The OpenMP® Common Core:
A hands on exploration

Tim Mattson
Intel Corp.
timothy.g.mattson@intel.com

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.
Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.
*Other names and brands may be claimed as the property of others.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Preliminaries: Systems for exercises

- Blue Gene
  ssh <<login_name>>@vesta.alcf.anl.gov

- The OpenMP compiler
  Uncomment the line in .soft then run the resoft command
  +mpiwrapper-xl
  xlc++_r –qsmp=omp << file names>>

- X86 cluster
  ssh <<login_name>>@cooley.alcf.anl.gov

- The OpenMP compiler
  Add the line to “.soft.cooley” and then run the resoft command
  +intel-composer-xe
  icc –qopenmp –O3 << file names>>

- Copy the exercises to your home directory
  $ cp /projects/ATPESC2017/OMP_Exercises

- You can just run on the login nodes or use qsub (to get good timing numbers)

- To get a single node for 30 minutes in interactive mode
  qsub –A ATPESC2017 –n 1 –t 30 -Ik

Use either system or even your laptop if you wish
Preliminaries: Part 1

• Disclosures
  – The views expressed in this tutorial are those of the people delivering the tutorial.
    – We are not speaking for our employers.
    – We are not speaking for the OpenMP ARB

• We take these tutorials VERY seriously:
  – Help us improve … tell us how you would make this tutorial better.
Preliminaries: Part 2

• Our plan for the day .. Active learning!
  – We will mix short lectures with short exercises.
  – You will use your laptop to connect to a multiprocessor server.

• Please follow these simple rules
  – Do the exercises that we assign and then change things around and experiment.
    – Embrace active learning!
  – **Don’t cheat**: Do Not look at the solutions before you complete an exercise … even if you get really frustrated.
Outline

• Introduction to OpenMP
• Creating Threads
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap
• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Thread private data
OpenMP overview:

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

OpenMP: An API for Writing Multithreaded Applications

- C$OMP THREADPRIVATE (/ABC/)
- CALL OMP_SET_NUM_THREADS (10)
- C$OMP PARALLEL COPYIN (/blk/)
- C$OMP DO lastprivate (XX)
- Nthrds = OMP_GET_NUM_PROCS()
The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for application programmers more versed in their area of science than computer science.

- The complexity has grown considerably over the years!

The full spec is overwhelming. Master the 16 constructs OpenMP programmers use all the time (OpenMP Common Core), then dive into the rest of the spec. as needed.
OpenMP basic definitions: Basic Solution stack

End User

Application

Directives, Compiler

OpenMP library

Environment variables

OpenMP Runtime library

OS/system support for shared memory and threading

Proc1

Proc2

Proc3

ProcN

Shared Address Space
OpenMP basic syntax

• Most of the constructs in OpenMP are compiler directives. 
  #pragma omp construct [clause [clause]…]
  – Example
    #pragma omp parallel private(x)

• Function prototypes and types in the file:
  #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.
  – It’s OK to have an exit() within the structured block.
Exercise, Part A: Hello world

Verify that your environment works

• Write a program that prints “hello world”.

```c
#include<stdio.h>
int main()
{
    printf("hello ");
    printf(" world \n");
}
```
Exercise, Part B: Hello world
Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

```c
#include <omp.h>
#include <stdio.h>
int main()
{
    #pragma omp parallel
    {
        printf(" hello ");
        printf(" world \n");
    }
}
```

Switches for compiling and linking

- `gcc -fopenmp` - Gnu (Linux, OSX)
- `pgcc -mp pgi` - PGI (Linux)
- `icl /Qopenmp` - Intel (windows)
- `icc -fopenmp` - Intel (Linux, OSX)
Solution

A multi-threaded “Hello world” program

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

```c
#include <omp.h>
#include <stdio.h>
int main()
{
#pragma omp parallel
{
    printf("hello ");
    printf(" world 
");
}
}
```

Sample Output:

```
hello hello world
world
hello hello world
world
```

The statements are interleaved based on how the operating schedules the threads.
Outline

• Introduction to OpenMP
• Creating Threads
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap
• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Thread private data
OpenMP programming model:

**Fork-Join Parallelism:**

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.
Thread creation: Parallel regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:

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

- Each thread executes a copy of the code within the structured block.
- Each thread calls `pooh(ID,A)` for $ID = 0$ to $3$
Thread creation: Parallel regions example

- Each thread executes the same code redundantly.

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel {
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done
");
```

A single copy of A is shared between all threads.

```c
pooh(0,A)  pooh(1,A)  pooh(2,A)  pooh(3,A)
```

Threads wait here for all threads to finish before proceeding (i.e., a barrier)

* The name “OpenMP” is the property of the OpenMP Architecture Review Board
Thread creation: How many threads did you actually get?

- Request a number of threads with `omp_set_num_threads()`
- The number requested may not be the number you actually get.
  - An implementation may silently give you fewer threads than you requested.
  - Once a team of threads has launched, it will not be reduced.

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

- Each thread executes a copy of the code within the structured block
- Each thread calls `pooh(ID, A)` for `ID = 0` to `nthrds-1`

Runtime function to request a certain number of threads
Runtime function to return actual number of threads in the team
Internal control variables & the number of threads

• There are a few ways to control the number of threads.
  – omp_set_num_threads(4)

• What does omp_set_num_threads() actually do?
  – It resets an “internal control variable” the system queries to select the default number of threads to request on subsequent parallel constructs.

• To change this internal control variable without re-compilation.
  – When an OpenMP program starts up, it queries an environment variable OMP_NUM_THREADS and sets the appropriate internal control variable to the value of OMP_NUM_THREADS

• For example, to set the initial, default number of threads to request in OpenMP from my apple laptop
  > export OMP_NUM_THREADS=12
An interesting problem to play with Numerical integration

Mathematically, we know that:

\[ \int_{0}^{1} \frac{4.0}{(1+x^2)} \, dx = \pi \]

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 \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).
Serial PI program

```c
static long num_steps = 100000;
double step;
int main ()
{
    int i;    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

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

    pi = step * sum;
}
```

See OMP_exercises/pi.c
Serial PI program

```c
#include <omp.h>
static long num_steps = 100000;
double step;
int main ()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;
    double tdata = omp_get_wtime();
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    tdata = omp_get_wtime() - tdata;
    printf(" pi = %f in %f secs\n",pi, tdata);
}
```

See OMP_exercises/pi.c

The library routine `omp_get_wtime()` is used to find the elapsed “wall time” for blocks of code.
Exercise: the parallel Pi program

• Create a parallel version of the pi program using a parallel construct:
  
  #pragma omp parallel.

• 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();
  - int omp_get_thread_num();
  - double omp_get_wtime();
  - omp_set_num_threads();

  Number of threads in the team
  Thread ID or rank
  Time in Seconds since a fixed point in the past
  Request a number of threads in the team
Hints: the Parallel Pi program

• Use a parallel construct:
  #pragma omp parallel

• The challenge is to:
  – divide loop iterations between threads (use the thread ID and the number of threads).
  – Create an accumulator for each thread to hold partial sums that you can later combine to generate the global sum.

• In addition to a parallel construct, you will need the runtime library routines
  – int omp_set_num_threads();
  – int omp_get_num_threads();
  – int omp_get_thread_num();
  – double omp_get_wtime();
Results*

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: A simple Parallel pi program

```c
#include <omp.h>
static long num_steps = 1000000; double step;
#define NUM_THREADS 2
void main()
{
    int i, nthreads; double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        int i, id, nthrd;
        double x;
        id = omp_get_thread_num();
        nthrd = omp_get_num_threads();
        if (id == 0) nthreads = nthrd;
        for (i=id, sum[id]=0.0; i<num_steps; i=i+nthrd) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }

    for (i=0, pi=0.0; i<nthreads; i++) pi += sum[i] * step;
}
```

<table>
<thead>
<tr>
<th>threads</th>
<th>1st SPMD*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.86</td>
</tr>
<tr>
<td>2</td>
<td>1.03</td>
</tr>
<tr>
<td>3</td>
<td>1.08</td>
</tr>
<tr>
<td>4</td>
<td>0.97</td>
</tr>
</tbody>
</table>

*SPMD: Single Program Multiple Data

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.
Why such poor scaling? 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”.

- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines … Results in poor scalability.

- Solution: Pad arrays so elements you use are on distinct cache lines.
Example: Eliminate false sharing by padding the sum array

```c
#include <omp.h>
static long num_steps = 100000;   double step;
#define PAD 8            // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{
    int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id,nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0)  nthreads = nthrds;
        for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id][0] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
 Pad the array so each sum value is in a different cache line
```
Results*: pi program padded accumulator

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: eliminate False sharing by padding the sum array

