

PHYS52015 Introduction to Scientific and High Performance Computing



PHYS52015 Core Ib: Introduction to High Performance Computing (HPC)

Session II: OpenMP (1/3)

Christopher Marcotte

Michaelmas term 2023

Outline



OpenMP
Loop scheduling

pollev.com/christophermarcotte820

OpenMP is a ...

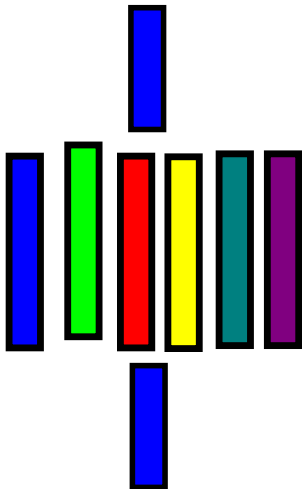
- ▶ abbreviation for *Open Multi Processing*
- ▶ allows programmers to annotate their C/FORTRAN code with parallelism specs
- ▶ portability stems from compiler support
- ▶ standard defined by a consortium (www.openmp.org)
- ▶ driven by AMD, IBM, Intel, Cray, HP, Nvidia, ...
- ▶ “old” thing currently facing its fifth (5.2) generation (1st: 1997, 2nd: 2000; 3rd: 2008; 4th: 2013, 5th: 2018)
- ▶ standard to program embarrassingly parallel accelerators – i.e., GPGPU
We’ve come a long way from manual loop unrolls in OpenCL...
- ▶ Way to iteratively parallelise serial code by identifying opportunities for concurrency

A first example

```
const int size = ...  
int a[size];  
#pragma omp parallel for  
for( int i=0; i<size; i++ ) {  
    a[i] = i;  
}
```

- ▶ OpenMP for C is a preprocessor(pragma)-based extension (annotations, not API)
 - ▶ Implementation is up to the compiler (built into recent GNU and Intel compilers; before additional precompiler required)
 - ▶ Implementations internally rely on libraries (such as pthreads)
 - ▶ Annotations that are not understood should be ignored (don't break old code)
- ▶ Syntax conventions
 - ▶ OpenMP statements always start with `#pragma omp`
 - ▶ Before GCC 4, source-to-source compiler replaced only those lines
- ▶ OpenMP originally written for BSP/Fork-Join applications
- ▶ OpenMP usually abstracts from machine characteristics
(number of cores, threads, ...)

A first example



```
const int size = ...  
int a[size];  
#pragma omp parallel for  
for( int i=0; i<size; i++ ) {  
    a[i] = i;  
}
```

Compilation and execution:

- ▶ GCC: `-fopenmp`
- ▶ Intel: `-openmp`
- ▶ Some systems require
 `#include <omp.h>`
- ▶ Set threads: `export OMP_NUM_THREADS=2`

What happens — usage

```
const int size = ...  
int a[size];  
#pragma omp parallel for  
for( int i=0; i<size; i++ ) {  
    a[i] = i;  
}
```

```
$ gcc -fopenmp test.c  
$ export OMP_NUM_THREADS=4  
$ ./a.out
```

- ▶ Code runs serially until it hits the *#pragma*
- ▶ System splits up for loop into chunks (we do not yet know how many)
- ▶ Chunks then are deployed among the available (four) threads
- ▶ All threads wait until loop has terminated on all threads, i.e. *it synchronises the threads* \Rightarrow bulk synchronous processing (bsp)
- ▶ Individual threads may execute different instructions from the loop concurrently (designed for MIMD machines)

OpenMP execution model

Explicit scoping:

```
#pragma omp parallel for
{
    for( int i=0; i<size; i++ ) {
        a[i] = i;
    }
}
```

Implicit scoping:

```
#pragma omp parallel for
for( int i=0; i<size; i++ ) {
    a[i] = i;
}
```

- ▶ Manager thread vs. worker threads
- ▶ Fork/join execution model with implicit synchronisation (barrier) at end of scope
- ▶ Nested parallel loops possible (though sometimes very expensive)
- ▶ Shared memory paradigm (everybody may access everything)

Some OMP functions

```
int numberOfThreads = omp_get_num_procs();

#pragma omp parallel for
for( int i=0; i<size; i++ ) {

    int thisLineCodeIsRunningOn = omp_get_thread_num();

}
```

- ▶ No explicit initialisation of OpenMP required in source code
- ▶ Abstracted from hardware threads—setting thread count is done by OS
- ▶ Error handling (to a greater extent) not specified by standard
- ▶ Functions almost never required (perhaps for debugging)

OpenMP function example

```
#include <stdio.h>
#include <omp.h>
int main(void)
{
    int mthread = omp_get_max_threads();
    int nthread = omp_get_num_threads();
    int thread;
    #pragma omp parallel private(thread) shared(mthread,nthread)
    {
        thread = omp_get_thread_num();
        printf("Hello, World! I am thread %d of %d of %d\n",
               thread, nthread, mthread);
    }
    return 0;
}
```

