Parallel Programming using OpenMP

Qin Liu

The Chinese University of Hong Kong
Overview

Why Parallel Programming?

Overview of OpenMP

Core Features of OpenMP

More Features and Details...

One Advanced Feature
Introduction

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

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

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

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

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

- OpenMP is one of the most common parallel programming models in use today
- It is relatively easy to use which makes a great language to start with when learning to write parallel programs
- Assumptions:
  - We assume you know C++ (OpenMP also supports Fortran)
  - We assume you are new to parallel programming
  - We assume you have access to a compiler that supports OpenMP (like gcc)
Why Parallel Programming?
Growth in processor performance since the late 1970s

- Good old days: 17 years of sustained growth in performance at an annual rate of over 50%
The Hardware/Software Contract

- We (SW developers) learn and design sequential algorithms such as quick sort and Dijkstra’s algorithm.
The Hardware/Software Contract

- We (SW developers) learn and design sequential algorithms such as quick sort and Dijkstra’s algorithm
- Performance comes from hardware
The Hardware/Software Contract

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

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

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

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

But...
The Hardware/Software Contract

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

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

But...

- In 2004, Intel canceled its high-performance uniprocessor projects and joined others in declaring that the road to higher performance would be via multiple processors per chip rather than via faster uniprocessors
• Growth in power is unsustainable \((\text{power} = \text{perf}^{1.74})\)
• Partial solution: simple low power cores

The rest of the solution - Add Cores

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

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

A new HW/SW contract:

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

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

A new HW/SW contract:

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

Process:

1. We have a sequential algorithm

2. Split the program into tasks and identify shared and local data

3. Use some algorithm strategy to break dependencies between tasks

4. Implement the parallel algorithm in C++/Java/...

Can this process be automated by the compiler? Unlikely... We have to do it manually.
Overview of OpenMP
OpenMP: an API for writing multi-threaded applications

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

• Most of the constructs in OpenMP are compiler directives:
  
  #pragma omp <construct> [clause1 clause2 ...]

• Example:
  
  #pragma omp parallel num_threads(4)

• Include file for runtime library: #include <omp.h>

• Most OpenMP constructs apply to a “structured block”
  
  ▶ Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom
Exercise 1: Hello World

A multi-threaded “hello world” program

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

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

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

- More information:
  http://openmp.org/wp/openmp-compilers/
Symmetric Multiprocessing (SMP)

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

Source: https://moinakg.wordpress.com/2013/06/05/findings-by-google-on-numa-performance/
Symmetric Multiprocessing (SMP)

- SMP computers are everywhere... Most laptops and servers have multi-core multiprocessor CPUs.
Symmetric Multiprocessing (SMP)

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

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

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

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

Source: https://computing.llnl.gov/tutorials/pthreads/
SMP Programming

Threads:
- "light weight processes"
- share process state
- reduce the cost of switching context

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

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

Source: https://computing.llnl.gov/tutorials/pthreads/
Shared Memory Model

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

Source: https://computing.llnl.gov/tutorials/pthreads/
Exercise 1: Hello World

A multi-threaded “hello world” program

Sample Output:

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

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

Threads interleave and give different outputs every time
How Do Threads Interact in OpenMP?

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

Fork-Join Parallelism Model:

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

Source: https://computing.llnl.gov/tutorials/openMP/
Thread Creation: Parallel Regions

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

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

- Each thread calls `pooh(ID, A)` for ID from 0 to 3
Thread Creation: Parallel Regions

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

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

- Each thread calls `pooh(ID, A)` for ID from 0 to 3
Thread Creation: Parallel Regions

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

A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a barrier)
Compute $\pi$ using Numerical Integration

Mathematically, we know that:

$$\int_{0}^{1} \frac{4}{1 + x^2} \, dx = \pi$$

We can approximate the integral as a sum of rectangles:

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

Where each rectangle has width $\Delta x$ and height $F(x_i)$ at the middle of interval $i$. 
```c
#include <stdio.h>

const long num_steps = 100000000;

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

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

    double pi = step * sum;

    printf("pi is %f \n", pi);
}
```
Exercise 2: First Parallel $\pi$ Program