```c
#include <omp.h>
static long num_steps = 100000; double step;
#define PAD 8 // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main()
{
    int i, nthreads; double pi, sum[NUM_THREADS][PAD];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id,nthrs;
        double x;
        id = omp_get_thread_num();
        nthrs = omp_get_num_threads();
        if (id == 0) nththreads = nthrs;
        for (i=id, sum[id][0]=0.0;i<num_steps;i=i+nthrs) {
            x = (i+0.5)*step;
            sum[id][0] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

<table>
<thead>
<tr>
<th>threads</th>
<th>1st SPMD</th>
<th>1st SPMD padded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.86</td>
<td>1.86</td>
</tr>
<tr>
<td>2</td>
<td>1.03</td>
<td>1.01</td>
</tr>
<tr>
<td>3</td>
<td>1.08</td>
<td>0.69</td>
</tr>
<tr>
<td>4</td>
<td>0.97</td>
<td>0.53</td>
</tr>
</tbody>
</table>

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.
Outline

• Introduction to OpenMP
• Creating Threads
• Quantifying Performance and Amdahl’s law
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap
• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Threadprivate data
Synchronization

• High level synchronization included in the common core (the full OpenMP specification has MANY more):
  – critical
  – barrier

Synchronization is used to impose order constraints and to protect access to shared data
Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a critical region.

```c
float res;
#pragma omp parallel
{
    float B; int i, id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for(i=id;i<niters;i+=nthrds){
        B = big_job(i);
        #pragma omp critical
        res += consume(B);
    }
}
```
Synchronization: barrier

- Barrier: a point in a program all threads must reach before any threads are allowed to proceed.
- It is a “stand alone” pragma meaning it is not associated with user code ... it is an executable statement.

```c
double Arr[8], Brr[8]; int numthrds;
omp_set_num_threads(8)
#pragma omp parallel
{
    int id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id==0) numthrds = nthrds;
    Arr[id] = big_ugly_calc(id, nthrds);
    #pragma omp barrier
    Brr[id] = really_big_and_ugly(id, nthrds, A);
}
```

Threads wait until all threads hit the barrier. Then they can go on.
Exercise

• In your first Pi program, you probably used an array to create space for each thread to store its partial sum.

• If array elements happen to share a cache line, this leads to false sharing.
  – Non-shared data in the same cache line so each update invalidates the cache line … in essence “sloshing independent data” back and forth between threads.

• Modify your “pi program” to avoid false sharing due to the partial sum array.
  
  ```c
  int omp_get_num_threads();
  int omp_get_thread_num();
  double omp_get_wtime();
  omp_set_num_threads();
  #pragma parallel
  #pragma critical
  ```
Pi program with false sharing*

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: A simple Parallel pi program

```c
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main()
{
    int i, nthreads; double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        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);
        }
    }
    for(i=0, pi=0.0; i<nthreads; i++) pi += sum[i] * step;
}
```

Recall that promoting sum to an array made the coding easy, but led to false sharing and poor performance.

<table>
<thead>
<tr>
<th>threads</th>
<th>1st SPMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.86</td>
</tr>
<tr>
<td>2</td>
<td>1.03</td>
</tr>
<tr>
<td>3</td>
<td>1.08</td>
</tr>
<tr>
<td>4</td>
<td>0.97</td>
</tr>
</tbody>
</table>

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.*
Example: Using a critical section to remove impact of false sharing

```c
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
    int nthreads; double pi=0.0;            step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int i, id, nthrds; double x, sum,
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
#pragma omp critical
        pi += sum * step;
    }
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don’t conflict.
Results*: pi program critical section

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: Using a critical section to remove impact of false sharing

```c
#include <omp.h>
static long num_steps = 100000; double step;
define NUM_THREADS 2
void main ()
{
    int nthreads; double pi=0.0; step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds; double x, sum;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        pi += sum * step;
    }
}
```

<table>
<thead>
<tr>
<th>threads</th>
<th>1st SPMD</th>
<th>1st SPMD padded</th>
<th>SPMD critical</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.86</td>
<td>1.86</td>
<td>1.87</td>
</tr>
<tr>
<td>2</td>
<td>1.03</td>
<td>1.01</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>1.08</td>
<td>0.69</td>
<td>0.68</td>
</tr>
<tr>
<td>4</td>
<td>0.97</td>
<td>0.53</td>
<td>0.53</td>
</tr>
</tbody>
</table>

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    int nthreads; double pi=0.0; step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int i, id,nthrds; double x;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id == 0) nthreads = nthrds;
    for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
        x = (i+0.5)*step;
        #pragma omp critical
        pi += 4.0/(1.0+x*x);
    }
}
pi *= step;
}
Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl’s law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Threadprivate data
The loop worksharing constructs

- The loop worksharing construct splits up loop iterations among the threads in a team

```c
#pragma omp parallel
{
#pragma omp for
  for (I=0; I<N; I++){
    NEAT_STUFF(I);
  }
}
```

Loop construct name:
- C/C++: for
- Fortran: do

The loop control index I is made “private” to each thread by default.

Threads wait here until all threads are finished with the parallel loop before any proceed past the end of the loop.
Loop worksharing constructs
A motivating example

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];}
Loop worksharing constructs: The schedule clause

- 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.
  - #pragma omp for schedule(dYNAMIC, CHUNK)

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Pre-determined and predictable by the programmer</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
</tbody>
</table>
Combined parallel/worksharing construct

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

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

These are equivalent

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

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

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

Note: loop index "i" is private by default

Remove loop carried dependence
Reduction

• How do we handle this case?

```c
double ave=0.0, A[MAX]; int i;
for (i=0; i< MAX; i++) {
    ave += A[i];
}
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.
Reduction

• OpenMP reduction clause:
  
  reduction (op : list)

• Inside a parallel or a work-sharing 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.

```c
double ave=0.0, A[MAX]; int i;

#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++) {
   ave += A[i];
}
ave = ave/MAX;
```
OpenMP: Reduction operands/initial-values

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>min</td>
<td>Largest pos. number</td>
</tr>
<tr>
<td>max</td>
<td>Most neg. number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>.true.</td>
</tr>
<tr>
<td>.OR.</td>
<td>.false.</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>.false.</td>
</tr>
<tr>
<td>.IEOR.</td>
<td>0</td>
</tr>
<tr>
<td>.IOR.</td>
<td>0</td>
</tr>
<tr>
<td>.IAND.</td>
<td>All bits on</td>
</tr>
<tr>
<td>.EQV.</td>
<td>.true.</td>
</tr>
</tbody>
</table>
Exercise: Pi with loops

• Go back to the serial pi program and parallelize it with a loop construct
• Your goal is to minimize the number of changes made to the serial program.

```
#pragma omp parallel
#pragma omp for
#pragma omp parallel for
#pragma omp for reduction(op:list)
#pragma omp critical
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
```
#include <omp.h>

static long num_steps = 100000;  double step;

void main ()
{
    int i;  double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double 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 = step * sum;
}
Results*: pi with a loop and a reduction

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

```c
#include <omp.h>
static long num_steps = 1000000;
void main ()
{
    int i; double x, pi, sum;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double 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 = step * sum;
}
```
The nowait clause

- Barriers are really expensive. You need to understand when they are implied and how to skip them when it's safe to do so.

```c
double A[big], B[big], C[big];

#pragma omp parallel
{
    int id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #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 parallel region
- implicit barrier at the end of a for worksharing construct
- no implicit barrier due to nowait
Outline

• Introduction to OpenMP
• Creating Threads
• Quantifying Performance and Amdahl’s law
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap
• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Thread private data
Data environment: Default storage attributes

• Shared memory programming model:
  – Most variables are shared by default

• Global variables are SHARED among threads
  – Fortran: COMMON blocks, SAVE variables, MODULE variables
  – C: File scope variables, static
  – Both: dynamically allocated memory (ALLOCATE, malloc, new)

• But not everything is shared...
  – Stack variables in subprograms (Fortran) or functions (C) called from parallel regions are PRIVATE
  – Automatic variables within a statement block are PRIVATE.
Data sharing: Examples

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

A, index, count

A, index, count
Data sharing: Changing storage attributes

• One can selectively change storage attributes for constructs using the following clauses* (note: list is a comma-separated list of variables)
  – shared(list)
  – private(list)
  – firstprivate(list)
• These can be used on parallel and for constructs … other than shared which can only be used on a parallel construct

• Force the programmer to explicitly define storage attributes
  – default (none)

*default() can only be used on parallel constructs
Data sharing: Private clause

- `private(var)` creates a new local copy of `var` for each thread.
  - The value of the private copies is uninitialized.
  - The value of the original variable is unchanged after the region.

```c
void wrong() {
    int tmp = 0;
    #pragma omp parallel for private(tmp)
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
}
```

- `tmp` was not initialized when referenced.
- `tmp` is 0 here.

When you need to reference the variable `tmp` that exists prior to the construct, we call it the **original variable**.
Data sharing: Private and the original variable

• The original variable’s value is unspecified if it is referenced outside of the construct
  – Implementations may reference the original variable or a copy ….. a dangerous programming practice!
  – For example, consider what would happen if the compiler inlined work()? 

