POLITECNICO DI MILANO

OpenMP

Algoritmi e Calcolo Parallelo

Daniele Loiacono

References

- Useful references
  - *Using OpenMP: Portable Shared Memory Parallel Programming*, Barbara Chapman, Gabriele Jost and Ruud van der Pas
  - *OpenMP.org*
    http://openmp.org/
  - *OpenMP Tutorial*
    https://computing.llnl.gov/tutorials/openMP/

Introduction

What is OpenMP?

- OpenMP (Open Multi-Processing) is an API for multi-platform shared memory multiprocessing programming
  - Supports C/C++ and Fortran
  - Available on Unix and on MS Windows
  - Provides compiler directives, library routines, and environment variables
- OpenMP is a portable and scalable for platforms ranging from the desktop to the supercomputer
- An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI)

History of OpenMP

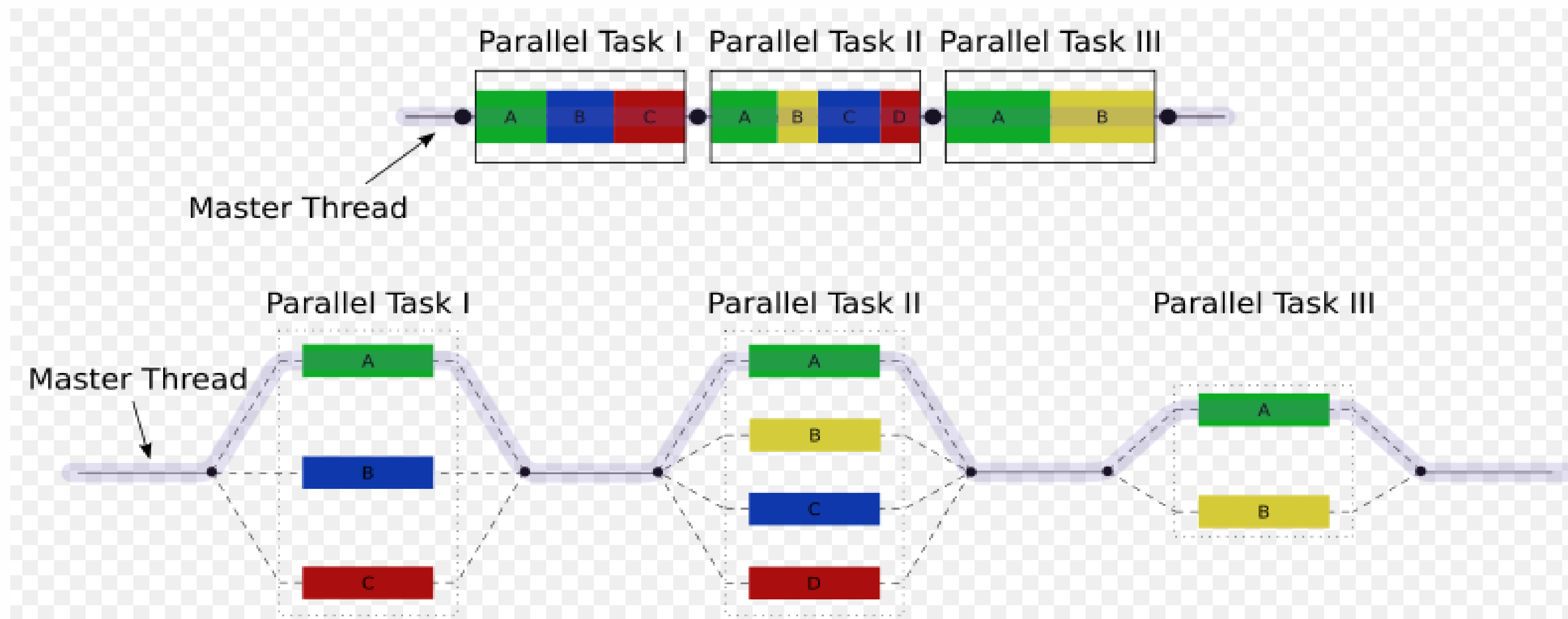
- The OpenMP Architecture Review Board (ARB) published its first API specifications, OpenMP for Fortran 1.0, in October 1997. October the following year they released the C/C++ standard.
- 2000 saw version 2.0 of the Fortran specifications with version 2.0 of the C/C++ specifications being released in 2002.
- Version 2.5 is a combined C/C++/Fortran specification that was released in 2005.
- Version 3.0, released in May, 2008, is the current version of the API specifications. Included in the new features in 3.0 is the concept of tasks and the task construct. These new features are summarized in Appendix F of the OpenMP 3.0 specifications.

