Introduction to Parallel Programming with OpenMP

By Dave Stampf & Nick D'Imperio

BROOKHAVEN NATIONAL LABORATORY
a passion for discovery

U.S. DEPARTMENT OF ENERGY | Office of Science

Lets start with some motivation

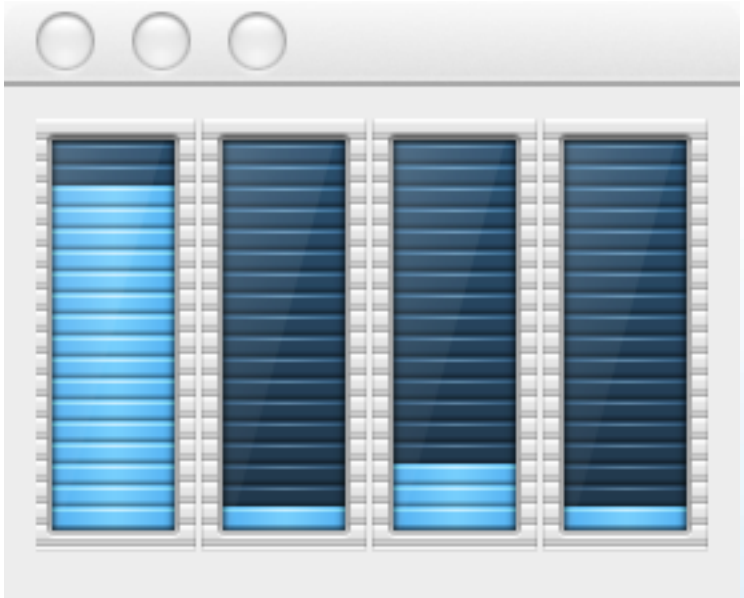
```
#include <stdio.h>
#include <limits.h>

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

   for (int i = INT_MIN; i < INT_MAX; i++) ;
       // your code replaces the ;
   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 third pass\n");

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

Serial Results (2-core laptop with hyperthreading...)

```
$ time ./Motivation
Done with first pass
Done with second pass
Done with third pass
Done with fourth pass

real 0m29.098s
user 0m29.080s
sys  0m0.011s
$
```

Serial Results (2-core laptop with hyperthreading...)

```
$ time ./Motivation
Done with first pass
Done with second pass
Done with third pass
Done with fourth pass

real 0m29.098s
user 0m29.080s
sys  0m0.011s
$
```



Now, with some very simple mods

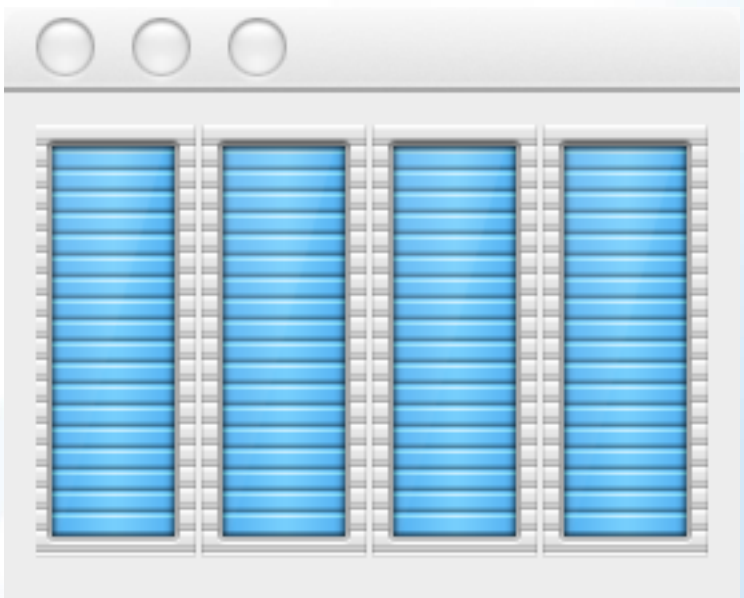
```
#include <stdio.h>
#include <limits.h>
#include <omp.h>

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

omp_set_num_threads(4);
#pragma omp parallel
   {
      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 1
Done with pass 2

real  0m10.121s
user  0m38.312s
sys   0m0.037s
$
```

And running with OpenMP

```
$ time ./MotivationOMP
Done with pass 0
Done with pass 3
Done with pass 1
Done with pass 2

