Introduction to OpenMP
Outline

Overview of OpenMP

Mechanics

   Hello World!

   Simple loops

   Nested loops

Calculating PI

Tips and Scheduling

Consider checking out this great online tutorial:

https://computing.llnl.gov/tutorials/openMP/
Overview of OpenMP
Levels of Parallelism

- main()
- task 1
  - thread 1
  - thread 2
  - thread m
- task 2
  - thread 1
  - thread 2
  - thread m
- task n
  - thread 1
  - thread 2
  - thread m

 MPI

Threads

Coarse grain

Fine grain

Compiler and CPU
OpenMP

An application programming interface (API) for parallel programming on *multiprocessors*

- Compiler directives
- Library functions
- Environmental variables

Open works in conjunction with Fortran, C, or C++

Frees you from the details of programming threads (e.g. POSIX)

http://openmp.org
Shared-memory Model

Processors interact and synchronize with each other through shared variables
JANUS

memory control

memory

memory control

memory
Fork/Join Parallelism

Initially only master thread is active

Master thread executes sequential code

Fork: Master thread creates or awakens additional threads to execute parallel code

Join: At end of parallel code created threads die or are suspended
Master Thread

Other threads

fork

join

fork

join
Shared-memory vs. Message Passing

Shared-memory model

Number active threads 1 at start and finish of program, changes dynamically during execution

Incrementally make your program parallel

Programs may only have a single parallel loop

Message-passing model

All processes active throughout execution of program

Sequential-to-parallel transformation requires major effort
Compiling OpenMP

Gnu

    gfortran -fopenmp -g basic.f90
    gcc -fopenmp -g basic.c

Intel

    ifort -openmp -g basic.f90
    icc -openmp -g basic.c

Environmental variables

    setenv OMP_NUM_THREADS 12
    export OMP_NUM_THREADS=12
Execution Context

Address space containing all of the variables a thread may access.

Every thread has its own execution context.

Contents:

- static variables
- dynamically allocated variables on the heap
- variables on the stack

Shared variable: every thread has the same address in execution context.

Private variable: every thread has a different address.

A thread cannot access the private variables of another thread.
Shared and Private Variables
Mechanics

Functions, directives, and clauses
Functions

Threads

```c
int thread = omp_get_thread_num();
```

Thread id

```c
int size = omp_get_num_threads();
```

How many threads?

```c
omp_set_num_threads(4);
```

What is another way to set the number of threads?
Functions

Threads

int thread = omp_get_thread_num();

Thread id

int size = omp_get_num_threads();

How many threads?
omp_set_num_threads(4);

What is another way to set the number of threads?

export OMP_NUM_THREADS=12
Directives

A way for the programmer to communicate with the compiler

Compiler free to ignore directives (they are hints)

#pragma directive is used to instruct the compiler to use pragmatic or implementation-dependent features (e.g. OpenMP)

#pragma omp parallel

#pragma omp parallel private(var1, var2,...)
Parallel

#pragma omp parallel [clause ...]
if (scalar_expression)
private (list)
shared (list)
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
num_threads (integer-expression)

structured_block
Hello World!

```cpp
#include <omp.h>
#include <iostream>
using namespace std;

int main (int argc, char *argv[]) {

#pragma omp parallel private(thread, size)
{
    // Obtain thread number
    int thread = omp_get_thread_num();
    int size = omp_get_num_threads();
    cout << "hello from thread " << thread;
    cout << " of " << size << endl;
}
// All threads join master thread and disband
}
```
Hello World!

```cpp
#include <omp.h>
#include <iostream>
using namespace std;

int main (int argc, char *argv[]) {

    {
        // Obtain thread number
        int thread = omp_get_thread_num();
        int size = omp_get_num_threads();
        cout << "hello from thread " << thread;
        cout << " of " << size << endl;

    } // All threads join master thread and disband
}
```
Hello World!

```cpp
#include <omp.h>
#include <iostream>
using namespace std;

int main (int argc, char *argv[]) {
    #pragma omp parallel private(thread, size)
    {
        // Obtain thread number
        int thread = omp_get_thread_num();
        int size = omp_get_num_threads();
        cout << "hello from thread " << thread;
        cout << " of " << size << endl;
    } // All threads join master thread and disband
}
```
General form