```c
int tmp;
void danger() {
    tmp = 0;
    #pragma omp parallel private(tmp)
    work();
    printf("%d\n", tmp);
}
```

```c
extern int tmp;
void work() {
    tmp = 5;
}
```

`tmp` has unspecified value

unspecified which copy of `tmp`
Firstprivate clause

- Variables initialized from a shared variable
- C++ objects are copy-constructed

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
    if ((i%2)==0) incr++;
    A[i] = incr;
}
```

Each thread gets its own copy of incr with an initial value of 0
Data sharing: A data environment test

• Consider this example of PRIVATE and FIRSTPRIVATE

variables:  \( A = 1, B = 1, C = 1 \)

#pragma omp parallel private(B) firstprivate(C)

• Are A, B, C private to each thread or shared inside the parallel region?
• What are their initial values inside and values after the parallel region?

Inside this parallel region ...

• “A” is shared by all threads; equals 1
• “B” and “C” are private to each thread.
  – B’s initial value is undefined
  – C’s initial value equals 1

Following the parallel region ...

• B and C revert to their original values of 1
• A is either 1 or the value it was set to inside the parallel region
Data sharing: Default clause

- **default(none):** Forces you to define the storage attributes for variables that appear inside the static extent of the construct … if you fail the compiler will complain. Good programming practice!
- You can put the default clause on parallel and parallel + workshare constructs.

```c
#include <omp.h>
int main()
{
    int i, j=5;     double x=1.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0; i<N; i++){
        for(j=0; j<3; j++)
            x+= foobar(i, j, y);
    }
    printf(" x is %f\n",(float)x);
}
```

The static extent is the code in the compilation unit that contains the construct.

The full OpenMP specification has other versions of the default clause, but they are not used very often so we skip them in the common core.
Exercise: Mandelbrot set area

• The supplied program (mandel.c) computes the area of a Mandelbrot set.

• The program has been parallelized with OpenMP, but we were lazy and didn’t do it right.

• Find and fix the errors (hint … the problem is with the data environment).

• Once you have a working version, try to optimize the program.
  – Try different schedules on the parallel loop.
  – Try different mechanisms to support mutual exclusion … do the efficiencies change?
The Mandelbrot area program

```c
#include <omp.h>
#define NPOINTS 1000
#define MXITR 1000
struct d_complex{
    double r; double i;
};
void testpoint(struct d_complex c);
struct d_complex c;
int numoutside = 0;

void testpoint(struct d_complex);

int main(){
    int i, j;
    double area, error, eps = 1.0e-5;
    #pragma omp parallel for default(shared) private(c, j) firstprivate(eps)
    for (i=0; i<NPOINTS; i++) {
        for (j=0; j<NPOINTS; j++) {
            c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
            c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
            testpoint(c);
        }
    }
    area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-numoutside)/(double)(NPOINTS*NPOINTS);
    error=area/(double)NPOINTS;
}
```
Outline

• Introduction to OpenMP
• Creating Threads
• Quantifying Performance and Amdahl’s law
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap
• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Thread private data
OpenMP memory model

- OpenMP supports a shared memory model
- All threads share an address space, but it can get complicated:

Multiple copies of data may be present in various levels of cache, or in registers.
OpenMP and relaxed consistency

- OpenMP supports a **relaxed-consistency** shared memory model
  - Threads can maintain a **temporary view** of shared memory that is not consistent with that of other threads
  - These temporary views are made consistent only at certain points in the program
  - The operation that enforces consistency is called the **flush operation**
Flush operation

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory
  - All previous read/writes by this thread have completed and are visible to other threads
  - No subsequent read/writes by this thread have occurred

- A flush operation is analogous to a fence in other shared memory APIs
Flush and synchronization

• A flush is implied where it makes sense:
  – at entry/exit of parallel regions
  – at implicit and explicit barriers
  – at entry/exit of critical regions
  ....
  (but not at entry to worksharing regions)

This means if you are mixing reads and writes of a variable across multiple threads, you cannot assume the reading threads see the results of the writes unless:

• the writing threads follow the writes with a construct that implies a flush.
• the reading threads proceed the reads with a construct that implies a flush.

This is a rare event … or putting this another way, you should avoid writing code that depends on ordering reads/writes around flushes.
Outline

• Introduction to OpenMP
• Creating Threads
• Quantifying Performance and Amdahl’s law
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap
• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Thread private data
What are tasks?

- Tasks are independent units of work

- Tasks are composed of:
  - code to execute
  - data to compute with

- Threads are assigned to perform the work of each task.
  - The thread that encounters the task construct may execute the task immediately.
  - The threads may defer execution until later
What are tasks?

- The task construct includes a structured block of code.

- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution.

- Tasks can be nested: i.e. a task may itself generate tasks.

A common Pattern is to have one thread create the tasks while the other threads wait at a barrier and execute the tasks.
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).

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

```c
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        fred();
        #pragma omp task
        daisy();
        #pragma omp task
        billy();
    }
}
```

Create some threads

One Thread packages tasks

Tasks executed by some thread in some order

All tasks complete before this barrier is released
Exercise: Simple tasks

• Write a program using tasks that will “randomly” generate one of two strings:
  – I think race cars are fun
  – I think car races are fun

• Hint: use tasks to print the indeterminate part of the output (i.e. the “race” or “car” parts).

• This is called a “Race Condition”. It occurs when the result of a program depends on how the OS schedules the threads.

• NOTE: A “data race” is when threads “race to update a shared variable”. They produce race conditions. Programs containing data races are undefined (in OpenMP but also ANSI standards C++’11 and beyond).

```cpp
#pragma omp parallel
#pragma omp task
#pragma omp single
```
Racey cars: solution

```c
#include <stdio.h>
#include <omp.h>
int main()
{
    printf("I think");
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            printf(" car");
            #pragma omp task
            printf(" race");
        }
    }
    printf("s");
    printf(" are fun!\n");
}```
Linked list traversal

\[
p = \text{listhead} ; \\
\text{while} \ (p) \ { \\
    \ \ \ \text{process}(p) ; \\
    \ \ \ p=\text{next}(p) ; \\
}\]

- Classic linked list traversal
- Do some work on each item in the list
- Assume that items can be processed independently
- Cannot use an OpenMP loop directive
Parallel linked list list traversal

```c
#pragma omp parallel
{
    #pragma omp single
    {
        p = listhead;
        while (p) {
            #pragma omp task firstprivate(p)
            {
                process (p);
            }
            p = next (p);
        }
    }
}
```

Only one thread packages tasks

makes a copy of `p` when the task is packaged
Data scoping with tasks

• Variables can be shared, private or firstprivate with respect to task

• These concepts are a little bit different compared with threads:
  – If a variable is shared on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered
  – If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
  – If a variable is firstprivate on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered
Data scoping defaults

• The behavior you want for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are private when the task construct is encountered are firstprivate by default
• Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```c
#pragma omp parallel shared(A) private(B)
{
    ...
    #pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

A is shared  
B is firstprivate  
C is private
When/where are tasks complete?

• At thread barriers (explicit or implicit)
  – applies to all tasks generated in the current parallel region up to the barrier

• At taskwait directive
  – i.e. Wait until all tasks defined in the current task have completed.
    #pragma omp taskwait
  – Note: applies only to tasks generated in the current task, not to “descendants”.
Example

```c
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        fred();
        #pragma omp task
daisy();
        #pragma taskwait
        #pragma omp task
        billy();
    }
}
```

fred() and daisy() must complete before billy() starts
Example: Fibonacci numbers

\[ F_n = F_{n-1} + F_{n-2} \]

Inefficient \( O(n^2) \) recursive implementation!