real  0m10.121s
user  0m38.312s
sys   0m0.037s
$
```



I don’t know about you, but...

I don’t know about you, but...

I WANT THAT!!!!!

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

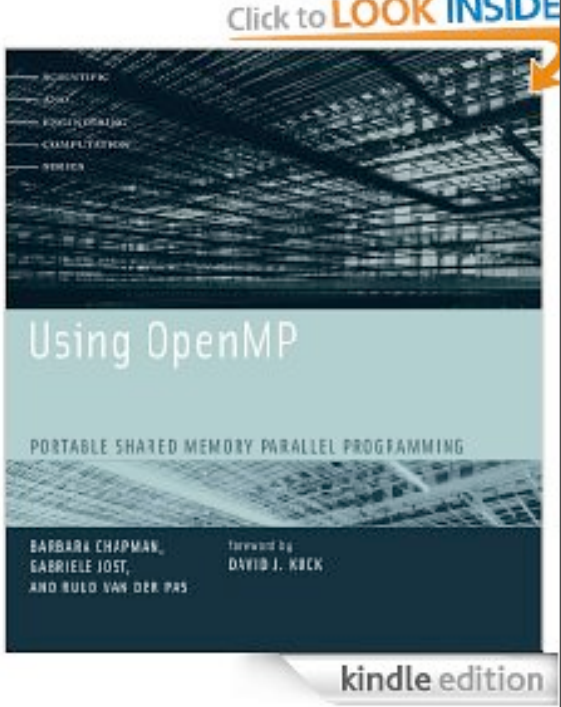
Plus! Hands On Code Examples & Exercises!

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

Reference

- I found “Using OpenMP - Portable Shared Memory Parallel Programming” by David J. Kuck to be quite useful - covers both C and Fortran)

- But don’t forget omp.org & formal spec:
  - 160 pages of specifications
  - 160 pages of examples

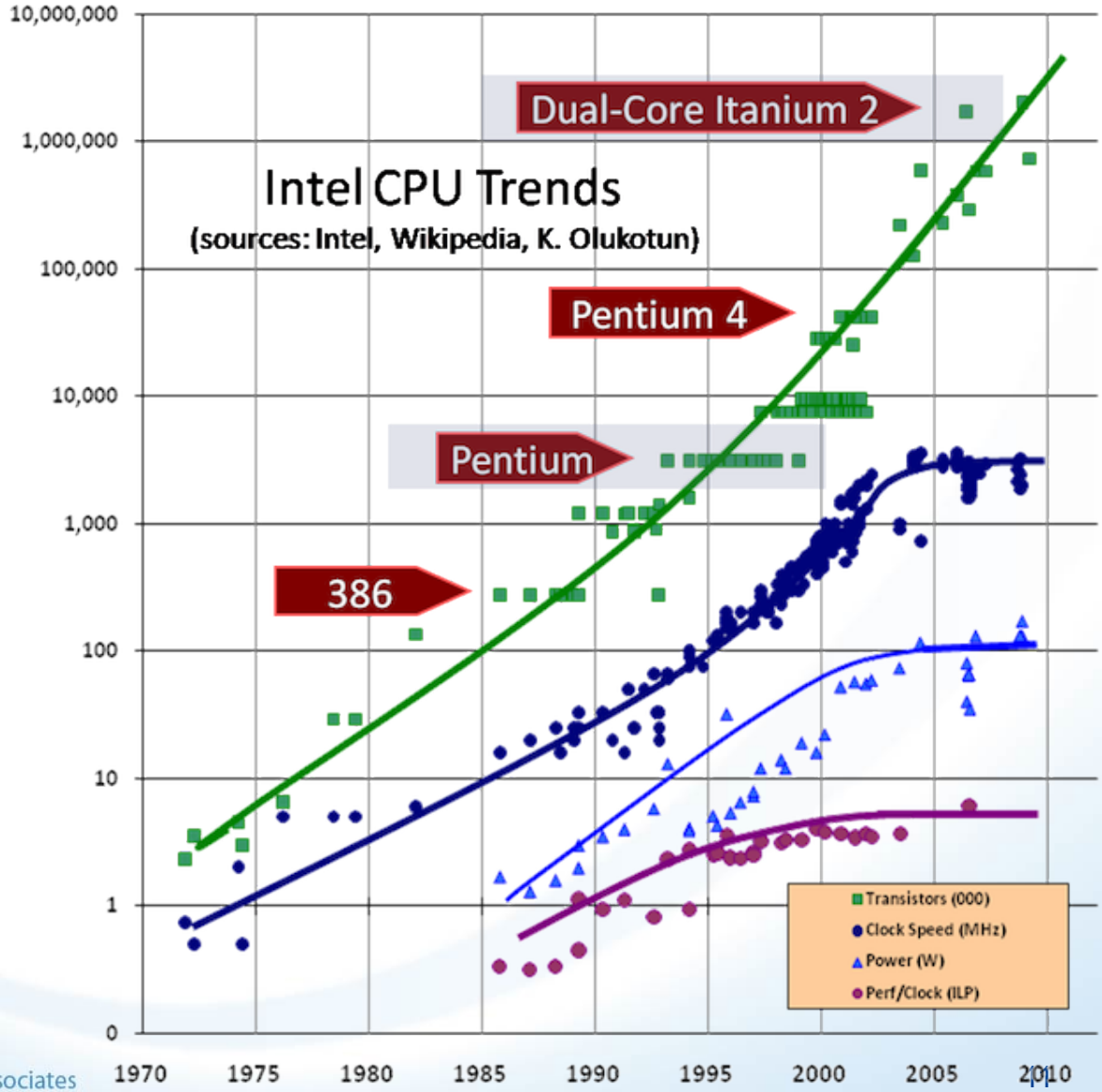


Some History & Nomenclature Cores and Threads

