Parallel Programming using OpenMP

Qin Liu

The Chinese University of Hong Kong

Overview

Why Parallel Programming?

Overview of OpenMP

Core Features of OpenMP

More Features and Details...

One Advanced Feature

Introduction

- OpenMP is one of the most common parallel programming models in use today

Introduction

- OpenMP is one of the most common parallel programming models in use today
- It is relatively easy to use which makes a great language to start with when learning to write parallel programs

Introduction

- OpenMP is one of the most common parallel programming models in use today
- It is relatively easy to use which makes a great language to start with when learning to write parallel programs
- Assumptions:

Introduction

- OpenMP is one of the most common parallel programming models in use today
- It is relatively easy to use which makes a great language to start with when learning to write parallel programs
- Assumptions:
  - We assume you know C++ (OpenMP also supports Fortran)

Introduction

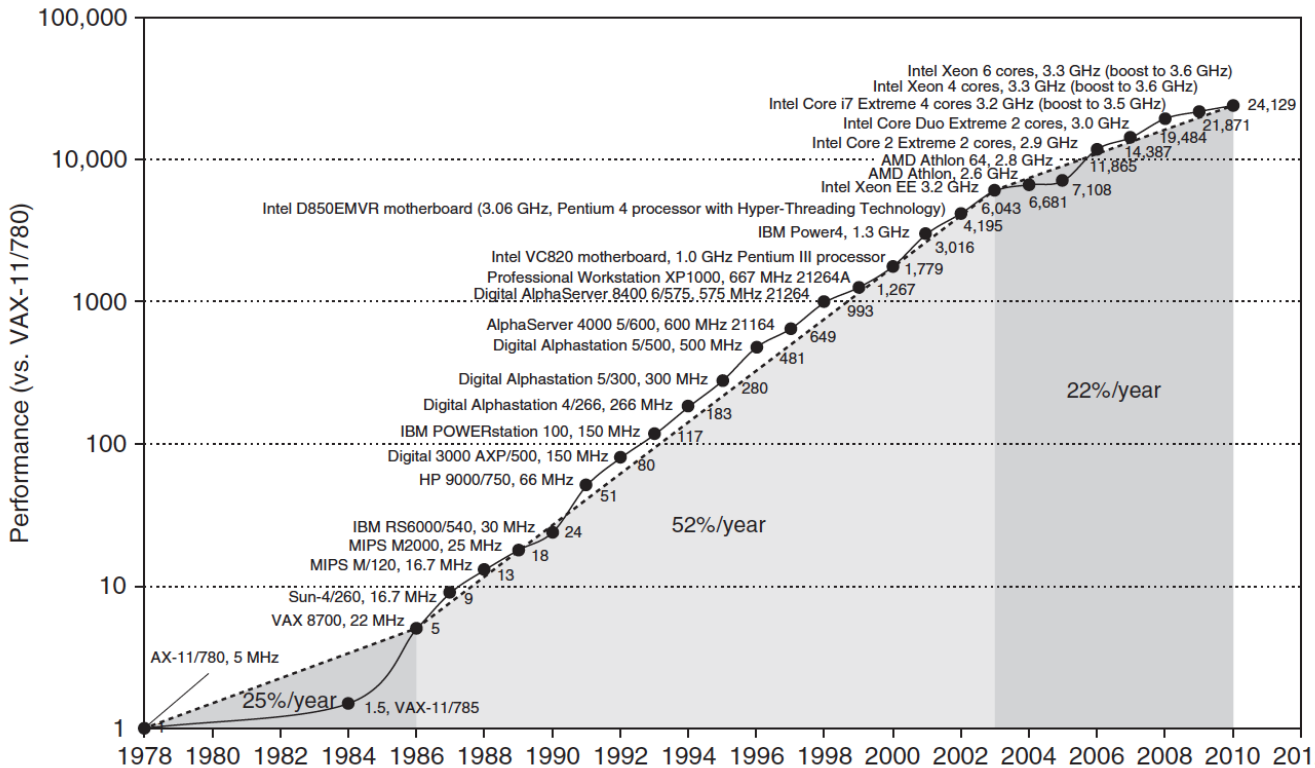
- OpenMP is one of the most common parallel programming models in use today
- It is relatively easy to use which makes a great language to start with when learning to write parallel programs
- Assumptions:
  - We assume you know C++ (OpenMP also supports Fortran)
  - We assume you are new to parallel programing

Introduction

- OpenMP is one of the most common parallel programming models in use today
- It is relatively easy to use which makes a great language to start with when learning to write parallel programs
- Assumptions:
  - We assume you know C++ (OpenMP also supports Fortran)
  - We assume you are new to parallel programing
  - We assume you have access to a compiler that supports OpenMP (like gcc)

Why Parallel Programming?

Growth in processor performance since the late 1970s



Source: Hennessy, J. L., & Patterson, D. A. (2011). Computer architecture: a quantitative approach. Elsevier.

- Good old days: 17 years of sustained growth in performance at an annual rate of over 50%

The Hardware/Software Contract

- We (SW developers) learn and design sequential algorithms such as quick sort and Dijkstra's algorithm

The Hardware/Software Contract

- We (SW developers) learn and design sequential algorithms such as quick sort and Dijkstra's algorithm
- Performance comes from hardware

The Hardware/Software Contract

- We (SW developers) learn and design sequential algorithms such as quick sort and Dijkstra's algorithm
- Performance comes from hardware

Results: Generations of performance ignorant software engineers write serial programs using performance-handicapped languages (such as Java)... This was OK since performance was a hardware job

The Hardware/Software Contract

- We (SW developers) learn and design sequential algorithms such as quick sort and Dijkstra's algorithm
- Performance comes from hardware

Results: Generations of performance ignorant software engineers write serial programs using performance-handicapped languages (such as Java)... This was OK since performance was a hardware job

But...

The Hardware/Software Contract

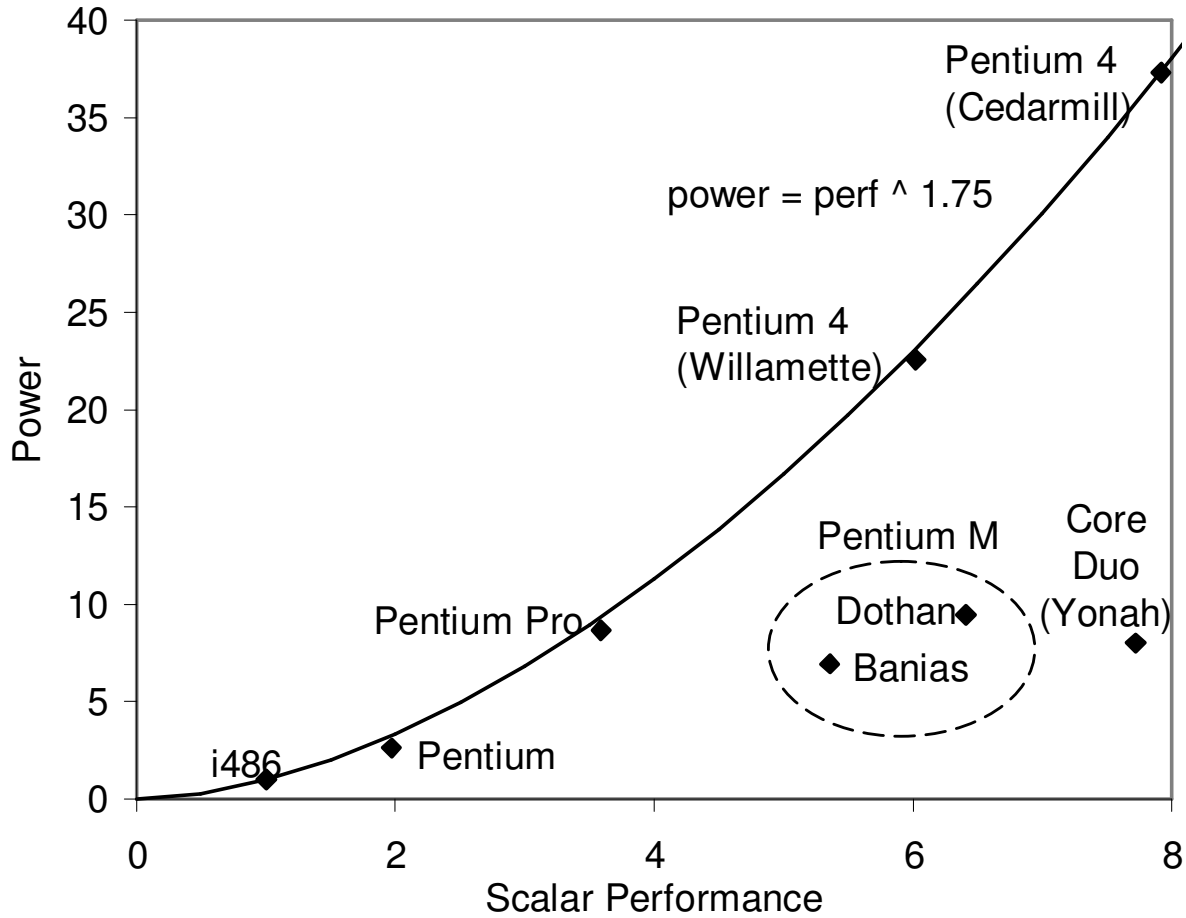
- We (SW developers) learn and design sequential algorithms such as quick sort and Dijkstra's algorithm
- Performance comes from hardware