- Create a parallel version of the pi program using a parallel construct
- Pay close attention to shared versus private variables
- In addition to a parallel construct, you will need the runtime library routines
  - `int omp_get_num_threads()`: number of threads in the team
  - `int omp_get_thread_num()`: ID of current thread
Exercise 2: First Parallel $\pi$ Program

```c
#include <stdio.h>
#include <omp.h>
const long num_steps = 100000000;
#define NUM_THREADS 4
double sum[NUM_THREADS];
int main () {
    double step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        sum[id] = 0.0;
        for (int i = id; i < num_steps; i += NUM_THREADS) {
            double x = (i+0.5) * step;
            sum[id] += 4.0 / (1.0 + x*x);
        }
    }
    double pi = 0.0;
    for (int i = 0; i < NUM_THREADS; i++)
        pi += sum[i] * step;
    printf("pi is \%f \n", pi);
}```
Algorithm Strategy: SPMD

The SPMD (single program, multiple data) technique:

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

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

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

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>1.29</td>
<td>0.72</td>
<td>0.47</td>
<td>0.48</td>
</tr>
</tbody>
</table>
Results

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

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>1.29</td>
<td>0.72</td>
<td>0.47</td>
<td>0.48</td>
</tr>
</tbody>
</table>

Why such poor scaling?
Reason: False Sharing

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


- Solution: pad arrays so elements you use are on distinct cache lines
# include <stdio.h>
# include <omp.h>
const long num_steps = 100000000;
#define NUM_THREADS 4
#define PAD 8 // assume 64 bytes L1 cache
double sum[NUM_THREADS][PAD];
int main () {
    double step = 1.0 / (double) num_steps;
   omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        sum[id][0] = 0.0;
        for (int i = id; i < num_steps; i += NUM_THREADS) {
            double x = (i+0.5) * step;
            sum[id][0] += 4.0 / (1.0 + x*x);
        }
    }
    double pi = 0.0;
    for (int i = 0; i < NUM_THREADS; i++)
        pi += sum[i][0] * step;
    printf("pi is %f \n", pi);
Results

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

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>1.29</td>
<td>0.72</td>
<td>0.47</td>
<td>0.48</td>
</tr>
<tr>
<td>Padding</td>
<td>1.27</td>
<td>0.65</td>
<td>0.43</td>
<td>0.33</td>
</tr>
</tbody>
</table>
Do We Really Need to Pad Our Arrays?

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

Recall: to control race conditions

- use synchronization to protect data conflicts

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

- **Barrier**: each thread wait at the barrier until all threads arrive

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

**Barrier**: each thread wait until all threads arrive

```c
#pragma omp parallel
{
  int id = omp_get_thread_num();
  A[id] = big_calc1(id);
  #pragma omp barrier
  B[id] = big_calc2(id, A); // depend on A calculated by every thread
}
```
Synchronization: critical

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

```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);
        // only one at a time calls consume() and modify res
    }
}
```

Synchronization: atomic

**Atomic** provides mutual exclusion but only applies to the update of a memory location and only supports \( x += \text{expr}, \) 
\( x++, --x \ldots \)

```cpp
float res;
#pragma omp parallel
{
    float tmp, B;
    B = big();
    tmp = calc(B);
    #pragma omp atomic
    res += tmp;
}
```
# include <stdio.h>
#include <omp.h>
const long num_steps = 100000000;
#define NUM_THREADS 4
int main () {
    double step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    double pi = 0.0;
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        double sum = 0.0; // local scalar not array
        for (int i = id; i < num_steps; i += NUM_THREADS) {
            double x = (i+0.5) * step;
            sum += 4.0 / (1.0 + x*x); // no false sharing
        }
        #pragma omp critical
        pi += sum * step; // must do summation here
    }
    printf("pi is %f \n", pi);
Results

• Setup: gcc with no optimization on Ubuntu 14.04 with a quad-core Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz and 16 GB RAM

• The serial version ran in 1.25 seconds

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>1.29</td>
<td>0.72</td>
<td>0.47</td>
<td>0.48</td>
</tr>
<tr>
<td>Padding</td>
<td>1.27</td>
<td>0.65</td>
<td>0.43</td>
<td>0.33</td>
</tr>
<tr>
<td>Critical</td>
<td>1.26</td>
<td>0.65</td>
<td>0.44</td>
<td>0.33</td>
</tr>
</tbody>
</table>
Mutual Exclusion Done Wrong

Be careful where you put a critical section.

