Parallel Programming Course
OpenMP

Paul Guermonprez
www.Intel-Software-Academic-Program.com
paul.guermonprez@intel.com
Intel Software
2012-03-14
OpenMP
OpenMP is an open standard: OpenMP.org

- You add magic “pragma” comments to your code. No need to change the C/C++/Fortran.
- You compile with an OpenMP aware compiler
- Your binary will execute in parallel!

It's a simple, clean and well known technology in both research and industry.
Pi with OpenMP pragmas

Serial version:

```c
for (i=0; i<num_steps; i++) {
    x = (i + .5)*step;
    sum = sum + 4.0/(1.+ x*x);
}
```

Parallel version with OpenMP pragmas:

```c
#pragma omp parallel for
for (i=0; i<num_steps; i++) {
    x = (i + .5)*step;
    sum = sum + 4.0/(1.+ x*x);
}
```
#pragma omp parallel for

- `#` : it's a simple comment line, my C/C++ code is unchanged.
- `pragma` : this comment is special, (compiler please read this line)
- `omp` : read it as an OpenMP pragma (compiler, if you are not OpenMP aware, do not read it)
- `parallel` : it's a parallel region
- `for` : the kind of parallel region is a for loop
OpenMP is using the Fork/Join Model
Fork/Join Model

- When your main() function begins, that's the beginning of the first thread of your software. You are in serial mode.

- When you enter the parallel part of your multi-threaded software, threads are launched. Master thread forks worker threads.

- When all the worker threads have finished, they join.

- You are in serial mode again.

- End of main() function, end of process.
Fork/Join Model

Sequential  Parallel  Sequential

Fork    Join

Master Thread

Worker Thread #1
Worker Thread #2
Worker Thread #3
Worker Thread #4
parallel regions
This pragma will execute in parallel what's next: next line, next loop, next block of code between brackets.

But the parallel keyword alone won't distribute the workload on different threads. For that we'll see the constructs `for`, `task`, `section`.

`parallel` will execute the same thing several times in parallel.
```c
#pragma omp parallel
printf("before\n");
#pragma omp parallel
printf("parallel\n");
printf("after\n");
```

If you compile as usual (without the -openmp flag), will return:

before
parallel
after

A pragma is not regular code and require a special flag to be used by the compiler.
#pragma omp parallel
printf("before\n");
#pragma omp parallel
printf("parallel\n");
printf("after\n");

If compile with the -openmp flag, will return on a quad-core machine:
before
parallel
parallel
parallel
parallel
parallel
after
#pragma omp parallel
{
    printf("start ");
    printf("end ");
}
Will return on a dual-core machine:
start start end end
or (depending on various conditions)
start end end start end

Parallel execution does NOT implicate anything about the order of execution.
single, master and wait clauses
#pragma omp single

single will execute the next block of code once by the first available thread.

#pragma omp parallel
{
    printf("start ");
    #pragma omp single
    printf("end ");
}

Will return on a dual-core machine: 
start start start end
```c
#pragma omp master
single will execute the next block of code once by the master thread.

#pragma omp parallel
{
    printf("start ");
    #pragma omp master
    printf("end ");
}
Will return on a dual-core machine:
start start start end
```
#pragma omp ... nowait

`nowait` will prevent the join phase at the end of a parallel region (an implicit barrier) from blocking execution.

```c
#pragma omp parallel for nowait
for (int i=0; i<n; i++) {
    printf("i");
}
#pragma omp parallel for
for (int j=0; j<m; j++) {
    printf("j");
}
Could print iiiiiiiijijiiijjjjjjjjjjjjjjjjjjjjjjjj
for worksharing construct data decomposition
for worksharing construct

```c
#pragma omp parallel for
for (i=0 ; i<N ; i++) {
    printf("loop %d\n",i);
}
```

Will start a parallel region, create an optimal number of threads, maintain a queue of iterations to execute and distribute them to the threads as needed.

When all the iterations are executed, the parallel region will end and the code will go back to serial execution.
Good side

*parallel for* is a simple and flexible way to implement *data decomposition*:

- keep a simple *loop* structure
- all the *queue* management is done automatically
- *worksharing* is handle automatically and there's many settings to change the algorithm and settings
To keep in mind

But:

- you can't expect iterations to execute in any specific order or to show the same behavior on different software or hardware environments.

- If some of your variables are defined before the parallel region, they are shared between threads. Sharing to read is safe, but sharing and writing leads to parallel bugs. We'll see how to share variables safely.
sections
worksharing construct

simple task decomposition
Task decomposition

With task decomposition, you have to analyze your algorithm to understand what parts can be executed in parallel:

```plaintext
a = alice();  # no dependancy
b = bob();    # no dependancy
s = boss(a,b); # s depends on a,b
c = cy();      # no dependancy
# bigboss depends on s and c
printf("%6.2f\n", bigboss(s,c));
```

alice, bob, and cy can be computed in parallel
#pragma omp sections

#pragma omp sections
Must be inside a parallel region
Precedes a code block containing of N blocks of code that may be executed concurrently by N threads
Encompasses each omp section
#pragma omp section

Precedes each block of code within the encompassing block described above

May be omitted for first parallel section after the parallel sections pragma

Enclosed program segments are distributed for parallel execution among available threads
sections + section example

```c
#pragma omp parallel sections
{
    #pragma omp section
        a = alice();
    #pragma omp section
        b = bob();
    #pragma omp section
        c = cy();
}

s = boss(a, b);
printf("%6.2f\n", bigboss(s, c));
```
task
worksharing construct
flexible task decomposition
#pragma omp task

Allows parallelization of irregular problems
- unbounded loops
- recursive algorithms
- producer/consumer

A task is composed of:
- **Code** to execute
- **Data** environment
- Internal control variables (ICV)
```c
#pragma omp task

#pragma omp parallel
{
#pragma omp single private(p)
{
  while (p) {
#pragma omp task
    processwork(p);
    p = p->next;  // not in the task
  }
}
}  // end of the single region

}  // end of the parallel region
```
synchronizations

Tasks are guaranteed to be complete:

- At thread or task barriers
- At the directive:
  `#pragma omp barrier`
- At the directive:
  `#pragma omp taskwait`
#pragma omp parallel
{
  #pragma omp task
  foo();
  #pragma omp barrier

  #pragma omp single
  {
    #pragma omp task
    bar();
  }
}
Sharing variables
Problems
OpenMP shared-memory model

OpenMP worker threads and the master thread are sharing the same process and some variables.

If your variable scope is including the parallel region, it is shared by default: all the threads will read and write to the same memory location.
Shared by default

float x, y;
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
    x = a[i]; y = b[i];
    c[i] = x + y;
}

This code is executing correctly in serial, but may give **wrong** results in parallel.

Let's see why!
A race condition is **nondeterministic behavior** caused by the times at which two or more threads access a **shared variable**.

Let's suppose we have 2 threads executing:

\[
x = a[i]; \quad y = b[i]; \quad c[i] = x + y;
\]

If a thread can execute the two lines without having the other thread changing variables \(x\) and \(y\), good. But it's **not guaranteed**.

If the two threads have a mixed execution, the result \(c\) will be **wrong**.
Problem 1 : Race condition

A race condition is **nondeterministic behavior** caused by the times at which two or more threads access a **shared variable**.

Let's suppose we have 2 threads executing:

\[ x = a[i]; \quad y = b[i]; \quad c[i] = x + y; \]

If a thread can execute the two lines without having the other thread changing variables \( x \) and \( y \), good. But it's **not guaranteed**.

If the two threads have a mixed execution, the result \( c \) will be **wrong**.
Problem 1 : Race condition

A race condition is a different kind of bug, independent from the your regular serial bugs.

Race conditions may or may not be visible depending on various experimental conditions (number of cores, other software running, luck, ...).

Your software may run fine 10,000 times on your development machine and crash during deployment on the production server because of a race condition.
Problem 2 : Corruption

Independently from race conditions, writing to the same object or memory location from different threads without protection is risky.

Example : Different threads write to the serial output (console) at the same time.

• If you are lucky, messages will intercalate nicely.
• If you are not, the output may become garbled as bits of information representing the output text will be mixed together.
Problem 3 : Initialization

If you use local copies instead of global variables to prevent race conditions and corruption, the last problem is initialization.

Local variables created by the OpenMP layer may or may not be initialized, or initialized differently depending on the directive used:

- PRIVATE
- THREADPRIVATE
- FIRSTPRIVATE
- ...

...
Sharing variables
Solutions
Solutions

The solutions exist:

- **Large**: force the execution of a block of code in serial with a *critical section*. Only one thread will execute this block at a given time.

- **Finer**: protect a variable (force serial execution of write operations).

- **Finest**: protect a variable with detailed information about the kind of operation. Example: *reduction*.

- **Ideal**: recode to prevent sharing.
Scalability

Thanks to Amdahl's law, we know serial parts will rapidly decrease the scalability of your parallel software.

But forcing the serial execution of a block of code or serial access to a variable is a serial part, even if it's inside a parallel region.

Try to rewrite or rethink your algorithm to prevent variable sharing and synchronization.

If you have to synchronize, select the finest granularity to minimize the serial part.
Solution 1: explicitly changing the scope