Results: Generations of performance ignorant software engineers write serial programs using performance-handicapped languages (such as Java)... This was OK since performance was a hardware job

But...

- In 2004, Intel canceled its **high-performance uniprocessor projects** and joined others in declaring that the road to higher performance would be via **multiple processors per chip** rather than via **faster uniprocessors**

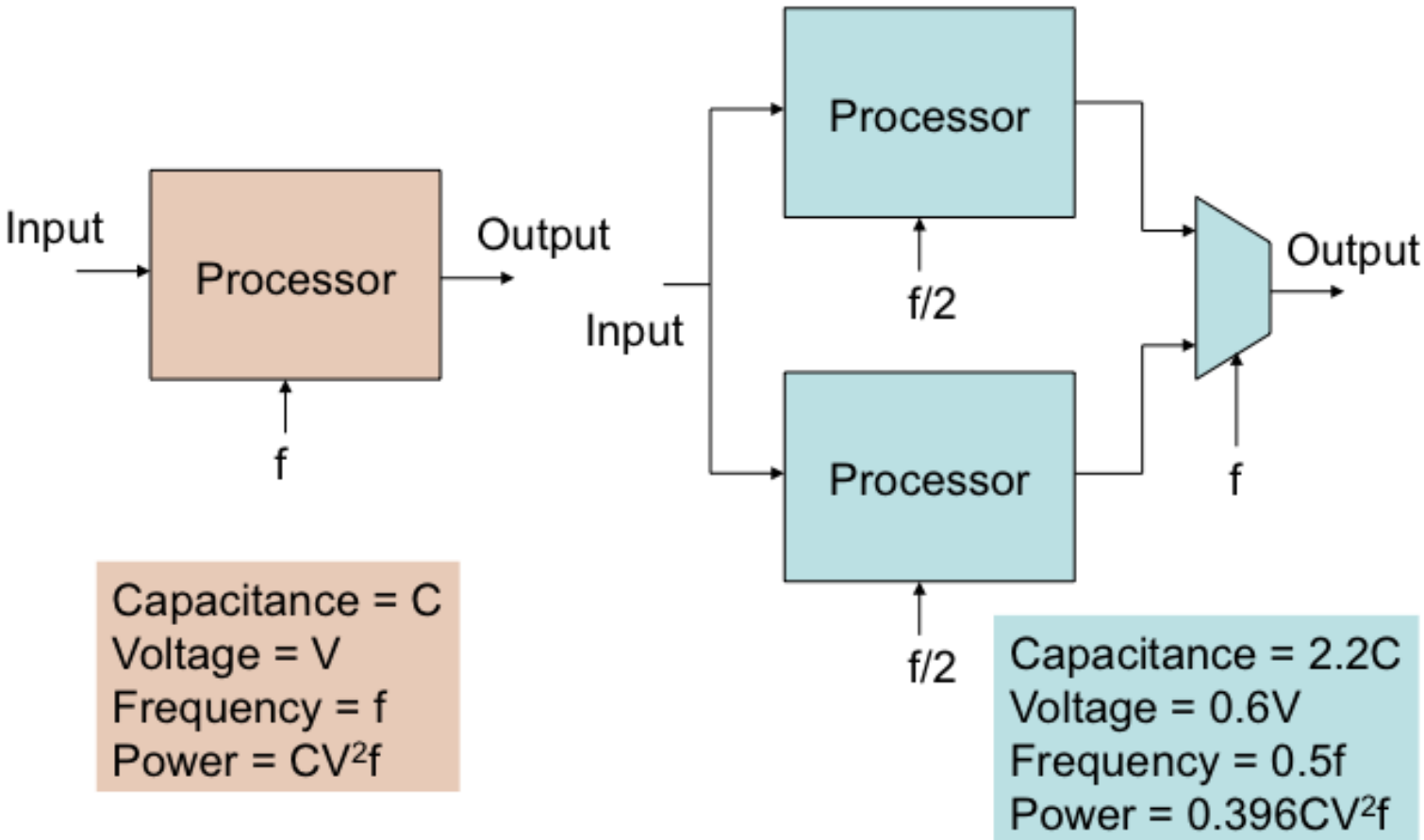
Computer Architecture and the Power Wall



Source: Grochowski, Ed, and Murali Annavaram. "Energy per instruction trends in Intel microprocessors." Technology@Intel Magazine 4, no. 3 (2006): 1-8.

- Growth in power is unsustainable (power $=$ perf$^{1.74}$)
- Partial solution: simple low power cores

The rest of the solution - Add Cores



Source: Multi-Core Parallelism for Low-Power Design - Vishwani D. Agrawal

Microprocessor Trends

Individual processors are **many core** (and often heterogeneous) processors from Intel, AMD, NVIDIA

Microprocessor Trends

Individual processors are **many core** (and often heterogeneous) processors from Intel, AMD, NVIDIA
A new HW/SW contract:

- HW people will do what's natural for them (lots of simple cores) and SW people will have to adapt (rewrite everything)

Microprocessor Trends

Individual processors are **many core** (and often heterogeneous) processors from Intel, AMD, NVIDIA
A new HW/SW contract:

- HW people will do what's natural for them (lots of simple cores) and SW people will have to adapt (rewrite everything)
- The problem is this was presented as an ultimatum... nobody asked us if we were OK with this new contract... which is kind of rude

Parallel Programming

Process:

1. We have a sequential algorithm
2. Split the program into tasks and identify shared and local data
3. Use some algorithm strategy to break dependencies between tasks
4. Implement the parallel algorithm in C++/Java/...

Can this process be automated by the compiler?
Unlikely... We have to do it manually.

