Module 2 Part 1
Application Design for Multicore Programming

Carnegie Mellon University
18-645

Jike Chong
Ian Lane

What we discussed last time:

Fast Platforms + **Good Techniques**

- **Multicore platforms**
- **Manycore platforms**
- **Cloud platforms**

- **Data structures**
- **Algorithms**
- **Software Architecture**

- Highlighted the difference between multicore and manycore platforms
- Discussed the multicore and manycore platform intricacies
- **Exposing** concurrency in the k-means algorithm
- **Exploiting** parallelism by **exploring** mappings from application to platform
- This lecture: **An abstraction for multicore parallel programming**

Outline

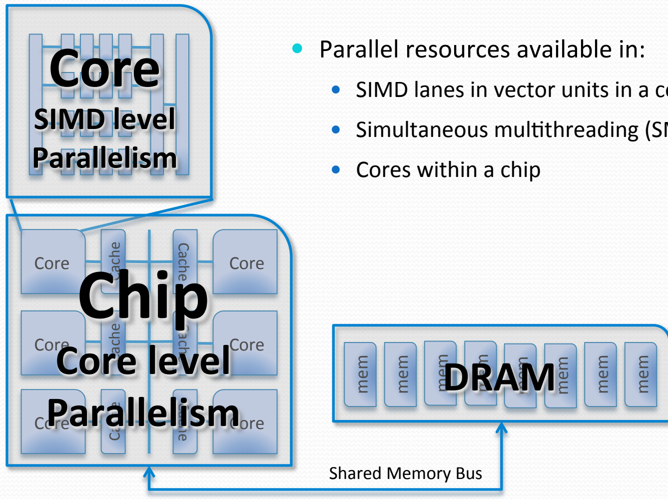
- Multicore Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multi-processing
  - Overview
  - Example: numerical integration of Pi
  - Shared variable synchronization
  - SPMD vs worksharing
  - Data environment options
  - Advanced worksharing options
  - Synchronization options
- How is This Relevant to Writing Fast Code?

Answers you should know after this…

- What are the exploitable levels of parallelism in a multicore processor?
- What is SPMD? And how to use OpenMP to do SPMD?
- What's the difference between "critical" and "atomic"?
- How to reduce synchronization cost and avoid "false sharing"?
- What are the scheduling, reduction, data sharing, and synchronization options for OpenMP?
- How is this relevant to writing fast code?

Multicore Shared-Memory Platforms

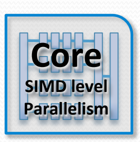
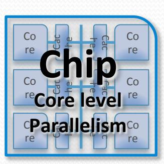
- Parallel resources available in:
  - SIMD lanes in vector units in a core
  - Simultaneous multithreading (SMT) in a core
  - Cores within a chip



18-645 – *How to Write Fast Code?* *Carnegie Mellon University (c) 2012-2014* 5

Exploiting Different Levels of Parallelism

- How do we write fast code to exploit the many levels of parallelism on a multicore platform?

	**SIMD-Level Parallelism**	Exploited using vectorizing compiler and hand-code intrinsics
	SMT-Level Parallelism	OS abstract it to core-level parallelism
	**Core-Level Parallelism**	Using **threads** to describe work done on different cores

18-645 – *How to Write Fast Code?* *Carnegie Mellon University (c) 2012-2014* 6

What is a thread?

- A thread of execution is a unit of processing scheduled by the OS
  - Allows system resources to be shared efficiently

Processes	Threads	Fiber
Exist independently in the OS	**Subsets of processes**	Subsets of threads
Independent states: virtual memory space, handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution	**Threads in a process share:** **virtual address space and system resources** **Independent:** **exception handlers, a scheduling priority, a unique thread identifier, and thread context**	Fibers within a thread share: virtual address space, system resources, thread context, a manually specified schedule Independent: exception handlers, a unique fiber identifier
Interact through system-provided inter-process communication mechanisms	**Interact through shared memory**	Interact through shared memory
OS Preemptive	**OS Preemptive**	Non-Preemptive

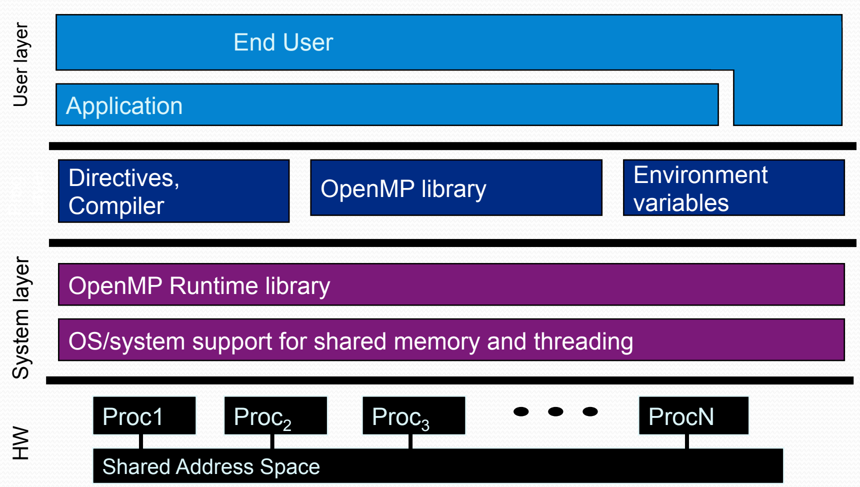
Landscape of Thread Programming

- POSIX threads
  - POSIX: Portable Operating System Interface for Unix – IEEE Standard
  - Defines a set of C programming language types, functions and constants
  - ***Manually expose and manage all concurrency with the API***
- OpenMP
  - OpenMP: Open Multi-Processing – an API for shared memory multiprocessing
  - ***Programmer hints at the concurrency***
  - ***Compiler manages the parallelism***
- Intel Cilk Plus, Intel Thread Building Block (TBB):
  - Fork-join parallelism ("spawn" and "sync" in Clik, Parallel loops/containers in TBB)
  - ***Programmer exposes the concurrency***
  - ***Runtime determines what is run in parallel***

Outline

- Mutlicore Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multi-processing
  - **Overview**
  - Example: numerical integration of Pi
  - Shared variable synchronization
  - SPMD vs worksharing
  - Data environment options
  - Advanced worksharing options
  - Synchronization options
- How is This Relevant to Writing Fast Code?