- 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.

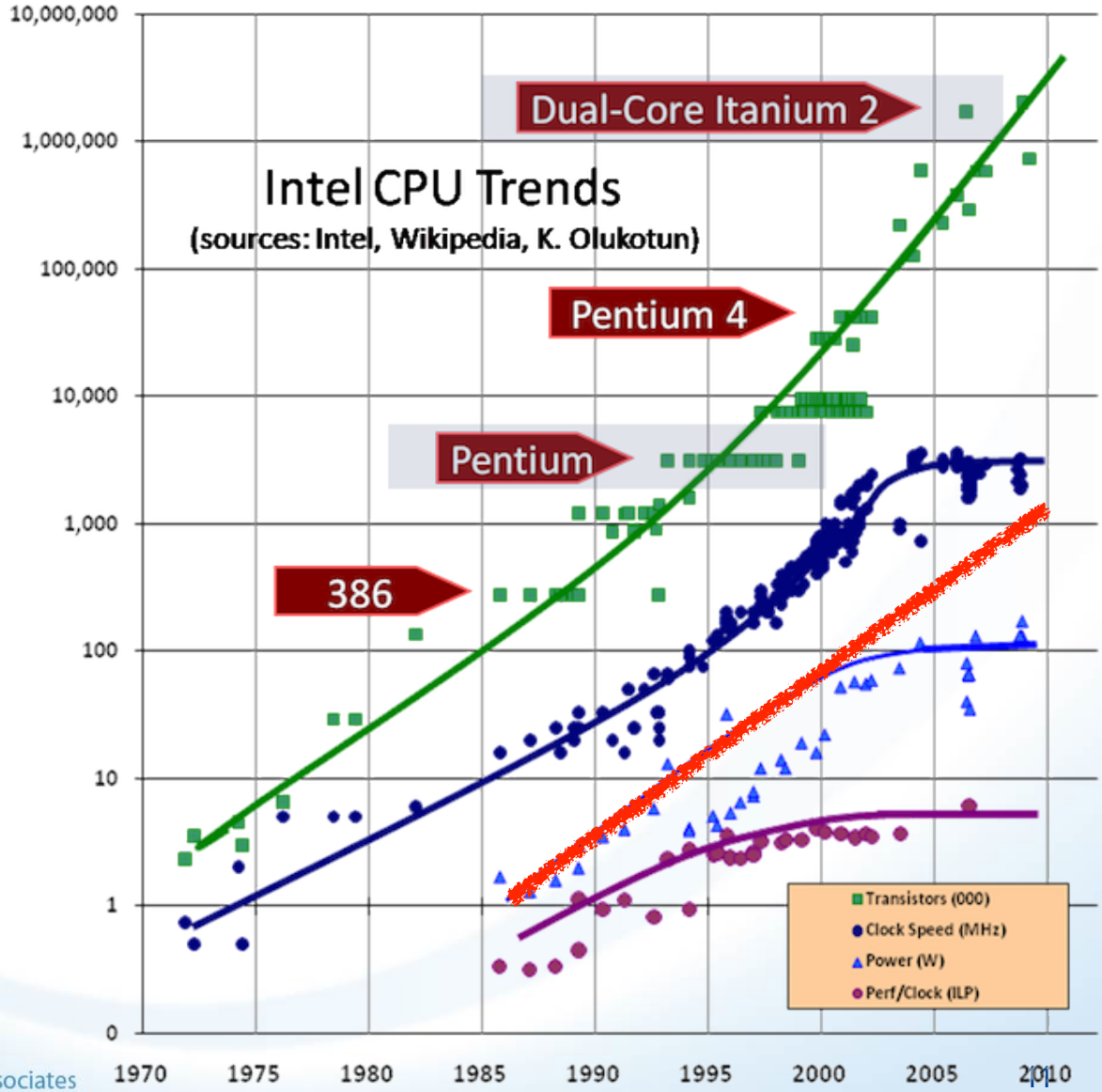
Classic (by now) Performance Graph

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



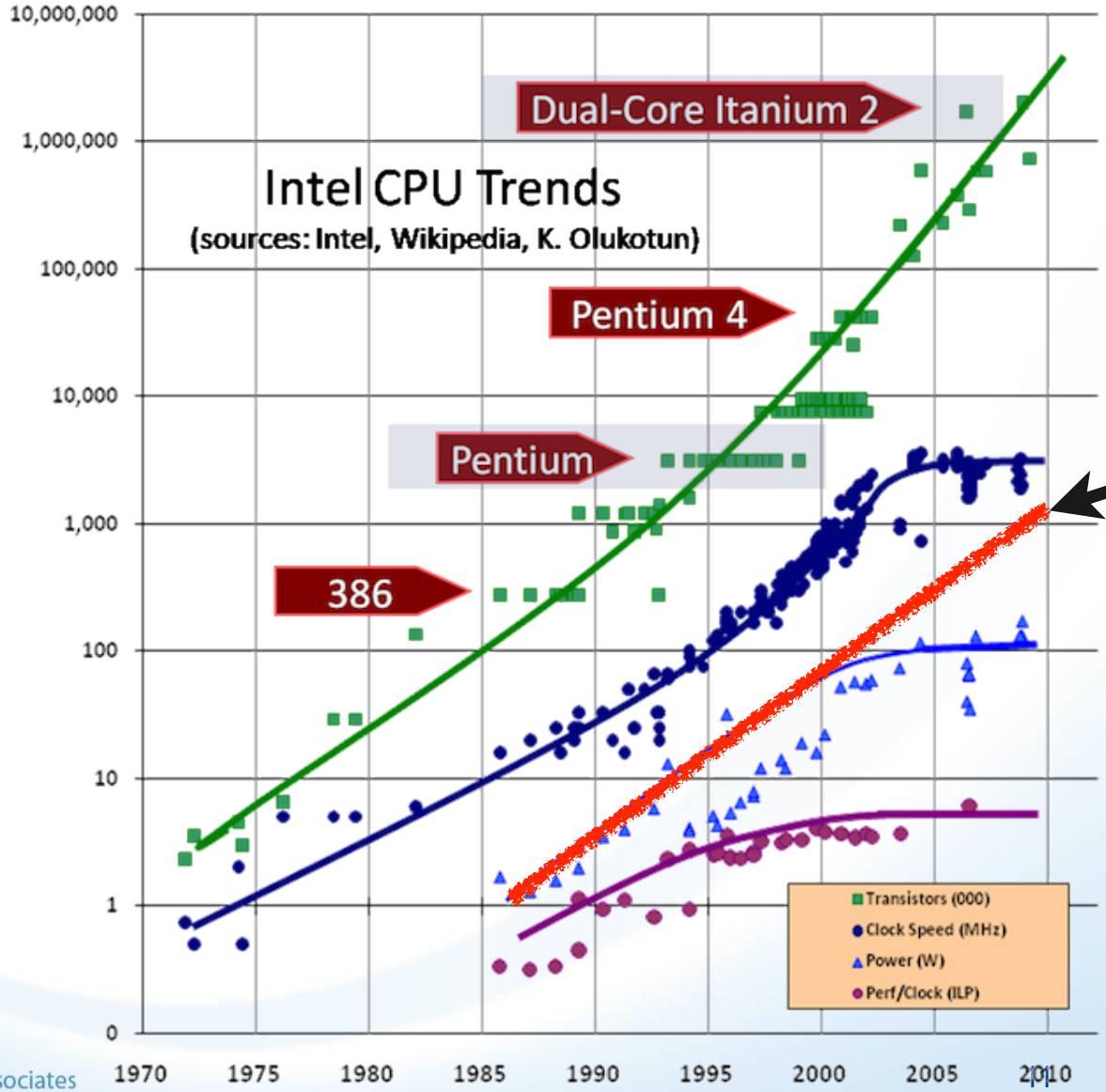
Classic (by now) Performance Graph

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



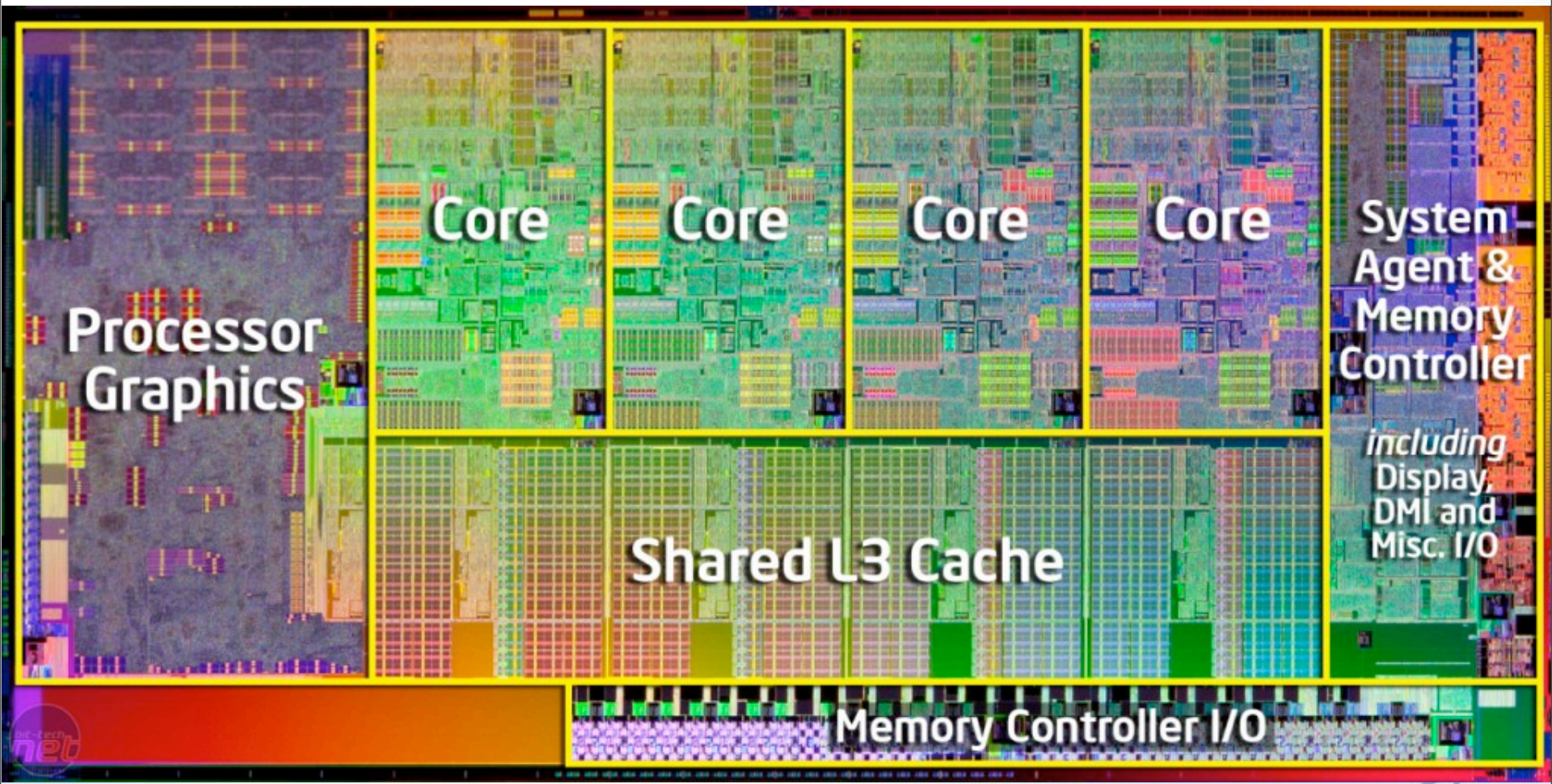
Classic (by now) Performance Graph

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

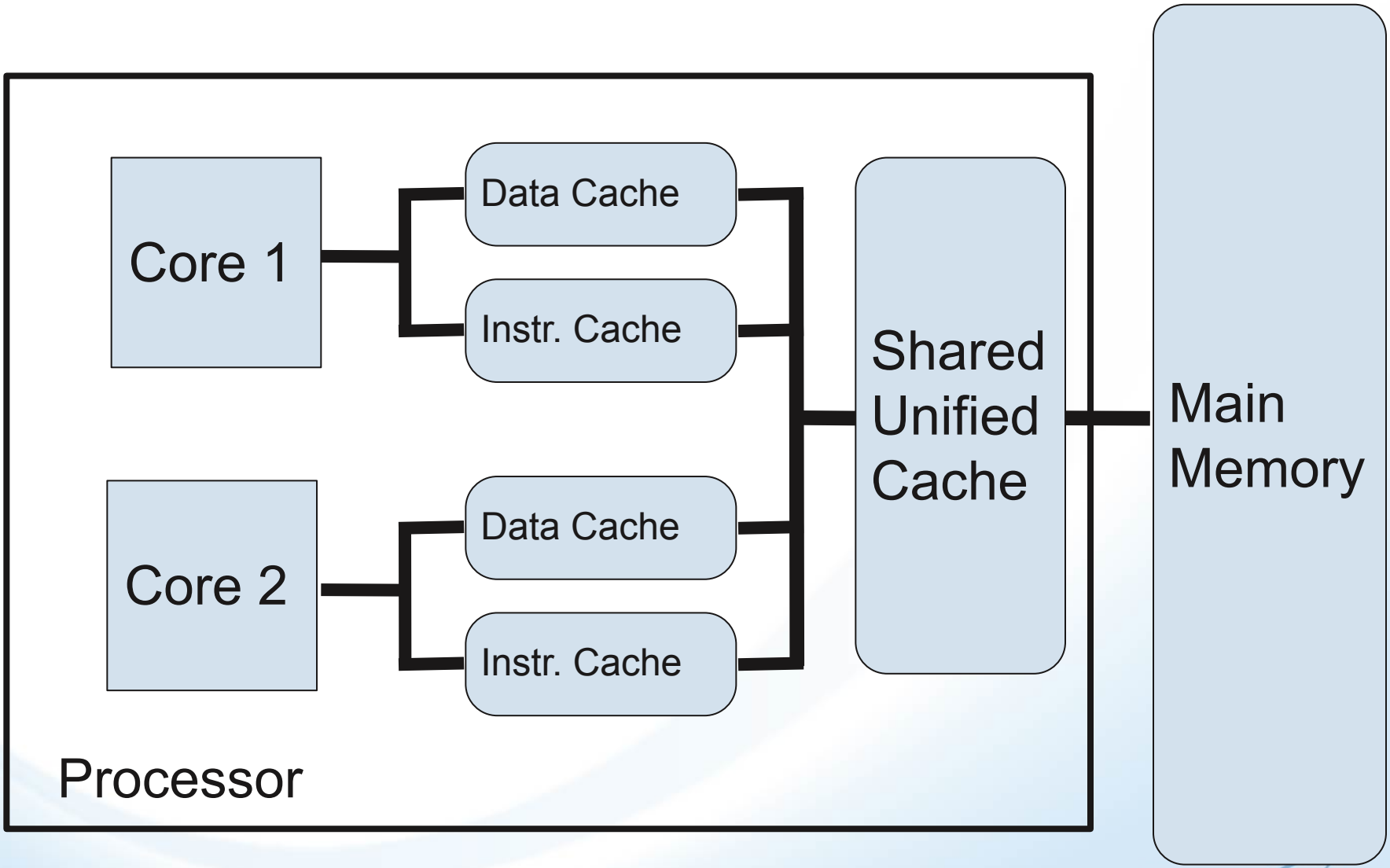


Enough to toast a bagel in 2010.

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..

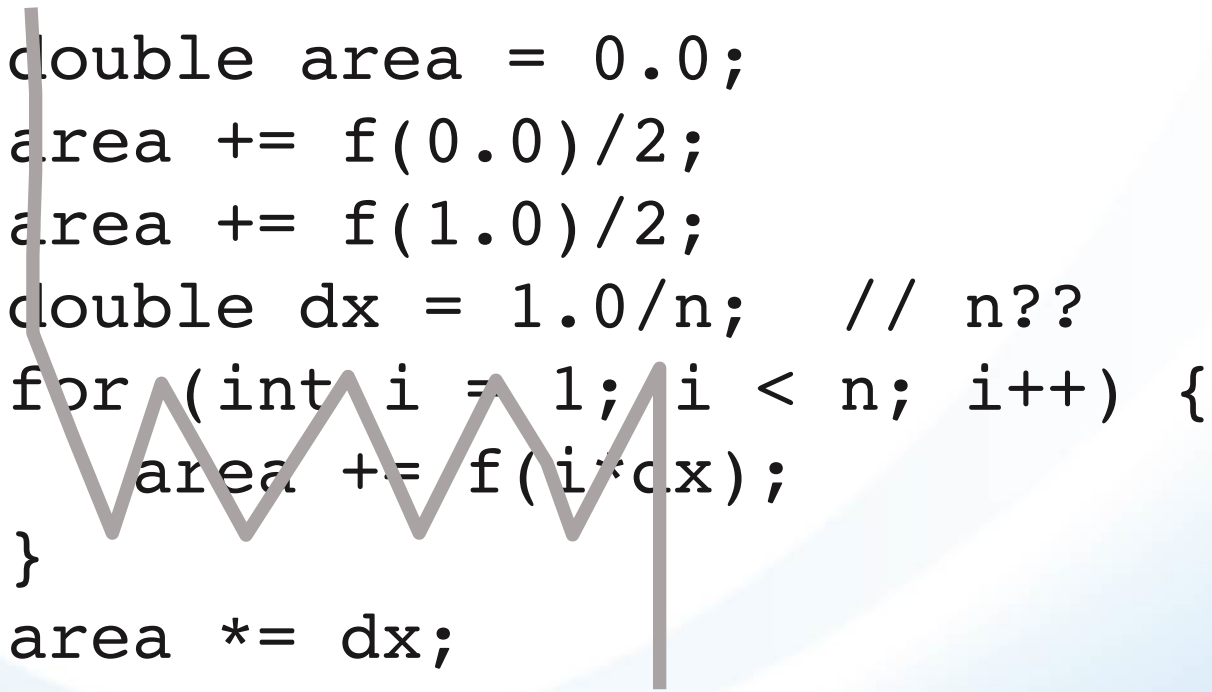
One Definition

- 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;
```