Overview of OpenMP

OpenMP: Overview

OpenMP: an API for writing multi-threaded applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded programs in Fortran and C/C++
- Standardizes last 20 years of symmetric multiprocessing (SMP) practice

OpenMP Core Syntax

- Most of the constructs in OpenMP are compiler directives:
  `#pragma omp <construct> [clause1 clause2 ...]`
- Example:
  `#pragma omp parallel num_threads(4)`
- Include file for runtime library: `#include <omp.h>`
- Most OpenMP constructs apply to a "structured block"
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom

Exercise 1: Hello World

A multi-threaded "hello world" program

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main() {
4 #pragma omp parallel
5   {
6     int ID = omp_get_thread_num();
7     printf(" hello(%d)", ID);
8     printf(" world(%d)\n", ID);
9   }
10 }
```

Compiler Notes

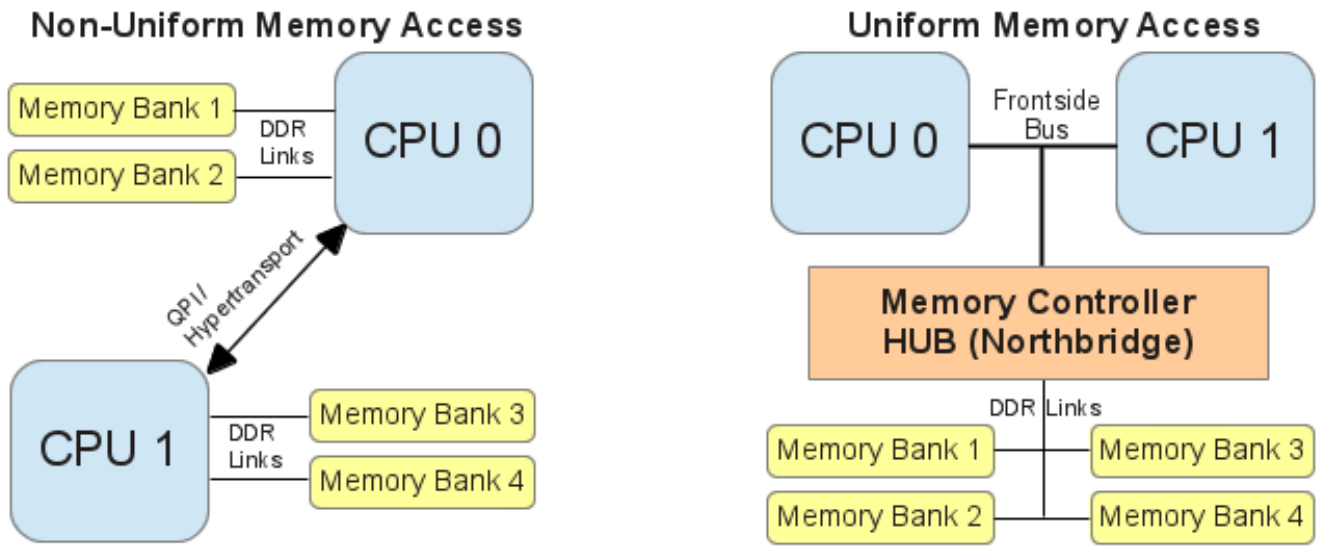
- On Windows, you can use Visual Studio C++ 2005 (or later) or Intel C Compiler 10.1 (or later)
- Linux and OS X with gcc (4.2 or later):