```c
int fib (int n) {
  int x, y;
  if (n < 2) return n;
  x = fib(n-1);
  y = fib(n-2);
  return (x+y);
}

int main() {
  int NW = 5000;
  fib(NW);
}
```
Parallel Fibonacci

```c
int fib (int n)
{
    int x,y;
    if (n < 2) return n;

    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return (x+y);
}

Int main()
{
    int NW = 5000;
    #pragma omp parallel
    {
        #pragma omp single
        fib(NW);
    }
}
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)
- \(x, y\) are local, and so by default they are private to current task
  - must be shared on child tasks so they don’t create their own firstprivate copies at this level!
Divide and conquer

- Split the problem into smaller sub-problems; continue until the sub-problems can be solve directly

3 Options:
- Do work as you split into sub-problems
- Do work only at the leaves
- Do work as you recombine
Exercise: Pi with tasks

• Consider the program Pi_recur.c. This program uses a recursive algorithm in integrate the function in the pi program.
  – Parallelize this program using OpenMP tasks

```c
#pragma omp parallel
#pragma omp task
#pragma omp taskwait
#pragma omp single
double omp_get_wtime()
int omp_get_thread_num();
int omp_get_num_threads();
```
Program: OpenMP tasks

```
#include <omp.h>

static long num_steps = 100000000;
#define MIN_BLK 10000000

double pi_comp(int Nstart, int Nfinish, double step)
{
    int i,iblk;
    double x, sum = 0.0,sum1, sum2;
    if (Nfinish-Nstart < MIN_BLK){
        for (i=Nstart;i< Nfinish; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    } else{
        iblk = Nfinish-Nstart;
        #pragma omp task shared(sum1)
            sum1 = pi_comp(Nstart, Nfinish-iblk/2,step);
        #pragma omp task shared(sum2)
            sum2 = pi_comp(Nfinish-iblk/2, Nfinish, step);
        #pragma omp taskwait
            sum = sum1 + sum2;
    }
    return sum;
}

int main ()
{
    int i;
    double step, pi, sum;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        #pragma omp single
        sum = pi_comp(0,num_steps,step);
    }
    pi = step * sum;
}  
```
## Results*: pi with tasks

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

<table>
<thead>
<tr>
<th>threads</th>
<th>1\textsuperscript{st} SPMD</th>
<th>SPMD critical</th>
<th>PI Loop</th>
<th>Pi tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.86</td>
<td>1.87</td>
<td>1.91</td>
<td>1.87</td>
</tr>
<tr>
<td>2</td>
<td>1.03</td>
<td>1.00</td>
<td>1.02</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>1.08</td>
<td>0.68</td>
<td>0.80</td>
<td>0.76</td>
</tr>
<tr>
<td>4</td>
<td>0.97</td>
<td>0.53</td>
<td>0.68</td>
<td>0.52</td>
</tr>
</tbody>
</table>
Using tasks

• Don’t use tasks for things already well supported by OpenMP
  – e.g. standard do/for loops
  – the overhead of using tasks is greater

• Don’t expect miracles from the runtime
  – best results usually obtained where the user controls the number and granularity of tasks
Outline

• Introduction to OpenMP
• Creating Threads
• Quantifying Performance and Amdahl’s law
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap
• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Thread private data
### The OpenMP Common Core: Most OpenMP programs only use these 16 constructs

<table>
<thead>
<tr>
<th>OMP Construct</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp parallel</td>
<td>parallel region, teams of threads, structured block, interleaved execution across threads</td>
</tr>
<tr>
<td>int omp_get_thread_num()</td>
<td>Create threads with a parallel region and split up the work using the number of threads and thread ID</td>
</tr>
<tr>
<td>int omp_get_num_threads()</td>
<td>Speedup and Amdahl's law. False Sharing and other performance issues</td>
</tr>
<tr>
<td>double omp_get_wtime()</td>
<td>internal control variables. Setting the default number of threads with an environment variable</td>
</tr>
<tr>
<td>setenv OMP_NUM_THREADS N</td>
<td>Synchronization and race conditions. Revisit interleaved execution.</td>
</tr>
<tr>
<td>#pragma omp barrier</td>
<td>worksharing, parallel loops, loop carried dependencies</td>
</tr>
<tr>
<td>#pragma omp critical</td>
<td>reductions of values across a team of threads</td>
</tr>
<tr>
<td>#pragma omp for</td>
<td>Loop schedules, loop overheads and load balance</td>
</tr>
<tr>
<td>#pragma omp parallel for</td>
<td>Data environment</td>
</tr>
<tr>
<td>reduction(op:list)</td>
<td>disabling implied barriers on workshare constructs, the high cost of barriers. The flush concept (but not the concept)</td>
</tr>
<tr>
<td>schedule(dynamic [,chunk])</td>
<td>Workshare with a single thread</td>
</tr>
<tr>
<td>schedule (static [,chunk])</td>
<td>tasks including the data environment for tasks.</td>
</tr>
<tr>
<td>private(list), firstprivate(list), shared(list)</td>
<td></td>
</tr>
</tbody>
</table>
There is much more to OpenMP than the Common Core.

• Synchronization mechanisms
  – locks, flush and several forms of atomic

• Data environment
  – lastprivate, threadprivate, default(private|shared)

• Fine grained task control
  – dependencies, tied vs. untied tasks, task groups, task loops …

• Vectorization constructs
  – simd, uniform, simdlen, inbranch vs. nobranch, …. 

• Map work onto an attached device
  – target, teams distribute parallel for, target data …

• … and much more. The OpenMP 4.5 specification is over 350 pages!!!

Don’t become overwhelmed. Master the common core and move on to other constructs when you encounter problems that require them.
OpenMP organizations

• OpenMP architecture review board URL, the “owner” of the OpenMP specification:
  www.openmp.org

• OpenMP User’s Group (cOMPunity) URL:
  www.compunity.org

Get involved, join cOMPunity and help define the future of OpenMP
Books about OpenMP

• A book about OpenMP by a team of authors at the forefront of OpenMP’s evolution.

• A book about how to “think parallel” with examples in OpenMP, MPI and java
A great book that explores key patterns with Cilk, TBB, OpenCL, and OpenMP (by McCool, Robison, and Reinders)

An excellent introduction and overview of multithreaded programming in general (by Clay Breshears)
Outline

• Introduction to OpenMP
• Creating Threads
• Quantifying Performance and Amdahl’s law
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap

• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Thread private data
Outline

• Introduction to OpenMP
• Creating Threads
• Quantifying Performance and Amdahl’s law
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap
• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Thread private data
The loop worksharing constructs

- The loop worksharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
{
#pragma omp for
    for (I=0;I<N;I++){
        NEAT_STUFF(I);
    }
}
```

Loop construct name:
- C/C++: for
- Fortran: do

The variable I is made “private” to each thread by default. You could do this explicitly with a “private(I)” clause.
Loop worksharing constructs: The schedule clause

- 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).
  - `schedule(auto)`
    - Schedule is left up to the runtime to choose (does not have to be any of the above).

OpenMP 4.5 added modifiers monotonic, nonmonotonic and simd.
## Loop Work-sharing Constructs: The Schedule Clause

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Pre-determined and predictable by the programmer</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
<tr>
<td>AUTO</td>
<td>When the runtime can “learn” from previous executions of the same loop</td>
</tr>
</tbody>
</table>

- **STATIC**: Least work at runtime: scheduling done at compile-time
- **DYNAMIC**: Most work at runtime: complex scheduling logic used at run-time
Nested loops

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

```c
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
        ......
    }
}
```

- Will form a single loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.
Sections worksharing Construct

• The Sections worksharing construct gives a different structured block to each thread.

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        X_calculation();
        #pragma omp section
        y_calculation();
        #pragma omp section
        z_calculation();
    }
}
```

By default, there is a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.
Array sections with reduce

#include <stdio.h>
#define N 100
void init(int n, float (*b)[N]);

int main(){
  int i,j; float a[N], b[N][N]; init(N,b);
  for(i=0; i<N; i++) a[i]=0.0e0;

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

  printf(" a[0] a[N-1]: %f %f\n", a[0], a[N-1]);
  return 0;

Works the same as the other reductions:
  • A private array is formed for each thread.
  • Initialized according to the identify for the operator.
  • Element wise combination across threads at end of the construct.
  • Elementwise combination with original array at the end of the construct.
Outline

• Introduction to OpenMP
• Creating Threads
• Quantifying Performance and Amdahl’s law
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap
• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Thread private data
Synchronization

- High level synchronization:
  - critical
  - barrier
  - atomic
  - ordered

- Low level synchronization
  - flush
  - locks (both simple and nested)

Synchronization is used to impose order constraints and to protect access to shared data.
Synchronization: atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```c
#pragma omp parallel
{
    double B;
    B = DOIT();

    #pragma omp atomic
    X += big_ugly(B);
}
```
Synchronization: atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of $X$ in the following example)

```cpp
#pragma omp parallel
{
    double B, tmp;
    B = DOIT();
    tmp = big_ugly(B);
    #pragma omp atomic
    X += tmp;
}
```

Additional forms of atomic were added in 3.1 (discussed later)
Exercise

• In your first Pi program, you probably used an array to create space for each thread to store its partial sum.

• You fixed this by using a critical section instead of updating the array (remember .. the array you created by promoting the scalar “sum” to an array).

• Use and atomic instead. Does the performance improve?
Parallel loop with ordered region

• An **ordered clause** on a loop worksharing construct
  – indicates that the loop contains an ordered region

• The **ordered construct** defines an ordered region
  – The Statements in ordered region execute in iteration order

```c
#pragma omp for ordered
for (i=0; i<N; i++) {
    float res = work(i);
    #pragma omp ordered
    {
        printf("result for %d was %f\n", i, res);
        fflush(stdout);
    }
}
```
Parallelizing nested loops

• Will these nested parallel loops execute correctly?