OpenMP Basic Defs: Solution Stack



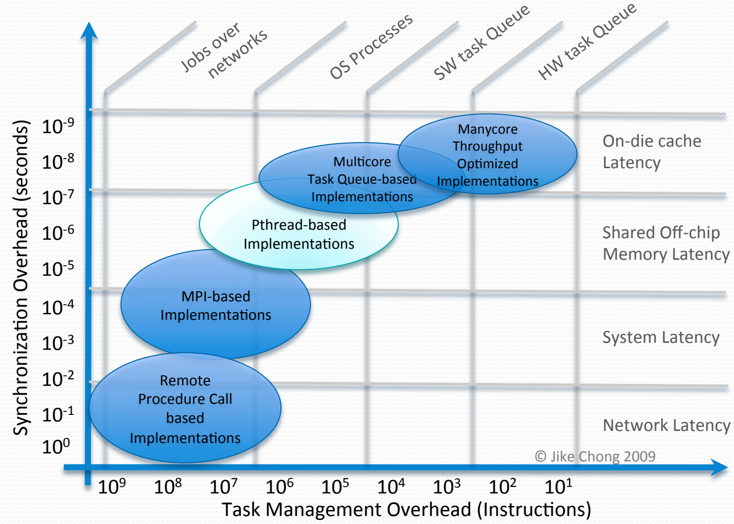
Mattson et al, A "Hands-on" Introduction to OpenMP

Scope of discussion for OpenMP

Topic	Concepts
I. OMP Intro	**Parallel regions**
II. Creating threads	**Parallel, default data environment, runtime library calls**
III. Synchronization	**False sharing, critical, atomic**
IV. Parallel loops	**For, schedule, reduction**
V. Odds and ends	Single, sections, master, runtime libraries, environment variables, synchronization, etc.
VI. Data Environment	Data environment details, software optimization
VII. OpenMP 3 and tasks	Tasks and other OpenMP 3 features
VIII. Memory model	The need for flush, but its actual use is left to an appendix
IX. Threadprivate	Modular software

Mattson et al, A "Hands-on" Introduction to OpenMP

Spectrum of Granularity



Example 1: Hello World

Verifying that your OpenMP environment works

- Write a multithreaded program where **each thread** prints “hello world”.

```
void main()
{

    int ID = 0;
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);

}
```

Mattson et al, A “Hands-on” Introduction to OpenMP

Example 1: SPMD with “omp parallel”

Verifying that your OpenMP environment works

- Write a multithreaded program where each thread prints “hello world”.

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

Switches for compiling and linking

- gcc -fopenmp gcc
- pgcc -mp pgi
- icl /Qopenmp intel (windows)
- icc –openmp intel (linux)

Mattson et al, A “Hands-on” Introduction to OpenMP

Example 1: SPMD with "omp parallel"

A multi-threaded "Hello world" program

- Write a multithreaded program where each thread prints "hello world".

OpenMP include file

Parallel region with default number of threads

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

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)

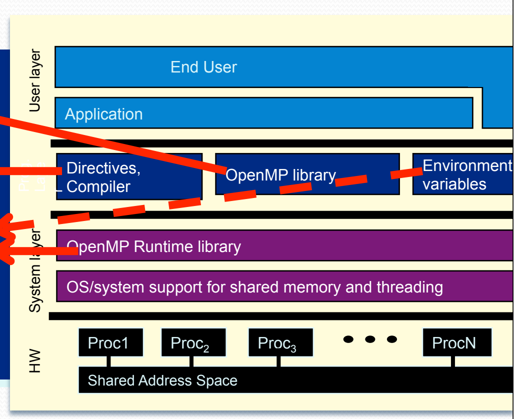
Mattson et al, A "Hands-on" Introduction to OpenMP

Example 1: SPMD with "omp parallel"

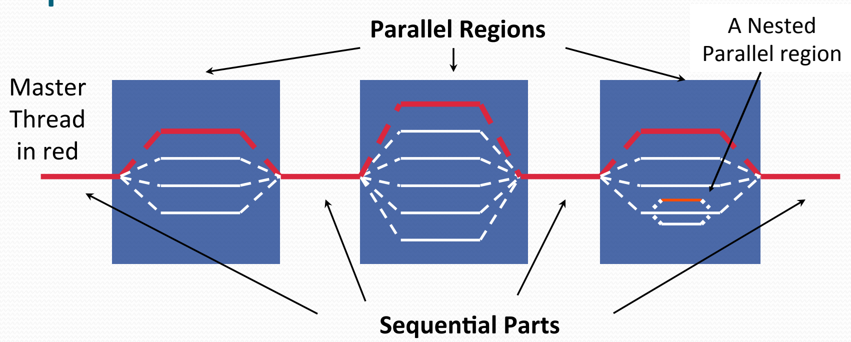
A multi-threaded "Hello world" program

- Write a multithreaded program where each thread prints "hello world".

