Imperial College London

Advanced Programming

Mini-Introduction to Parallel Programming with C++

Adriana Paluszny

[Many of the slides are adapted from the original Bjarne Stroustrup slides on C++]

Imperial College London

What is Parallel Programming?

- Using more than one processor or computer to complete a task
- Each processor works on its section of the problem (functional parallelism)
- Each processor works on its section of the data (data parallelism)
- Processors can exchange information

Imperial College London

Why Parallel Computing?

- This is a legitime question! Parallel computing is complex and will make debugging much more complex.

- The primary reasons for using parallel computing:
  - Save time - wall clock time
  - Solve larger problems
  - Provide concurrency (do multiple things at the same time)

Why Parallel Computing?

- Other reasons might include:
  - Taking advantage of non-local resources - using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
  - Cost savings - using multiple "cheap" computing resources instead of paying for time on a supercomputer.
  - Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

Terminology

- **serial**: code is a single thread of execution working on a single data item at any one time
- **parallel**: code has more than one thing happening at a time. This could be
  - A single thread of execution operating on multiple data items simultaneously
  - Multiple threads of execution in a single executable
  - Multiple executables all working on the same problem
  - Any combination of the above
- **task**: is the name we use for an instance of an executable. Each task has its own virtual address space and may have multiple threads.

Imperial College London

Terminology

- **node**: a discrete unit of a computer system that typically runs its own instance of the operating system
- **core**: a processing unit on a computer chip that is able to support a thread of execution; can refer either to a single core or to all of the cores on a particular chip
- **cluster**: a collection of machines or nodes that function in someway as a single resource.
- **grid**: the software stack designed to handle the technical and social challenges of sharing resources across networking and institutional boundaries. grid also applies to the groups that have reached agreements to share their resources.

Imperial College London

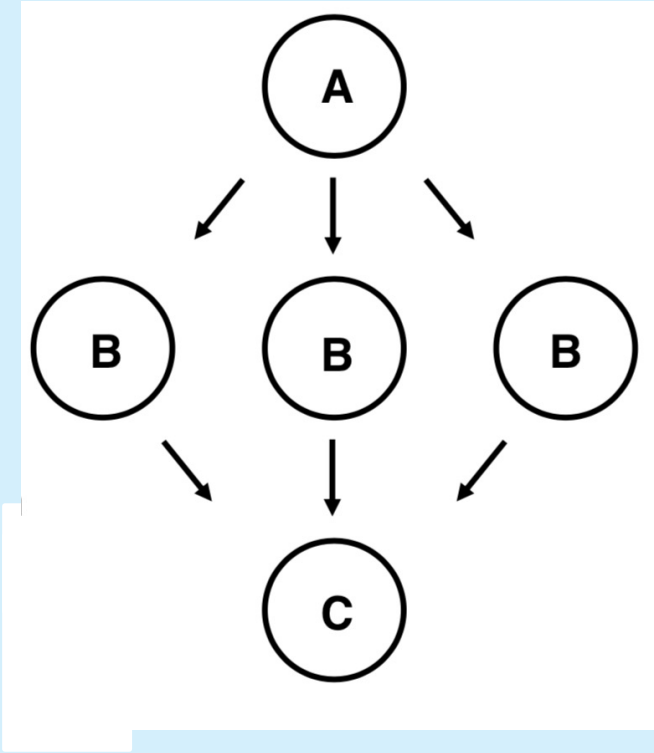
Types of parallelism

- Data parallelism
- Functional parallelism
- Task parallelism
- Pipeline parallelism

Imperial College London

Data parallelism

- Definition: each process does the same work on unique and independent pieces of data
- Examples:
  - 2 brothers mow the lawn
  - 8 farmers paint a barn
- Usually more scalable than functional parallelism
- Can be programmed at a high level with OpenMP, or at a lower level using a message-passing library like MPI.