One Definition

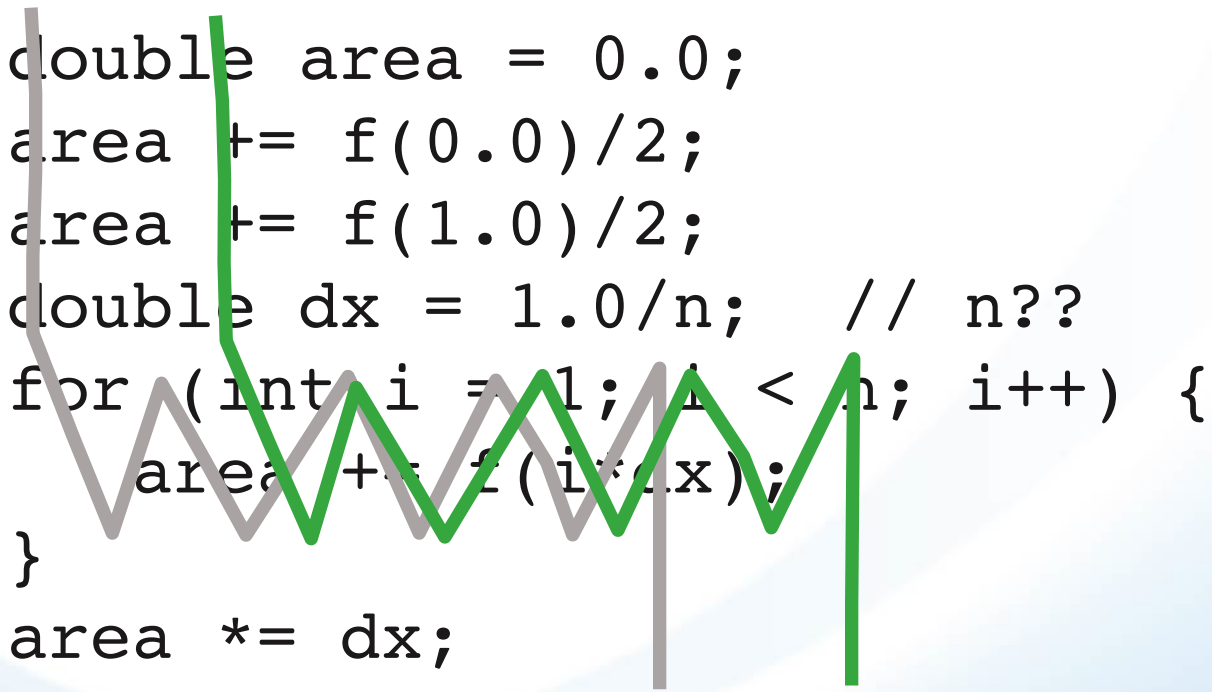
- A thread is a run time entity that is able to independently execute a stream of instructions (think Robot)



```
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;
```

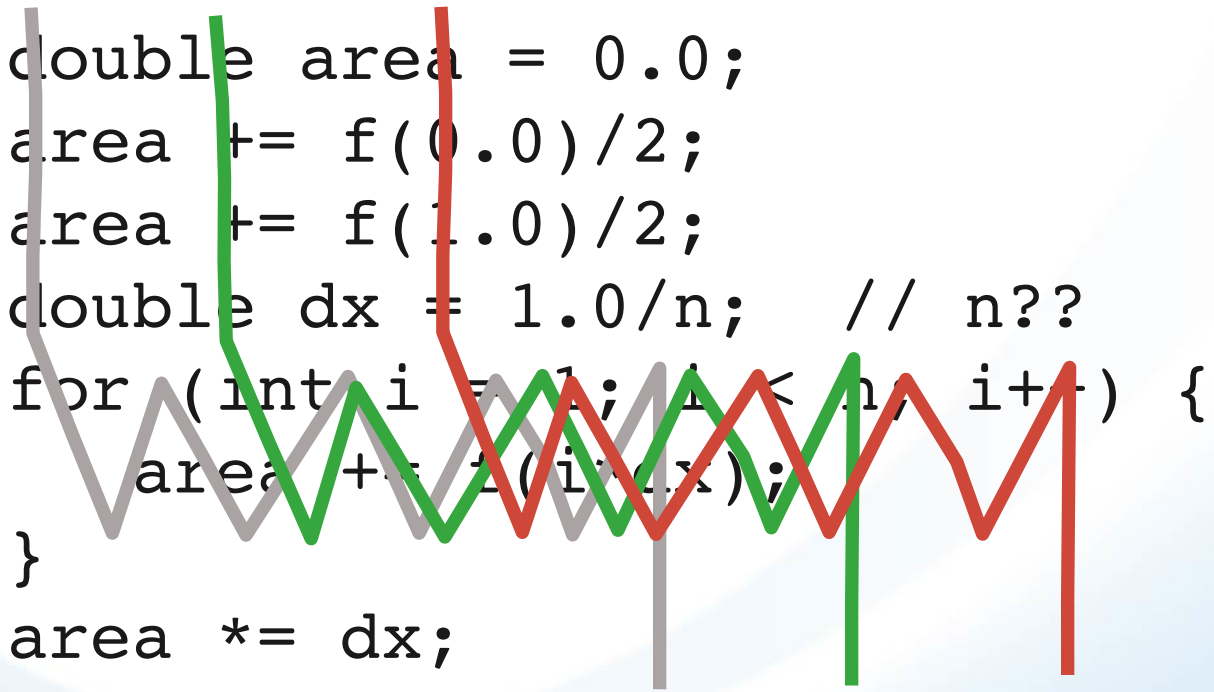
One Definition

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



One Definition

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



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.

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.

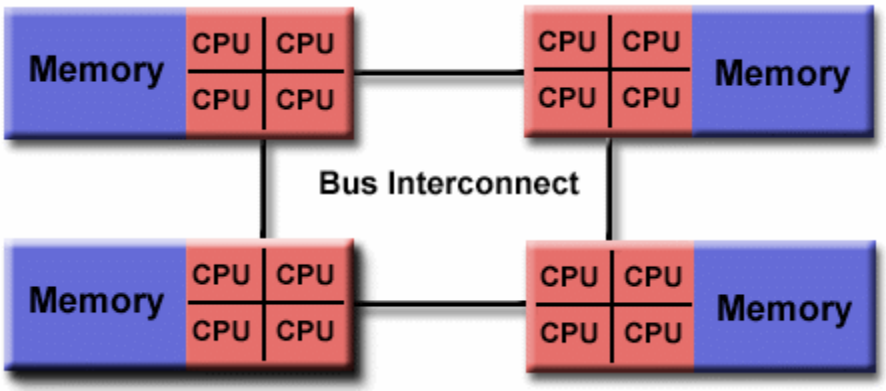
On a side note - in a Java Virtual Machine, it is almost trivial to create a thread but not so a process.

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, the 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.

OpenMP Defined

OpenMP is a Parallel Programming Model for Shared memory and distributed shared memory multiprocessors.



Tomorrow, you will learn about MPI which can cover this model, but more typically deals with systems where the CPUs and Memories are separated by network class interconnects.

Other architectures (CUDA) use more of a processor/ coprocessor arrangement.

OpenMP Concepts

OpenMP is not a computer language

Works in conjunction with C/C++ or Fortran

Comprised of compiler directives and supporting library

```
#pragma omp parallel      (in C)

!$omp parallel            (in Fortran)
```

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 ofter 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.

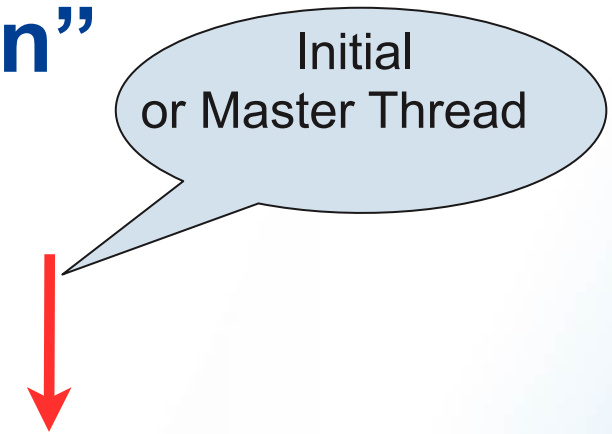
OMP's "Big Idea" - Execution Model Simple "fork/join"

```
// serial code

#pragma omp parallel ...
{
   // a block with regular C
   // and perhaps some
   // calls to omp functions
}
```

OMP's "Big Idea" - Execution Model Simple "fork/join"

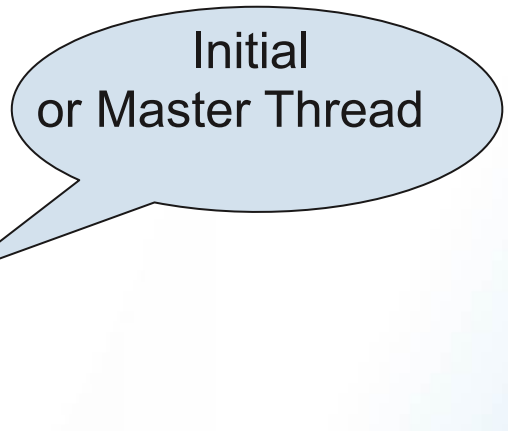
```
// serial code

#pragma omp parallel ...
{
   // a block with regular C
   // and perhaps some
   // calls to omp functions
}
```