```c
float x, y;
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
    x = a[i]; y = b[i];
    c[i] = x + y;
}
```

**Before**: variables defined with global scope from the master thread, shared between threads.

**After**: local variables defined locally. Nothing shared.

**Efficient and safe.**
Solution 2: changing the scope with OpenMP

```c
float x, y;
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
    x = a[i];  y = b[i];
    c[i] = x + y;
}
```

Before: global variables shared between threads.

After: local copies of global variables. Nothing shared.

```c
float x, y;
int i;
#pragma omp parallel for private (x,y)
for(i=0; i<N; i++) {
    x = a[i];  y = b[i];
    c[i] = x + y;
}
```

Efficient and safe.
Solution 3: forcing serial execution of the critical block

float x, y;
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
    x = a[i]; y = b[i];
    c[i] = x + y;
}

Before: variables defined with global scope from the master thread, shared between threads.

After: same thing, but serial execution forced.

Safe but not scaling.
Solution 3: forcing serial execution of the `critical` block

Only one thread from a thread team can execute a `critical` section at a time. A critical section is a way to force serial execution for a block a code specifically. You can prevent the parallel execution of different but interacting blocks of code using the `critical` directive several times in your code.

```c
int a;
#pragma omp parallel for
for (i=0 ; i++ ; i<n) {
    #pragma omp critical
    a++;
    int b = 4+5; // safe execution in //
    #pragma omp critical
    a++;
}
```
Solution 3: forcing serial execution of the *critical* block

If your code has independent blocks of code, you can define different critical sections for each, giving them a name.

```c
#pragma omp parallel for
for (i=0 ; i++ ; i<n) {
  #pragma omp critical (first_lock)
    a++;
  #pragma omp critical (second_lock)
    b++;
  #pragma omp critical (first_lock)
    a++;
}
```
Solution 4: \textit{atomic}

\begin{verbatim}
float x, y;
int i;
#pragma omp parallel for
for(i=0; i<N; i++) {
    x = a[i]; y = b[i];
    c[i] = x + y;
}

float x, y;
int i;
#pragma omp parallel for private (x,y)
for(i=0; i<N; i++) {
    x = a[i]; y = b[i];
    c[i] = x + y;
}
\end{verbatim}

\textbf{Before}: global variables shared between threads.
\textbf{After}: local copies of global variables. Nothing shared.

\textbf{Efficient and safe}.
Solution 5 : reduction

Instead of protecting an entire block of code, is it enough to protect write accesses to a single shared variable only?

If yes, use *atomic* it will be a lot faster than critical:

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

*atomic* is like a mini *critical* section for a variable.
schedule clause

performance and scalability
schedule static

schedule(static [,chunk])

- Blocks of iterations of size "chunk" to threads
- Round robin distribution
- Low overhead, may cause load imbalance
**schedule dynamic**

`schedule(dynamic [,chunk])`

- Threads grab “chunk” iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance
schedule guided

schedule(guided [,chunk])

• Dynamic schedule starting with large block
• Size of the blocks shrink, no smaller than “chunk”
OpenMP Environment variables
OMP Standard Variables

You may add to your launching shell:

**OMP_NUM_THREADS=8**
will set to 8 the number of threads

**OMP_SCHEDULE="guided, 6"**
will ask to use the guided scheduler

**OMP_DYNAMIC=true**
will adjust the number of threads

**OMP_NESTED=true**
will allow nested parallelism
**KMP Intel Variables**

You may add to your launching shell:

```
KMP_AFFINITY=...
```

to define an alternate affinity pattern by hand

```
KMP_CPUINFO_FILE=cpuinfo.txt
```

will give an alternate cpuinfo file to tune affinity

OpenMP API
You can use OpenMP set of pragmas: It's safe as your code can be analyzed as regular serial code by your compiler if you do not use the -openmp flag. It's flexible, as you can define a lot of parameters in the code or from command line.

But in some cases, you may want to write code aware of the OpenMP execution: use the OpenMP API on top of pragmas.

```c
#include <omp.h>
int omp_get_thread_num(void);
int omp_get_num_threads(void);
```
License Creative Commons - By 2.5

You are free:
- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work
- to make commercial use of the work

Under the following conditions:
- Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

With the understanding that:
- Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.
- Public Domain — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- Other Rights — In no way are any of the following rights affected by the license: