PARALLEL COMPUTING AT THE DESKTOP

Aaron Smith – March 2015 GSPS

Outline

```
/usr/local
|---Why Parallel?
|---Closer Look @
|    |---Hardware
|    |---Software
|---Language Considerations
|---Parallel Paradigms
|---Example Code
     |---Serial
     |---MPI
     |---OpenMP
```

Why parallel?

>> Speed up code [processing power]

Slow is relative (minutes/days/months)

>> Share the workload [big/distributed data]

Big is relative (MB, GB, TB)

Amdahl's Law

>> Serial sections limit the parallel effectiveness

$$\text{Speedup} = \frac{1}{f_s + f_p / p}$$

f_s = serial fraction
f_p = parallel fraction
p = number of processors

What resources do you have?

Hardware

>> Know the basic architecture.

>> What exactly is multi-core?

CPU = Central Processing Unit

SMP = Simultaneous Multiprocessing

CMP = Chip-level Multiprocessing

Big pool of slower cache and
separate fast memory/cycles

SMT = Simultaneous Multithreading

e.g. quad-core, hyperthreaded processors

Effectively 2x4x2 – lower latency

>> Distributed and Shared Memory

What processor owns the data?

Race conditions and other problems

Communication overhead / bottlenecks

Software

>> Compilers are smart!

We don't have to try as hard.

>> Who's developing?

Open source community

Well-established standards

>> Version Control (git/hg)