```c
#include <stdio.h>
#include <omp.h>
const long num_steps = 100000000;
#define NUM_THREADS 4
int main () {
    double step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    double pi = 0.0;
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i = id; i < num_steps; i += NUM_THREADS) {
            double x = (i+0.5) * step;
            #pragma omp critical
            pi += 4.0 / (1.0 + x*x) * step;
        }
    }
    printf("pi is %f \n", pi);
}
```

Ran in 10 seconds with 4 threads.
Example: Using Atomic

```c
#include <stdio.h>
#include <omp.h>
const long num_steps = 100000000;
#define NUM_THREADS 4
int main () {
    double step = 1.0 / (double) num_steps ;
    omp_set_num_threads(NUM_THREADS);
    double pi = 0.0;
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        double sum = 0.0;
        for (int i = id; i < num_steps; i += NUM_THREADS) {
            double x = (i+0.5) * step ;
            sum += 4.0 / (1.0 + x*x);
        }
        sum *= step;
    #pragma omp atomic
        pi += sum;
    }
    printf("pi is %f \n", pi);
} 
```
Serial $\pi$ program:

```c
#include <stdio.h>

const long num_steps = 100000000;

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

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

    double pi = step * sum;

    printf("pi is %f \n", pi);
}
```

What we want to parallelize:
```
for (int i = 0; i < num_steps; i++)
```
Two equivalent directives:

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

```c
double res[MAX];
#pragma omp parallel for
for (int i = 0; i < MAX; i++)
  res[i] = huge();
```
Working with For Loops

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

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

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

Each iteration depends on the previous one.
Working with For Loops

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

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

Each iteration depends on the previous one.

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

Remove dependency and $j$ is now local to each iteration.
The Schedule Clause

The schedule clause affects how loop iterations are mapped onto threads

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

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

```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 $N \times M$ and then parallelize that

- Useful if $N$ is $O$ (num of threads) so parallelizing the outer loop makes balancing the load difficult
Reduction

- How do we handle this case?

```c
double ave = 0.0, A[MAX];
for (int 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 such as MapReduce and MPI.
Reduction

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

```c
double ave=0.0, A[MAX];
#pragma omp parallel for reduction(+:ave)
for (int i = 0; i < MAX; i++) {
    ave += A[i];
}
ave = ave/MAX;
```
# include <stdio.h>
#include <omp.h>
const long num_steps = 100000000;
#define NUM_THREADS 4
int main () {
    double sum = 0.0;
    double step = 1.0 / (double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for reduction(:+sum)
    for (int i = 0; i < num_steps; i++) {
        double x = (i+0.5) * step;
        sum += 4.0 / (1.0 + x*x);
    }
    double pi = step * sum;
    printf("pi is %f \n", pi);
}
Results

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

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>1.29</td>
<td>0.72</td>
<td>0.47</td>
<td>0.48</td>
</tr>
<tr>
<td>Padding</td>
<td>1.27</td>
<td>0.65</td>
<td>0.43</td>
<td>0.33</td>
</tr>
<tr>
<td>Critical</td>
<td>1.26</td>
<td>0.65</td>
<td>0.44</td>
<td>0.33</td>
</tr>
<tr>
<td>For</td>
<td>1.27</td>
<td>0.65</td>
<td>0.43</td>
<td>0.33</td>
</tr>
</tbody>
</table>
More Features and Details…
**Synchronization: barrier**

**Barrier:** each thread wait until all threads arrive

