Open MP - Basics

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

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.
Outline

• Preamble: On becoming a parallel programmer
• Introduction to OpenMP
• Creating Threads
• Synchronization
• Parallel Loops
• Synchronize single masters and stuff
• Data environment
Parallel programming is really hard

• Programming is hard whether you write serial or parallel code.
  – Parallel programming is just a new wrinkle added to the already tough problem of writing high quality, robust and efficient code.

• Why does Parallel programming seems so complex?
  – The literature overwhelms with hundreds of languages/APIs and a countless assortment of algorithms.
  – Experienced parallel programmers love to tell “war stories” of Herculean efforts to make applications scale … which can scare people away.
  – It’s new: synchronization, scalable algorithms, distributed data structures, concurrency bugs, memory models … hard or not it’s a bunch of new stuff to learn.
But it’s really not that bad (part 1): parallel libraries

Programming Challenges and NITRD Solutions

• Application complexity grew due to parallelism and more ambitious science problems (e.g., multiphysics, multiscale)

• Scientific libraries enable these applications

Source: Kathy Yelick

Third party names are the property of their owners.
But its really not that bad: part 2

• Don’t let the glut of parallel programming languages confuse you.

• Leave research languages to C.S. researchers and stick to the small number of broadly used languages/APIs:
  – Industry standards:
    – Pthreads and OpenMP
    – MPI
    – OpenCL
    – TBB (… and maybe Cilk?)
  – or a broadly deployed solutions tied to your platform of choice
    – CUDA (for NVIDIA platforms and PGI compilers)
    – .NET and C++ AMP (Microsoft)
  – For HPC programmers dreaming of Exascale … maybe a PGAS language/API?
    – UPC
    – GA
But it’s really not that bad: part 3

- Most algorithms are based on a modest number of recurring patterns (see Kurt Kreutzer's lecture tomorrow).

- Almost every parallel program is written in terms of just 7 basic patterns:
  - SPMD
  - Kernel Parallelism
  - Fork/join
  - Actors
  - Vector Parallelism
  - Loop Parallelism
  - Work Pile
Parallel programming is easy

• So all you need to do is:
  – **Pick** your language.
    – I suggest sticking to industry standards and open source so you can move around between hardware platforms:
      – pthreads
      – OpenMP
      – OpenCL
      – MPI
      – TBB
  – **Learn** the key 7 patterns
    – SPMD
    – Kernel Parallelism
    – Fork/join
    – Actors
  – **Master** the few patterns common to your platform and application domain … for example, most application programmers just use these three patterns
    – SPMD
    – Kernel Parallelism
    – Loop Parallelism

Third party names are the property of their owners.
If you become overwhelmed during this course ... 

• Come back to this slide and remind yourself ... things are not as bad as they seem

Parallel programming is easy

• So all you need to do is:
  - **Pick** your language.
    - I suggest sticking to industry standards and open source so you can move around between hardware platforms:
      - pthreads
      - OpenMP
      - OpenCL
      - MPI
      - TBB
  - **Learn** the key 7 patterns
    - SPMD
    - Kernel Parallelism
    - Fork/join
    - Actors
  - **Master** the few patterns common to your platform and application domain ... for example, most application programmers just use these three patterns
    - SPMD
    - Kernel Parallelism
    - Loop Parallelism
Outline

• Preamble: On becoming a parallel programmer
• Introduction to OpenMP
• Creating Threads
• Synchronization
• Parallel Loops
• Synchronize single masters and stuff
• Data environment
Recapitulation

• You know about parallel architectures … multicore chips have made them very common.

• You know about threads and cache coherent shared address spaces

• … and you know about the Posix Threads API (Pthreads) for writing multithreaded programs.

Third party names are the property of their owners.
A simple running example: 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 $i$. 

$$F(x) = \frac{4.0}{1+x^2}$$

Graph showing rectangles approximating the integral.
#define NUMSTEPS = 100000;
double step;
void main ()
{
    int i;  double x, pi, sum = 0.0;

    step = 1.0/(double) NUMSTEPS;
x = 0.5 * step;
for (i=0;i<= NUMSTEPS; i++)
{
    x+=step;
    sum += 4.0/(1.0+x*x);
}
pi = step * sum;
}
Let’s turn this into a parallel program using the Pthreads API.

```c
#define NUMSTEPS = 100000;
double step;
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) NUMSTEPS;
    x = 0.5 * step;
    for (i=0;i<= NUMSTEPS; i++){
        x+= step;
        sum += 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

**Variable to accumulate thread results must be shared**

**Package this into a function**

**Assign loop iterations to threads**

**Assure safe update to sum ... correct for any thread schedule**
Numerical Integration: PThreads

#include <stdio.h>
#include <pthread.h>
#define NUMSTEPS 10000000
#define NUMTHREADS 4

double step = 0.0, Pi = 0.0;

pthread_mutex_t gLock;

void *Func(void *pArg)
{
    int myRank = *((int *)pArg);
    double partialSum = 0.0, x;
    for (int i = myRank; i < NUMSTEPS; i += NUMTHREADS)
    {
        x = (i + 0.5f) * step;
        partialSum += 4.0f / (1.0f + x*x);
    }

    pthread_mutex_lock(&gLock);
    Pi += partialSum * step;
    pthread_mutex_unlock(&gLock);

    return 0;
}
int main()
{
    pthread_t thrds[NUMTHREADS];
    int tNum[NUMTHREADS], i;
    pthread_mutex_init(&gLock, NULL);
    Step = 1.0 / NUMSTEPS;
    for ( i = 0; i < NUMTHREADS; ++i )
    {
        tRank[i] = i;
        pthread_create(&thrds[i], NULL, Func, (void *)&tRank[i]);
    }
    for ( i = 0; i < NUMTHREADS; ++i )
    {
        pthread_join(thrds[i], NULL);
    }
    pthread_mutex_destroy(&gLock);
    printf("Computed value of Pi: %12.9f\n", Pi );
    return 0;
}

Source: Michael Wrinn of Intel
The fork-join pattern

- This is an instance of the well-known **Fork join pattern**:
  1. Start as a serial program.
  2. When work to do in parallel is encountered, pack it into a function.
  3. Fork a number of threads to execute the function.
  4. When the functions have completed, the threads join back together.
  5. Program continues as a serial program.

- If this pattern with such “mechanical” transformations is so common, can’t we come up with an easier, less intrusive way for this style of programming?
- Yes we can … and it’s called OpenMP
OpenMP* Overview:

OpenMP: An API for Writing Multithreaded Applications

- 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 last 20 years of SMP practice

---

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.
OpenMP Execution Model:

Fork-Join pattern:

- **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.
Example: Hello world

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

```c
void main()
{
    int ID = 0;
    printf(" hello(%d )", ID);
    printf(" world(%d ) \n", ID);
}
```
Example: Hello world Solution

- Tell the compiler to pack code into a function, fork the threads, and join when done …

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

Sample Output:
- hello(1) hello(0) world(1)
- world(0)
- hello (3) hello(2) world(3)
- world(2)
Example: Hello world Solution

Calling the OpenMP compiler

• Tell the compiler to pack code into a function, fork the threads, and join when done …

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

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compiler Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux and OS X</td>
<td>gcc -fopenmp</td>
</tr>
<tr>
<td>PGI Linux</td>
<td>pgcc -mp</td>
</tr>
<tr>
<td>Intel windows</td>
<td>icl /Qopenmp</td>
</tr>
<tr>
<td>Intel Linux and OS X</td>
<td>icpc –openmp</td>
</tr>
</tbody>
</table>
OpenMP core syntax

• Most of the constructs in OpenMP are compiler directives.

  \#pragma omp construct [clause [clause]…]
  
  – Example

  \#pragma omp parallel num_threads(4)

• 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.
OpenMP BasicDefs: Solution Stack

User layer

End User

Application

Prog. Layer

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

Proc1

Proc2

Proc3

ProcN

Shared Address Space
OpenMP Overview:
How do threads interact?

• OpenMP is a multi-threading, shared address model.
  – Threads communicate by sharing variables.

• Unintended sharing of data causes race conditions:
  – race condition: when the program’s outcome changes as
    the threads are scheduled differently.

• To control race conditions:
  – Use synchronization to protect data conflicts.

• Synchronization is expensive so:
  – Change how data is accessed to minimize the need for
    synchronization.
Outline

- Preamble: On becoming a parallel programmer
- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
OpenMP Execution Model:

Fork-Join pattern:

- **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 calls `pooh(ID,A)` for $ID = 0$ to $3$

---

* The name “OpenMP” is the property of the OpenMP Architecture Review Board
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];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
```

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

* The name “OpenMP” is the property of the OpenMP Architecture Review Board
Thread Creation: Parallel Regions

- Each thread executes the same code redundantly.

```c
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID =omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
omp_set_num_threads(4)
pooh(0,A) pooh(1,A) pooh(2,A) pooh(3,A)
printf("all done\n");
```

A single copy of A is shared between all threads.