```
1 $ g++ hello.cpp -fopenmp # add -fopenmp to enable it
2 $ export OMP_NUM_THREADS=16 # set the number of threads
3 $ ./a.out # run our parallel program
```

- More information:
  `http://openmp.org/wp/openmp-compilers/`

Symmetric Multiprocessing (SMP)

- **A SMP system**: multiple identical processors connect to a single, shared main memory. Two classes:
  - **Uniform Memory Access (UMA)**: all the processors share the physical memory uniformly
  - **Non-Uniform Memory Access (NUMA)**: memory access time depends on the memory location relative to a processor



Source: `https://moinakg.wordpress.com/2013/06/05/findings-by-google-on-numa-performance/`

Symmetric Multiprocessing (SMP)

- SMP computers are everywhere... Most laptops and servers have multi-core multiprocessor CPUs

Symmetric Multiprocessing (SMP)

- SMP computers are everywhere... Most laptops and servers have multi-core multiprocessor CPUs
- The shared address space and (as we will see) programming models encourage us to think of them as UMA systems

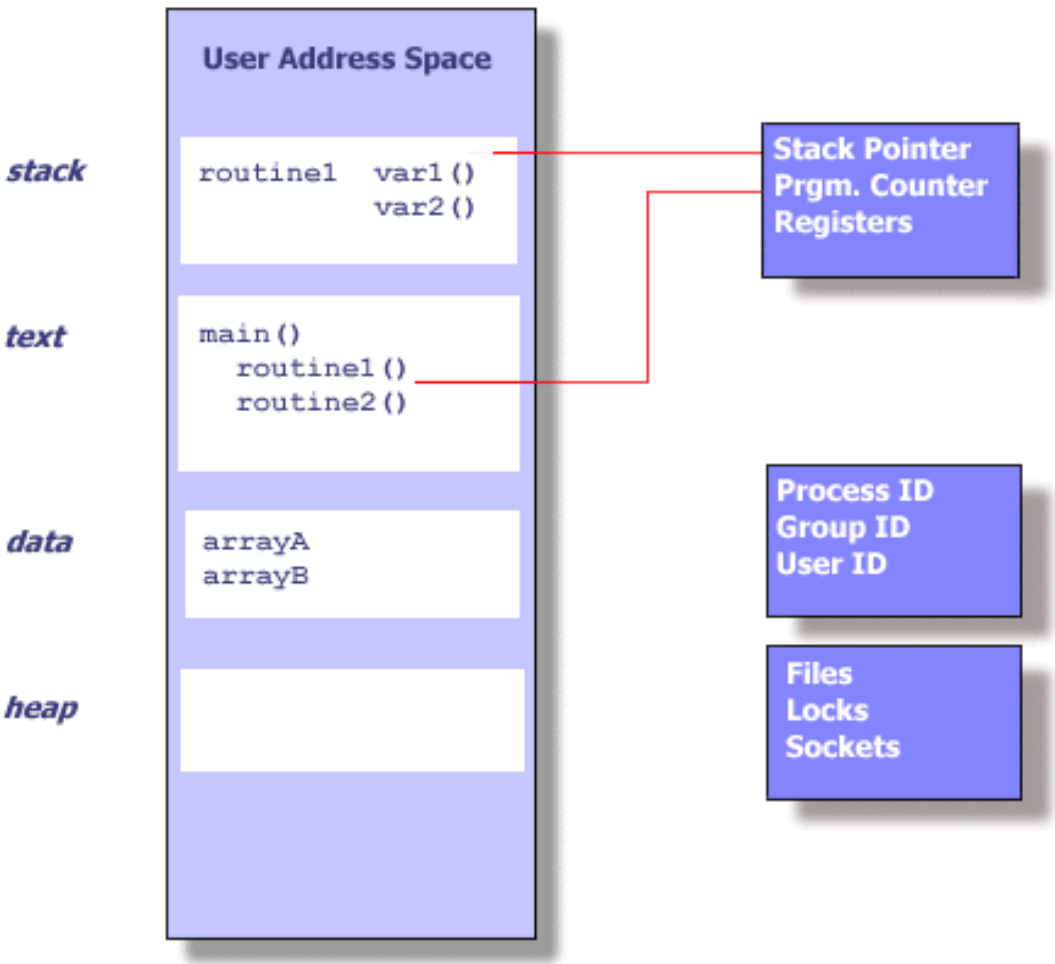
Symmetric Multiprocessing (SMP)

- SMP computers are everywhere... Most laptops and servers have multi-core multiprocessor CPUs
- The shared address space and (as we will see) programming models encourage us to think of them as UMA systems
- Reality is more complex... Any multiprocessor CPU with a cache is a NUMA system

Symmetric Multiprocessing (SMP)

- SMP computers are everywhere... Most laptops and servers have multi-core multiprocessor CPUs
- The shared address space and (as we will see) programming models encourage us to think of them as UMA systems
- Reality is more complex... Any multiprocessor CPU with a cache is a NUMA system
- Start out by treating the system as a UMA and just accept that much of your optimization work will address cases where that case breaks down

SMP Programming

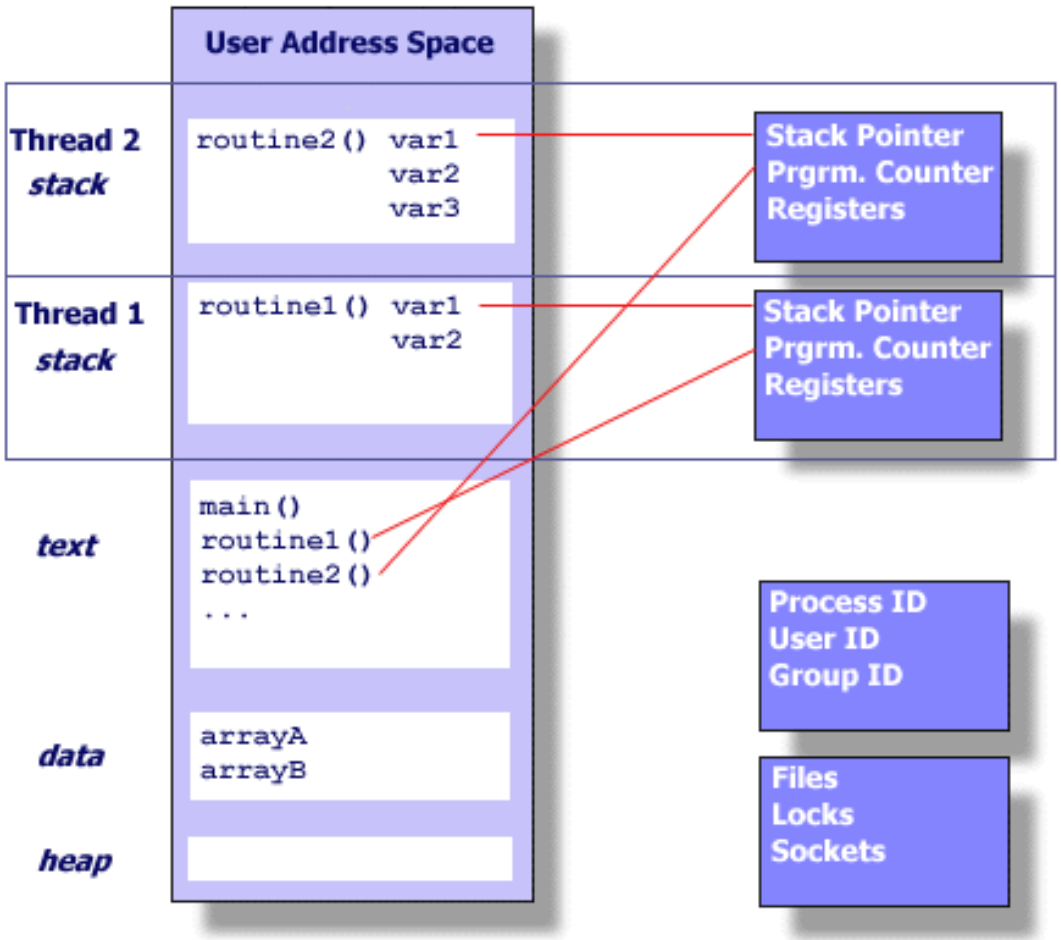


Source: `https://computing.llnl.gov/tutorials/pthreads/`

Process:

- an instance of a program execution
- contain information about program resources and program execution state

SMP Programming



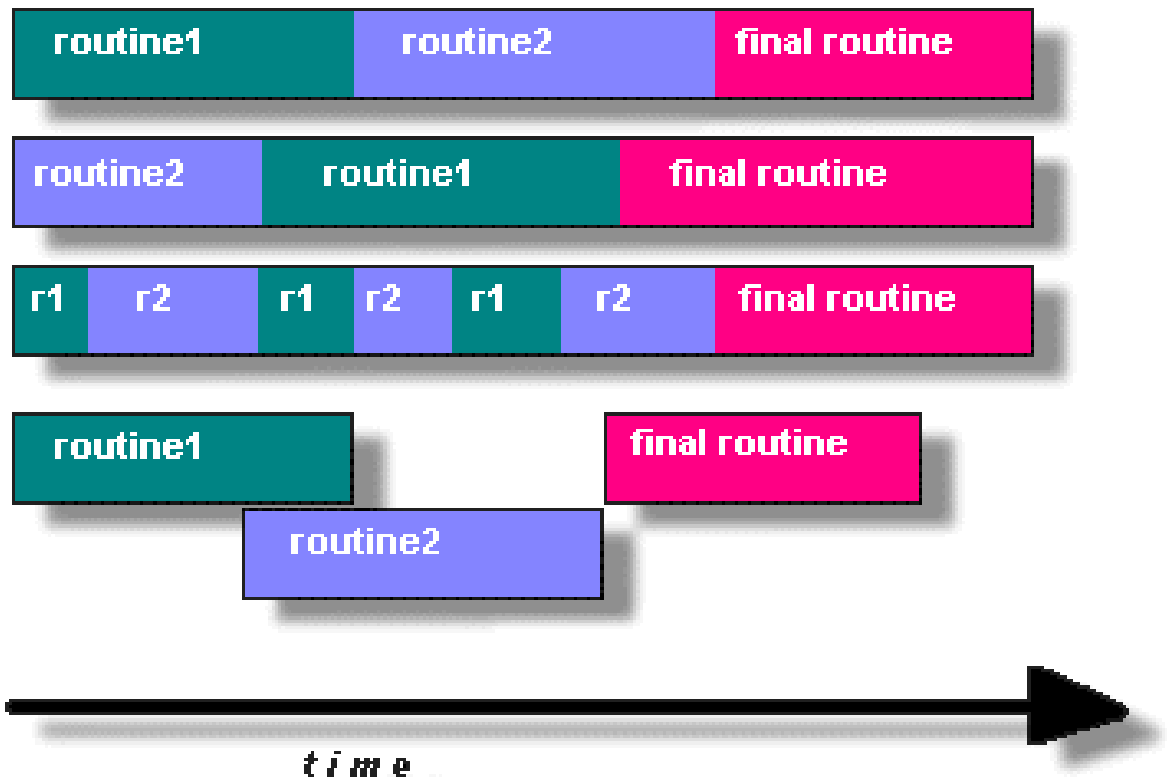
Source: `https://computing.llnl.gov/tutorials/pthreads/`

Threads:

- "light weight processes"
- share process state
- reduce the cost of swithcing context

Concurrency

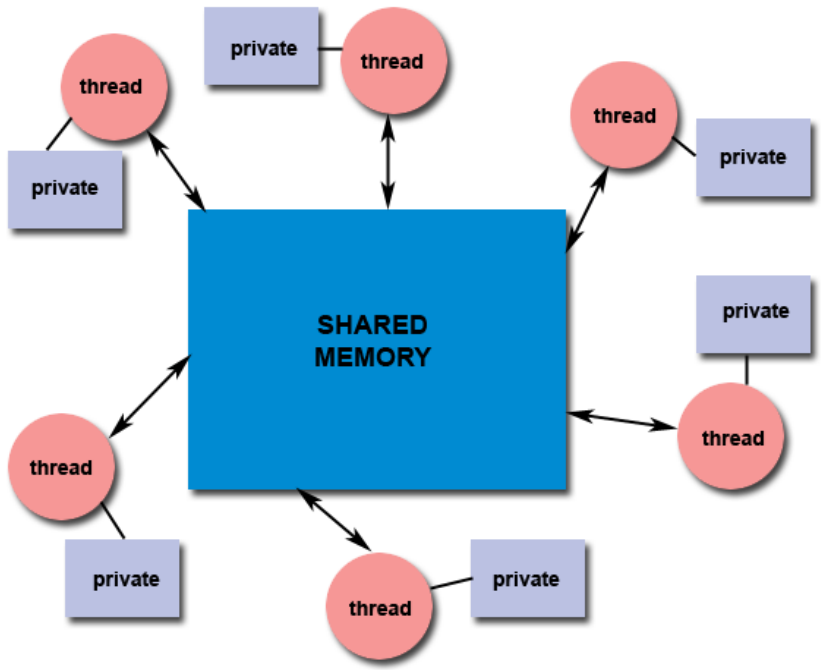
Threads can be interchanged, interleaved and/or overlapped in real time.



Source: `https://computing.llnl.gov/tutorials/pthreads/`

Shared Memory Model

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data



Source: `https://computing.llnl.gov/tutorials/pthreads/`

Exercise 1: Hello World

A multi-threaded "hello world" program

