Concurrent and Distributed Programming (CA4006)

A Design Pattern Based Approach to Concurrency and Parallelization (Part 3)

2022/2023

Graham Healy

These course slides are partly adapted from the original course slides prepared by:
Dr Martin Crane, Dr Rob Brennan and Dr Takfarinas Saber

Introduction to OpenMP

- Stands for *Open Multi-Processing*, or *Open specifications for Multi-Processing*
- Represents collaboration between interested parties from h/w and s/w - industry, government and academia.
- An API to facilitate explicitly direct multi-threaded, shared memory parallelism.
- Supported in C, C++, and Fortran, and on most processor architectures and OS.
- Comprises a set of compiler directives, library routines, and environment variables affecting run-time behaviour.
- Introduce it here as complementary to and usable in conjunction with MPI (more later on this) to achieve speedup

Motivations to use OpenMP

- Provides a standard among a variety of shared memory architectures/platforms.
    - Currently at OpenMP Version 5.1 stable (as of Nov 2020)
    - More details at `openmp.org/resources/`
- Establishes a simple and limited set of directives for programming shared memory machines.
    - (like MPI) we can get quite good parallelism using 3 or 4 directives ...
- Unlike MPI:
    - Facilitates incremental parallelization of a serial program,
    - Does not require ‘all or nothing’ approach to parallelization,
    - MPI scales well but is non-trivial to implement for codes originally written for serial machines & not good for shared memory

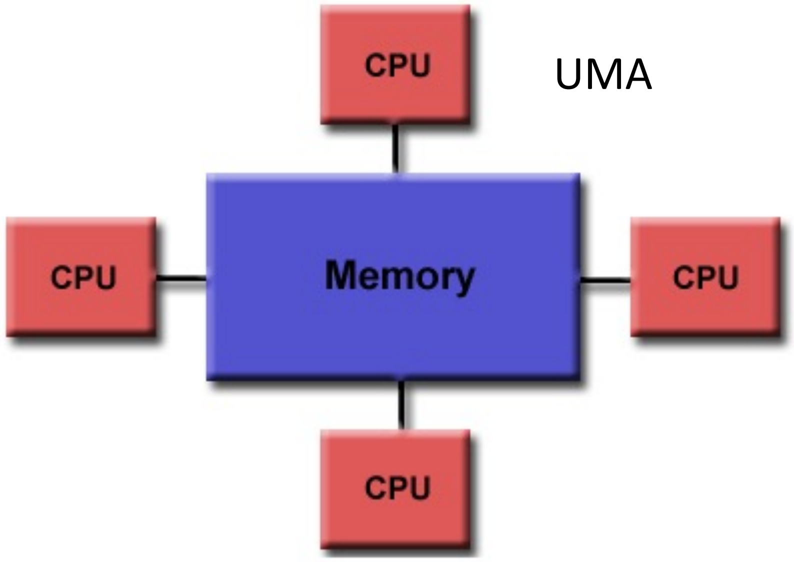
What OpenMP is not

- OpenMP Is not:
  - Meant for distributed memory parallel systems (by itself)
  - Guaranteed to make the most efficient use of shared memory

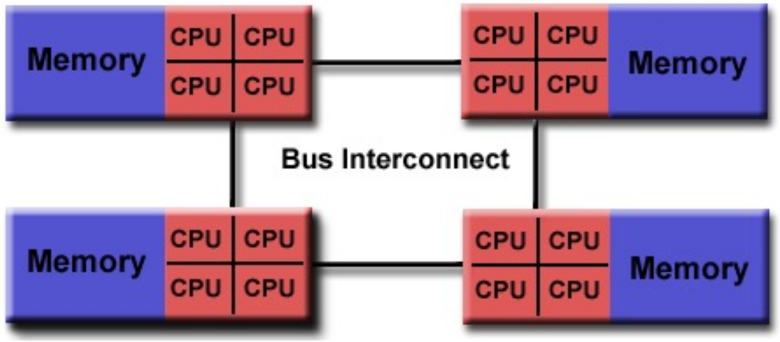
- NB OpenMP *will not:*
  - Check for data dependencies, data conflicts, race conditions, or deadlocks
  - Check for code lines that cause program to be classified as non-conforming
  - Cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization

OpenMP Programming Model

- Shared Memory Model:
  - OpenMP is designed for multi-processor/core, shared memory machines.
  - The underlying architecture can be shared memory UMA or NUMA.