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
}
```

OMP's "Big Idea" - Execution Model
Simple "fork/join"



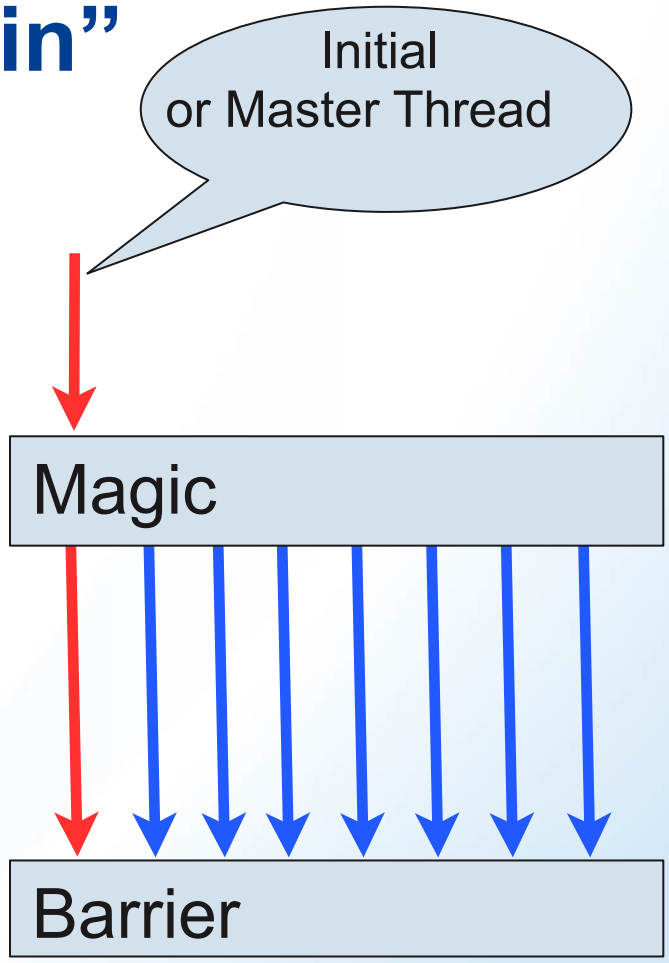

```
// serial code

#pragma omp parallel ...
{
   // a block with regular C
   // and perhaps some
   // calls to omp functions
}
```

Magic

Barrier

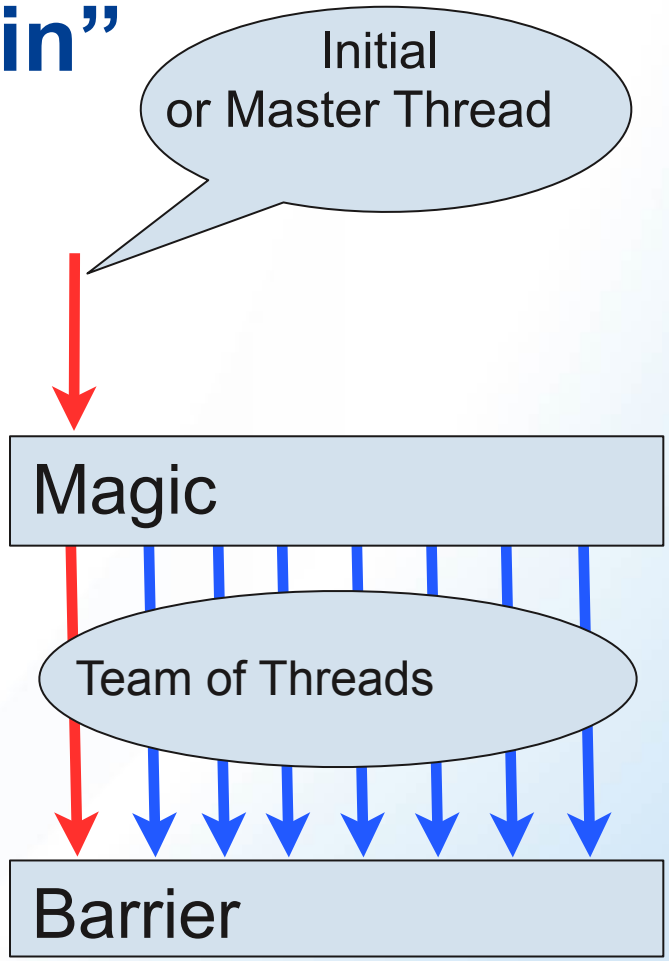
OMP's "Big Idea" - Execution Model
Simple "fork/join"

```
// serial code

#pragma omp parallel ...
{
   // a block with regular C
   // and perhaps some
   // calls to omp functions
}
```



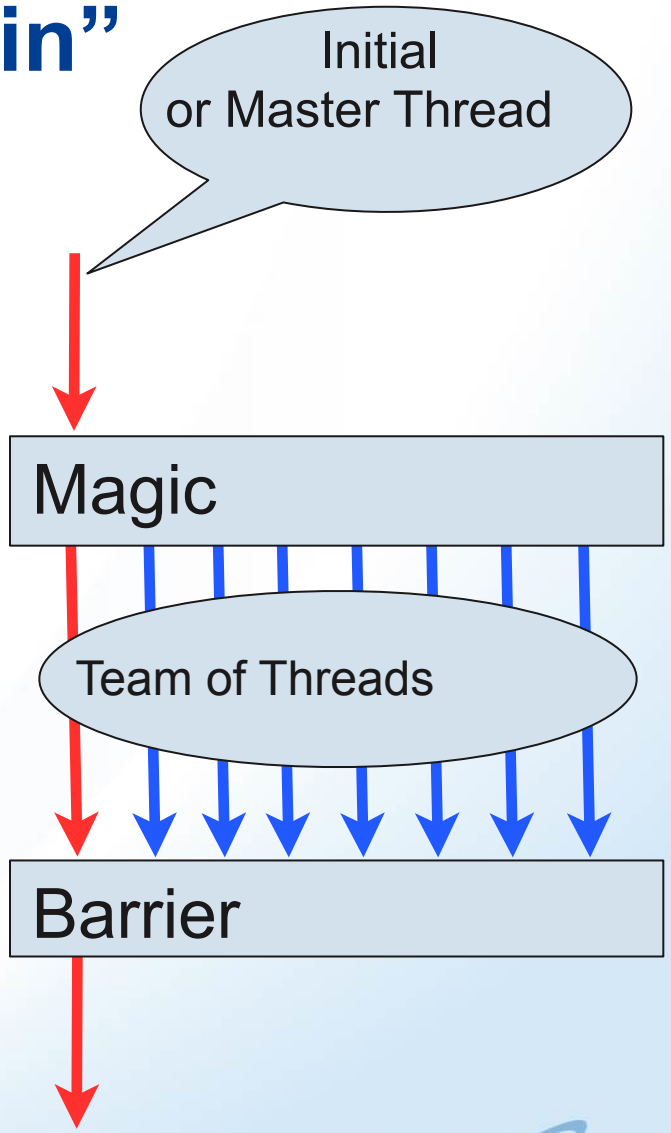
OMP's "Big Idea" - Execution Model
Simple "fork/join"

```
// serial code

#pragma omp parallel ...
{
   // a block with regular C
   // and perhaps some
   // calls to omp functions
}
```



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
}
```