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



OpenMP: Execution Pattern



- Managed **Fork-Join** Parallelism:
  - **Master thread** creates a **team of threads** as needed
  - Parallelism added incrementally until performance goals are met
  - i.e. the **sequential program evolves into a parallel program**

Mattson et al, A "Hands-on" Introduction to OpenMP

Thread Creation: Parallel Regions

- Threads in OpenMP are created with the parallel construct
  - To create a 4 thread Parallel region:

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

Each thread calls foo(ID,A) **for** ID = 0 **to** 3

Mattson et al, A "Hands-on" Introduction to OpenMP

Thread Creation: Parallel Regions

- Threads in OpenMP are created with the parallel construct
  - To create a 4 thread Parallel region:

Runtime function to request a certain number of threads

Each thread executes a copy of the code within the structured block

```
double A[1000];

#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    foo(ID,A);
}
```

Runtime function returning a thread ID

Each thread calls foo(ID,A) for ID = 0 to 3

Mattson et al, A "Hands-on" Introduction to OpenMP

Outline

- Mutlicore Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multi-processing
  - Overview
  - **Example: numerical integration of Pi**
  - Shared variable synchronization
  - SPMD vs worksharing
  - Data environment options
  - Advanced worksharing options
  - Synchronization options
- How is This Relevant to Writing Fast Code?

Example: Numerical Integration of Pi

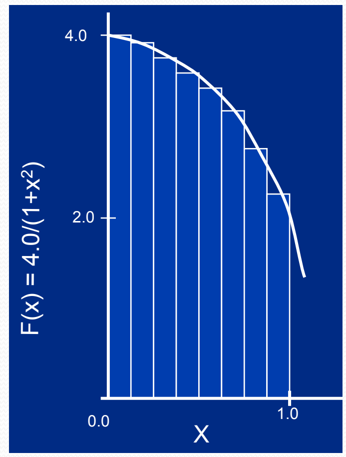
Mathematically, we know that:

$$\int_0^1 \frac{4}{1+x^2}dx = 4 \cdot \tan^{-1} x \Big|_0^1 = \pi$$

We can approximate the integral as a sum of rectangles:

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

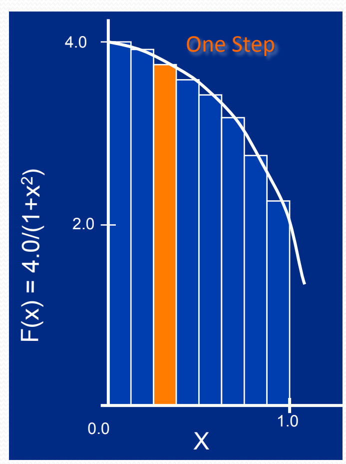
Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.



Mattson et al, A "Hands-on" Introduction to OpenMP

Numerical Integration of Pi

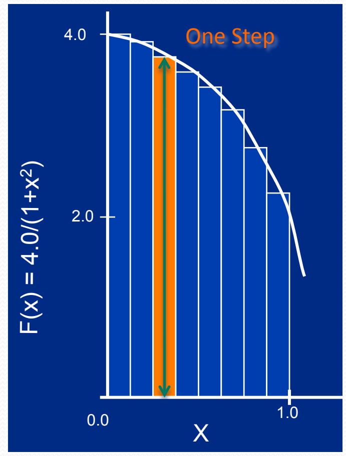
```
#include <stdio.h>
static long num_steps = 100000; double step;

int main ()
{
    int i; double pi, sum;
    step = 1.0/(double) num_steps;


    int i; double x;

    for (i=0, sum=0.0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }


    pi = sum * step;
    printf("pi = %f \n", pi);
    return 0;
}
```



Mattson et al, A "Hands-on" Introduction to OpenMP

Numerical Integration of Pi

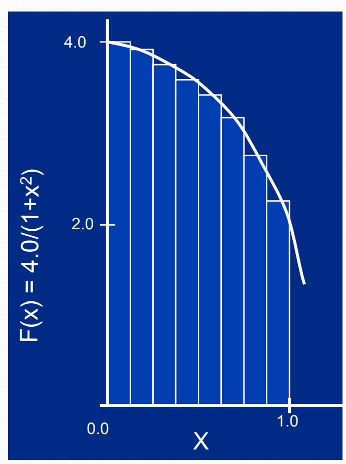
```
#include <stdio.h>
static long num_steps = 100000; double step;

int main ()
{
    int i; double pi, sum;
    step = 1.0/(double) num_steps;


    int i; double x;

    for (i=0, sum=0.0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }

    pi = sum * step;
    printf("pi = %f \n", pi);
    return 0;
}
```



Mattson et al, A "Hands-on" Introduction to OpenMP

Concurrency Opportunities

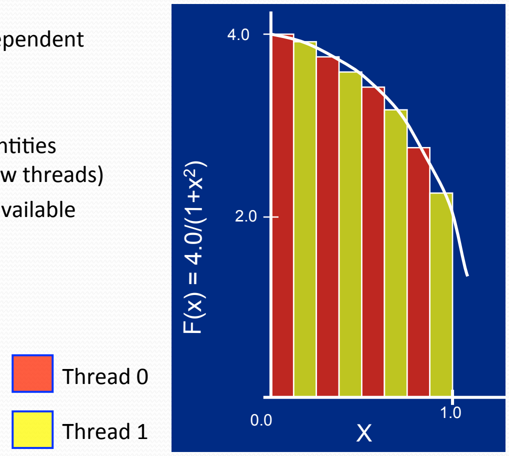
- What is parallelizable?
  - Calculation of each step is independent
- One can map each of the steps to an independent thread
  - **What's wrong with this approach?**
- Significant thread management overhead
  - Compute each step as a separate thread involves significant thread management overhead (hundreds of cycles)