C/C++

`#pragma omp <directive> [clause]`

Fortran

`!$OMP <directive> [clause]`
<table>
<thead>
<tr>
<th>Directives</th>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel</td>
<td>private</td>
</tr>
<tr>
<td>master</td>
<td>shared</td>
</tr>
<tr>
<td>for</td>
<td>num_threads</td>
</tr>
<tr>
<td>atomic</td>
<td>schedule</td>
</tr>
<tr>
<td>critical</td>
<td>reduction</td>
</tr>
<tr>
<td>barrier</td>
<td></td>
</tr>
<tr>
<td>sections</td>
<td></td>
</tr>
<tr>
<td>ordered</td>
<td></td>
</tr>
</tbody>
</table>
#pragma omp for [clause ...]
schedule (type [,chunk])
ordered
private (list)
firstprivate (list)
lastprivate (list)
shared (list)
reduction (operator: list)
collapse (n)
nowait
Simple loop

```c
for (int i = 0; i < large_number; i++)
{
    // do something
}
```

```c
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < large_number; i++)
    {
        // do something
    }
}
```
Simple loop

```c
#pragma omp parallel for
for (int i = 0; i < large_number; i++)
{
    // do something
}
```

```c
for (int i = 0; i < large_number; i++)
{
    // do something
}
```
Simple loop

```c
#pragma omp parallel for
for (int i = 0; i < large_number; i++)
{
    // do something
}

#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < large_number; i++)
    {
        // do something
    }
}
Example

```c
for(int i=0; i<num_rows; ++i)
{
    for(int j=0; j<num_cols; ++j)
    {
        a(i,j) = a(i,j) + tmp
    }
}
```
Either loop could be executed in parallel

```c
for(int i=0; i<num_rows; ++i)
{
    for(int j=0; j<num_cols; ++j)
    {
        a(i,j) = a(i,j) + tmp
    }
}
```
Either loop could be executed in parallel

We prefer to make outer loop parallel, to reduce number of forks/joins

```c
for(int i=0; i<num_rows; ++i)
{
    for(int j=0; j<num_cols; ++j)
    {
        a(i,j) = a(i,j) + tmp
    }
}
```
Either loop could be executed in parallel

We prefer to make outer loop parallel, to reduce number of forks/joins

We then must give each thread its own private copy of variable i and j

```c
for(int i=0; i<num_rows; ++i)
{
    for(int j=0; j<num_cols; ++j)
    {
        a(i,j) = a(i,j) + tmp
    }
}
```
Directs compiler to make one or more variables private

```c
#pragma omp parallel for private(i,j)
for(int i=0; i<num_rows; ++i)
{
    for(int j=0; j<num_cols; ++j)
    {
        a(i,j) = a(i,j) + tmp
    }
}
```
Private
directs compiler to make one or more variables private

firstprivate
private variables having initial values identical to the variable controlled by the master thread as the loop is entered

Variables are initialized once per thread, not once per loop iteration

lastprivate
Copies the private copy of the variable from the thread that executed the sequentially last iteration... back to the master
Calculating PI

double x, sum = 0.0;
const double step = 1.0/static_cast<double>(large_number);

for (int i=1; i<= large_number; i++){
    x = (i-0.5)*step;
    sum += 4.0/(1.0+x*x)*step;
}
R version 2.14.1 (2011-12-22)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> source("pi.R")
Calculating PI

If we simply parallelize the loop...

double x, sum = 0.0;
const double step = 1.0/static_cast<double>(large_number);

#pragma omp parallel private(x)
for (int i=1; i<= large_number; i++){
    x = (i-0.5)*step;
    sum += 4.0/(1.0+x*x);
}
double pi = step * sum;
Race condition

... we set up a race condition in which one process may “race ahead” of another and not see its change to shared variable `sum`