Goals of OpenMP

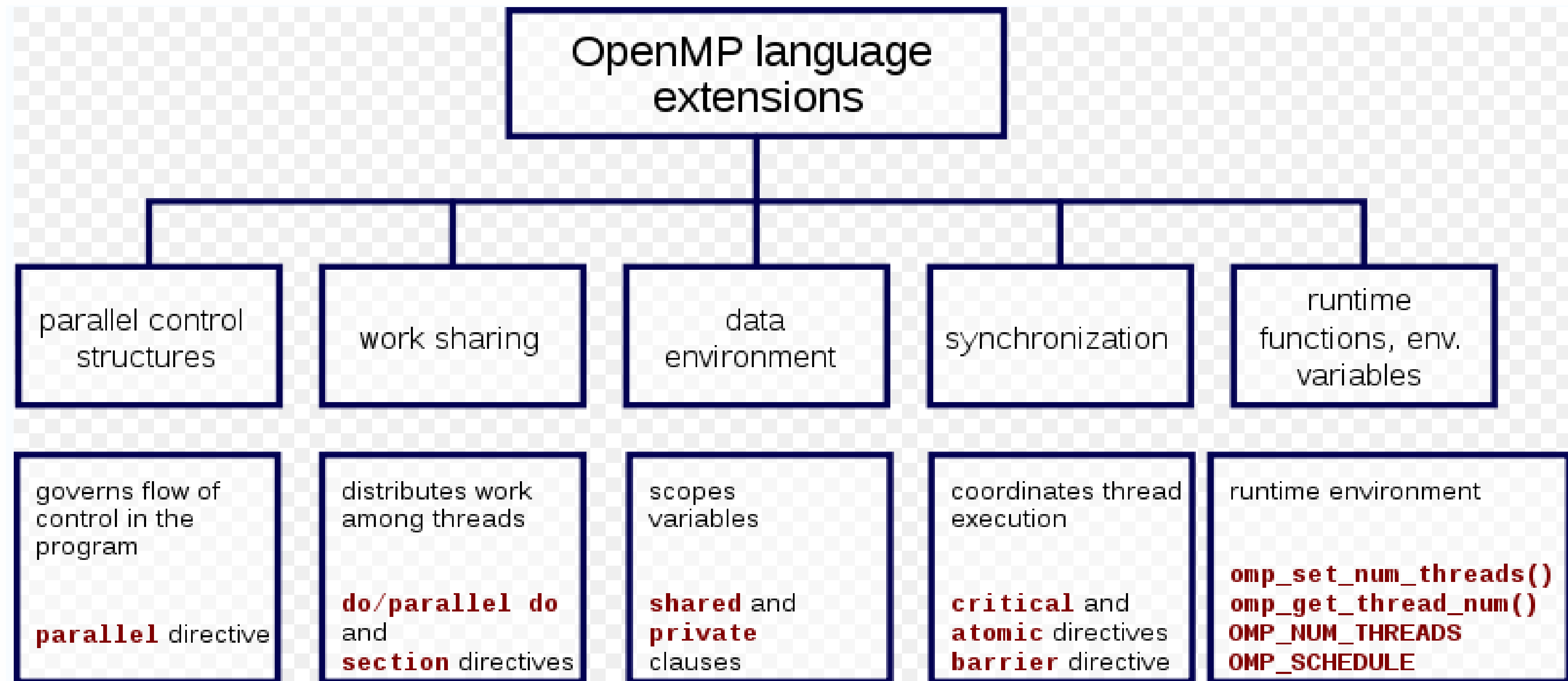
- Standardization
  - Provide a standard among a variety of shared memory architectures/platforms
- Lean and Mean
  - Establish a simple and limited set of directives for programming shared memory machines (significant parallelism can be implemented by using just 3 or 4 directives)
- Ease of Use
  - Provide capability to incrementally parallelize a serial program (unlike message-passing libraries)
  - Provide the capability to implement both coarse-grain and fine-grain parallelism
- Portability
  - Supports Fortran (77, 90, and 95), C, and C++

How does it work?