>> Documentation

>> User-friendliness

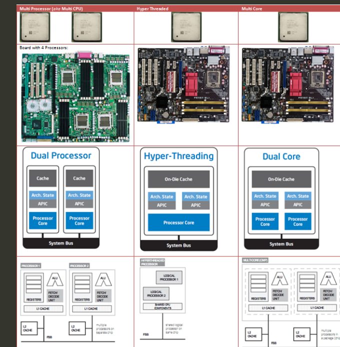
Unified codebase

Trustworthy

Unit Testing

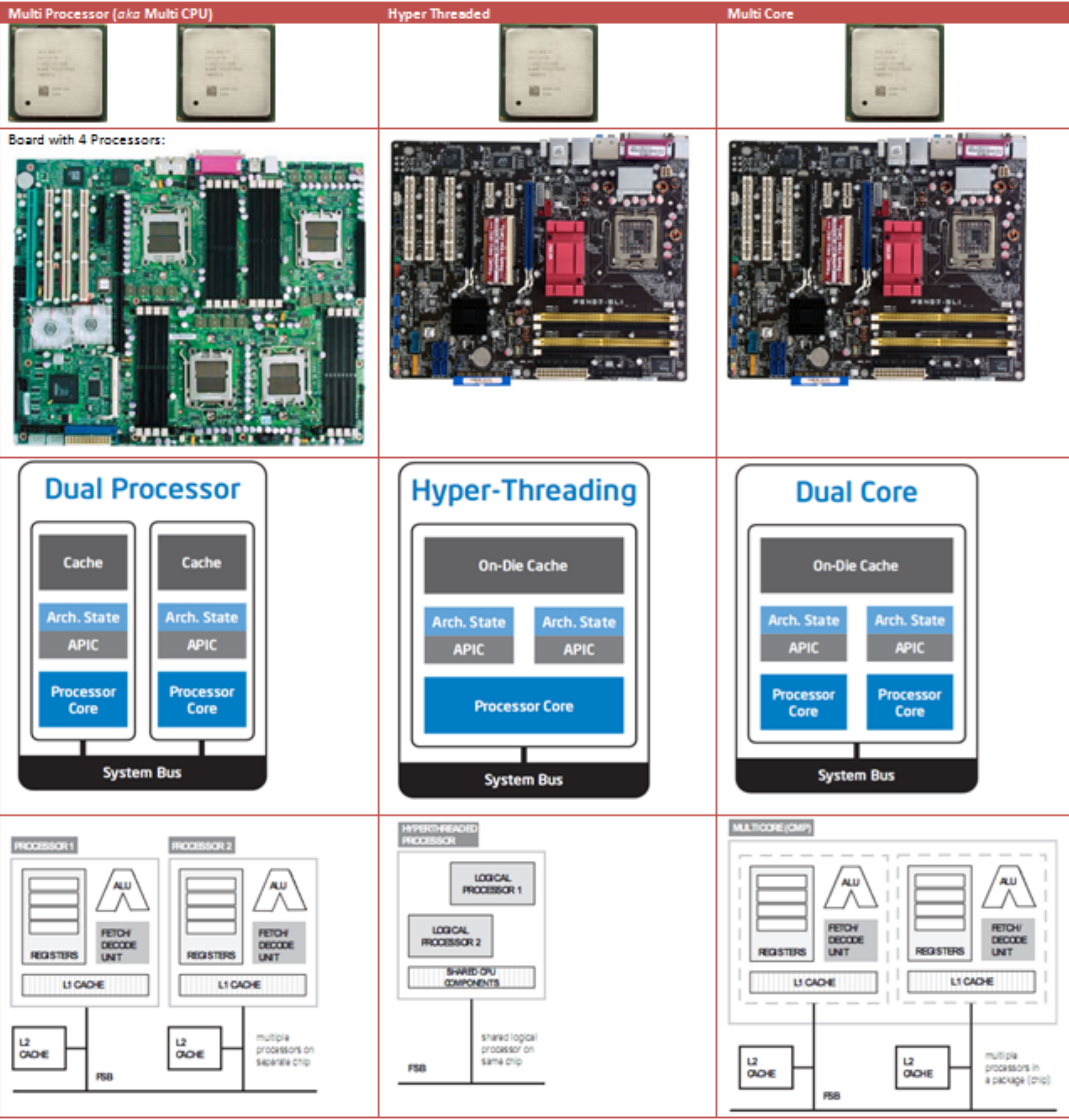
Installation

Languages…



What
Ha

>> Know the ba
>> What exactly
CPU = Central Pro
SMP = Simultaneo
CMP = Chip-level
Big pool of
separate fa
SMT = Simultaneo
e.g. quad-core, hy
Effectively
>> Distributed a
What processor o
Race conditions a
Communication o



The Language Landscape

Compiled vs. Interpreted

Compiled

>> C/C++ and FORTRAN

>> Code is reduced to machine-specific instructions (executable)

>> Faster runtimes, easy to optimize

>> Low-level access to data structures

>> Less flexible -- static types

Interpreted

>> Python, Java, C#, bash

>> Code is saved as written and must be translated at runtime.

>> Faster develop times

>> Convenient high-level functions

>> Extra freedom – dynamic types, type checking, extra information

>> Web-based applications (Java)

>> Ongoing development & support

Just-In-Time (JIT)

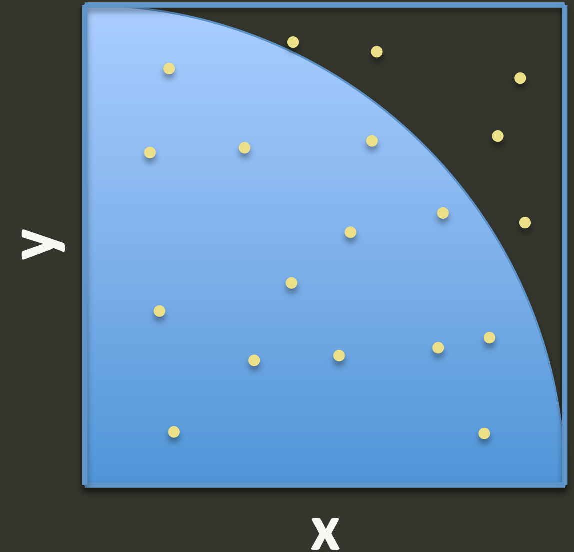
>> Julia – smart compiler, still under development, read the docs thoroughly to avoid pitfalls

Paradigms in Parallel Programming

1. Run several serial programs
   e.g. shell scripting – not processor or memory limited
2. Message-Passing Interface (MPI)
   STANDARD – "necessary" for large clusters and supercomputers
3. Open Multi Processing (OpenMP)
   STANDARD – incremental parallelization, easy, shared memory
4. Hybrid Programming
   Important enough to be it's own category – more memory & processors
5. Graphics Processing Units (GPU)
   Very efficient for certain kinds of operations but not everything
6. Useful but more obscure methods
   Native to languages, architecture-centric, many integrated cores (MIC) …