- ▶ `omp_get_max_threads()` – maximum number of threads, in this case reads `OMP_NUM_THREADS`
- ▶ `omp_get_num_threads()` – current number of threads, in this case 1
- ▶ `omp_get_thread_num()` – current thread index, in this case between 1 & `OMP_NUM_THREADS`

OpenMP function example

```
#include <stdio.h>
#include <omp.h>
int main(void)
{
    int mthread = omp_get_max_threads();
    int nthread;
    int thread;
    #pragma omp parallel private(thread,nthread) shared(mthread)
    {
        thread = omp_get_thread_num();
        nthread = omp_get_num_threads();
        printf("Hello, World! I am thread %d of %d of %d\n",
               thread, nthread, mthread);
    }
    return 0;
}
```

- What is the output of running this example with OMP_NUM_THREADS=2?

Parallel loops in action

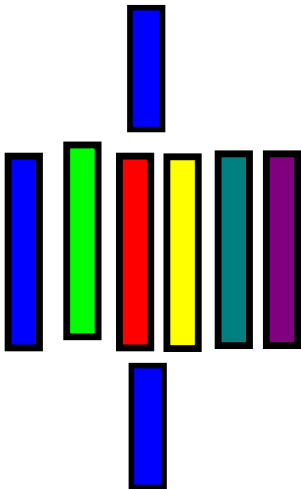
```
#pragma omp parallel           // parallel, but doesn't split the work!
{
    for( int i=0; i<size; i++ ) {
        a[i] = a[i]*2;
    }
}

#pragma omp parallel for      // splits work to be done in parallel
{
    for( int i=0; i<size; i++ ) {
        a[i] = a[i]*2;
    }
}
```

Observations:

- ▶ Global loop count is either `size` or `threads-size`
- ▶ We may run into race conditions
- ▶ These result from dependencies (read-write, write-read, write-write) on `a[i]`

Parallel loops and BSP



```
#pragma omp parallel
{
    #pragma omp for
    for( int i=0; i<size; i++ ) {
        a[i] = a[i]*2;
    }
}
```

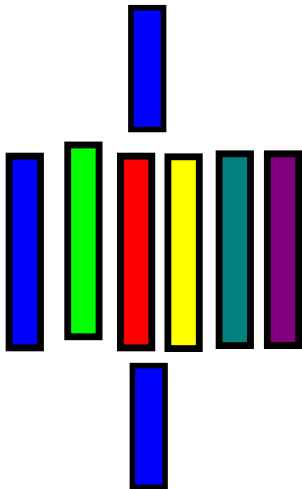
- ▶ `omp parallel` triggers fork technically, i.e. spawns the threads
- ▶ `for` decomposes the iteration range (logical fork part)
- ▶ `omp parallel for` is a shortcut
- ▶ BSP's join/barrier is done implicitly at end of the parallel section

Requirements for parallel loops

```
#pragma omp parallel for
{
    for( int i=0; i<size; i++ ) {
        a[i] = a[i]*2;
    }
}
```

- ▶ Loop has to follow plain initialisation-condition-increment pattern:
 - ▶ Only integer counters
 - ▶ Only plain comparisons
 - ▶ Only increment and decrement (no multiplication or any arithmetics)
- ▶ Loop has to be countable (otherwise splitting is doomed to fail).
- ▶ Loop has to follow single-entry/single-exit pattern.

Data consistency in OpenMP's BSP model



```
#pragma omp parallel
{
    #pragma omp for
    for( int i=0; i<size; i++ ) {
        a[i] = a[i]*2;
    }
}
```

- ▶ No assumptions which statements run technically concurrent
 - ▶ Shared memory without any consistency model
 - ▶ No inter-thread communication (so far)
- ⇒ Data consistency is developer's responsibility

Concept of building block: OpenMP Introduction

- ▶ Content
 - ▶ OpenMP syntax basics
 - ▶ OpenMP runtime model
 - ▶ OpenMP functions
- ▶ Expected Learning Outcomes
 - ▶ The student can *translate* and use an application with OpenMP support
 - ▶ The student can *explain* with the OpenMP execution model
- ▶ Further Reading:
 - ▶ We will not talk about vectorisation (using OpenMP or otherwise) in this course, but a good resource for an introduction (and many other topics!) is here: *Algorithmica* <https://en.algorithmica.org/hpc/>

Remaining agenda

Starting point:

- ▶ Data analysis allows us to identify candidates for data parallelism/BSP
- ▶ Concurrency analysis plus speedup laws determine potential real-world speedup
- ▶ OpenMP allows us to annotate serial code with BSP logic

Open questions:

- ▶ How is work technically split?
- ▶ How is work assigned to compute cores?
- ▶ What speedup can be expected in practice?