Mattson et al, A "Hands-on" Introduction to OpenMP

Concurrency Opportunities

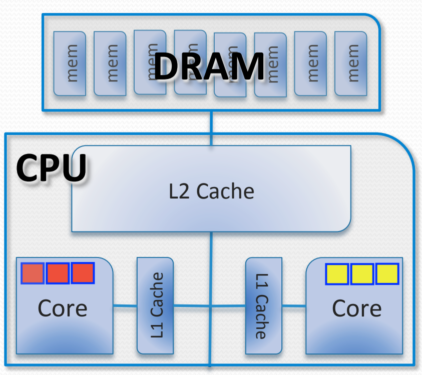
- What is parallelizable?
  - Calculation of each step is independent
- Alternative:
  - Specify a fixed set of parallel entities
    e.g. two threads (red and yellow threads)
  - Parallelize workload over the available number of threads

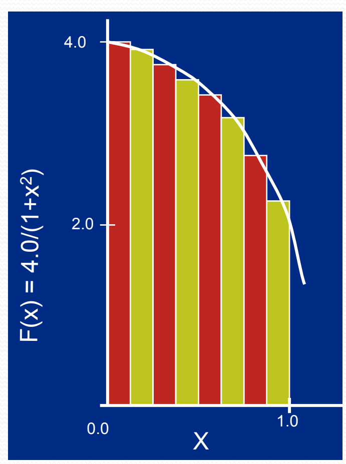


18-645 – *How to Write Fast Code?* *Carnegie Mellon University (c) 2012-2014* 25

Implementation View

- Work is evenly distributed across two processors



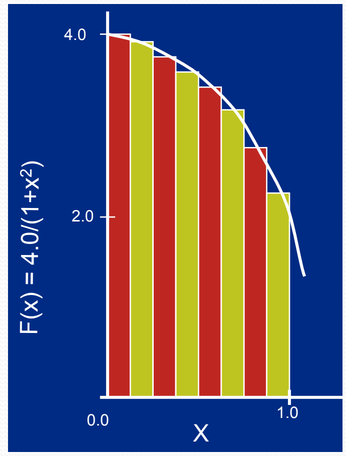


18-645 – *How to Write Fast Code?* *Carnegie Mellon University (c) 2012-2014* 26

Example 2: Numerical Integration of Pi

Sequential: 2.12 seconds

```
#include <stdio.h>
static long num_steps = 100000000; double step;

int main ()
{
     int i; double pi, sum = 0.0;
     step = 1.0/(double) num_steps;


     double x;

     for (i=0;i< num_steps; i++){
          x = (i+0.5)*step;
          sum += 4.0/(1.0+x*x);
     }


     pi = sum * step;
     printf("pi = %f \n", pi);
     return 0;
}
```



Mattson et al, A "Hands-on" Introduction to OpenMP

Outline

- Mutlicore Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multi-processing
  - Overview
  - Example: numerical integration of Pi
  - **Shared variable synchronization**
  - SPMD vs worksharing
  - Data environment options
  - Advanced worksharing options
  - Synchronization options
- How is This Relevant to Writing Fast Code?

Example 3: Critical Region

Sequential: 2.12 seconds

64.45 seconds

Why so slow?

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
     int i; double pi; int nthreads; double sum=0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel num_threads(NUM_THREADS)
     {
          int i, id = omp_get_thread_num(); double x;
          int nthrds = omp_get_num_threads();
          for (i=id;i< num_steps; i=i+nthrds) {
               x = (i+0.5)*step;
               #pragma omp critical
               {sum += 4.0/(1.0+x*x);}
          }
          if (id == 0)   nthreads = nthrds;
     }
     pi = sum * step;
     printf("pi = %f \n", pi);
     return 0;
}
```

Anything in the critical region, OpenMP would make sure it is sequentialized.

18-645 – *How to Write Fast Code?* *Carnegie Mellon University (c) 2012-2014* 29

Example 4: Atomic Operation

Sequential: 2.12 seconds

4.93 seconds

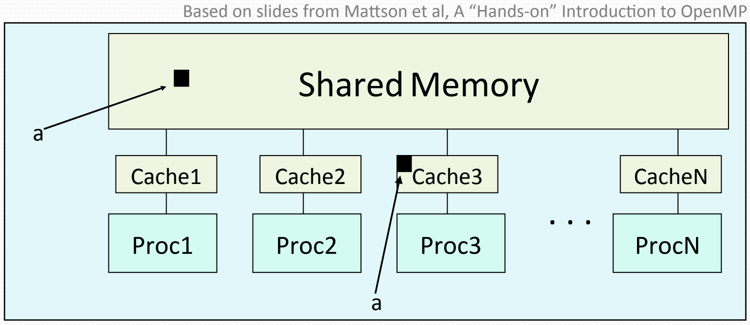
Better, why still so slow?

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
     int i; double pi; int nthreads; double sum=0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel num_threads(NUM_THREADS)
     {
          int i, id = omp_get_thread_num(); double x, tmp;
          int nthrds = omp_get_num_threads();
          for (i=id;i< num_steps; i=i+nthrds) {
               x = (i+0.5)*step;
               tmp = 4.0/(1.0+x*x);
               #pragma omp atomic
               {sum += tmp}
          }
          if (id == 0)   nthreads = nthrds;
     }
     pi = sum * step;
     printf("pi = %f \n", pi);
     return 0;
}
```