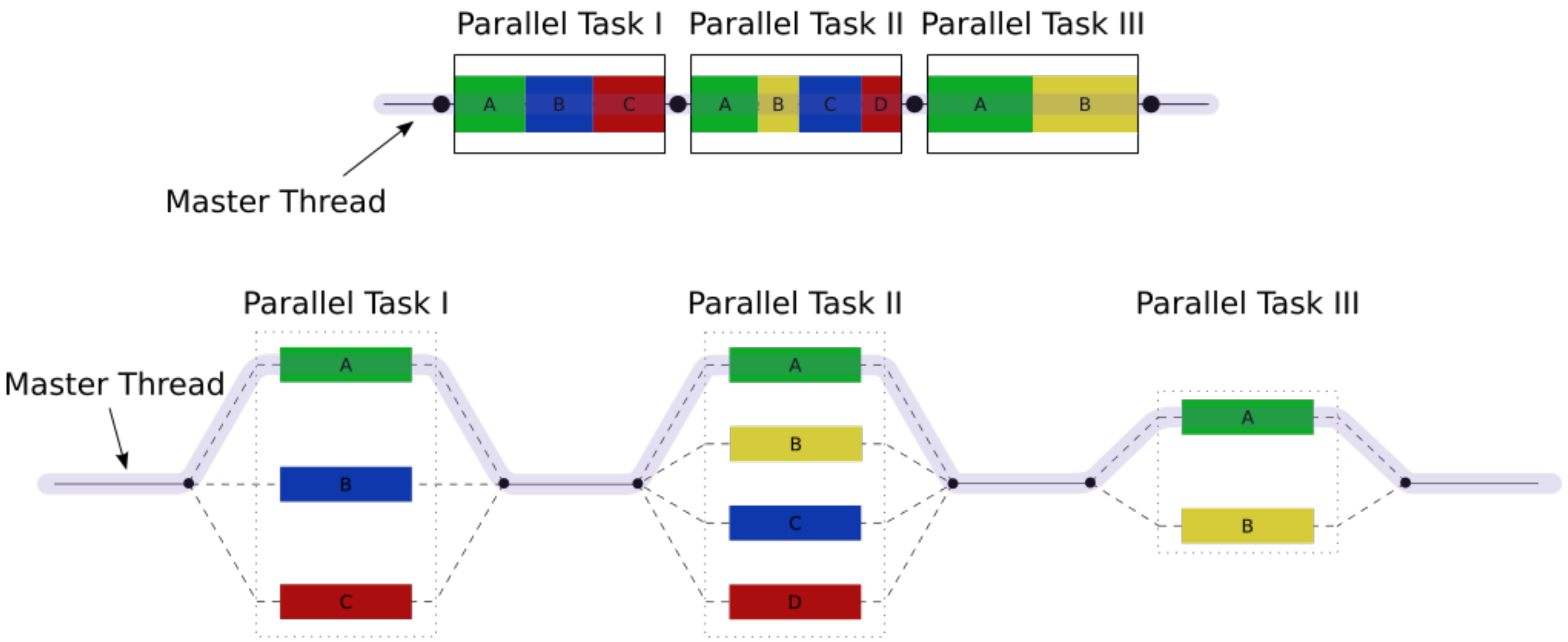
UMA



NUMA

OpenMP Programming Model (/2)

The *Fork-Join* Model



Parallelism in OpenMP

- *Thread-Based Parallelism*
  - OpenMP programs accomplish parallelism solely using *threads*.
  - A *thread of execution* is smallest processing unit schedulable by OS.
    - Analogous conceptually to a subroutine that can be scheduled to run autonomously.
  - These threads exist within resources of a single process, without which they cannot exist.
  - Usually number of threads match the number of machine processors/cores.
    - However, the actual use of threads is up to the application.

Parallelism in OpenMP (/2)

- *Explicit Parallelism*
  - OpenMP is an **explicit** (not automatic) programming model, offering the programmer full control over parallelization.
  - Parallelization can be as simple as taking a serial program and inserting compiler directives....
  - The general form of these are:

```
#pragma omp construct [clause [clause]…]
```

    - Example of this:

```
#pragma omp parallel num_threads(4)
```

  - Note about `#pragma`
    - These are special preprocessor instructions.
    - Typically added to system to allow behaviours that aren't part of the basic language specification.
    - Compilers that don't support the pragmas ignore them.

