value of the variable to the master thread's copy upon exit.

OpenMP Runtime Library

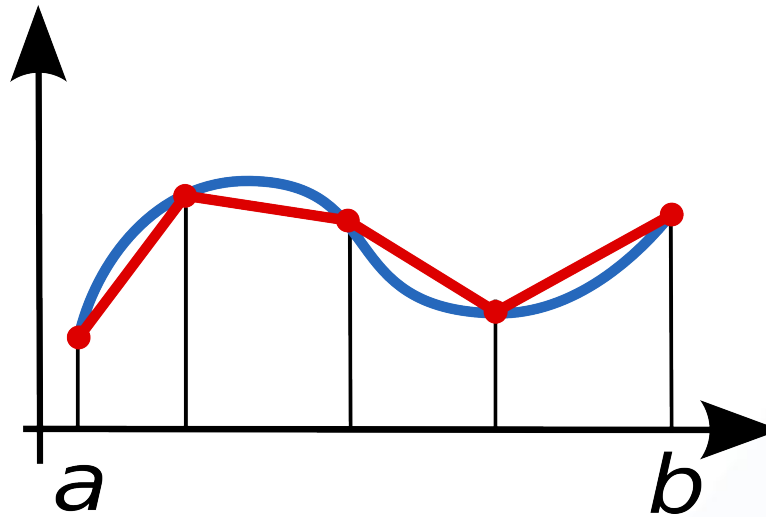
`omp_get_num_threads` returns the number of threads executing in the parallel region.

`omp_get_thread_num` returns the thread ID of calling thread. Master thread has ID=0.

`omp_set_num_threads(int)` sets the number of threads to use. Must be called from a serial portion of the code.

`omp_get_max_threads` returns the maximum number of threads available to parallel regions.

Trapezoid Rule



$$I = h * [f(x_0)/2 + f(x_n)/2 + f(x_1) + \dots + f(x_{n-1})]$$

Exercise - implement a parallel version of this using either a critical construct or a reduction - whichever you feel less comfortable with!

Trapezoid Rule Serial Code

```
#include <stdio.h>
#include <math.h>

double f(double x)
{
    return exp(x*x);
}

int main()
{
    double integral; /*definite integral*/
    const double a=0.0; /*left end point*/
    const double b=1.0; /*right end point*/
    const int N=100000; /*subdivisions*/
    double h; /*base width of subdivision*/
    double x;
    int i;
```

```
    h = (b-a)/N;
    integral = (f(a)+f(b))/2.0;
    x = a;

    for(i = 1; i <= N-1; i++)
    {
        x = x+h;
        integral = integral + f(x);
    }

    integral = integral*h;

    printf("%s%d%s%f\n", "WITH N=", N,
        " TRAPEZOIDS, INTEGRAL=",
        integral);

    return 0;
}
```

Trapezoid Rule Parallel Code in C

```
#include <stdio.h>
#include <math.h>
#include <omp.h>    /*openmp api*/

double f(double x)
{
    return exp(x*x);
}

int main()
{
    double integral;    /*definite integral result*/
    const double a=0.0; /*left end point*/
    const double b=1.0; /*right end point*/
    const int N=100000; /*number of subdivisions*/
    double h;           /*base width of subdivision*/

    h = (b-a)/N;
    integral = 0.0;
```

Trapezoid Rule Parallel Code in C

```
integral = (integral+(f(a)+f(b))/2.0)*h;
```

```
printf("%s%d%s%f\n", "WITH N=", N, " TRAPEZOIDS, INTEGRAL=",  
integral);
```