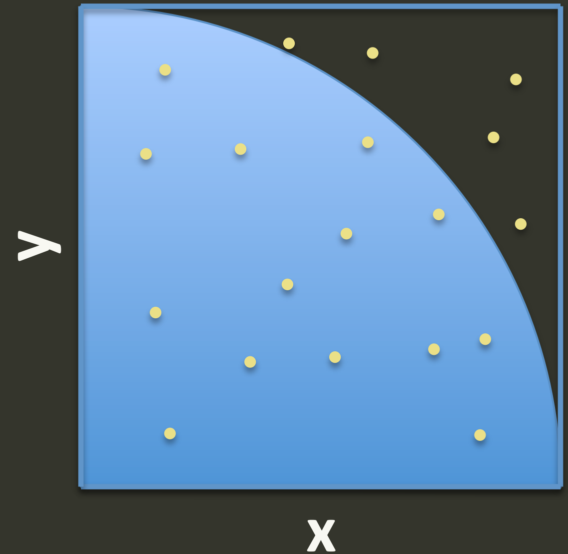
Example: MC integration

$$\pi = \frac{4 \times \#\,\text{Hits}}{\#\,\text{Attempts}}$$



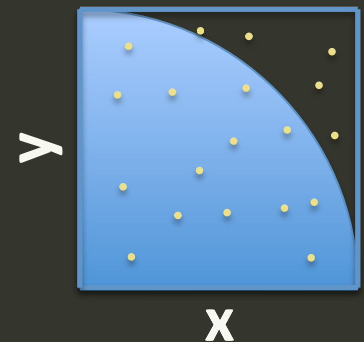
Example: Serial MC integration

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

int main (int argc, char* argv[])
{
    double x, y, r, pi;
    int i, count = 0, niter = 1e8;
    srand(time(NULL)); /* set random seed */

    /* main loop */
    for ( i = 0; i < niter; ++i )
    {
        /* get random points */
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;
        r = sqrt(x*x + y*y);

        /* check to see if point is in unit circle */
        if ( r <= 1 ) ++count;
    } /* end main loop */

    pi = 4.0 * ( (double)count / (double)niter );
    printf("Pi: %f\n", pi);  // p = 4(m/n)
    return 0;
}
```



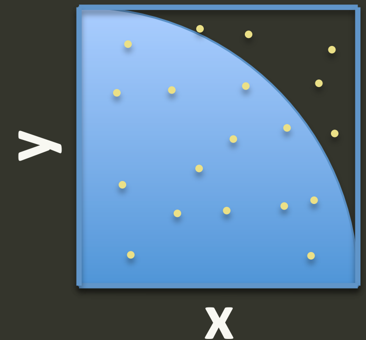
Example: Serial MC integration

```
#include <...>

int main (int argc, char* argv[])
{
    /* declare variables */




    srand(time(NULL)                        );     // random seed


    for ( i = 0; i < niter; ++i )
    { /* test if random points are in unit circle */ }



    pi = 4.0 * ( (double)count  / (double)niter  );

        printf("Pi: %f\n", pi);  // p = 4(m/n)


    return 0;
}
```



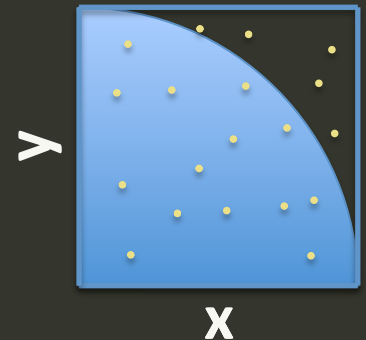
Example: MPI MC integration

```c
#include <...>
#include "mpi.h"

int main (int argc, char* argv[])
{
    /* declare variables */
    int my_rank, process;
    int total_count, total_niter;
    MPI_Init(&argc, &argv);                        // start MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  // get rank and
    MPI_Comm_size(MPI_COMM_WORLD, &process);  // number of processes
    srand(time(NULL)*(my_rank+17887527));     // random seed


    for ( i = 0; i < niter; ++i )
    { /* test if random points are in unit circle */ }

    /* reduce count and niter totals */
    MPI_Reduce(&count, &countT, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&niter, &niterT, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    pi = 4.0 * ( (double)countT / (double)niterT );
    if ( !my_rank ) /* root */
        printf("Pi: %f\n", pi);  // p = 4(m/n)

    MPI_Finalize();
    return 0;
}
```



Example: OpenMP MC integration

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

int main (int argc, char* argv[])
{
    /* declare variables */

  #pragma omp parallel
  {
    int my_rank = omp_get_thread_num();
    int process = omp_get_num_threads();
    srand(time(NULL)*(my_rank+17887527));     // random seed

    #pragma omp for private(x, y, r, i) reduction(+:count)
    for ( i = 0; i < niter; ++i )
    { /* test if random points are in unit circle */ }
  }



    pi = 4.0 * ( (double)count  / (double)niter  );

        printf("Pi: %f\n", pi);  // p = 4(m/n)


    return 0;
}
```



Summary

```
/.Trashes
|---Likely number of cores
|     on your desktop: 4
|---Likely number of cores
|     on local cluster: 16+
|---Is the effort worth it?
|    |---Many codes have already
|          done the work for you.
|---Additional resources
     |----TACC
     |----Fellow students
```