```c
#pragma omp parallel for collapse(2)
for (r=1; r<N; r++) {
    for (c=1; c<N; c++) {
        x[r][c] += fn(x[r-1][c], x[r][c-1]);
    }
}
```

• Pattern of dependencies between elements of x prevent straightforward parallelization
• is there a way to manage the synchronization so we can parallelize this loop?
Ordered stand-alone directive

- Specifies cross-iteration dependencies in a doacross loop nest … i.e. loop level parallelism over nested loops with a regular pattern of synchronization to manage dependencies.

```
#pragma omp ordered depend(sink : vec)
#pragma omp ordered depend(source)
```

- **Depend** clauses specify the order the threads execute **ordered** regions.
  - The **sink dependence-type**
    - specifies a cross-iteration dependence, where the iteration vector `vec` indicates the iteration that satisfies the dependence.
  - The **source dependence-type**
    - specifies the cross-iteration dependences that arise from the current iteration.
Parallelizing DOACROSS loops

2 loops contribute to the pattern of dependencies … so the dependency relations for each `depend(sink)` is of length 2

```c
#pragma omp for ordered(2) collapse(2)
for (r=1; r<N; r++) {
    for (c=1; c<N; c++) {
        // other parallel work ...
        #pragma omp ordered depend(sink:r-1,c) \ depend(sink:r,c-1)
        x[r][c] += fn(x[r-1][c], x[r][c-1]);
        #pragma omp ordered depend(source)
    }
}
```

Threads wait here until `x[r-1][c]` and `x[r][c-1]` have been released.

`x[r][c]` is complete and released for use by other threads.
OpenMP memory model

- OpenMP supports a shared memory model
- All threads share an address space, where variable can be stored or retrieved:

  ![Diagram](https://via.placeholder.com/150)

  - Threads maintain their own temporary view of memory as well … the details of which are not defined in OpenMP but this temporary view typically resides in caches, registers, write-buffers, etc.
Flush operation

• Defines a sequence point at which a thread enforces a consistent view of memory.

• For variables visible to other threads and associated with the flush operation (the.flush-set)
  – The compiler can’t move loads/stores of the flush-set around a flush:
    – All previous read/writes of the flush-set by this thread have completed
    – No subsequent read/writes of the flush-set by this thread have occurred
  – Variables in the flush set are moved from temporary storage to shared memory.
  – Reads of variables in the flush set following the flush are loaded from shared memory.

IMPORTANT POINT: The flush makes the calling threads temporary view match the view in shared memory. Flush by itself does not force synchronization.
Memory consistency: flush example

- Flush forces data to be updated in memory so other threads see the most recent value

```c
double A;
A = compute();
#pragma omp flush(A)
// flush to memory to make sure other
// threads can pick up the right value
```

Flush without a list: flush set is all thread visible variables

Flush with a list: flush set is the list of variables

Note: OpenMP’s flush is analogous to a fence in other shared memory APIs
Flush and synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.,
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions
  - whenever a lock is set or unset

  ....

  (but not at entry to worksharing regions or entry/exit of master regions)
Example: prod_cons.c

- Parallelize a producer/consumer program
  - One thread produces values that another thread consumes.

```c
int main()
{
    double *A, sum, runtime;    int flag = 0;

    A = (double *) malloc(N*sizeof(double));

    runtime = omp_get_wtime();

    fill_rand(N, A);           // Producer: fill an array of data

    sum = Sum_array(N, A);     // Consumer: sum the array

    runtime = omp_get_wtime() - runtime;

    printf(" In %lf secs, The sum is %lf \n",runtime,sum);
}
```

- Often used with a stream of produced values to implement “pipeline parallelism”
- The key is to implement pairwise synchronization between threads
Pairwise synchronization in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.
- When needed, you have to build it yourself.
- Pairwise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory
int main()
{
    double *A, sum, runtime;   int numthreads, flag = 0;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel
    sections
    {
        #pragma omp section
        {
            fill_rand(N, A);
            flag = 1;
        }
        #pragma omp section
        {
            while (flag == 0){
            }
            sum = Sum_array(N, A);
        }
    }
}

Put the flushes in the right places to make this program race-free.

Do you need any other synchronization constructs to make this work?
Solution (try 1): Producer/consumer

```c
int main()
{
    double *A, sum, runtime;   int numthreads, flag = 0;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            fill_rand(N, A);
            #pragma omp flush
            flag = 1;
            #pragma omp flush (flag)
        }
        #pragma omp section
        {
            #pragma omp flush (flag)
            while (flag == 0){
                #pragma omp flush (flag)
            }
            #pragma omp flush
            sum = Sum_array(N, A);
        }
    }
}
```

Use flag to Signal when the “produced” value is ready

Flush forces refresh to memory; guarantees that the other thread sees the new value of A

Flush needed on both “reader” and “writer” sides of the communication

Notice you must put the flush inside the while loop to make sure the updated flag variable is seen

This program works with the x86 memory model (loads and stores use relaxed atomics), but it technically has a race … on the store and later load of flag
The OpenMP 3.1 atomics (1 of 2)

• Atomic was expanded to cover the full range of common scenarios where you need to protect a memory operation so it occurs atomically:

  # pragma omp atomic [read | write | update | capture]

• Atomic can protect loads

  # pragma omp atomic read
  v = x;

• Atomic can protect stores

  # pragma omp atomic write
  x = expr;

• Atomic can protect updates to a storage location (this is the default behavior … i.e. when you don’t provide a clause)

  # pragma omp atomic update
  x++; or ++x; or x--; or –x; or
  x binop= expr; or x = x binop expr;
The OpenMP 3.1 atomics (2 of 2)

- Atomic can protect the assignment of a value (its capture) AND an associated update operation:
  
  ```
  # pragma omp atomic capture
  statement or structured block
  ```

- Where the statement is one of the following forms:
  
  ```
  v = x++;       v = ++x;        v = x--;       v = −x;       v = x binop expr;
  ```

- Where the structured block is one of the following forms:
  
  ```
  {v = x;  x binop = expr;}
  {v=x;  x=x binop expr;}
  {v = x;  x++;}
  {++x;  v=x:}
  {v = x;  x--;}
  {--x;  v = x;}
  {x binop = expr;  v = x;}
  {X = x binop expr;  v = x;}
  {v=x;  ++x:}
  {x++;  v = x;}
  {v= x;  --x;}
  {x--;  v = x;}
  {--x;  v = x;}
  ```

The capture semantics in atomic were added to map onto common hardware supported atomic operations and to support modern lock free algorithms.
Atoms and synchronization flags

```c
int main()
{
    double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            fill_rand(N, A);
            #pragma omp flush
            #pragma omp atomic write
            flag = 1;
            #pragma omp flush (flag)
        }
        #pragma omp section
        {
            while (1)
            {
                #pragma omp flush(flag)
                #pragma omp atomic read
                flg_tmp = flag;
                if (flg_tmp==1) break;
            }
            #pragma omp flush
            sum = Sum_array(N, A);
        }
    }
}
```

This program is truly race free … the reads and writes of flag are protected so the two threads cannot conflict

Still painful and error prone due to all of the flushes that are required
OpenMP 4.0 Atomic: Sequential consistency

• Sequential consistency:
  – The order of loads and stores in a race-free program appear in some interleaved order and all threads in the team see this same order.

• OpenMP 4.0 added an optional clause to atomics
  – #pragma omp atomic [read | write | update | capture] [seq_cst]

• In more pragmatic terms:
  – If the seq_cst clause is included, OpenMP adds a flush without an argument list to the atomic operation so you don’t need to.

• In terms of the C++’11 memory model:
  – Use of the seq_cst clause makes atomics follow the sequentially consistent memory order.
  – Leaving off the seq_cst clause makes the atomics relaxed.

Advice to programmers: save yourself a world of hurt … let OpenMP take care of your flushes for you whenever possible … use seq_cst
int main()
{
    double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            fill_rand(N, A);

            #pragma omp atomic write seq_cst
            flag = 1;
        }

        #pragma omp section
        {
            while (1){

                #pragma omp atomic read seq_cst
                flg_tmp= flag;
                if (flg_tmp==1) break;
            }
        }

        sum = Sum_array(N, A);
    }
}
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()`

Note: a thread always accesses the most recent copy of the lock, so you don’t need to use a flush on the lock variable.

Locks with hints were added in OpenMP 4.5 to suggest a lock strategy based on intended use (e.g. contended, uncontended, speculative, unspeculative)
Lock Example from Gafort (SpecOMP’2001)

- Genetic algorithm
- Most “interesting” loop: shuffle the population.
  - Original loop is not parallel; performs pair-wise swap of an array element with another, randomly selected element. There are 40,000 elements.
  - Parallelization idea:
    - Perform the swaps in parallel
    - Need to prevent simultaneous access to same array element: use one lock per array element $\rightarrow$ 40,000 locks.
Parallel loop Gafort shuffle routine

```c
omp_lock_t chrom_lcks[NPOP];
pop_t pop[NPOP];

#pragma omp parallel for
for(i=0;i<NPOP;i++){
    oother = (int) pick_random(i,NPOP);
    if(i<oother){
        omp_set_lock(&pop_lcks[i]);
        omp_set_lock(&pop_lcks[oother]);
    }
    else{
        omp_set_lock(&pop_lcks[oother]);
        omp_set_lock(&pop_lcks[i]);
    }
    mix_chroms(i, oother, pop);
    keep_fittest(l, oother, pop);
    if(i<oother){
        omp_unset_lock(&pop_lcks[oother]);
        omp_unset_lock(&pop_lcks[i]);
    }
    else{
        omp_unset_lock(&pop_lcks[i]);
        omp_unset_lock(&pop_lcks[oother]);
    }
}
```

Allocate then initialize locks

Exclusive access to array elements. Ordered locking prevents deadlock.
Exercise

- We provide a program in the file hist.c
- This program tests our random number generator by calling it many times and producing a histogram of the results.
- Parallelize this program.