Trapezoid Rule Parallel Code in C Using Reduction Clause

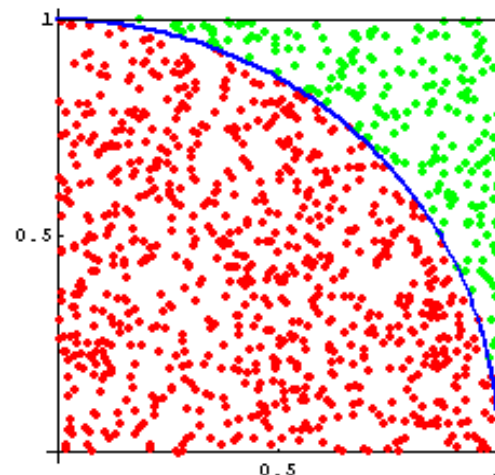
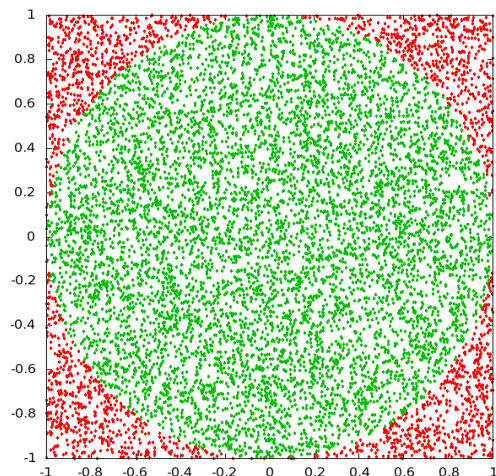
```
#pragma omp parallel reduction(+:integral)
{

}

printf("%s%d%s%f\n", "WITH N=", N, " TRAPEZOIDS, INTEGRAL=",
integral);
```

Monte Carlo to Calculate Pi

$$\frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi R^2}{(2R)^2} = \frac{\pi}{4}$$



If we randomly assign points inside the unit square and take the ratio of points that fall inside the circle to the total number of points, we can calculate π with the following formula:

$$\pi = 4 * N/M$$

We have a problem though...

- ◆The pseudo random number generator is NOT thread safe - the function has state - namely the last number generated (or the last “seed”).
- ◆So, we will write our own - starting with another non-thread safe version

Random Number Generator

```
#include <stdio.h>

unsigned int seed = 1; /* random number seed */
const unsigned int rand_max = 32768;

double rannum()
{
    unsigned int rv;
    seed = seed * 1103515245 + 12345;
    rv = ((unsigned)(seed/65536) % rand_max);
    return (double)rv/rand_max;
}
```

```
int main()
{
    const int N = 10;
    int i;

    for(i = 0; i < N; i++)
    {
        printf("%g\n",rannum());
    }

    return;
}
```

Random number between 0 and 1

***seed must never be initialized to zero**

Threadprivate Directive

The **threadprivate** directive identifies a global variable as being private to each thread. In essence, it's similar to the **private** clause except it applies to the entire program and not just a parallel region.

It gives us a way to declare global or static data to be private to every thread

Threadprivate Directive cont.

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

unsigned int seed; /* random seed */
const unsigned int rand_max = 32768;

double rannum()
{
#pragma omp threadprivate(seed)
    unsigned int rv;
    seed = seed * 1103515245 + 12345;
    rv = ((unsigned)(seed/65536) % rand_max);

    return (double)rv/rand_max;
}
```

Threadprivate Directive cont.

```
int main()
{
    const int N = 10; /*# of random
numbers*/
    int i;

#pragma omp threadprivate(seed)

#pragma omp parallel
    {
        seed = omp_get_thread_num()+1;
```

```
#pragma omp for
    for(i = 0; i < N; i++)
    {
        printf("%d\t%g\n",
            omp_get_thread_num(),
            rannum());
    }

    return;
}
```

Serial Version 1/2

```
/* Monte Carlo simulation of pi*/
```

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

```
unsigned int Seed = 1; /* random number seed */  
const unsigned int rand_max = 32768;
```

```
double rannum()  
{  
    unsigned int rv;  
    seed = seed * 1103515245 + 12345;  
    rv = ((unsigned)(seed/65536) % rand_max);  
    return (double)rv/rand_max;  
}
```

Serial Version 2/2

```
int main()
{
    const int N = 1000000000; /* number of random numbers */
    const double r = 1.0; /* radius of unit circle */

    double x, y; /* function inputs */
    double sum = 0.0;
    double Q = 0.0;

    for(int i = 0; i < N; i++)
    {
        x = rannum();
        y = rannum();

        if((x*x + y*y) < r) sum = sum+1.0;
    }

    Q = 4.0*sum*1.0/N;
    printf("%.9g\n", Q);
}
```

Monte Carlo - OpenMP version - 1/3