Example 1: My first OpenMP Code.

```
#include <stdio.h>
int main(void)
{
        #pragma omp parallel num_threads(2)
        printf("Hello, world.\n");
        return 0;
}
```

- *Thread Creation*
  - `pragma omp parallel` used to fork additional threads (here 2) to carry out the work enclosed in the construct in parallel.
  - The original thread is denoted as `master` thread with thread ID 0.
  - `num_threads(2)` is one of a number of clauses that can be specified .... e.g. `private` variables, `shared` variables, `reduction` operation
  - Simple Example: Display "Hello, world." using multiple threads.
  - Complex: insert subroutines to set multiple levels of parallelism, locks and even nested locks.

Example 1: My first OpenMP Code (/2).

- *Thread Creation (/2)*
  - When a thread reaches a `parallel` directive, creates a team of threads & becomes master of the team.
  - From the start of this parallel region, code is duplicated and all threads execute that code.
  - Implicit barrier at the end of a parallel section.
    - Only the master thread continues execution past this point.
  - If any thread terminates in a parallel region, all threads in team stop
  - If this happens, the work done up until that point is *undefined*.

Running this Example in OpenMP

- Use flag `-fopenmp` to compile using GCC:

```
$ gcc -fopenmp hello.c -o hello
```

- Outputs on a computer with 2 cores, and thus 2 threads:

```
Hello, world.
Hello, world.
```

- However, output may also be garbled due to race condition caused from the two threads sharing the standard output:

```
Hello, wHello, woorld.
rld.
```

- A helpful step by step example on how to run can be found at `dartmouth.edu/~rc/classes/intro_openmp/compile_run.html`

Example 2: More Complex OpenMP Code.

```
#include <stdio.h>
int main(int argc, char **argv) {
int a[100];
  #pragma omp parallel for
  for (int i = 0; i < 100; i++)
        a[i] = 2 * i;
return 0;
}
```

- Work-sharing constructs
  - `omp for`/ `omp do` for forking extra threads to do work enclosed in `parallel` (aka *loop* constructs).
  - This is equivalent to:

```
{   // stuff here                                    }
#pragma omp parallel //nb omp twice

#pragma omp for (int i = 0; i < 100; i++)
    a[i] = 2 * i;
return 0;
```

Data Dependencies

- Data on one thread can be dependent on data on another one
- This can result in wrong answers
  - Thread 0 may require a variable that is calculated on thread 1
  - Answer depends on timing – When thread 0 does the calculation, has thread 1 calculated it's value yet?
- Example – Fibonacci Sequence $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ more bunnies!

- Parallelize on 2 threads
  - Thread 0 gets $i\ = 3$ to 51, Thread 1 gets $i\ =\ 52$ to 100
    - Look carefully at calculation for $i\ =\ 52$ on thread 1
    - What will be values of for $i\ - 1$ and $i\ -\ 2$?

```
A [1] = 0;
A [2] = 1;
for(i = 3; i <= 100; i++){
    A [i] =  A[i-1] + A[i-2];
}
```

Data Dependencies (/2)

- *A Test for Dependency:*
  - If serial loop is executed in reverse order, will it give same result?
  - If so, it’s ok
  - You can test this on your serial code

- What about subprogram calls?

```
for(i = 0; i < 100; i++){
    mycalc(i,x,y);
}
```

Other Work Constructs in OpenMP

- **`sections`**

  Used to assign consecutive but independent code blocks to different threads

- **`single`**

  Specifying a code block that is executed by only one thread, a barrier is implied in the end

  Uses first thread that encounters the construct.