- OpenMP is an implementation of multithreading
  - A **master** thread "forks" a specified number of **slave** threads
  - Tasks are divided among slaves
  - Slaves run concurrently as the runtime environment allocating threads to different processors



Overview of OpenMP



- A compiler directive in C/C++ is called a *pragma* (pragmatic information). It is a preprocessor directive, thus it is declared with a hash (#). Compiler directives specific to OpenMP in C/C++ are written in codes as follows:

```
#pragma omp <rest of pragma>
```

OpenMP programming model

- Based on compiler directives
- Nested Parallelism Support
  - API allows parallel constructs inside other parallel constructs
- Dynamic Threads
  - API allows to dynamically change the number of threads which may used to execute different parallel regions
- I/O
  - OpenMP specifies nothing about parallel I/O
  - It is up to the programmer to insure that I/O is conducted correctly within the context of a multi-threaded program
- Memory consistency
  - Threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time
  - When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is FLUSHed by all threads as needed

Basic Elements

Hello World in OpenMP

```
#include <omp.h>
#include <iostream>
using namespace std;
int main()
{
   #pragma omp parallel num_threads(3)
   {
      cout << "Hello World" << endl;
   }
}
```

```
$ ./hello
Hello World
Hello World
Hello World
```

OpenMP Directives

- A valid OpenMP directive must appear after the `#pragma omp`
- `name` is the name of OpenMP directive
- The directive name can be followed by optional clauses (in any order)
- An OpenMP directive precedes the structured block which is enclosed by the directive itself

```
#pragma omp name [clause, ...]
{
    …
}
```

- Example

```
#pragma omp parallel num_threads(3)
{
   cout << "Hello World\n";
}
```

Directive Scoping

- Static (Lexical) Extent
  - The code textually enclosed between the beginning and the end of a structured block following a directive
  - The static extent of a directives does not span multiple routines or code files
- Orphaned Directive
  - An OpenMP directive that appears independently from another enclosing directive. It exists outside of another directive's static (lexical) extent.
  - Will span routines and possibly code files
- Dynamic Extent
  - The dynamic extent of a directive includes both its static (lexical) extent and the extents of its orphaned directives.

Directive Scoping: example

```
void f(){

   #pragma omp name2 [clause, ...]
   {
      …
   }
}

int main(){

   #pragma omp name1 [clause, ...]
   {
      f();
      …
   }
}
```

Directive Scoping: example

```
void f(){

   #pragma omp name2 [clause, ...]
   {
      …
   }
}

int main(){

   #pragma omp name1 [clause, ...]
   {
      f();
      …
   }
}
```

Static extent of name1

Directive Scoping: example

```
void f(){
  #pragma omp name2 [clause, ...]
  {
      …
  }
}

int main(){

  #pragma omp name1 [clause, ...]
  {
      f();
      …
  }
}
```

Orphaned directive

Dynamic extent of name1

`parallel` directive

```
#pragma omp parallel [if (exp) | num_threads (exp)| … ]
{
}
```

- When a thread reaches a `parallel` directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.
- Threads are numbered from 0 (master thread) to N-1

`parallel` directive: how many threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - Evaluation of the `if` clause
  - Setting of the `num_threads` clause
  - Use of the `omp_set_num_threads()` library function
  - Setting of the `OMP_NUM_THREADS` environment variable
  - Implementation default, e.g., the number of CPUs on a node.
- When present, the `if` clause must evaluate to true (i.e., non-zero integer) in order for a team of threads to be created; otherwise, the region is executed serially by the master thread.

Run-Time Library Routines

Overview

- The OpenMP standard defines an API for library calls that perform a variety of functions:
  - Query the number of threads/processors, set number of threads to use
  - General purpose locking routines (semaphores)
  - Portable wall clock timing routines
- Set execution environment functions: nested parallelism, dynamic adjustment of threads.
- It may be necessary to specify the include file "omp.h".
- For the Lock routines/functions:
  - The lock variable must be accessed only through the locking routines
  - The lock variable must have type omp_lock_t or type omp_nest_lock_t, depending on the function being used.