18-645 – *How to Write Fast Code?* *Carnegie Mellon University (c) 2012-2014* 30

Shared Memory Model

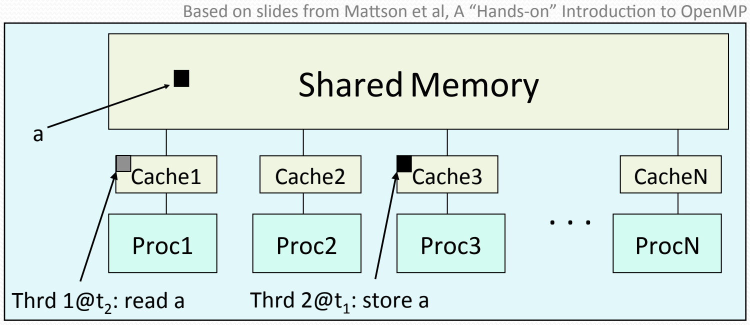
Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP



- All threads share an address space
- Multiple copies of data may be present in various levels of cache

Shared Memory Model

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

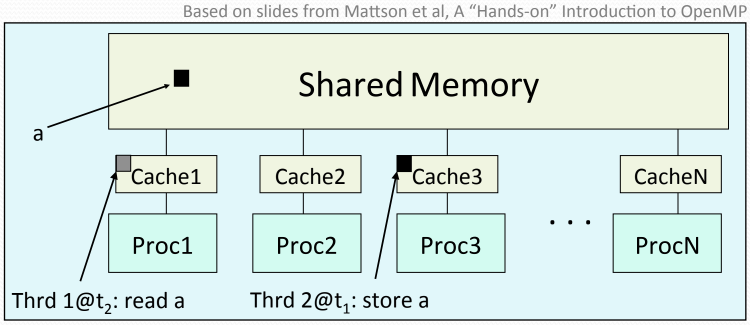


- A piece of data (at address "a") can be:
  - Written into memory by Thrd 2 at t_1
  - Then, read by Thrd 1 at t_2
- We say the data at address "a" is in a ***shared*** state

Mattson et al, A "Hands-on" Introduction to OpenMP

Managing Coherence Costly

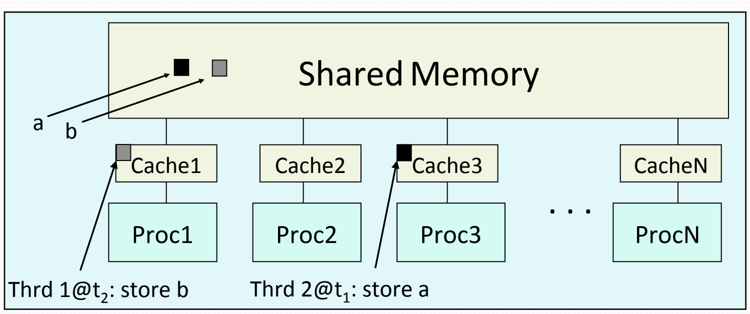
Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP



- Synchronization protocols are implemented in hardware to manage coherence between caches
  - e.g. **MESI Protocol** (Papamarcos & Patel 1984)
  - Allows data to be in one of four states: *Modified, Exclusive, Shared, Invalid*
- Transparent to software, but could have **severe performance penalties**

Mattson et al, A "Hands-on" Introduction to OpenMP

Threads can Write to Different Addresses…



- Avoid cache **synchronization protocols** induced **severe performance penalties** when no sharing is required
- Technique:
  - Write to separate addresses during parallel computation