```c
omp_lock_t lck;
omp_init_lock(&lck);
omp_set_lock(&lck);
omp_unset_lock(&lck);
omp_destroy_lock(&lck);
```
Synchronization: Simple locks

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

```c
#pragma omp parallel for
for (i=0; i<NBUCKETS; i++){
    omp_init_lock(&hist_locks[i]);
    hist[i] = 0;
}
#pragma omp parallel for
for (i=0; i<NVALS; i++){
    ival = (int) sample(arr[i]);
    omp_set_lock(&hist_locks[ival]);
    hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
}
for (i=0; i<NBUCKETS; i++)
    omp_destroy_lock(&hist_locks[i]);
```

- One lock per element of hist
- Enforce mutual exclusion on update to hist array
- Free-up storage when done.
Outline

• Introduction to OpenMP
• Creating Threads
• Quantifying Performance and Amdahl’s law
• Synchronization
• Parallel Loops
• Data environment
• Memory model
• Irregular Parallelism and tasks
• Recap
• Beyond the common core:
  – Worksharing revisited
  – Synchronization: More than you ever wanted to know
  – Thread private data
Data sharing: Threadprivate

• Makes global data private to a thread
  – Fortran: COMMON blocks
  – C: File scope and static variables, static class members
• Different from making them PRIVATE
  – with PRIVATE global variables are masked.
  – THREADPRIVATE preserves global scope within each thread
• Threadprivate variables can be initialized using COPYIN or at time of definition (using language-defined initialization capabilities)
A threadprivate example (C)

Use threadprivate to create a counter for each thread.

```c
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```
Data copying: Copyin

You initialize threadprivate data using a copyin clause.

```fortran
parameter (N=1000)
common,buf/A(N)
!$OMP THREADPRIVATE(/buf/)

!$ Initialize the A array
call init_data(N,A)

!$OMP PARALLEL COPYIN(A)

... Now each thread sees threadprivate array A initialized
... to the global value set in the subroutine init_data()

!$OMP END PARALLEL

end
```
Data copying: Copyprivate

Used with a single region to broadcast values of privates from one member of a team to the rest of the team

```
#include <omp.h>
void input_parameters (int, int); // fetch values of input parameters
void do_work(int, int);

void main()
{
    int Nsize, choice;

    #pragma omp parallel private (Nsize, choice)
    {
        #pragma omp single copyprivate (Nsize, choice)
        input_parameters (*Nsize, *choice);

        do_work(Nsize, choice);
    }
}
```
Exercise: Monte Carlo calculations
Using random numbers to solve tough problems

• Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
• Example: Computing \( \pi \) with a digital dart board:

\[
\begin{align*}
&\text{Throw darts at the circle/square.} \\
&\text{Chance of falling in circle is proportional to ratio of areas:} \\
&A_c = r^2 \cdot \pi \\
&A_s = (2r) \times (2r) = 4 \cdot r^2 \\
&P = \frac{A_c}{A_s} = \frac{\pi}{4}
\end{align*}
\]

• Compute \( \pi \) by randomly choosing points; \( \pi \) is four times the fraction that falls in the circle

<table>
<thead>
<tr>
<th>( N )</th>
<th>( \pi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.8</td>
</tr>
<tr>
<td>100</td>
<td>3.16</td>
</tr>
<tr>
<td>1000</td>
<td>3.148</td>
</tr>
</tbody>
</table>
Exercise: Monte Carlo pi (cont)

• We provide three files for this exercise
  – pi_mc.c: the Monte Carlo method pi program
  – random.c: a simple random number generator
  – random.h: include file for random number generator

• Create a parallel version of this program without changing the interfaces to functions in random.c
  – This is an exercise in modular software … why should a user of your parallel random number generator have to know any details of the generator or make any changes to how the generator is called?
  – The random number generator must be thread-safe.

• Extra Credit:
  – Make your random number generator numerically correct (non-overlapping sequences of pseudo-random numbers).
Appendices

• Challenge Problems
• Challenge Problems: solutions
  – Monte Carlo PI and random number generators
  – Molecular dynamics
  – Matrix multiplication
  – Linked lists
  – Recursive matrix multiplication
• Fortran and OpenMP
Challenge problems

• Long term retention of acquired skills is best supported by “random practice”.
  – i.e., a set of exercises where you must draw on multiple facets of the skills you are learning.

• To support “Random Practice” we have assembled a set of “challenge problems”
  1. Parallel molecular dynamics
  2. Optimizing matrix multiplication
  3. Traversing linked lists in different ways
  4. Recursive matrix multiplication algorithms
Challenge 1: Molecular dynamics

- The code supplied is a simple molecular dynamics simulation of the melting of solid argon.
- Computation is dominated by the calculation of force pairs in subroutine `forces` (in `forces.c`).
- Parallelise this routine using a parallel for construct and atomics; think carefully about which variables should be SHARED, PRIVATE or REDUCTION variables.
- Experiment with different schedule kinds.
Challenge 1: MD (cont.)

• Once you have a working version, move the parallel region out to encompass the iteration loop in main.c
  – Code other than the forces loop must be executed by a single thread (or workshared).
  – How does the data sharing change?

• The atomics are a bottleneck on most systems.
  – This can be avoided by introducing a temporary array for the force accumulation, with an extra dimension indexed by thread number
  – Which thread(s) should do the final accumulation into f?
Another option is to use locks
- Declare an array of locks
- Associate each lock with some subset of the particles
- Any thread that updates the force on a particle must hold the corresponding lock
- Try to avoid unnecessary acquires/releases
- What is the best number of particles per lock?
Challenge 2: Matrix multiplication

- Parallelize the matrix multiplication program in the file matmul.c
- Can you optimize the program by playing with how the loops are scheduled?
- Try the following and see how they interact with the constructs in OpenMP
  - Alignment
  - Cache blocking
  - Loop unrolling
  - Vectorization
- Goal: Can you approach the peak performance of the computer?
Challenge 3: Traversing linked lists

• Consider the program linked.c
  – Traverses a linked list, computing a sequence of Fibonacci numbers at each node

• Parallelize this program two different ways
  1. Use OpenMP tasks
  2. Use anything you choose in OpenMP other than tasks.

• The second approach (no tasks) can be difficult and may take considerable creativity in how you approach the problem (why its such a pedagogically valuable problem)
Challenge 4: Recursive matrix multiplication

• The following three slides explain how to use a recursive algorithm to multiply a pair of matrices
• Source code implementing this algorithm is provided in the file matmul_recur.c
• Parallelize this program using OpenMP tasks
Challenge 4: Recursive matrix multiplication

- Quarter each input matrix and output matrix
- Treat each submatrix as a single element and multiply
- 8 submatrix multiplications, 4 additions

\[
C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}
\]
\[
C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}
\]
\[
C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}
\]
\[
C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}
\]
Challenge 4: Recursive matrix multiplication
How to multiply submatrices?

• Use the same routine that is computing the full matrix multiplication
  – Quarter each input submatrix and output submatrix
  – Treat each sub-submatrix as a single element and multiply

\[
C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}
\]

\[
C_{11_{1,1}} = A_{11_{1,1}} \cdot B_{11_{1,1}} + A_{11_{1,2}} \cdot B_{11_{2,1}} + A_{11_{2,1}} \cdot B_{11_{1,1}} + A_{11_{2,2}} \cdot B_{11_{2,1}}
\]
Challenge 4: Recursive matrix multiplication

Recursively multiply submatrices

\begin{align*}
C_{1,1} &= A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\
C_{2,1} &= A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\
C_{1,2} &= A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\
C_{2,2} &= A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}
\end{align*}

- Need range of indices to define each submatrix to be used

```c
void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                double **A, double **B, double **C)
{
    // Dimensions: A[mf..ml][pf..pl]  B[pf..pl][nf..nl]  C[mf..ml][nf..nl]

    // C11 += A11*B11
    matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A,B,C);
    // C11 += A12*B21
    matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A,B,C);
    ...
}
```

- Also need stopping criteria for recursion
Appendices

• Challenge Problems
• Challenge Problems: solutions
  – Monte Carlo PI and random number generators
    – Molecular dynamics
    – Matrix multiplication
    – Linked lists
    – Recursive matrix multiplication
• Fortran and OpenMP
Computers and random numbers

• We use “dice” to make random numbers:
  – Given previous values, you cannot predict the next value.
  – There are no patterns in the series … and it goes on forever.