```c
#pragma omp parallel
{
  id =omp_get_thread_num();
  A[id] = big_calc1(id);
  #pragma omp barrier
  #pragma omp for
  for (int i = 0; i < N; i++) {
    C[i] = big_calc3(i, A);
  } // implicit barrier at the end of a for worksharing construct

  #pragma omp for nowait
  for (int i = 0; i < N; i++) {
    B[i] = big_calc2(C, i);
  } // no implicit barrier due to nowait
  A[id] = big_calc4(id);
} // implicit barrier at the end of a parallel region
```
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 Construct

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

```c
#pragma omp parallel
{
    do_many_things();
#pragma omp single
    { exchange_boundaries(); }
    do_many_other_things();
}
```
Low Level Synchronization: lock routines

- Simple lock routines: a simple lock is available if it is unset
  - `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`, `omp_destroy_lock()`

- Nested locks: a nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
  - `omp_init_nest_lock()`, `omp_set_nest_lock()`, `omp_unset_nest_lock()`, `omp_test_nest_lock()`, `omp_destroy_nest_lock()`
Synchronization: Simple Locks

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

```c
#pragma omp parallel for
for (int i = 0; i < NBUCKETS; i++) {
    omp_init_lock(&hist_locks[i]); // one lock per elements of hist
    hist[i] = 0;
}
#pragma omp parallel for
for (int i = 0; i < NVALS; i++) {
    ival = (int) sample(arr[i]);
    omp_set_lock(&hist_locks[ival]);
    hist[ival]++;// mutual exclusion, less wait compared to critical
    omp_unset_lock(&hist_locks[ival]);
}
#pragma omp parallel for
for (i = 0; i < NBUCKETS; i++)
    omp_destroy_lock(&hist_locks[i]); // free locks when done
```
Sections Worksharing

- 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();
} // implicit barrier that can be turned off with nowait
```

Shorthand: `#pragma omp parallel sections`
Runtime Library Routines

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

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

Examples:

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

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

A, index, count are shared.
temp is private to each thread.
Data Scope Attribute Clauses

- Change scope attributes using:
  - shared, private, firstprivate, lastprivate
- The default attributes can be overridden with:
  - default(shared|none)
- Skip details...
int main() {
    std::string a = "a", b = "b", c = "c";
    #pragma omp parallel firstprivate(a) private(b) shared(c)
    num_threads(2)
    {
        a += "k"; // a is initialized with "a"
        b += "k"; // b is initialized with std::string()
        #pragma omp critical
        {
            c += "k"; // c is shared
            std::cout << omp_get_thread_num() << ": " << a
                    << ", " << b << ", " << c << "\n";
        }
    }
    std::cout << a << ", " << b << ", " << c << "\n";
}
One Advanced Feature
OpenMP Tasks

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

Traversal of a tree:

```c
struct node { node *left, *right; };
extern void process(node* );
void traverse(node* p) {
    if (p->left)
        #pragma omp task // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
        traverse(p->right);
    process(p);
}

#pragma omp parallel
{
    #pragma omp single nowait
    { traverse(root); }
}
```
What if we want to force a postorder traversal of the tree?

```c
struct node { node *left, *right; };

extern void process(node*);

void traverse(node* p) {
    if (p->left)
        #pragma omp task // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
        traverse(p->right);
    #pragma omp taskwait // a barrier only for tasks
    process(p);
}
```
Process elements of a linked list in parallel:

```c
struct node { int data; node* next; };
extern void process(node*);
void increment_list_items(node* head)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (node* p = head; p; p = p->next) {
                #pragma omp task
                process(p); // p is firstprivate by default
            }
        }
    }
}
```
References

- Guide into OpenMP: Easy multithreading programming for C++:
  http://bisqwit.iki.fi/story/howto/openmp
- Introduction to OpenMP - Tim Mattson (Intel):
  - videos: https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwinoEG
- OpenMP specification: http://openmp.org