Example 5: Separate Result Storage

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
      int i; double pi; int nthreads; double sum[NUM_THREADS];
      step = 1.0/(double) num_steps;
      #pragma omp parallel num_threads(NUM_THREADS)
      {
            int i, id = omp_get_thread_num(); double x;
            int nthrds = omp_get_num_threads();
            for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                  x = (i+0.5)*step;
                  sum[id] += 4.0/(1.0+x*x);
            }
            if (id == 0)    nthreads = nthrds;
      }
      for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
      printf("pi = %f \n", pi);
      return 0;
}
```

Sequential: 2.12 seconds

1.47 seconds

Better, but why slower 2x speedup?

Structure of a Cache

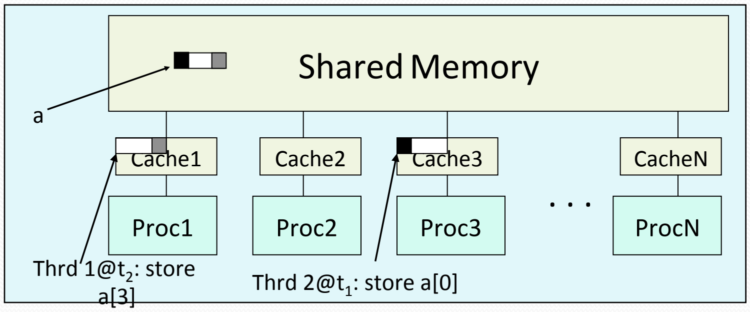
- **Principle of Locality**
  - The phenomenon of the same value or related storage locations being frequently accessed, usually amenable to performance optimization
- **Temporal Locality**
  - Reuse of specific data and/or resources within relatively small time durations
- **Spatial Locality**
  - Use of data elements within relatively close storage locations

$$A = \begin{pmatrix} 4 & 5 & -5 \\ -1 & -4 & -2 \\ -3 & 1 & 5 \\ 2 & 1 & 4 \end{pmatrix} \quad B = \begin{pmatrix} -4 & 4 & -1 & 3 & -4 \\ -1 & -5 & -5 & 4 & -5 \\ -1 & -1 & 0 & -1 & 1 \end{pmatrix}$$

$$AB = \begin{pmatrix} -16 & -4 & -29 & 37 & -46 \\ 10 & 18 & 21 & -17 & 22 \\ 6 & -22 & -2 & -10 & 12 \\ -13 & -1 & -7 & 6 & -9 \end{pmatrix}$$

$$\begin{array}{ccccc} (-1)(3) & + & (-4)(4) & + & (-2)(-1) \\ -3 & + & -16 & + & 2 \\ & & -17 & & \end{array}$$

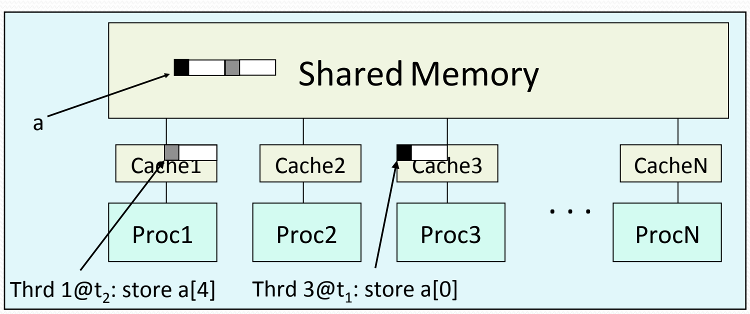
Is data being share?



- Cache loads and stores work with 4-16 word long cache lines (64B for Intel)
  - If two threads are writing to the same cache line, conflicts occurs
- Even if the address differs, one will still suffer **performance penalty**
- This is called **false sharing**

18-645 – *How to Write Fast Code?* *Carnegie Mellon University (c) 2012-2014* 37

Solution for False Sharing



- Be aware of the cache line sizes for a platform
- Avoid accessing the same cache line from different threads

18-645 – *How to Write Fast Code?* *Carnegie Mellon University (c) 2012-2014* 38

Example 6: Eliminate False Sharing

Sequential: 2.12 seconds

1.15 seconds

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
     int i; double pi; int nthreads;
     step = 1.0/(double) num_steps;
     #pragma omp parallel num_threads(NUM_THREADS)
     {
         int i, id = omp_get_thread_num(); double x, local_sum=0.0;
         int nthrds = omp_get_num_threads();
         for (i=id;i< num_steps; i=i+nthrds) {
           x = (i+0.5)*step;
           local_sum += 4.0/(1.0+x*x);
         }
         #pragma omp critical
         pi += local_sum * step;
     }
     printf("pi = %f \n", pi);
     return 0;
}
```

Based on slides from Mattson et al, A “Hands-on” Introduction to OpenMP

Example 6: Eliminate False Sharing

Sequential: 2.12 seconds

1.15 seconds

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
     int i; double pi, x, local_sum; int nthreads;
     step = 1.0/(double) num_steps;
     #pragma omp parallel num_threads(NUM_THREADS) private(x, local_sum)
     {
         int i, id = omp_get_thread_num(); local_sum=0.0;
         int nthrds = omp_get_num_threads();
         for (i=id;i< num_steps; i=i+nthrds) {
           x = (i+0.5)*step;
           local_sum += 4.0/(1.0+x*x);
         }
         #pragma omp critical
         pi += local_sum * step;
     }
     printf("pi = %f \n", pi);
     return 0;
}
```

Great! But is this the best we can do?

Based on slides from Mattson et al, A “Hands-on” Introduction to OpenMP

Example 7: Sequential Optimization

Sequential: 2.12 seconds

0.46 seconds

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
     int i; double pi, x, local_sum; int nthreads;
     step = 1.0/(double) num_steps;
     #pragma omp parallel num_threads(NUM_THREADS) private(x, local_sum)
     {
         int i, id = omp_get_thread_num(); local_sum=0.0;
         int nthrds = omp_get_num_threads();
         double hoop = nthrds*step;
         for (i=id, x=(i+0.5)*step;i< num_steps; i=i+nthrds) {
           x += hoop;
           local_sum += 4.0/(1.0+x*x);
         }
         #pragma omp critical
         pi += local_sum * step;
     }
     printf("pi = %f \n", pi);
     return 0;
}
```

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Outline

- Mutlicore Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multi-processing
  - Overview
  - Example: numerical integration of Pi
  - Shared variable synchronization
  - **SPMD vs worksharing**
  - Data environment options
  - Advanced worksharing options
  - Synchronization options
- How is This Relevant to Writing Fast Code?

SPMD vs. Worksharing

- A parallel construct by itself creates an SPMD
  - "Single Program Multiple Data"
  - Programmer must explicitly specify what each thread must do differently
  - The division of work is hard-coded in the program
- Opportunity:
  - Many parallel regions are loops
- Question:
  - Can we make it easier to parallelize these loops?