```
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel
  {
    int ID =
        omp_get_thread_num();
    printf(" hello(%d)", ID);
    printf(" world(%d)\n",
        ID);
  }
}
```

```
$ g++ hello.cpp -fopenmp
$ export OMP_NUM_THREADS=16
$ ./a.out
```

Sample Output:

```
hello(7) world(7)
hello(1) hello(9) world(9)
world(1)
hello(13) world(13)
hello(14) hello(4) hello(15)
    world(15)
world(4)
hello(2) world(2)
hello(10) world(10)
hello(11) world(11)
world(14)
hello(6) world(6)
hello(5) world(5)
hello(3) world(3)
hello(0) world(0)
hello(12) world(12)
hello(8) world(8)
```

Threads interleave and give different outputs every time

How Do Threads Interact in OpenMP?

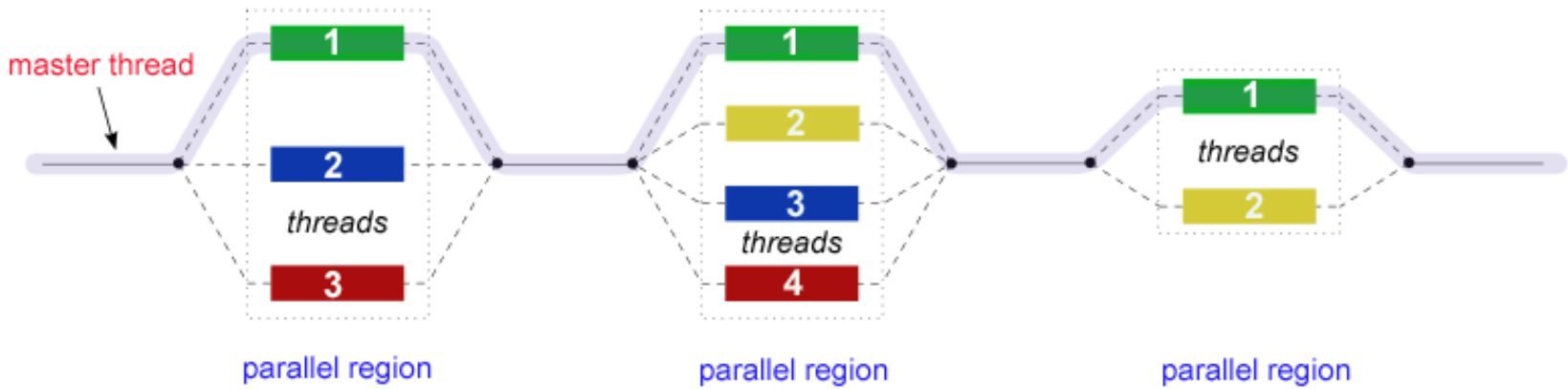
- OpenMP is a multi-threading, shared address model
  - Threads communicate by sharing variables
- Unintended sharing of data causes race conditions:
  - Race condition: when the program's outcome changes as the threads are scheduled differently
- To control race conditions:
  - Use synchronization to protect data conflicts
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization

Core Features of OpenMP

OpenMP Programming Model

Fork-Join Parallelism Model:

- OpenMP programs start with only the master thread
- FORK: the master thread creates a team of parallel threads when encounter a parallel region construct
- The parallel region construct are then executed in parallel
- JOIN: When the team threads complete, they synchronize and terminate, leaving only the master thread



Source: `https://computing.llnl.gov/tutorials/openMP/`

Thread Creation: Parallel Regions

- Create threads in OpenMP with the parallel construct
- For example, to create a 4-thread parallel region:

```
1 double A[1000];
2 omp_set_num_threads(4); // declared in omp.h
3 #pragma omp parallel
4 {
5   int ID = omp_get_thread_num();
6   pooh(ID,A);
7 }
8 printf("all done\n");
```

- Each thread calls `pooh(ID, A)` for `ID` from 0 to 3

Thread Creation: Parallel Regions

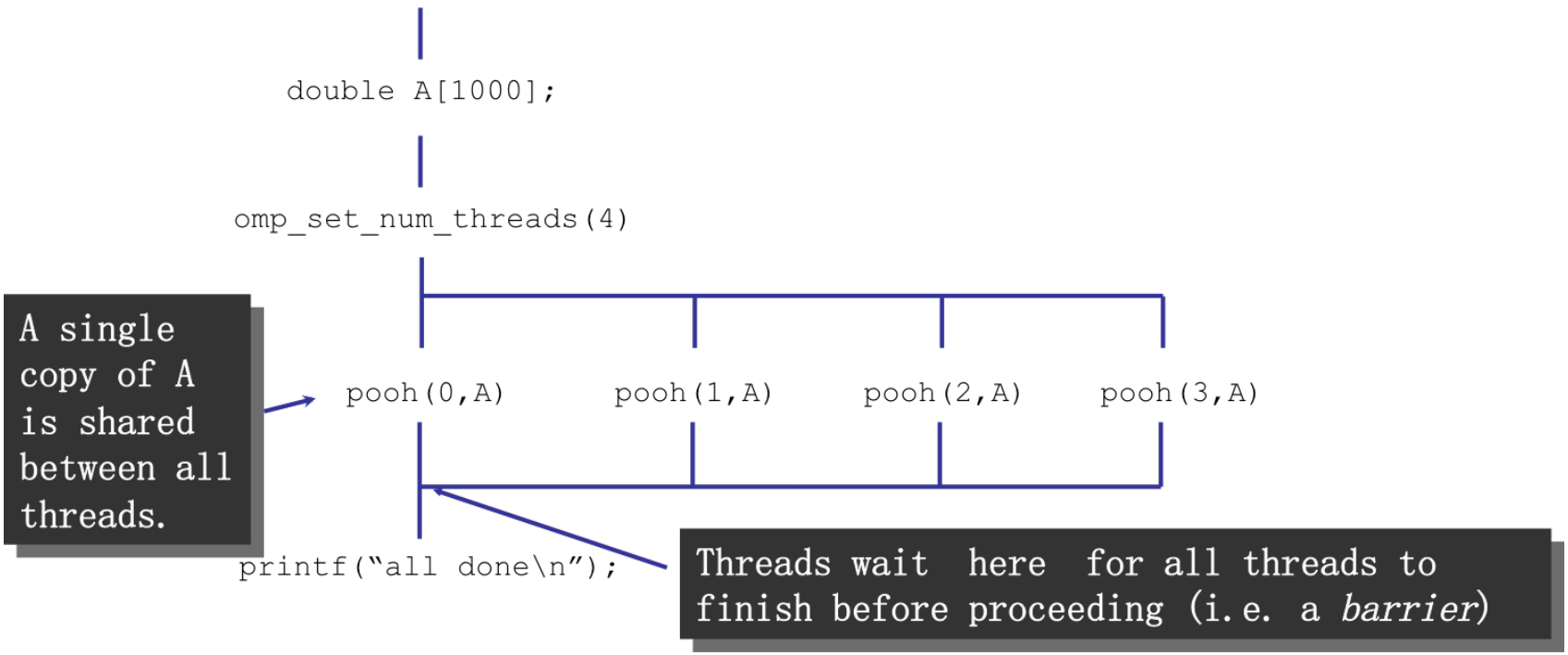
- Create threads in OpenMP with the parallel construct
- For example, to create a 4-thread parallel region:

```
1 double A[1000];
2 // specify the number of threads using a clause
3 #pragma omp parallel num_threads(4)
4 {
5   int ID = omp_get_thread_num();
6   pooh(ID,A);
7 }
8 printf("all done\n");
```

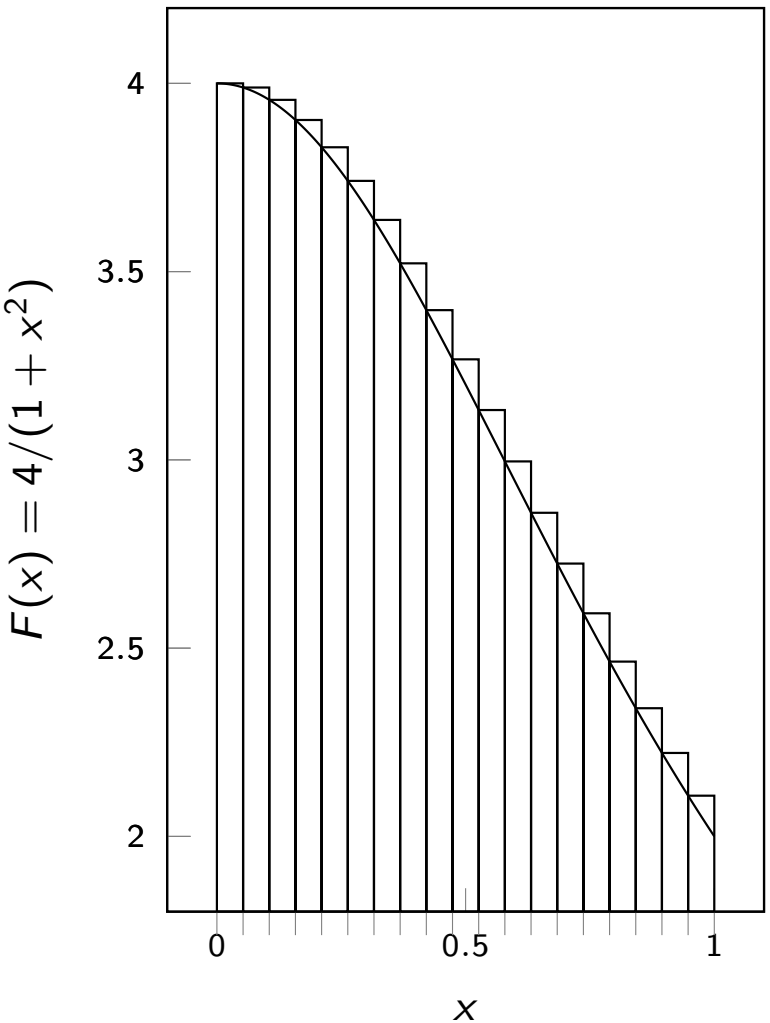
- Each thread calls `pooh(ID, A)` for `ID` from 0 to 3

Thread Creation: Parallel Regions

```
1 double A[1000];
2 omp_set_num_threads(4); // declared in omp.h
3 #pragma omp parallel
4 {
5   int ID = omp_get_thread_num();
6   pooh(ID,A);
7 }
8 printf("all done\n");
```



Compute π using Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial π Program

```
1 #include <stdio.h>
2
3 const long num_steps = 100000000;
4
5 int main () {
6     double sum = 0.0;
7     double step = 1.0 / (double) num_steps;
8
9     for (int i = 0; i < num_steps; i++) {
10        double x = (i+0.5) * step;
11        sum += 4.0 / (1.0 + x*x);
12    }
13
14    double pi = step * sum;
15
16    printf("pi is %f \n", pi);
17 }
```

Exercise 2: First Prallel π Program

- Create a parallel version of the pi program using a parallel construct
- Pay close attention to shared versus private variables
- In addition to a parallel construct, you will need the runtime library routines
  - `int omp_get_num_threads()`: number of threads in the team
  - `int omp_get_thread_num()`: ID of current thread

Exercise 2: First Prallel π Program

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3 const long num_steps = 100000000;
 4 #define NUM_THREADS 4
 5 double sum[NUM_THREADS];
 6 int main () {
 7     double step = 1.0 / (double) num_steps;
 8     omp_set_num_threads(NUM_THREADS);
 9 #pragma omp parallel
10     {
11         int id = omp_get_thread_num();
12         sum[id] = 0.0;
13         for (int i = id; i < num_steps; i += NUM_THREADS) {
14             double x = (i+0.5) * step;
15             sum[id] += 4.0 / (1.0 + x*x);
16         }
17     }
18     double pi = 0.0;
19     for (int i = 0; i < NUM_THREADS; i++)
20         pi += sum[i] * step;
21     printf("pi is %f \n", pi);
22 }
```

Algorithm Strategy: SPMD

The SPMD (single program, multiple data) technique:

- Run the same program on P processing elements where P can be arbitrarily large
- Use the rank (an ID ranging from 0 to $P - 1$) to select between a set of tasks and to manage any shared data structures

This pattern is very general and has been used to support most (if not all) parallel software.

Results

- Setup: gcc with no optimization on Ubuntu 14.04 with a quad-core Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz and 16 GB RAM
- The serial version ran in 1.25 seconds

Threads	1	2	3	4
SPMD	1.29	0.72	0.47	0.48

Results

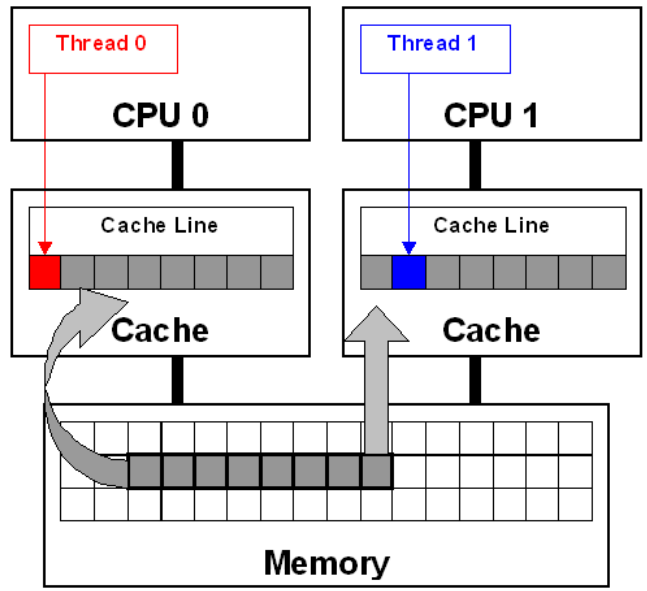
- Setup: gcc with no optimization on Ubuntu 14.04 with a quad-core Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz and 16 GB RAM
- The serial version ran in 1.25 seconds

Threads	1	2	3	4
SPMD	1.29	0.72	0.47	0.48

Why such poor scaling?

Reason: False Sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "slosh back and forth" between threads... This is called **"false sharing"**



Source: `https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads`

- Solution: pad arrays so elements you use are on distinct cache lines

Eliminate False Sharing by Padding

```
1 #include <stdio.h>
2 #include <omp.h>
3 const long num_steps = 100000000;
4 #define NUM_THREADS 4
5 #define PAD         8 // assume 64bbytes L1 cache
6 double sum[NUM_THREADS][PAD];
7 int main () {
8     double step = 1.0 / (double) num_steps;
9     omp_set_num_threads(NUM_THREADS);
10 #pragma omp parallel
11     {
12         int id = omp_get_thread_num();
13         sum[id][0] = 0.0;
14         for (int i = id; i < num_steps; i += NUM_THREADS) {
15             double x = (i+0.5) * step;
16             sum[id][0] += 4.0 / (1.0 + x*x);
17         }
18     }
19     double pi = 0.0;
20     for (int i = 0; i < NUM_THREADS; i++)
21         pi += sum[i][0] * step;
22     printf("pi is %f \n", pi);
23 }
```

Results

- Setup: gcc with no optimization on Ubuntu 14.04 with a quad-core Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz and 16 GB RAM
- The serial version ran in 1.25 seconds

Threads	1	2	3	4
SPMD	1.29	0.72	0.47	0.48
Padding	1.27	0.65	0.43	0.33

Do We Really Need to Pad Our Arrays?