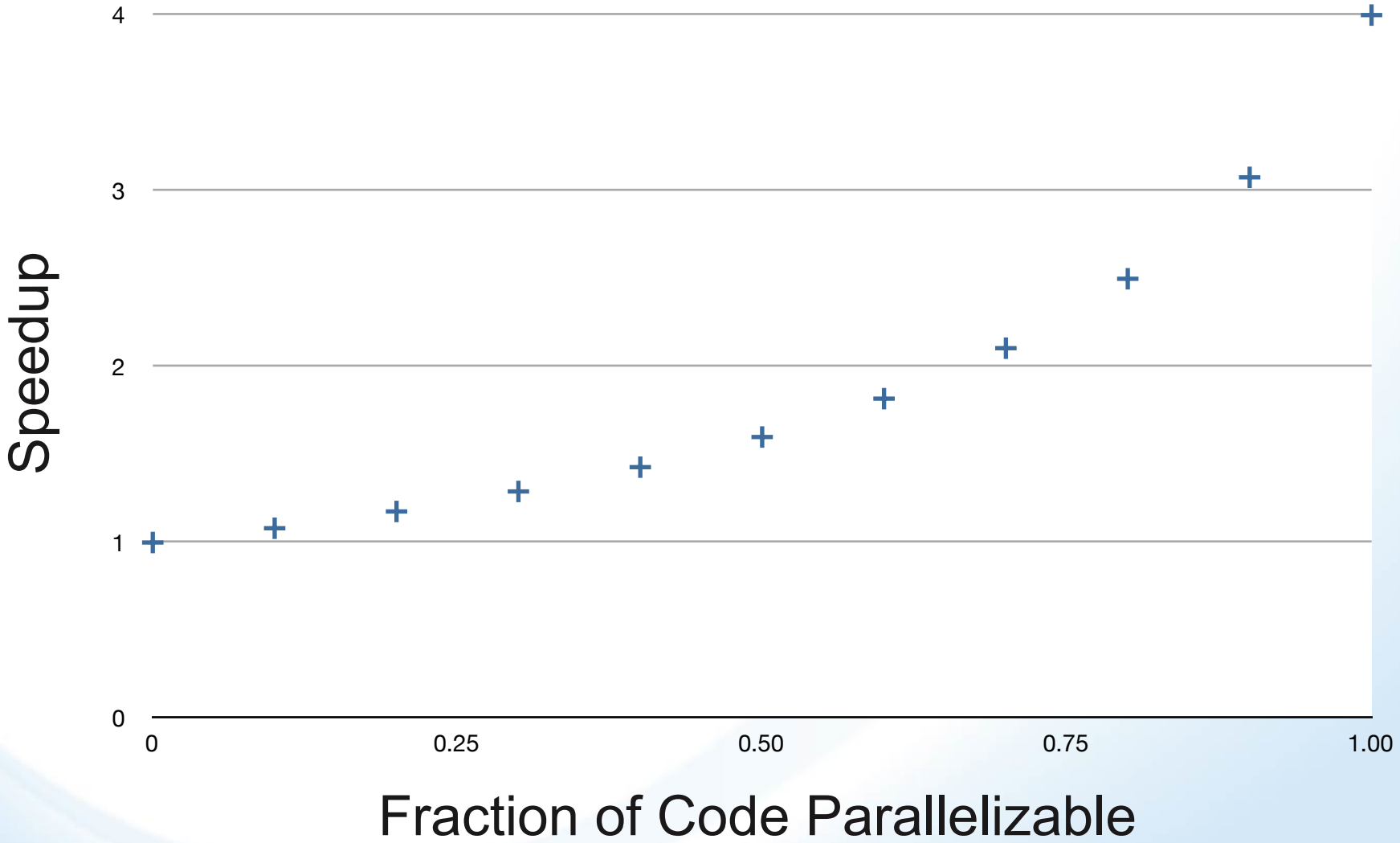
A Quick Reality Check

- 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 algorithms 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



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!

Programming Tip

- It is really easy to think you are running a program using OpenMP when in fact you are running a serial executable.
- 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!

```
#ifdef _OPENMP
   printf("#ifdef _OPENMP
   printf("You are running OpenMP version %d\n",
        _OPENMP);
#else
   printf("You are not running OpenMP");
#endif
```

Serial Hello World

In C:

```
#include <stdio.h>

int main()
{
  printf("hello world\n");
  return 0;
}
```

Exercise 1 - Parallel Hello World Prerequisites

- Environment:
  - export OMP_NUM_THREADS=4
- Includes
  - #include <omp.h>
- Directive:
  - #pragma parallel omp
- 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
  - Make sure your compiler supports OpenMP!!!
  - -fopenmp -Wall

Exercise 1 - The Problem

- Write a parallel Hello World that outputs “N” lines of “Hello World from Thread # [n]“ where N is the number of threads you create using the above OpenMP Includes, Directives and functions and environment values.
- Explorations:
  - Try running without the environment variable (OMP_NUM_THREADS) set
  - Try varying the number of threads via environment
  - Try misspelling omp in the directive
  - Try compiling without the -fopenmp

Parallel Hello World:

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

int main()
{

#pragma omp parallel
  {
    int threadID = omp_get_thread_num();
    printf("%s %d\n", "hello parallel world from thread #",
                  threadID);
  }

  return 0;
}
```

Other Ways to Specify # of Threads

- Environment
  - export OMP_NUM_THREADS=#
- Function call (permitting one to react to data)
  - omp_set_num_threads(int)
- A clause on an #pragma omp parallel
  - num_threads(#)

Exercise 1a - The Problem (Closer to real world...)

- You have a program that has to read in two large files as part of the initialization. The files each contain 1,000,000 lines of double precision values. When you read the first file, million1.dat, you are to compute the mean and variance of the data in the array. When you read the second file, million2.dat, you are to compute the minimum and maximum values.
  - Use my serial program that does this (for reference)
  - Using just what you know now, write a parallel program that does this. Make sure your answer matches.
  - Open question - how to write a single program that performs this tasks, but will work both serially and as a parallel program

Serial Solution (1 of 2)

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

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

   // read in million1 and determine the mean and variance of
   // the data within.

   FILE *f = fopen("million1.dat","r");
   double sum = 0, sumSquared = 0;
   for (int i = 0; i < 1000000; i++) {
      double x;
      fscanf(f,"%le",&x);
      sum += x;
      sumSquared += x*x;
   }
   fclose(f);
```

Serial Solution (2 of 2)

```
  double mean = sum/1000000;
  double var = (sumSquared/1000000) - mean*mean;
  printf("mean = %e variance = %e\n",mean, var);

  // read in million2 and determine the max and min
  // of the data within.

  f = fopen("million2.dat","r");
  double max = -1.0, min = 1.0;
  for (int i = 0; i < 1000000; i++) {
     double x;
     fscanf(f,"%le",&x);
     if (x < min) min = x;
     else if (x > max) max = x;
  }
  printf("max = %e min = %e\n",max,min);
}
```

Parallel Solution (1 of 3)

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

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

   // read in million1 and determine the mean and variance of
   // the data within.


#pragma omp parallel
  {
   FILE *f;
```

Parallel Solution (2 of 3)

```
if (omp_get_thread_num() == 0) {
   f = fopen("million1.dat","r");
   double sum = 0, sumSquared = 0;
   for (int i = 0; i < 1000000; i++) {
      double x;
      fscanf(f,"%le",&x);
      sum += x;
      sumSquared += x*x;
   }
   fclose(f);
   double mean = sum/1000000;
   double var = (sumSquared/1000000) - mean*mean;
   printf("mean = %e variance = %e\n",mean, var);

}
```

Parallel Solution (3 of 3)

```
  } else if (omp_get_thread_num() == 1) {

     // read in million2 and determine the max and min
     // of the data within.

     f = fopen("million2.dat","r");
     double max = -1.0, min = 1.0;
     for (int i = 0; i < 1000000; i++) {
        double x;
        fscanf(f,"%le",&x);
        if (x < min) min = x;
        else if (x > max) max = x;
     }
     printf("max = %e min = %e\n",max,min);
  }
 }
}
```

But for Sequential Equivalence... 1/2

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      {
         FILE *f = fopen("million1.dat","r");
         double sum = 0, sumSquared = 0;
         for (int i = 0; i < 1000000; i++) {
            double x;
            fscanf(f,"%le",&x);
            sum += x;
            sumSquared += x*x;
         }
         fclose(f);
         double mean = sum/1000000;
         double var = (sumSquared/1000000) - mean*mean;
         printf("mean = %e variance = %e\n",mean, var);
      }
```