OpenMP: Parallel For

- One of the worksharing constructs OpenMP provide is **"omp for"**

Sequential code	`for(i=0;i<N;i++) { a[i] = a[i] + b[i];}`
OpenMP **"parallel"** region	see code below
OpenMP **"parallel"** region and a worksharing **"for"** construct	see code below

Sequential code:

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP "parallel" region:

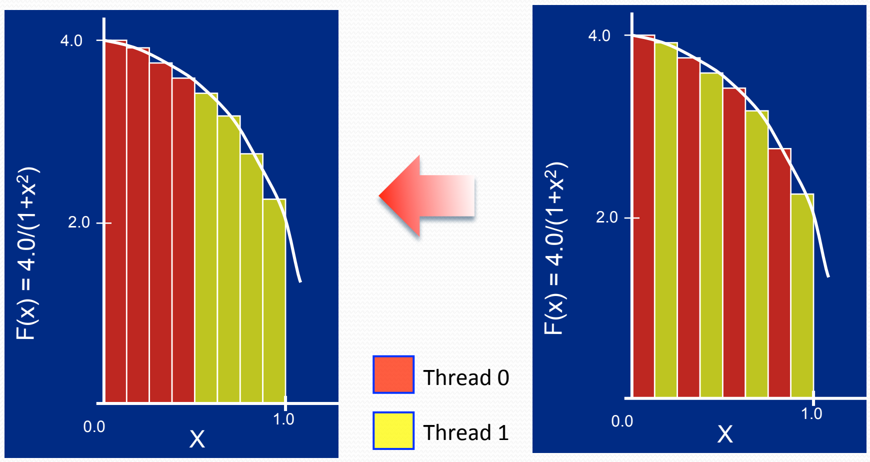
```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP "parallel" region and a worksharing "for" construct:

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

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Other Concurrency Opportunities



Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Example 8: Parallel For and Reduction

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
      int i; double pi, x, sum = 0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel private(x)
      {
            #pragma omp for reduction(+:sum)
            for (i=0;i< num_steps; i++){
                  x = (i+0.5)*step;
                  sum = sum + 4.0/(1.0+x*x);
            }
      }
      Pi = sum * step;
      printf("pi = %f \n", pi);
      return 0;
}
```

Sequential: 2.12 seconds

1.22 seconds

Reduction is more scalable than critical sections.

Loops must not have loop carry dependency. **Limitation of compiler technology.**

"i" is private by default

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Combined Parallel/Worksharing Construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

```
double  res[MAX];  int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
      res[i] = huge();
   }
}
```

```
double  res[MAX];  int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
      res[i] = huge();
   }
```

Based on slides from Mattson et al, A “Hands-on” Introduction to OpenMP

OpenMP: Working With Loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
double hoop = nthrds*step;
for (i=id, x=(i+0.5)*step;i< num_steps; i=i+nthrds) {
  x += hoop;
  sum = sum + 4.0/(1.0+x*x);
}
```

```
for (i=0;i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
```

Based on slides from Mattson et al, A “Hands-on” Introduction to OpenMP

OpenMP: Reductions

```
for (i=0;i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
```

```
#pragma omp for reduction(+:sum)
for (i=0;i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
```

reduction (op : list)

1. A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+")
2. Updates occur on the local copy
3. Local copies are reduced into a single value and combined with the original global value

- Accumulating values into a single variable (sum) creates 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.

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

OpenMP: Reduction Operators

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0

Operator	Initial value
&	~0
		0
^	0
&&	1
			0

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Example 8: Parallel For and Reduction

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;

int main ()
{
     int i; double pi, x, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel for private(x) reduction(+:sum)
     for (i=0;i< num_steps; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
     }
     Pi = sum * step;
     printf("pi = %f \n", pi);
     return 0;
}
```

Sequential: 2.12 seconds

1.22 seconds

Advantage: we created a parallel program without changing any code and by adding 2 simple lines of text!

Disadvantage:
1) Lot's of pitfalls if you don't understand system architecture
2) The basic result may not the fastest one can do

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Outline

- Mutlicore Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multi-processing
  - Overview
  - Example: numerical integration of Pi
  - Shared variable synchronization
  - SPMD vs worksharing
  - **Data environment options**
  - Advanced worksharing options
  - Synchronization options
- How is This Relevant to Writing Fast Code?

Data Environment in OpenMP

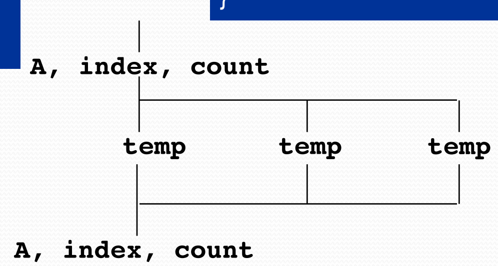
- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are **SHARED** among threads
  - File scope variables, static
  - Dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Functions called from parallel regions are **PRIVATE**
  - Automatic variables within a statement block are **PRIVATE**.

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Example: Data Sharing

```
double A[10];
int main() {
    int index[10];
    #pragma omp parallel
            work(index);
    printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```



A, index and count are shared by all threads.

temp is local to each thread

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Changing Storage Attributes

- One can selectively change storage attributes for constructs using the following clauses:
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - LASTPRIVATE

Example 9: firstprivate example

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
     int i; double pi, x, local_sum = 0.0; int nthreads;
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
     #pragma omp parallel private(x) firstprivate(local_sum)
     {
          int i, id = omp_get_thread_num(); local_sum=0.0;
          int nthrds = omp_get_num_threads();
          double hoop = nthrds*step;
          for (i=id, x=(i+0.5)*step;i< num_steps; i=i+nthrds) {
            x += hoop;
            local_sum += 4.0/(1.0+x*x);
          }
          #pragma omp critical
          pi += local_sum * step;
     }
     printf("pi = %f \n", pi);
     return 0;
}
```

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Outline

- Mutlicore Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multi-processing
  - Overview
  - Example: numerical integration of Pi
  - Shared variable synchronization
  - SPMD vs worksharing
  - Data environment options
  - **Advanced worksharing options**
  - Synchronization options
- How is This Relevant to Writing Fast Code?