• Computers are deterministic machines … set an initial state, run a sequence of predefined instructions, and you get a deterministic answer
  – By design, computers are not random and cannot produce random numbers.

• However, with some very clever programming, we can make “pseudo random” numbers that are as random as you need them to be … but only if you are very very careful.

• Why do I care? Random numbers drive statistical methods used in countless applications:
  – Sample a large space of alternatives to find statistically good answers (Monte Carlo methods).
Monte Carlo Calculations
Using Random numbers to solve tough problems

• Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
• Example: Computing π with a digital dart board:

![](image)

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:
  \[ A_c = r^2 \times \pi \]
  \[ A_s = (2r) \times (2r) = 4 \times r^2 \]
  \[ P = \frac{A_c}{A_s} = \frac{\pi}{4} \]
- Compute π by randomly choosing points, count the fraction that falls in the circle, compute π.

<table>
<thead>
<tr>
<th>N</th>
<th>π</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.8</td>
</tr>
<tr>
<td>100</td>
<td>3.16</td>
</tr>
<tr>
<td>1000</td>
<td>3.148</td>
</tr>
</tbody>
</table>
Parallel Programmers love Monte Carlo algorithms

#include "omp.h"
static long num_trials = 10000;
int main ()
{
    long i;   long Ncirc = 0;   double pi, x, y;
double r = 1.0;   // radius of circle. Side of square is 2*r
seed(0,-r, r);  // The circle and square are centered at the origin
#pragma omp parallel for private (x, y) reduction (+:Ncirc)
for(i=0;i<num_trials; i++)
{
    x = random();   y = random();
    if ( x*x + y*y) <= r*r)  Ncirc++;
}

pi = 4.0 * ((double)Ncirc/(double)num_trials);
printf("%d trials, pi is %f \n",num_trials, pi);
}
Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

```plaintext
random_next = (MULTIPLIER * random_last + ADDEND) % PMOD;
random_last = random_next;
```

- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025
LCG code

static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;

    return ((double)random_next/(double)PMOD);
}
Running the PI_MC program with LCG generator

Log$_{10}$ number of samples

Log$_{10}$ Relative error

Run the same program the same way and get different answers!
That is not acceptable!

Issue: my LCG generator is not threadsafe

Program written using the Intel C/C++ compiler (10.0.659.2005) in Microsoft Visual studio 2005 (8.0.50727.42) and running on a dual-core laptop (Intel T2400 @ 1.83 Ghz with 2 GB RAM) running Microsoft Windows XP.
LCG code: threadsafe version

```c
static long MULTIPLIER  = 1366;
static long ADDEND     = 150889;
static long PMOD       = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;

    return ((double)random_next/(double)PMOD);
}
```

random_last carries state between random number computations,
To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

random_last carries state between random number computations,
To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.
Thread safe random number generators

Thread safe version gives the same answer each time you run the program. But for large number of samples, its quality is lower than the one thread result!

Why?
Pseudo Random Sequences

- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG.

- In a typical problem, you grab a subsequence of the RNG range.

- Grab arbitrary seeds and you may generate overlapping sequences:
  - E.g. three sequences ... last one wraps at the end of the RNG period.

- Overlapping sequences = over-sampling and bad statistics ... lower quality or even wrong answers!
Parallel random number generators

• Multiple threads cooperate to generate and use random numbers.
• Solutions:
  – Replicate and Pray
  – Give each thread a separate, independent generator
  – Have one thread generate all the numbers.
  – Leapfrog … deal out sequence values “round robin” as if dealing a deck of cards.
  – Block method … pick your seed so each threads gets a distinct contiguous block.
• Other than “replicate and pray”, these are difficult to implement. Be smart … get a math library that does it right.

If done right, can generate the same sequence regardless of the number of threads …

Nice for debugging, but not really needed scientifically.

Intel’s Math kernel Library supports a wide range of parallel random number generators.

For an open alternative, the state of the art is the Scalable Parallel Random Number Generators Library (SPRNG): http://www.sprng.org/ from Michael Mascagni’s group at Florida State University.
MKL Random number generators (RNG)

- MKL includes several families of RNGs in its vector statistics library.
- Specialized to efficiently generate vectors of random numbers

```
#define BLOCK 100
double buff[BLOCK];
VSLStreamStatePtr stream;

vslNewStream(&ran_stream, VSL_BRNG_WH, (int)seed_val);
vdRngUniform (VSL_METHOD_DUNIFORM_STD, stream,
              BLOCK, buff, low, hi)

vslDeleteStream( &stream );
```

- Initialize a stream or pseudo random numbers
- Select type of RNG and set seed
- Fill buff with BLOCK pseudo rand. nums, uniformly distributed with values between lo and hi.
- Delete the stream when you are done
Wichmann-Hill generators (WH)

- WH is a family of 273 parameter sets each defining a non-overlapping and independent RNG.
- Easy to use, just make each stream threadprivate and initiate RNG stream so each thread gets a unique WG RNG.

```c
VSLStreamStatePtr stream;
#pragma omp threadprivate(stream)
...
vsINewStream(&ran_stream, VSL_BRNG_WH+Thrd_ID, (int)seed);
```
Independent Generator for each thread

Notice that once you get beyond the high error, small sample count range, adding threads doesn’t decrease quality of random sampling.
Leap Frog method

- Interleave samples in the sequence of pseudo random numbers:
  - Thread i starts at the i
    th
  number in the sequence
  - Stride through sequence, stride length = number of threads.
- Result … the same sequence of values regardless of the number of threads.

```c
#pragma omp single
{ 
  nthreads = omp_get_num_threads();
  iseed = PMOD/MULTIPLIER;    // just pick a seed
  pseed[0] = iseed;
  mult_n = MULTIPLIER;
  for (i = 1; i < nthreads; ++i)
  { 
    iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
    pseed[i] = iseed;
    mult_n = (mult_n * MULTIPLIER) % PMOD;
  }
}

random_last = (unsigned long long) pseed[id];
```

One thread computes offsets and strided multiplier

LCG with Addend = 0 just to keep things simple

Each thread stores offset starting point into its threadprivate “last random” value
Same sequence with many threads.

- We can use the leapfrog method to generate the same answer for any number of threads

<table>
<thead>
<tr>
<th>Steps</th>
<th>One thread</th>
<th>2 threads</th>
<th>4 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>3.156</td>
<td>3.156</td>
<td>3.156</td>
</tr>
<tr>
<td>10000</td>
<td>3.1168</td>
<td>3.1168</td>
<td>3.1168</td>
</tr>
<tr>
<td>100000</td>
<td>3.13964</td>
<td>3.13964</td>
<td>3.13964</td>
</tr>
<tr>
<td>1000000</td>
<td>3.140348</td>
<td>3.140348</td>
<td>3.140348</td>
</tr>
<tr>
<td>10000000</td>
<td>3.141658</td>
<td>3.141658</td>
<td>3.141658</td>
</tr>
</tbody>
</table>

Used the MKL library with two generator streams per computation: one for the x values (WH) and one for the y values (WH+1). Also used the leapfrog method to deal out iterations among threads.
Appendices

• Challenge Problems
• Challenge Problems: solutions
  – Monte Carlo PI and random number generators
  – Molecular dynamics
    – Matrix multiplication
    – Linked lists
    – Recursive matrix multiplication
• Fortran and OpenMP
Molecular dynamics: Solution

```c
#pragma omp parallel for default (none) \ 
    shared(x,f,npart,rcoll,side) \ 
    reduction(+:epot,vir) \ 
    schedule (static,32)
for (int i=0; i<npart*3; i+=3) {
    ..........
```

Compiler will warn you if you have missed some variables

Loop is not well load balanced: best schedule has to be found by experiment.
Molecular dynamics: Solution (cont.)

```
#pragma omp atomic
    f[j]   -= forcex;
#pragma omp atomic
    f[j+1] -= forcey;
#pragma omp atomic
    f[j+2] -= forcez;

#pragma omp atomic
    f[i]   += fxi;
#pragma omp atomic
    f[i+1] += fyi;
#pragma omp atomic
    f[i+2] += fzi;
```
Implicit barrier needed to avoid race condition with update of reduction variables at end of the for construct

All variables which used to be shared here are now implicitly determined
Molecular dynamics: With array reduction

```c
ftemp[myid][j] -= forcex;
ftemp[myid][j+1] -= forcey;
ftemp[myid][j+2] -= forcez;
}
}
ftemp[myid][i] += fxi;
ftemp[myid][i+1] += fyi;
ftemp[myid][i+2] += fzi;
}
```

Replace atomics with accumulation into array with extra dimension
Molecular dynamics: With array reduction

....