```
/* Monte Carlo simulation of pi*/

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

unsigned int seed = 1; /* random number seed */
const unsigned int rand_max = 32768;

double rannum()
{
#pragma omp threadprivate(seed)
    unsigned int rv;
    seed = seed * 1103515245 + 12345;
    rv = ((unsigned)(seed/65536) % rand_max);

    return (double)rv/rand_max;
}
```

Monte Carlo - OpenMP version - 2/3

```
int main()  
{  
    const int N = 1000000000; /* # of random numbers */  
    const double r = 1.0;      /* radius of unit circle */  
  
    int i;  
  
    double x, y; /* function inputs */  
    double sum = 0.0;  
    double Q = 0.0;
```


Monte Carlo - OpenMP version - 3/3

```
printf("%.9g\n", Q);
```

Data Dependencies, Recurrences

```
for(i = 0; i < N-1; i++)  
{  
    y[i] = y[i+1] - y[i];  
}
```

For N=4 and 2 Threads ...

Iteration	Thread 1	Thread2
0	y[0]=y[1]-y[0]	y[2]=y[3]-y[2]
1	y[1]=y[2]-y[1]	no-op

In Iteration 1, Thread 1 reads y[2] which has already been written by Thread 2.

Add to this that we don't really know the order in which instructions execute... We have a problem!

Final Project - getting down and dirty

- ◆ We are given an array of double precision numbers - for our purposes, they represent the value of a function at equally placed points along the domain. The points are a distance “h” apart.
- ◆ The goal is to compute the derivative without using 2x the memory!!!
- ◆ In general $(y[i+1]-y[i])/h$ is what we need to compute and to replace $y[i]$ with that value.
- ◆ A parallel uncontrolled group of threads won't work, even a parallel for worksharing won't help. You are going to have to work with individual threads.
- ◆ Hint - draw a picture!!!

Forward Difference - Serial 1/2

```
/* Recurrence, finite differences */
```

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

```
#include <math.h>
```

```
int main()
```

```
{
```

```
    const int N = 10000;    // # of intervals
```

```
    const double h = 0.001; // size of interval
```

```
    double y[N];           // value of function
```

```
    const int prune = 1;    // print every...
```

```
    for(int i = 0; i < N; i++) y[i] = sin(i*h);
```

```
    for(int i = 0; i < N; i++){
```

```
        if(i%prune == 0) printf("%g\t%g\n", i*h, y[i]);
```

```
}
```

Forward Difference - Serial 2/2

```
for(int i = 0; i < N - 1; i++) {  
    y[i] = (y[i+1]-y[i])/h;    // forward difference  
}  
  
y[N-1] = y[N-2];    // kind of kluge  
  
printf("\n\n");  
  
for(int i = 0; i < N; i++) {  
    if(i%prune == 0)  
        printf("%g\t%g\n", i*h, y[i]);  
}  
  
return 0;  
}
```

Forward Difference - Parallel 1/5

```
/* Recurrence, finite differences */

#include <stdio.h>
#include <math.h>
#include <omp.h>

int main()
{
    const int N = 10000;    // # of points in domain
    const double h = 0.001; // distance between points
    double y[N];            // value of fn & later derivative
    const int prune = 1;    // print every...

    // load up the function values
```

Forward Difference - Parallel 2/5

Forward Difference - Parallel 3/5

Forward Difference - Parallel 4/5

Forward Difference - Parallel 5/5

```
return 0;
```

Let's wrap it up

- ◆ Parallel programming IS mainstream programming. Every computer from phones to supercomputers has the capability.
 - ◆ Lowest level - get a good compiler
 - ◆ Next - OpenMP (1-5x speedup)
 - ◆ Next - MPI ($O(N)$ speedup where N = # of computers in cluster) - BUT - you can still use OpenMP on all of the distributed tasks
 - ◆ Next - Intel/Phi - a strange mixup of the above 2
 - ◆ Next - CUDA/OPEN-CL (10-1000 times speedup - but lots of programming required) And still you can make use of OpenMP and MPI
 - ◆ But also - learn some parallel algorithms - program with threads at every opportunity - it is the future after all!