But for Sequential Equivalence... 2/2

```
   // read in million2 and determine the max and min
   // of the data within.

   #pragma omp section
   {
      FILE *f = fopen("million2.dat","r");
      double max = -1.0, min = 1.0;
      for (int i = 0; i < 1000000; i++) {
         double x;
         fscanf(f,"%le",&x);
         if (x < min) min = x;
         else if (x > max) max = x;
      }
      printf("max = %e min = %e\n",max,min);
   }
}
```

Work-Sharing Construct - Section

- #pragma omp sections/#pragma opm section are a simple means of gaining some control over the threads.
  - Upon entering the parallel region (#pragma omp parallel), when a sections directive is encountered, 1 thread is assigned/section. The rest remain idle.
  - The closing of the sections block is an implicit barrier.
  - This is equivalent to the serial code however the order of things may change (of course).

Other Work-Sharing constructs

- Just having “N” threads suddenly come to life and execute the same lines of code is ... not that useful
- A basic pattern of parallel programming is to take a loop and have each iteration of the loop (or perhaps some subset of the iterations) executed by a separate thread.
  - This requires that each iteration of the loop is independent - which means they can be executed in any order without affecting the answer.

Dependent vs Independent Iterations

Dependent vs Independent Iterations

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

Dependent vs Independent Iterations

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

Order dependent

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;
}
```

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

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]);
}
```

Dependent vs Independent Iterations

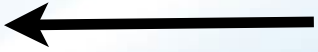
```
for (int i = 0; i < n; i++) {
   printf("%d\n",i);
}
```

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
}
```

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.

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
```

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.

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

to

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

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), (for i=1, n)

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.

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

```
#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.

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.

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

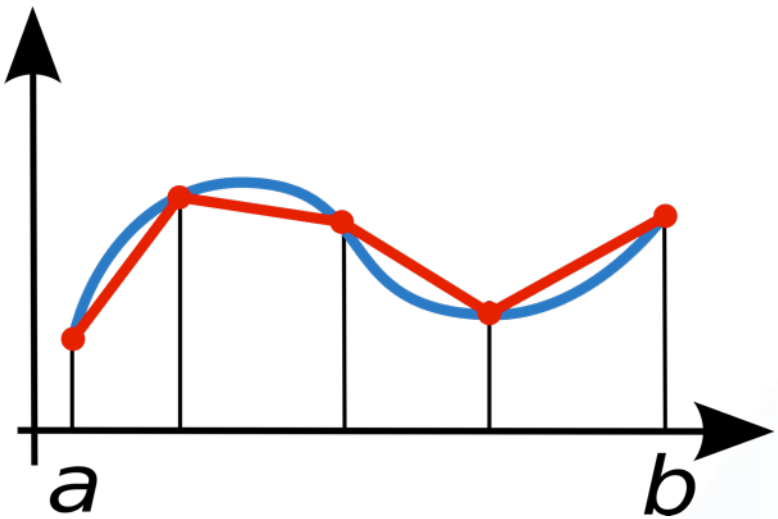
omp_get_num_threads returns the number of threads executing in the parallel region.

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



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

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

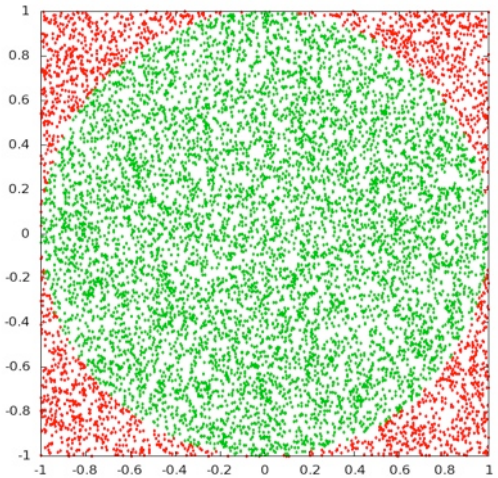
```
#pragma omp parallel shared(integral)
{

 double integral_priv = 0.0;

 #pragma omp for
     for(int i = 1; i <= N-1; i++) {
          double x = a+i*h;
          integral_priv = integral_priv + f(x);
     }

 #pragma omp critical
     integral = integral+integral_priv;

}

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

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

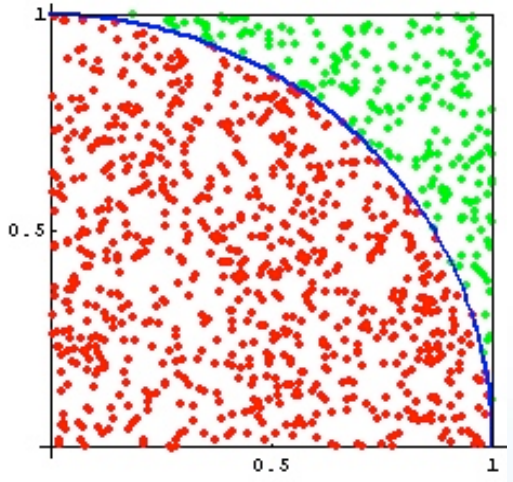
Trapezoid Rule Parallel Code in C Using Reduction Clause

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

 double integral_priv = 0.0;

 #pragma omp for
     for(int i = 1; i <= N-1; i++) {
        double x = a+i*h;
        integral_priv = integral_priv + f(x);
     }

     integral = integral+integral_priv;

}

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

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

Monte Carlo to Calculate Pi

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





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

```
#pragma omp threadprivate(seed)

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

#pragma omp parallel for private(x,y) reduction(+:sum)
  for(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);
```

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

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

Forward Difference - Parallel 2/5

```
// print - note cannot use parallel as order counts!

for(int i = 0; i < N; i++) {
      if(i%prune == 0)printf("%g\t%g\n", i*h, y[i]);
}

//
// The idea here - start off a bunch of threads. Most
// of them just deal with a contiguous range of values.
// Thread 0 does point 0 to N_local-1. Thread 1 does
// point N_local to 2*N_local-1. Thread 2 does
// 2*N_local to 3*N_local-1, ... Thread k does points
// from k*N_local to (k+1)*N_local-1. Have each of them
// compute their values as in the serial case, taking
// care to save the value needed for the last value in
// the previous group.
```

Forward Difference - Parallel 3/5

```
#pragma omp parallel
  {

  // every thread does this

    int N_local;
    double next;

  // N_local is the number of points each thread has
  // to deal with (last thread will pick up the extras)

    N_local = N/omp_get_num_threads();

  // now start and stop indices for each thread to
  // work on. The last thread picks up the extras
```

Forward Difference - Parallel 4/5

```
    int start = omp_get_thread_num()*N_local;
    int stop = start + N_local -1;
    if(omp_get_thread_num()==omp_get_num_threads()-1) {
       stop = stop + N%omp_get_num_threads();
       next = y[N];
    } else {
       next = y[start + N_local];
    }

  // ok, all threads meet here.

#pragma omp barrier

  // note - ordinary for, not parallel, but each
  // thread runs their own

    for(int i = start; i < stop ; i++)
        y[i] = (y[i+1]-y[i])/h;
```

Forward Difference - Parallel 5/5

```
 // next has been saved from the original

  y[stop] = (next - y[stop])/h;

}

 // end of parallel region

y[N-1] = y[N-2];// 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;
}
```

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!