#pragma omp for
for (int i = 0; i < (npart*3); i++) {
    for (int id = 0; id < nthreads; id++) {
        f[i] += ftemp[id][i];
        ftemp[id][i] = 0.0;
    }
}

Reduction can be done in parallel

Zero ftemp for next time round
Appendices

• Challenge Problems
• Challenge Problems: solutions
  – Monte Carlo PI and random number generators
  – Molecular dynamics
  – Matrix multiplication
    – Linked lists
    – Recursive matrix multiplication
• Fortran and OpenMP
Challenge: Matrix Multiplication

• Parallelize the matrix multiplication program in the file matmul.c
• Can you optimize the program by playing with how the loops are scheduled?
• Try the following and see how they interact with the constructs in OpenMP
  – Cache blocking
  – Loop unrolling
  – Vectorization
• Goal: Can you approach the peak performance of the computer?
Matrix multiplication

```c
#pragma omp parallel for private(tmp, i, j, k)
for (i=0; i<Ndim; i++){
    for (j=0; j<Mdim; j++){
        tmp = 0.0;
        for(k=0;k<Pdim;k++){
            /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
            tmp += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));
        }
        *(C+(i*Ndim+j)) = tmp;
    }
}
```

- On a dual core laptop
  - 13.2 seconds  153 Mflops  one thread
  - 7.5 seconds 270 Mflops two threads

There is much more that can be done. This is really just the first and most simple step.

Results on an Intel dual core 1.83 GHz CPU, Intel IA-32 compiler 10.1 build 2
Appendices

• Challenge Problems
• Challenge Problems: solutions
  – Monte Carlo PI and random number generators
  – Molecular dynamics
  – Matrix multiplication
  – Linked lists
  – Recursive matrix multiplication
• Fortran and OpenMP
Exercise: traversing linked lists

• Consider the program linked.c
  – Traverses a linked list computing a sequence of Fibonacci numbers at each node.

• Parallelize this program two different ways
  1. Use OpenMP tasks
  2. Use anything you choose in OpenMP other than tasks.

• The second approach (no tasks) can be difficult and may take considerable creativity in how you approach the problem (hence why its such a pedagogically valuable problem).
Linked lists with tasks
• See the file Linked_omp3_tasks.c

```c
#pragma omp parallel
{
    #pragma omp single
    {
        p=head;
        while (p) {
            #pragma omp task firstprivate(p)
            processwork(p);
            p = p->next;
        }
    }
}
```

Creates a task with its own copy of “p” initialized to the value of “p” when the task is defined
Exercise: traversing linked lists

• Consider the program linked.c
  – Traverses a linked list computing a sequence of Fibonacci numbers at each node.

• Parallelize this program two different ways
  1. Use OpenMP tasks
  2. Use anything you choose in OpenMP other than tasks.

• The second approach (no tasks) can be difficult and may take considerable creativity in how you approach the problem (hence why its such a pedagogically valuable problem).
Linked lists without tasks

• See the file Linked_omp25.c

```c
while (p != NULL) {
    p = p->next;
    count++;
}
p = head;
for(i=0; i<count; i++) {
    parr[i] = p;
    p = p->next;
}
#pragma omp parallel
{
#pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
        processwork(parr[i]);
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

<table>
<thead>
<tr>
<th></th>
<th>Default schedule</th>
<th>Static,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Thread</td>
<td>48 seconds</td>
<td>45 seconds</td>
</tr>
<tr>
<td>Two Threads</td>
<td>39 seconds</td>
<td>28 seconds</td>
</tr>
</tbody>
</table>

Results on an Intel dual core 1.83 GHz CPU, Intel IA-32 compiler 10.1 build 2
Linked lists without tasks: C++ STL

- See the file Linked_cpp.cpp

```cpp
std::vector<node *> nodelist;
for (p = head; p != NULL; p = p->next)
    nodelist.push_back(p);

int j = (int)nodelist.size();
#pragma omp parallel for schedule(static,1)
    for (int i = 0; i < j; ++i)
        processwork(nodelist[i]);
```

<table>
<thead>
<tr>
<th></th>
<th>C++, default sched.</th>
<th>C++, (static,1)</th>
<th>C, (static,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Thread</td>
<td>37 seconds</td>
<td>49 seconds</td>
<td>45 seconds</td>
</tr>
<tr>
<td>Two Threads</td>
<td>47 seconds</td>
<td>32 seconds</td>
<td>28 seconds</td>
</tr>
</tbody>
</table>

Results on an Intel dual core 1.83 GHz CPU, Intel IA-32 compiler 10.1 build 2
Appendices

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
Recursive matrix multiplication

• Could be executed in parallel as 4 tasks
  – Each task executes the two calls for the same output submatrix of C
• However, the same number of multiplication operations needed

```c
#define THRESHOLD 32768  // product size below which simple matmult code is called

void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                 double **A, double **B, double **C)

// Dimensions: A[mf..ml][nf..nl] B[pf..pl][nf..nl] C[mf..ml][nf..nl]
{
  if ((ml-mf)*(nl-nf)*(pl-pf) < THRESHOLD)
    matmult (mf, ml, nf, nl, pf, pl, A, B, C);
  else
    #pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
    {
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C11 += A11*B11
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C11 += A12*B21
    }
    #pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
    {
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C12 += A11*B12
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C12 += A12*B22
    }
    #pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
    {
      matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C21 += A21*B11
      matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C21 += A22*B21
    }
    #pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
    {
      matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C22 += A21*B12
      matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C22 += A22*B22
    }
    #pragma omp taskwait
}
```
Appendices

• Challenge Problems
• Challenge Problems: solutions
  – Monte Carlo PI and random number generators
  – Molecular dynamics
  – Matrix multiplication
  – Linked lists
  – Recursive matrix multiplication
• Fortran and OpenMP
Fortran and OpenMP

• We were careful to design the OpenMP constructs so they cleanly map onto C, C++ and Fortran.
• There are a few syntactic differences that once understood, will allow you to move back and forth between languages.
• In the specification, language specific notes are included when each construct is defined.
OpenMP:
Some syntax details for Fortran programmers

• Most of the constructs in OpenMP are compiler directives.
  – For Fortran, the directives take one of the forms:
    C$OMP construct [clause [clause]…]
    !$OMP construct [clause [clause]…]
    *$OMP construct [clause [clause]…]

• The OpenMP include file and lib module
  use omp_lib
  Include omp_lib.h
OpenMP: Structured blocks (Fortran)

– Most OpenMP constructs apply to structured blocks.

– Structured block: a block of code with one point of entry at the top and one point of exit at the bottom.

– The only “branches” allowed are STOP statements in Fortran and exit() in C/C++.

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
    res(id) = wrk(id)**2
    if(conv(res(id))) goto 10
C$OMP END PARALLEL
print *,id
```

A structured block

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
30    res(id)=wrk(id)**2
    if(conv(res(id)))goto 20
    go to 10
C$OMP END PARALLEL
    if(not_DONE) goto 30
20    print *, id
```

Not A structured block
OpenMP: Structured Block Boundaries

- In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```fortran
C$OMP PARALLEL
10    wrk(id) = garbage(id)
    res(id) = wrk(id)**2
   if(conv(res(id))) goto 10
C$OMP END PARALLEL

C$OMP PARALLEL DO
   do i=1,N
      res(i)=bigComp(i)
   end do
C$OMP END PARALLEL DO
```

- The “construct/end construct” pairs is done anywhere a structured block appears in Fortran. Some examples:

  - DO … END DO
  - PARALLEL … END PARALLEL
  - CRITICAL … END CRITICAL
  - SECTION … END SECTION
  - SECTIONS … END SECTIONS
  - SINGLE … END SINGLE
  - MASTER … END MASTER
Runtime library routines

• The include file or module defines parameters
  – Integer parameter omp_lock_kind
  – Integer parameter omp_nest_lock_kind
  – Integer parameter omp_sched_kind
  – Integer parameter openmp_version
    – With value that matches C’s _OPENMP macro
• Fortran interfaces are similar to those used with C
  – Subroutine omp_set_num_threads (num_threads)
  – Integer function omp_get_num_threads()
  – Integer function omp_get_thread_num()
  – Subroutine omp_init_lock(svar)
    – Integer(kind=omp_lock_kind) svar
  – Subroutine omp_destroy_lock(svar)
  – Subroutine omp_set_lock(svar)
  – Subroutine omp_unset_lock(svar)