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
OpenMP: what the compiler does

- The OpenMP compiler generates code logically analogous to that on the right of this slide, given an OpenMP pragma such as that on the top-left.
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for each parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```

```
void thunk ()
{
    foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
    pthread_create (
        &tid[i],0,thunk, 0);
thunk();
for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```
Exercise

• Create a parallel version of the pi program using a parallel construct.
• Pay close attention to shared versus private variables.
• In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();

   Number of threads in the team

   Thread ID or rank

   Time in Seconds since a fixed point in the past
Example: Serial PI Program

```c
static long num_steps = 100000;
double step;
void 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;
}
```
```c
#include <omp.h>

static long num_steps = 100000;   // number of steps
double step;

#define NUM_THREADS 2

void main ()
{
    int i, nthreads;   // number of threads
    double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        int i, id, nthrds;   // thread id and number of threads
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0)   // only one thread copies the number of threads
            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;
}
```

Example: A simple Parallel pi program

Promote scalar to an array dimensioned by number of threads to avoid race condition.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.
SPMD: Single Program Multiple Data

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

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern ... it is probably the most commonly used pattern in the history of parallel programming.
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, 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; i<num_steps; i=i+nthrs) {
            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>1&lt;sup&gt;st&lt;/sup&gt; 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.
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
#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;
}
Results*: pi program padded accumulator

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

```
#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, nthreads;
        double x;
        id = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        if (id == 0) nthreads = nthreads;
        for (i=id, sum[id][0]=0.0; i<num_steps; i=i+nthreads) {
            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

• Preamble: On becoming a parallel programmer
• Introduction to OpenMP
• Creating Threads
• Synchronization
• Parallel Loops
• Synchronize single masters and stuff
• Data environment
Synchronization

- High level synchronization:
  - critical
  - atomic
  - barrier
  - 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: 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);
    }
}
```

Threads wait their turn – only one at a time calls consume()
Synchronization: Atomic (basic form)

• **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 tmp, B;
  B = DOIT();
  tmp = big_ugly(B);
  #pragma omp atomic
  X += tmp;
"}
```

The statement inside the atomic must be one of the following forms:

• \( x \) binop= expr
• \( x++ \)
• \( ++x \)
• \( x-- \)
• \( --x \)

\( X \) is an lvalue of scalar type and binop is a non-overloaded built-in operator.

Additional forms of atomic were added in OpenMP 3.1. We will discuss these later.
Synchronization: Barrier

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

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

- Implicit barrier at the end of a parallel region
- Implicit barrier at the end of a for worksharing construct
- No implicit barrier due to nowait
Exercise

• Earlier, we 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” from before to avoid false sharing due to the sum array using a critical section.
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 ()
{
    double pi;
    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) nthrds = nthrds;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        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;
}
```
Results*: pi program critical section

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

```c
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main()
{
    double pi; 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) nthrds = nthrds;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        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>1\textsuperscript{st} SPMD</th>
<th>1\textsuperscript{st} 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.
Outline

• Preamble: On becoming a parallel programmer
• Introduction to OpenMP
• Creating Threads
• Synchronization
• Parallel Loops
• Synchronize single masters and stuff
• Data environment
SPMD vs. worksharing

• A **parallel construct** by itself creates an SPMD or “Single Program Multiple Data” program … i.e., each thread redundantly executes the same code.

• How do you split up pathways through the code between threads within a team?
  – This is called worksharing
    – Loop construct
    – Sections/section constructs
    – Single construct
    – Task construct
    
    *Discussed later*
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 variable I is made “private” to each thread by default. You could do this explicitly with a “private(I)” clause.
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.
  - 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).
# loop work-sharing constructs:
The schedule clause

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

Least work at runtime: scheduling done at compile-time

Most work at runtime: complex scheduling logic used at run-time
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();
    }
}
```

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

These are equivalent
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);
}
```

```c
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
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 may balancing the load difficult.
Reduction

How do we handle this case?

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:

\[
\text{reduction (op : list)}
\]

• 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. num.</td>
</tr>
<tr>
<td>Max</td>
<td>Most neg. num.</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>

C/C++ only
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.
#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.

<table>
<thead>
<tr>
<th>threads</th>
<th>1st SPMD</th>
<th>1st SPMD padded</th>
<th>SPMD critical</th>
<th>PI Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.86</td>
<td>1.86</td>
<td>1.87</td>
<td>1.91</td>
</tr>
<tr>
<td>2</td>
<td>1.03</td>
<td>1.01</td>
<td>1.00</td>
<td>1.02</td>
</tr>
<tr>
<td>3</td>
<td>1.08</td>
<td>0.69</td>
<td>0.68</td>
<td>0.80</td>
</tr>
<tr>
<td>4</td>
<td>0.97</td>
<td>0.53</td>
<td>0.53</td>
<td>0.68</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.
Parallel loops

• OpenMP 3.0 guarantees that this works … i.e. that the same schedule is used in the two loops:

```c
!$omp do schedule(static)
do i=1,n
  a(i) = ....
end do
!$omp end do nowait
!$omp do schedule(static)
do i=1,n
  .... = a(i)
end do
```
Loops (cont.)

• Made \textit{schedule(runtime)} more useful
  – can get/set it with library routines
    \begin{verbatim}
    omp_set_schedule()
    omp_get_schedule()
    \end{verbatim}
  – allow implementations to implement their own schedule kinds
• Added a new schedule kind \textbf{AUTO} which gives full freedom to the runtime to determine the scheduling of iterations to threads.
• Allowed C++ Random access iterators as loop control variables in parallel loops
Outline

- Preamble: On becoming a parallel programmer
- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
Master Construct

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

```c
#pragma omp parallel
{
    do_many_things();
#pragma omp master
    { exchange_boundaries(); }
#pragma omp barrier
    do_many_other_things();
}
```
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();
}
```
• 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.
Synchronization: ordered

• The **ordered** region executes in the sequential order.

```c
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
for (l=0;l<N;l++){
    tmp = NEAT_STUFF(l);
    #pragma ordered
    res += consum(tmp);
}
```
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.
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.
Runtime Library routines

• Runtime environment routines:
  – Modify/Check the number of threads
    – \texttt{omp\_set\_num\_threads()}, \texttt{omp\_get\_num\_threads()}, \texttt{omp\_get\_thread\_num\_num()}, \texttt{omp\_get\_max\_threads()}
  – Are we in an active parallel region?
    – \texttt{omp\_in\_parallel()}
  – Do you want the system to dynamically vary the number of threads from one parallel construct to another?
    – \texttt{omp\_set\_dynamic}, \texttt{omp\_get\_dynamic()};
  – How many processors in the system?
    – \texttt{omp\_num\_procs()}

...plus a few less commonly used routines.
Runtime Library routines

To use a known, fixed number of threads in a program, (1) tell the system that you don’t want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.

```c
#include <omp.h>
void main()
{
    int num_threads;
    omp_set_dynamic( 0 );
    omp_set_num_threads( omp_num_procs() );
    #pragma omp parallel
    {    int id=omp_get_thread_num();
        #pragma omp single
        num_threads = omp_get_num_threads();
        do_lots_of_stuff(id);
    }
}
```

Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.
Environment Variables

- Set the default number of threads to use.
  - OMP_NUM_THREADS int literal

- OpenMP added an environment variable to control the size of child threads’ stack
  - OMP_STACKSIZE

- Also added an environment variable to hint to runtime how to treat idle threads
  - OMP_WAIT_POLICY
    - ACTIVE keep threads alive at barriers/locks
    - PASSIVE try to release processor at barriers/locks

- Control how “omp for schedule(RUNTIME)” loop iterations are scheduled.
  - OMP_SCHEDULE “schedule[, chunk_size]”
Outline

• Preamble: On becoming a parallel programmer
• Introduction to OpenMP
• Creating Threads
• Synchronization
• Parallel Loops
• Synchronize single masters and stuff
• Data environment
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.
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
Data sharing: Changing storage attributes

• One can selectively change storage attributes for constructs using the following clauses*
  – SHARED
  – PRIVATE
  – FIRSTPRIVATE

• The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  – LASTPRIVATE

• The default attributes can be overridden with:
  – DEFAULT (PRIVATE | SHARED | NONE)

DEFAULT(PRIVATE) *is Fortran only*

*All data clauses apply to parallel constructs and worksharing constructs except “shared” which only applies to parallel constructs.

All the clauses on this page apply to the OpenMP construct NOT to the entire region.
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

`tmp` is 0 here
Firstprivate Clause

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

```cpp
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
Lastprivate Clause

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```c
void sq2(int n, double *lastterm)
{
    double x; int i;
    #pragma omp parallel for lastprivate(x)
    for (i = 0; i < n; i++) {
        x = a[i]*a[i] + b[i]*b[i];
        b[i] = sqrt(x);
    }
    *lastterm = x;
}
```

“x” has the value it held for the “last sequential” iteration (i.e., for i=(n-1))
Example: Pi program … minimal changes

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

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

For good OpenMP implementations, reduction is more scalable than critical.

i private by default
Conclusion

• OpenMP is one of the simplest APIs available for programming shared memory machines.
  • This simplicity means you can focus on mastering the key design patterns and applying them to your own problems

• We covered the following essential parallel programming design patterns:
  – Fork join
  – SPMD
  – Loop level parallelism

• Next steps?
  – Start writing parallel code … you can only learn this stuff by writing lots of code.
  – Let’s consider some of the newer and more advanced features of OpenMP.