Run-Time Routines

- `void omp_set_num_threads(int num_threads)`
  - Sets the number of threads that will be used in the next parallel region.
  - `num_threads` must be a postive integer
  - This routine can only be called from the serial portions of the code
- `int omp_get_num_threads(void)`
  - Returns the number of threads that are currently in the team executing the parallel region from which it is called.
- `int omp_get_thread_num(void)`
  - Returns the thread number of the thread, within the team, making this call. This number will be between 0 and OMP_GET_NUM_THREADS-1. The master thread of the team is thread 0

A better Hello World in OpenMP

```
#include <omp.h>
#include <iostream>
using namespace std;
int main()
{
   #pragma omp parallel num_threads(3)
   {
     cout << "Hello World. I am thread ";
     cout << omp_get_thread_num() << endl;
   }
}
```

```
$ ./hello
Hello World. I am thread 0
Hello World. I am thread 2
Hello World. I am thread 1
```

Data Scope Attribute Clauses

Variables Scope in OpenMP

- OpenMP is based upon the shared memory programming model, most variables are shared by default
- The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped. They include:
  - `private`
  - `shared`
  - `default`
  - `reduction`
- Data Scope Attribute Clauses are used in conjunction with several directives to
  - define how and which data variables in the serial section of the program are transferred to the parallel sections
  - define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads
- Data Scope Attribute Clauses are effective only within their lexical/static extent.

`private` clause

- `private (list)`
  - The `private` clause declares variables in its list to be private to each thread
- `private` variables behave as follows:
  - a new object of the same type is declared once for each thread in the team
  - all references to the original object are replaced with references to the new object
  - variables should be assumed to be uninitialized for each thread

`shared` clause

- `shared (list)`
  - The `shared` clause declares variables in its list to be shared among all threads in the team.
- A `shared` variable exists in only one memory location and all threads can read or write to that address.
- It is the programmer's responsibility to ensure that multiple threads properly access `shared` variables

`default` clause

- `default (shared | none)`
  - The default scope of all variables is either set to `shared` or `none`
- The `default` clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region
- Specific variables can be exempted from the default using specific clauses (e.g., `private`, `shared`, etc.)
- Using `none` as a default requires that the programmer explicitly scope all variables.

`reduction` clause

- `reduction` *`(operator: list)`*
  - performs a reduction on the variables that appear in its list
  - *`operator`* defines the reduction operation
- A private copy for each list variable is created for each thread
- At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable

A simple example

```
#include <omp.h>
#include <iostream>
using namespace std;
int main()
{
   int tid;
   #pragma omp parallel num_threads(3) private(tid)
   {
      tid = omp_get_thread_num();
      cout << "Hello World. I am thread " << tid << endl;
   }
}
```

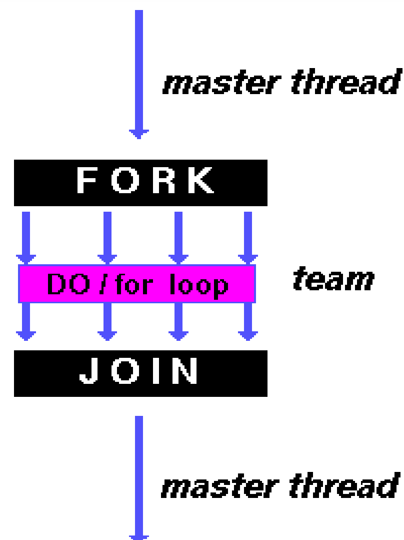
```
$ ./hello
Hello World. I am thread 0
Hello World. I am thread 2
Hello World. I am thread 1
```

Work-Sharing

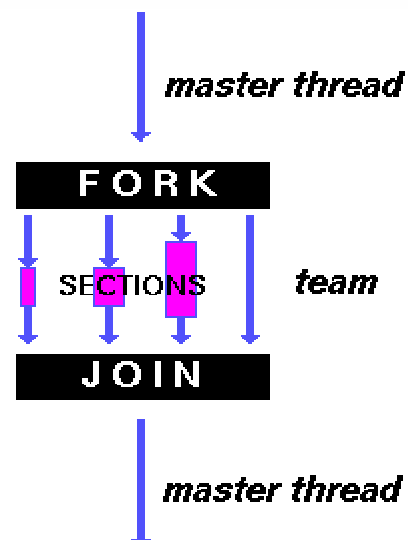
Work-sharing constructs

- Divide the execution of the enclosed code region among the members of the team that encounter it
  - A work-sharing construct must be enclosed dynamically within a `parallel` region in order for the directive to execute in parallel
  - Work-sharing constructs do not launch new threads
  - There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct

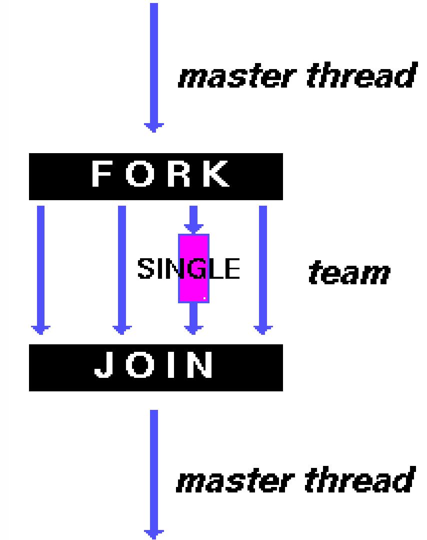
for: shares iterations of a loop across the team (data parallelism)



sections: breaks work into separate, discrete sections, each executed by a thread (functional parallelism)



single: serializes a section of code.



`for` directive

- `#pragma omp for [clause ...]`
  `for_loop`
- Most important clauses include:
  - `schedule (type [,chunk]):` edescribes how iterations of the loop are divided among the threads in the team (default schedule is implementation dependent)
  - `nowait:` if specified, then threads do not synchronize at the end of the parallel loop.
  - The data-scope clauses (`private, shared, reduction`)
- Most typical schedules are:
  - `static`: loop iterations are divided into blocks of size `chunk` and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
  - `dynamic`: loop iterations are divided into blocks of size `chunk`, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

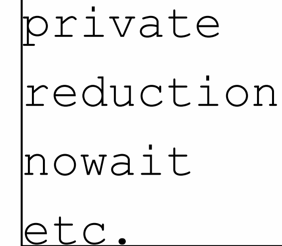
`for` directive: example

```
#include <omp.h>
#include <iostream>
#define CHUNKSIZE 100
#define N     1000
using namespace std;
int main(){
   float a[N], b[N], c[N];
   #pragma omp parallel shared(a,b,c)
   {
       #pragma omp for schedule(dynamic, CHUNKSIZE) nowait
       for (int i=0; i < N; i++)
              c[i] = a[i] + b[i];
   }// end of parallel region
}
```

`sections` directive

- `#pragma omp sections [clause ...]`

```
{
    #pragma omp section
    {
      …
    }
    #pragma omp section
    {
      …
    }
}
```



  - specifies that the enclosed section(s) of code are to be divided among the threads in the team
  - Each `section` is executed once by a thread in the team. Different sections may be executed by different threads. It is possible that for a thread to execute more than one section.

`sections` directive: example

```
#include <omp.h>
#include <iostream>
…
int main(){
    float a[N], b[N], c[N], d[N];
    #pragma omp parallel shared(a,b,c,d)
   {
       #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 region
}
```

`single` directive

- `#pragma omp single [private | nowait | …]`
```
{
…
}
#pragma omp single [private | nowait | …
```

Synchronization

`critical` directive

- ```
  #pragma omp critical [ name ]
  {
    …
  }
  ```
- The `critical` directive specifies a region of code that must be executed by only one thread at a time
- If a thread is currently executing inside a `critical` region and another thread reaches that `critical` region and attempts to execute it, it will block until the first thread exits that `critical` region.
- The optional `name` enables multiple different `critical` regions:
  - names act as global identifiers: different `critical` regions with the same name are treated as the same region
  - all `critical` sections which are unnamed, are treated as the same section
  ```

`critical` directive: example

```
#include <omp.h>
int main()
{
  int x;
  x = 0;
  #pragma omp parallel shared(x)
  {
    #pragma omp critical
    x = x + 1;
  }  // end of parallel region
}
```

barrier directive

- `#pragma omp barrier`
  - The `barrier` directive synchronizes all threads in the team.
  - When a `barrier` directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.
  - All threads in a team (or none) must execute the `barrier` region.