The Scheduling Cause

- The schedule clause affects how loop iterations are mapped onto threads
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size "chunk" to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs "chunk" iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0).

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

The Scheduling Cause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Example 9: Scheduling

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;

int main ()
{
     int i; double pi, x, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel for private(x) reduction(+:sum) schedule(static,
     10000000)
     for (i=0;i< num_steps; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
     }
     Pi = sum * step;
     printf("pi = %f \n", pi);
     return 0;
}
```

Sequential: 2.12 seconds

1.31 seconds

Based on slides from Mattson et al, A "Hands-on" Introduction to OpenMP

Outline

- Mutlicore Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multi-processing
  - Overview
  - Example: numerical integration of Pi
  - Shared variable synchronization
  - SPMD vs worksharing
  - Data environment options
  - Advanced worksharing options
  - **Synchronization options**
- How is This Relevant to Writing Fast Code?

Synchronization: ordered

- The **ordered** region executes in the sequential order
- Important for some scientific code and optimization code
  - Order of reduction may cause rounding differences

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
    for (I=0;I<N;I++){
        tmp = NEAT_STUFF(I);
#pragma ordered
        res += consum(tmp);
    }
```


62

Synchronization Barrier

- **Barrier**: Each thread waits until all threads arrive

```
#pragma omp parallel shared (A, B, C) private(id)
{
      id=omp_get_thread_num();
      A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
      for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
#pragma omp for nowait
      for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
      A[id] = big_calc4(id);
}
```

implicit barrier at the end of a for worksharing construct

No implicit barrier due to nowait

Implicit barrier at the end of a parallel region

"Single" Worksharing Construct

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block
  - can remove the barrier with a nowait clause

```
#pragma omp parallel
{
      do_many_things();
#pragma omp single
      {    exchange_boundaries();  }
      do_many_other_things();
}
```

```
#pragma omp parallel
{
      do_many_things();
#pragma omp single nowait
      {    exchange_boundaries();  }
      do_many_other_things();
}
```

Nested Parallelism

- **Q:** Is nested parallel possible with OpenMP?
- **A:** Yes. But be sure to understand why you want to use it.

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level) {
     #pragma omp single
     { printf("L %d: # threads in team %d\n", level, omp_get_num_threads());}
}
int main() {
     omp_set_dynamic(0);
     #pragma omp parallel num_threads(2)
     {
          report_num_threads(1);
          #pragma omp parallel num_threads(2)
          {
               report_num_threads(2);
               #pragma omp parallel num_threads(2)
                    report_num_threads(3);
          }
     }
     return(0);
}
```

Source from: http://download.oracle.com/docs/cd/E19205-01/819-5270/aewbc/index.html

Nested Parallelism

- Compiling and running this program with nested parallelism enabled produces the following (sorted) output:

```
$ export OMP_NESTED=TRUE
$ ./experimentN
L 1: # threads in team 2
L 2: # threads in team 2
L 2: # threads in team 2
L 3: # threads in team 2
L 3: # threads in team 2
L 3: # threads in team 2
L 3: # threads in team 2
```

```
$ export OMP_NESTED=FALSE
$ ./experimentN
L 1: # threads in team 2
L 2: # threads in team 1
L 3: # threads in team 1
L 2: # threads in team 1
L 3: # threads in team 1
```

OpenMP Environment Variables

- OMP_SCHEDULE=algorithm
  - dynamic[, n]
  - guided[, n]
  - Runtime
  - **static[, n]**
- OMP_NUM_THREADS=num
- OMP_NESTED=TRUE|FALSE
- OMP_DYNAMIC=TRUE|FALSE

Outline

- Mutlicore Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multi-processing
  - Overview
  - Example: numerical integration of Pi
  - Shared variable synchronization
  - SPMD vs worksharing
  - Data environment options
  - Advanced worksharing options
  - Synchronization options
- **How is This Relevant to Writing Fast Code?**

k-means Algorithm Concurrency

1. **Initialization:** Randomly select *k* cluster centers
2. **Expectation:** Assign closest center to each data point
   - N (independent)
   - k (min reduction)
   - D (sum reduction)
3. **Maximization:** Update centers based on assignments
   - D (independent)
   - N (Histogram computation into k bins)
4. **Evaluate:** Re-iterate steps 2-3 until convergence

Project 1: Search Among Alternatives

- What are the implementation **alternatives**?
  - Different mapping of **application concurrency** to **platform parallelism**
- The **search** process
  - More complex than one **application concurrency** to one **platform parallelism**
  - May want to sequentialize some operations:
    - Some parallel operations are as **"work-efficient"** as sequential operations
    - Reduction – sequential: O(N), Parallel: O(N logN)
  - One level of concurrency could map to multiple levels of parallelism

How is this relevant to writing fast code?

Fast Platforms + **Good Techniques**

- **Multicore platforms**
- **Manycore platforms**
- **Cloud platforms**

- **Data structures**
- **Algorithms**
- **Software Architecture**

- This lecture: **An abstraction for multicore parallel programming**
- **Abstractions:**
  - Help establish a mental model for the programmers
  - Make it easier to write parallel code
  - **Performance depend on deep understanding of the implementation platform**

Can You Answer These Questions Now?

- What are the exploitable levels of parallelism in a multicore processor?
- What is SPMD? And how to use OpenMP to do SPMD?
- What's the difference between "critical" and "atomic"?
- How to reduce synchronization cost and avoid "false sharing"?
- What are the scheduling, reduction, data sharing, and synchronization options for OpenMP?
- How is this relevant to writing fast code?