Scheduling: Assign work (loop fragments) to threads.

Pinning: Assign thread to core.

Technical remarks

On threads:

- ▶ A thread is a logically independent application part, i.e. it has its own call stack (local variables, local function history, ...)
- ▶ All threads share one common heap (pointers are replicated but not the area they are pointing to)
- ▶ OpenMP literally starts new threads when we hit `parallel for` \Rightarrow overhead
- ▶ OpenMP hides the scheduling from user code

On cores:

- ▶ Unix cores can host more than one thread though more than two (hyperthreading) becomes inefficient
- ▶ Unix OS may reassign threads from one core to the other \Rightarrow overhead
- ▶ Unix OS can restrict cores-to-thread mapping (task affinity)
- ▶ Unix OS can be forced to keep cores-to-thread mapping (pinning)

Grain size

Grain size: Minimal size of piece of work (loop range, e.g.).

- ▶ Concurrency is a theoretical metric, i.e. machine concurrency might/should be smaller
- ▶ Multithreading environment thus wrap up multiple parallel tasks into one job
- ▶ Grain size specifies how many tasks may be fused

Technical mapping of tasks

- ▶ Each thread has a queue of tasks (jobs to do)
- ▶ Each job has at least grain size
- ▶ Each thread processes tasks of its queues (dequeue)
- ▶ When all task queues are empty, BSP joins

Static scheduling

Definition:

1. Cut problem into pieces (constrained by prescribed grain size)
2. Distribute work chunks among queues
3. Disable any work stealing

In OpenMP:

- ▶ Default behaviour of `parallel for`
- ▶ Trivial grain size of 1 if not specified differently
- ▶ Problem is divided into `OMP_NUM_THREADS` chunks \Rightarrow at most one task per queue

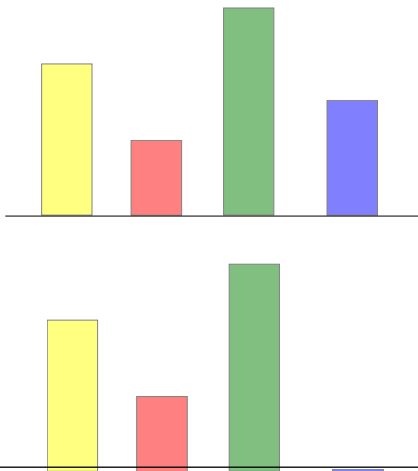
```
#pragma omp parallel for schedule(static,14)
```

Properties:

- ▶ Overhead
- ▶ Balancing
- ▶ Inflexibility w.r.t. inhomogeneous computations

Work stealing

Work stealing: When one thread runs out of work (work queue become empty), it tries to grab (steal) work packages from other threads.



Dynamic scheduling

Definition:

1. Cut problem into pieces of prescribed grain size
2. Distribute all work chunks among queues
3. Enable work stealing

In OpenMP:

- ▶ To be explicitly enabled in `parallel for`
- ▶ Trivial grain size of 1 if not specified differently
- ▶ Set of chunks is divided among `OMP_NUM_THREADS` queues first

```
#pragma omp parallel for schedule(dynamic)  
...  
#pragma omp parallel for schedule(dynamic,14)  
...
```

Properties:

- ▶ Overhead
- ▶ Balancing
- ▶ Flexibility w.r.t. inhomogeneous computations

Guided scheduling

Definition:

1. Cut problem into chunks of size N/p constrained by grain size and with $p = \text{OMP_NUM_THREADS}$
2. Cut remaining tasks into pieces of $(N/(2p))$ constrained by grain size
3. Continue cut process iteratively
4. Distribute all work chunks among queues; biggest jobs first
5. Enable work stealing

In OpenMP:

- ▶ To be explicitly enabled in `parallel for`
- ▶ Trivial grain size of 1 if not specified differently
- ▶ Set of chunks is divided among `OMP_NUM_THREADS` queues first

```
#pragma omp parallel for schedule(guided)  
...  
#pragma omp parallel for schedule(guided,14)  
...
```

Properties:

- ▶ Overhead
- ▶ Balancing
- ▶ Requires domain knowledge

Concept of building block: Loop scheduling

- ▶ Content
 - ▶ Introduce terminology
 - ▶ Discuss work stealing
 - ▶ Study OpenMP's scheduling mechanisms
 - ▶ Study OpenMP's two variants of dynamic scheduling
 - ▶ Conditional parallelisation
- ▶ Expected Learning Outcomes
 - ▶ The student **knows** technical terms tied to scheduling
 - ▶ The student can **explain** how work stealing conceptually works
 - ▶ The student can **identify problems** arising from poor scheduling/too small work packages
 - ▶ The student can **use** proper scheduling in OpenMP applications