- Padding arrays requires deep knowledge of the cache architecture
- Move to a machine with different sized cache lines and your software performance falls apart
- There has got to be a better way to deal with false sharing

High Level Synchronization

Recall: to control race conditions

- use synchronization to protect data conflicts

Synchronization: bringing one or more threads to a well defined and known point in their execution

- **Barrier**: each thread wait at the barrier until all threads arrive
- **Mutual exclusion**: define a block of code that only one thread at a time can execute

Synchronization: barrier

Barrier: each thread wait until all threads arrive

```
1 #pragma omp parallel
2 {
3   int id = omp_get_thread_num();
4   A[id] = big_calc1(id);
5 #pragma omp barrier
6   B[id] = big_calc2(id, A); // depend on A calculated by
        every thread
7 }
```

Synchronization: critical

Mutual exclusion: define a block of code that only one thread at a time can execute

```
1  float res;
2  #pragma omp parallel
3  {
4    float B; int i, id, nthrds;
5    id = omp_get_thread_num();
6    nthrds = omp_get_num_threads();
7    for (i = id; i < niters; i += nthrds) {
8      B = big_job(i);
9  #pragma omp critical
10     res += consume(B);
11     // only one at a time calls consume() and modify res
12   }
13 }
```

Synchronization: atomic

Atomic provides mutual exclusion but only applies to the update of a memory location and only supports `x += expr`, `x++`, `--x`...

```
1 float res;
2 #pragma omp parallel
3 {
4   float tmp, B;
5   B = big();
6   tmp = calc(B);
7 #pragma omp atomic
8   res += tmp;
9 }
```

Exercise 3: Rewrite π Program using Synchronization

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3 const long num_steps = 100000000;
 4 #define NUM_THREADS 4
 5 int main () {
 6     double step = 1.0 / (double) num_steps;
 7     omp_set_num_threads(NUM_THREADS);
 8     double pi = 0.0;
 9 #pragma omp parallel
10     {
11         int id = omp_get_thread_num();
12         double sum = 0.0; // local scalar not array
13         for (int i = id; i < num_steps; i += NUM_THREADS) {
14             double x = (i+0.5) * step;
15             sum += 4.0 / (1.0 + x*x); // no false sharing
16         }
17 #pragma omp critical
18         pi += sum * step; // must do summation here
19     }
20     printf("pi is %f \n", pi);
21 }
```

Results

- Setup: gcc with no optimization on Ubuntu 14.04 with a quad-core Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz and 16 GB RAM
- The serial version ran in 1.25 seconds

Threads	1	2	3	4
SPMD	1.29	0.72	0.47	0.48
Padding	1.27	0.65	0.43	0.33
Critical	1.26	0.65	0.44	0.33

Mutal Exclusion Done Wrong

Be careful where you put a critical section.

```
1 #include <stdio.h>
2 #include <omp.h>
3 const long num_steps = 100000000;
4 #define NUM_THREADS 4
5 int main () {
6     double step = 1.0 / (double) num_steps;
7     omp_set_num_threads(NUM_THREADS);
8     double pi = 0.0;
9 #pragma omp parallel
10    {
11        int id = omp_get_thread_num();
12        for (int i = id; i < num_steps; i += NUM_THREADS) {
13            double x = (i+0.5) * step;
14 #pragma omp critical
15            pi += 4.0 / (1.0 + x*x) * step;
16        }
17    }
18    printf("pi is %f \n", pi);
19 }
```

Ran in 10 seconds with 4 threads.

Example: Using Atomic

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3 const long num_steps = 100000000;
 4 #define NUM_THREADS 4
 5 int main () {
 6     double step = 1.0 / (double) num_steps;
 7     omp_set_num_threads(NUM_THREADS);
 8     double pi = 0.0;
 9 #pragma omp parallel
10     {
11         int id = omp_get_thread_num();
12         double sum = 0.0;
13         for (int i = id; i < num_steps; i += NUM_THREADS) {
14             double x = (i+0.5) * step;
15             sum += 4.0 / (1.0 + x*x);
16         }
17         sum *= step;
18 #pragma omp atomic
19         pi += sum;
20     }
21     printf("pi is %f \n", pi);
22 }
```

For Loop Worksharing

Serial π program:

```
1 #include <stdio.h>
2
3 const long num_steps = 100000000;
4
5 int main () {
6     double sum = 0.0;
7     double step = 1.0 / (double) num_steps;
8
9     for (int i = 0; i < num_steps; i++) {
10        double x = (i+0.5) * step;
11        sum += 4.0 / (1.0 + x*x);
12    }
13
14    double pi = step * sum;
15
16    printf("pi is %f \n", pi);
17 }
```

What we want to parallelize:

```
for (int i = 0; i < num_steps; i++)
```

For Loop Worksharing

Two equivalent directives:

```
1 double res[MAX];
2 #pragma omp parallel
3 {
4 #pragma omp for
5   for (int i = 0; i < MAX;
        i++)
6     res[i] = huge();
7 }
```

```
1 double res[MAX];
2 #pragma omp parallel for
3   for (int i = 0; i < MAX;
        i++)
4     res[i] = huge();
```

Working with For Loops

- Find computational intensive loops
- Make the loop iterations independent, so they can execute in any order
- Place the appropriate OpenMP directives and test

Working with For Loops

- Find computational intensive loops
- Make the loop iterations independent, so they can execute in any order
- Place the appropriate OpenMP directives and test

```
1 int j, A[MAX];
2 j = 5;
3 for (int i = 0; i < MAX; i++)
      {
4   j += 2;
5   A[i] = big(j);
6 }
```

Each iteration depends on the preivous one.

Working with For Loops

- Find computational intensive loops
- Make the loop iterations independent, so they can execute in any order
- Place the appropriate OpenMP directives and test

```
1 int j, A[MAX];
2 j = 5;
3 for (int i = 0; i < MAX; i++)
      {
4   j += 2;
5   A[i] = big(j);
6 }
```

Each iteration depends on the preivous one.

```
1 int A[MAX];
2 #pragma omp parallel for
3 for (int i = 0; i < MAX; i++)
      {
4   int j = 5 + 2*(i+1);
5   A[i] = big(j);
6 }
```

Remove dependency and j is now local to each iteration.

The Schdule Clause

The schedule clause affects how loop iterations are mapped onto threads

- `schedule(static, [chunk])`: each thread independently decides which which iterations of size "chunk" they will process
- `schedule(dynamic, [chunk])`: each thread grabs "chunk" iterations off a queue until all iterations have been handled
- guided, runtime, auto: skip...

Nested Loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```
1 #pragma omp parallel for collapse(2)
2 for (int i = 0; i < N; i++) {
3   for (int j=0; j<M; j++) {
4     .....
5   }
6 }
```

- Will form a single loop of length $N \times M$ and then parallelize that
- Useful if N is O(num of threads) so parallelizing the outer loop makes balancing the load difficult

Reduction

- How do we handle this case?