```java
for (int i=1; i<= num_steps; i++){
    x = double(i - 0.5)*step;
    sum += 4.0/(1.0+x*x);
}
```

<table>
<thead>
<tr>
<th>Value of <code>sum</code></th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ 3.765</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ 3.765</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.432</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ 3.563</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.230</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Critical directive

Critical section: a portion of code that only thread at a time may execute

```c
#pragma omp parallel for private(x)
for (int i=1; i<= num_steps; i++){
    x = double(i-0.5)*step;

    #pragma omp critical
    sum += 4.0/(1.0+x*x);
}
```

Only one thread at a time may execute the statement

sequential code

Limits speedup
Reductions

Reductions are so common that OpenMP provides support for them.

May add reduction clause to `parallel for` directive.

Specify reduction operation and reduction variable.

OpenMP takes care of storing partial results in private variables and combining partial results after the loop.

```
reduction(operation:var)
```

Operators:

`+`, `*`, `-`, ... and more
#pragma omp parallel for reduction(+:sum) private(x)
for (int i=1; i<= num_steps; i++) {
    x = double((i-0.5)*step);
    sum += 4.0/(1.0+x*x);
}

Divide sections among the threads in the team

```c
#pragma omp sections [clause ...]
  private (list)
  firstprivate (list)
  lastprivate (list)
  reduction (operator: list)
  nowait
{
  #pragma omp section
    structured_block
  #pragma omp section
    structured_block
}
```
#pragma omp parallel shared(a,b,c,d) private(i)
{

#pragma omp sections nowait
{

#pragma omp section
for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

#pragma omp section
for (i=0; i < N; i++)
    d[i] = a[i] * b[i];

}  /* end of sections */

}  /* end of parallel section */

}
Tips and Scheduling
Performance Improvement #1

Too many fork/joins can lower performance

Inverting loops may help performance if

Parallelism is in inner loop

After inversion, the outer loop can be made parallel

Inversion does not significantly lower cache hit rate

```c
#pragma omp parallel for private(j)
for(int i=0; i<num_rows; ++i)
{
    for(int j=0; j<num_cols; ++j)
    {
        a(i,j) = a(i,j) + tmp
    }
}
```
Performance Improvement #2

If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution

The if clause instructs compiler to insert code that determines at run-time whether loop should be executed in parallel;

```c
#pragma omp parallel if (num > 500)
```
Performance Improvement #3

We can use schedule clause to specify how iterations of a loop should be allocated to threads

Static schedule: all iterations allocated to threads before any iterations executed

Dynamic schedule: only some iterations allocated to threads at beginning of loop’s execution. Remaining iterations allocated to threads that complete their assigned iterations.
Scheduling: static and dynamic

Static scheduling

Low overhead

May exhibit high workload imbalance

Dynamic scheduling

Higher overhead

Can reduce workload imbalance

schedule(type[,chunk])

chunk specifies the size of iterations

type: dynamic, static
Chunks

A chunk is a contiguous range of iterations

Increasing chunk size
  reduces overhead
  may increase cache hit rate

Decreasing chunk size
  allows finer balancing of workloads
Scheduling options

schedule(static)

    block allocation of about n/t contiguous iterations to each thread

schedule(static,C)

    interleaved allocation of chunks of size C to threads

schedule(dynamic)

    dynamic one-at-a-time allocation of iterations to threads

schedule(dynamic,C)

    dynamic allocation of C iterations at a time to threads
Summary

OpenMP an API for shared-memory parallel programming

Shared-memory model based on fork/join parallelism

Data parallelism

    parallel for

    reduction clause