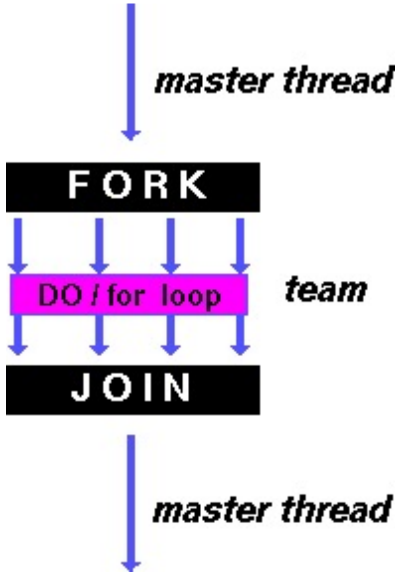
- **`master`**

  Similar to single, but code block is executed by **`master`** thread only – all others skip it

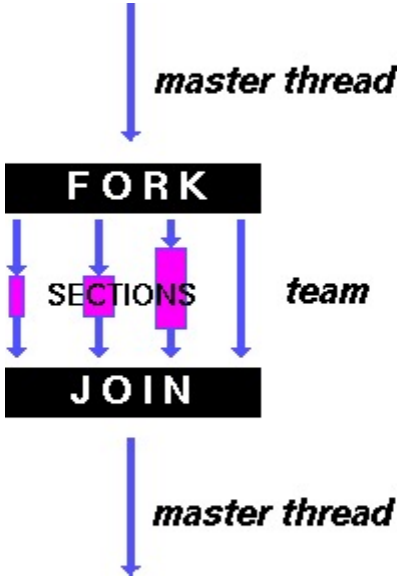
  No barrier implied in the end.

OpenMP Work Constructs (Summary)

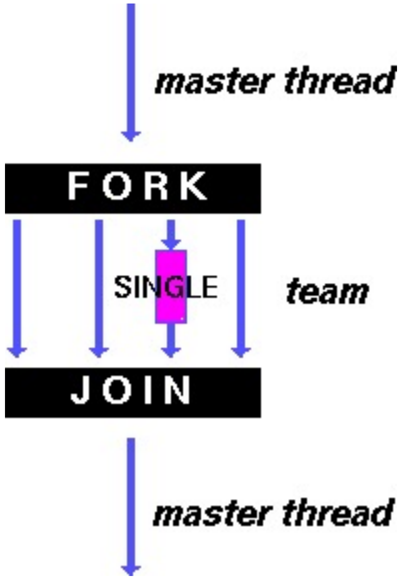
- `do` / `for` - shares loop iterations across team.
- Akin to "data parallelism"



- `sections` - breaks work into separate, discrete sections.
- Each executed by a thread.
- Can be used to implement a type of "functional parallelism".



- `single` - serializes a chunk of code



Example 3: `Sections` Construct.

```
void XAXIS(); void YAXIS(); void
ZAXIS();
void sect_example()
{
#pragma omp parallel sections
{
  #pragma omp section
        XAXIS();
  #pragma omp section
        YAXIS();
  #pragma omp section
        ZAXIS();
}
```

- Purpose
  - This `sections` directive is used to execute routines `XAXIS`, `YAXIS`, and `ZAXIS` concurrently

Example 4: single Construct.

```c
#include <stdio.h>
void work1() {}
void work2() {}
void single_example() {
#pragma omp parallel
        {
        #pragma omp single
                printf("Beginning work1.\n");
                work1();
        #pragma omp single
                printf("Finishing work1.\n");
        #pragma omp single nowait
                printf("Finished work1, starting work2.\n");
                work2();
        }
}
```

- Purpose
  - `single` directive specifies that the enclosed code is to be executed by only one thread in the team.
  - Useful dealing with sections of code that are not thread safe (such as I/O)
  - There is an implicit barrier at end of each except where a `nowait` clause is specified

Synchronisation Constructs in OpenMP

- **`critical`**

  Specifies a critical section i.e. a region of code that must be executed by only one thread at a time.

- **`atomic`**

  Commonly used to update counters and other simple variables that are accessed by multiple threads simultaneously.

- **`barrier`**

  Synchronizes all threads in the team.

  When reached, a thread waits there until all other threads have reached that barrier.

  All then resume executing in parallel the code that follows the barrier.

- **`master`**

  Strictly speaking, **`master`** is a synchronisation directive - master thread only and no barrier implied in the end.

Example 5: Data Scope Attributes

```
#include <stdio.h>
int a, b=0;
#pragma omp parallel for private(a) shared(b)
for(a=0; a<50; ++a)
{
  #pragma omp atomic // means that either happens or doesn't.
        b += a;    // one thread can't interrupt another here
}
```

- *Purpose*
  - These attribute clauses specify data scoping/ sharing.
  - As OpenMP based on shared memory programming model, most variables shared by default.
  - Used with directives e.g. `Parallel`, `Do`/ `for`, `Sections` to control the scope of enclosed variables.
  - `a` is explicitly specified `private` (each thread has own copy) and `b` is `shared` (each thread accesses same variable).

Example 6: A more Complex HelloWorld

```
#include <iostream>
using namespace std;
#include <omp.h>
int main(int argc, char *argv[])
{
 int th_id, nthreads;
 #pragma omp parallel private(th_id) shared(nthreads)
 {
         th_id = omp_get_thread_num(); // returns thread id
         #pragma omp critical // only one thread can access this at a time!
         {
                  cout << "Hello World from thread " << th_id << '\n';
         }
         #pragma omp barrier // one thread waits for all others
         #pragma omp master  // master thread access only!
         {
                  nthreads = omp_get_num_threads(); // returns number of thread
                  cout << "There are " << nthreads << " threads" << '\n';
         }
 }
 return 0;
}
```

- *Purpose*
  - `Private, shared` declares that threads have their own copy of the variable or share a copy, respectively.

Reduction Clauses

- *Reduction*
  - (Like MPI – you'll see later), OpenMP supports the Reduction operation.

```
int t;
#pragma omp parallel reduction(+:t)
{
        t = omp_get_thread_num() + 1;
        printf("local %d\n", t);
}
printf("reduction %d\n", t);
```

  - Reduction Operators: `+ * -` logical operators and `Min()`, `Max()`
  - The operation makes the specified variable private to each thread.
  - At the end of the computation it combines private results
  - Very useful when combined with `for` as shown below see below:

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
        for (i=0; i < 100; i++)     {
                sum += array[i];
}
```

Common Mistakes in OpenMP: #1 Missing `Parallel` keyword

```
#pragma omp for //this is incorrect as parallel keyword omitted
... // your code
```

- The code fragment will be successfully compiled, and the `#pragma omp for` directive will be simply ignored by the compiler.
- So only one thread executes the loop, and it could be tricky for a developer to uncover.
- The correct form should be:

```
#pragma omp parallel //this is correct
{
 #pragma omp for
 ... //your code
}
```

Common Mistakes in OpenMP: #2 Missing **`for`** keyword

- **`#pragma omp parallel`**
- This directive may be applied to a single code line as well as to a code fragment. This may cause unexpected behaviour of the `for` loop:

```
#pragma omp parallel num_threads(2) // incorrect as for keyword omitted
for (int i = 0; i < 10; i++)
  myFunc();
```

- If the developer wanted to share the loop between two threads, they should use the `#pragma omp parallel for` directive.
- Here the loop would have been executed 10 times indeed.
- However, the code above will be executed once in every thread. As the result, the `myFunc();` function will be called 20 times.
- The correct version of the code is provided below:

```
#pragma omp parallel for num_threads(2) // now correct
for (int i = 0; i < 10; i++)
  myFunc();
```

Common Mistakes in OpenMP:
#3 Redundant Parallization

- Applying the `#pragma omp parallel` directive to a large code fragment can lead to unexpected behaviour in cases like below:

```
#pragma omp parallel num_threads(2)
{
        ... // some lines of code
        #pragma omp parallel for
        for (int i = 0; i < 10; i++)
        {
                myFunc();
        }
}
```

- A naïve programmer wanting to share the loop execution between two threads placed the `parallel` keyword inside a parallel section.
- The result of execution is similar to previous example: the `myFunc` function will be called 20 times, not 10.
- The correct version of the code is the same as the above except for

```
#pragma omp parallel for
for (int i = 0; i < 10; i++)
```

Common Mistakes in OpenMP:
#4 Redefining the Number of Threads in a `Parallel` section

- By OpenMP 2.0 spec no. of threads cannot be redefined inside a `parallel` section without run-time errors and program termination:

```
#pragma omp parallel
{
omp_set_num_threads(2); // incorrect to call this routine here
#pragma omp for
for (int i = 0; i < 10; i++)
    myFunc();
}
```

- A correct version of the code is:

```
#pragma omp parallel num_threads(2) // correct!
{
#pragma omp for
for (int i = 0; i < 10; i++)
        myFunc();
}
```

- Or:

```
omp_num_threads(2)   // also
#pragma omp parallel // correct!
{
        #pragma omp for
        ... // more code
```