```
1 double ave=0.0, A[MAX];
2 for (int i = 0; i < MAX; i++) {
3   ave += A[i];
4 }
5 ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave). There is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation. It is called a "reduction"
- Support for reduction operations is included in most parallel programming environments such as MapReduce and MPI

Reduction

- OpenMP reduction clause: `reduction(op:list)`
- Inside a parallel or a worksharing construct:
  - A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+")
  - Updates occur on the local copy
  - Local copies are reduced into a single value and combined with the original global value
- The variables in "list" must be shared in the enclosing parallel region

```
1 double ave=0.0, A[MAX];
2 #pragma omp parallel for reduction(+:ave)
3 for (int i = 0; i < MAX; i++) {
4   ave += A[i];
5 }
6 ave = ave/MAX;
```

Exercise 4: π Program with Parallel For

```
1 #include <stdio.h>
2 #include <omp.h>
3 const long num_steps = 100000000;
4 #define NUM_THREADS 4
5 int main () {
6     double sum = 0.0;
7     double step = 1.0 / (double) num_steps;
8     omp_set_num_threads(NUM_THREADS);
9 #pragma omp parallel for reduction(+:sum)
10    for (int i = 0; i < num_steps; i++) {
11        double x = (i+0.5) * step;
12        sum += 4.0 / (1.0 + x*x);
13    }
14    double pi = step * sum;
15    printf("pi is %f \n", pi);
16 }
```

Quite simple...

Results

- Setup: gcc with no optimization on Ubuntu 14.04 with a quad-core Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz and 16 GB RAM
- The serial version ran in 1.25 seconds

Threads	1	2	3	4
SPMD	1.29	0.72	0.47	0.48
Padding	1.27	0.65	0.43	0.33
Critical	1.26	0.65	0.44	0.33
For	1.27	0.65	0.43	0.33

More Features and Details...

Synchronization: barrier

Barrier: each thread wait until all threads arrive

```
1 #pragma omp parallel
2 {
3   id=omp_get_thread_num();
4   A[id] = big_calc1(id);
5 #pragma omp barrier
6 #pragma omp for
7   for (int i = 0; i < N; i++) {
8     C[i] = big_calc3(i, A);
9   } // implicit barrier at the end of a for workesharing
        construct
10 #pragma omp for nowait
11   for (int i = 0; i < N; i++) {
12     B[i] = big_calc2(C, i);
13   } // no implicit barrier due to nowait
14   A[id] = big_calc4(id);
15 } // implicit barrier at the end of a parallel region
```

Master Construct

- The master construct denotes a structured block that is only executed by the master thread
- The other threads just skip it (no synchronization is implied)

```
1 #pragma omp parallel
2 {
3   do_many_things();
4 #pragma omp master
5   { exchange_boundaries(); }
6 #pragma omp barrier
7   do_many_other_things();
8 }
```

Single Construct

- The single construct denotes a structured block that is only executed by only one thread
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause)

```
1 #pragma omp parallel
2 {
3   do_many_things();
4 #pragma omp single
5   { exchange_boundaries(); }
6   do_many_other_things();
7 }
```

Low Level Synchronization: lock routines

- Simple lock routines: a simple lock is available if it is unset

  `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`, `omp_destroy_lock()`

- Nested locks: a nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function

  `omp_init_nest_lock()`, `omp_set_nest_lock()`, `omp_unset_nest_lock()`, `omp_test_nest_lock()`, `omp_destroy_nest_lock()`

Synchronization: Simple Locks

Example: conflicts are rare, but to play it safe, we must assure mutual exclusion for updates to histogram elements

```
1 #pragma omp parallel for
2     for (int i = 0; i < NBUCKETS; i++) {
3         omp_init_lock(&hist_locks[i]); // one lock per
              elements of hist
4         hist[i] = 0;
5     }
6 #pragma omp parallel for
7     for(int i = 0; i < NVALS; i++) {
8         ival = (int) sample(arr[i]);
9         omp_set_lock(&hist_locks[ival]);
10            hist[ival]++; // mutual exclusion, less wait
                  compared to critical
11        omp_unset_lock(&hist_locks[ival]);
12    }
13 #pragma omp parallel for
14    for(i=0;i<NBUCKETS; i++)
15        omp_destroy_lock(&hist_locks[i]); // free locks
              when done
```

Sections Worksharing

- The sections worksharing construct gives a different structured block to each thread

```
1 #pragma omp parallel
2 {
3 #pragma omp sections
4   {
5 #pragma omp section
6     x_calculation();
7 #pragma omp section
8     y_calculation();
9 #pragma omp section
10    z_calculation();
11  } // implicit barrier that can be turned off with nowait
12 }
```

Shorthand: `#pragma omp parallel sections`

Runtime Library Routines

- Modify/Check the number of threads - `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_get_thread_num()`, `omp_get_max_threads()`
- Are we in an active parallel region? - `omp_in_parallel()`
- How many processors in the system? - `omp_num_procs()`
- Many less commonly used routines

Data Sharing

- Most variables are SHARED by default: static variables, global variables, heap data (`malloc()`, `new`)
- But not everything is shared: stack variables in functions called from parallel regions are PRIVATE

Examples:

```
1 double A[10];
2 int main() {
3     int index[10];
4 #pragma omp parallel
5     work(index);
6     printf("%d\n", index[0]);
7 }
```

```
1 extern double A[10];
2 void work(int *index) {
3     double temp[10];
4     static int count;
5     // do something
6 }
```

`A`, `index`, `count` are shared.

`temp` is private to each thread.

Data Scope Attribute Clauses

- Change scope attributes using:
  `shared, private, firstprivate, lastprivate`
- The default attributes can be overridden with:
  `default(shared|none)`
- Skip details...

Data Scope Attribute Clauses: Example

```
1  int main() {
2      std::string a = "a", b = "b", c = "c";
3  #pragma omp parallel firstprivate(a) private(b) shared(c)
       num_threads(2)
4      {
5          a += "k"; // a is initialized with "a"
6          b += "k"; // b is initialized with std::string()
7  #pragma omp critical
8          {
9              c += "k"; // c is shared
10             std::cout << omp_get_thread_num() << ": " << a
                   << ", " << b << ", " << c << "\n";
11         }
12     }
13     std::cout << a << ", " << b << ", " << c << "\n";
14 }
```

Sample Output:

```
1 0: ak, k, ck
2 1: ak, k, ckk
3 a, b, ckk
```

One Advanced Feature

OpenMP Tasks

- The for and sections worksharings worked well for many cases. However,
  - loops need a known length at run time
  - finite number of parallel sections
- This didn't work well with certain common problems:
  - linked list, recursive algorithms, etc.
- Introduce the task directive (only for OpenMP 3.0+)

Task: Example

Traversal of a tree:

```
1  struct node { node *left, *right; };
2  extern void process(node* );
3  void traverse(node* p) {
4      if (p->left)
5  #pragma omp task // p is firstprivate by default
6          traverse(p->left);
7      if (p->right)
8  #pragma omp task // p is firstprivate by default
9          traverse(p->right);
10     process(p);
11 }
```

```
1 #pragma omp parallel
2 {
3     #pragma omp single nowait
4     { traverse(root); }
5 }
```

Task: Example

What if we want to force a postorder traversal of the tree?

```
1  struct node { node *left, *right; };
2  extern void process(node* );
3  void traverse(node* p) {
4      if (p->left)
5  #pragma omp task // p is firstprivate by default
6          traverse(p->left);
7      if (p->right)
8  #pragma omp task // p is firstprivate by default
9          traverse(p->right);
10 #pragma omp taskwait // a barrier only for tasks
11     process(p);
12 }
```

Task: Example

Process elements of a linked list in parallel:

```
1  struct node { int data; node* next; };
2  extern void process(node* );
3  void increment_list_items(node* head) {
4      #pragma omp parallel
5      {
6          #pragma omp single
7          {
8              for(node* p = head; p; p = p->next) {
9                  #pragma omp task
10                 process(p); // p is firstprivate by default
11             }
12         }
13     }
14 }
```

References

- Guide into OpenMP: Easy multithreading programming for C++: `http://bisqwit.iki.fi/story/howto/openmp`
- Introduction to OpenMP - Tim Mattson (Intel):
  - slides: `http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf`
  - videos: `https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG`
- OpenMP specification: `http://openmp.org`