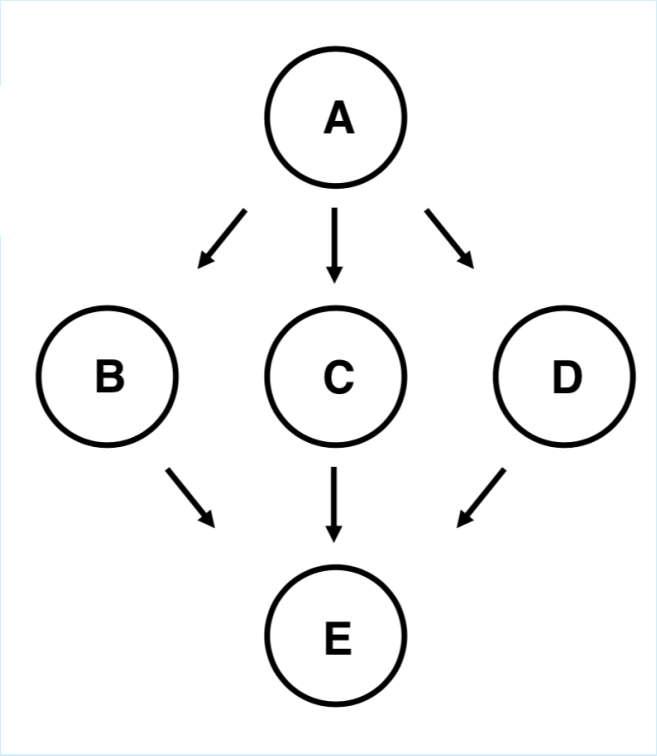


Imperial College London

Functional parallelism

- Definition: each process performs a different "function" or execute a different code sections that are independent.
- Commonly programmed with message passing libraries

- Examples:
  - 2 brothers do yard work (1 edges & 1 mows)
  - 8 farmers build a barn



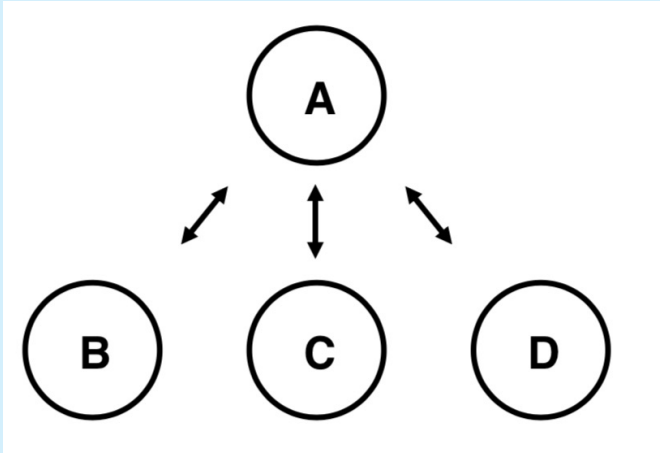
Imperial College London

Task parallelism

- Definition: each process perform the same functions but do not communicate with each other, only with a "Master" Process. are often called "Embarrassingly Parallel".
- Examples:

  - Independent Monte Carlo Simulations

  - ATM Transactions



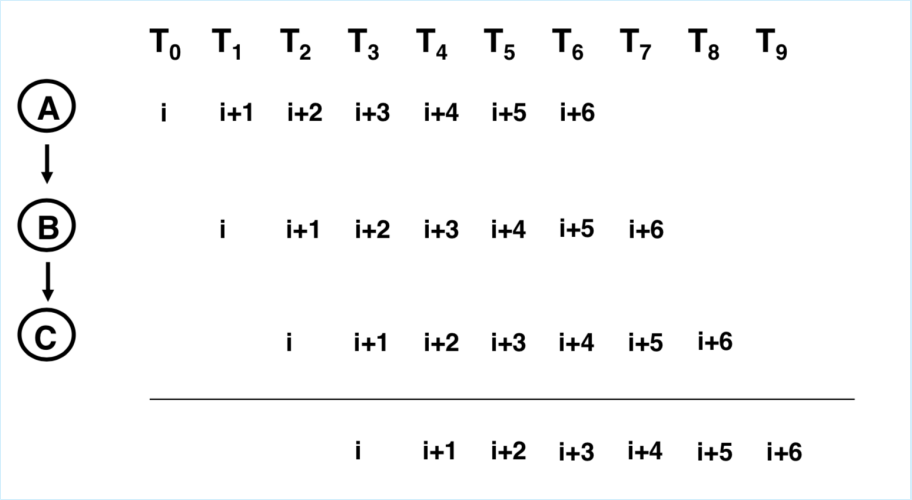
Imperial College London

Pipeline parallelism

- **Definition**: each Stage works on a part of a solution. The output of one stage is the input of the next.

(Note: This works best when each stage takes the same amount of time to complete)

- **Examples**: Assembly lines, Computing partial sums



Imperial College London

Is it really worth it to go Parallel?

- Writing effective parallel applications is difficult!!
  - Load balance is important
  - Communication can limit parallel efficiency
  - Serial time can dominate

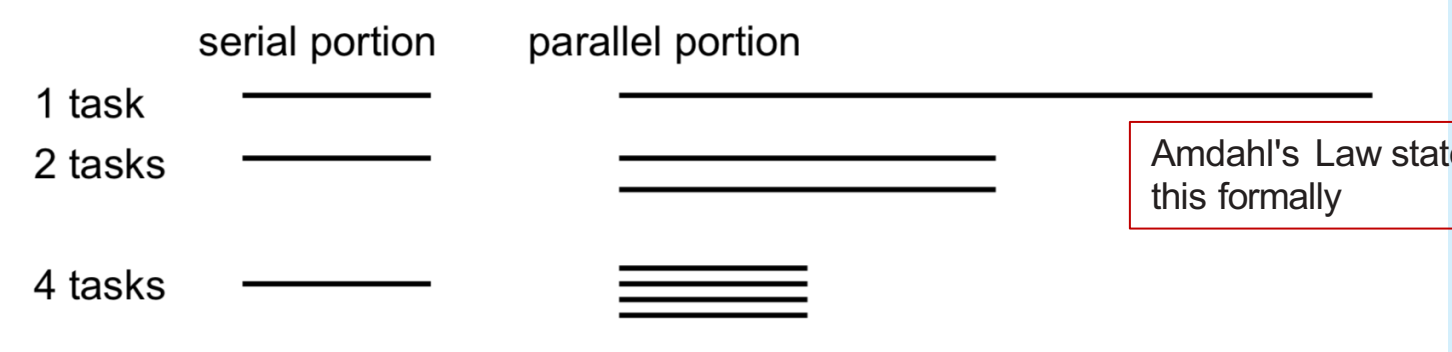
- Is it worth your time to rewrite your application?
  - Do the CPU requirements justify parallelization? Is your problem really 'large'?
  - Is there a library that does what you need (parallel FFT, linear system solving)
  - Will the code be used more than once?

Imperial College London

Theoretical Upper Limits to Performance

- All parallel programs contain:
  - parallel sections (we hope!)
  - serial sections (unfortunately)

- Serial sections limit the parallel effectiveness



Imperial College London

Amdahl's Law

- Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors.
- Effect of multiple processors on run time

$$t_n = (f_p / N + f_s)\, t_1$$

where

f_s = serial fraction of code

f_p = parallel fraction of code

N = number of processors

t_1 = time to run on one processor

Imperial College London

Limit Cases of Amdahl's Law

- Speed up formula:
  - $S = 1 / (f_s + f_p/N)$

  where

  f_s = serial fraction of code

  f_p = parallel fraction of code

  N = number of processors
- Case:

  1. $f_s = 0$, $f_p = 1$, then S = N

  2. N -> infinity: $S = 1/f_s$; if 10% of the code is sequential, you will never speed up by more than 10, no matter the number of processors

More terminology

- **synchronization**: the temporal coordination of parallel tasks. It involves waiting until two or more tasks reach a specified point (a sync point) before continuing any of the tasks.

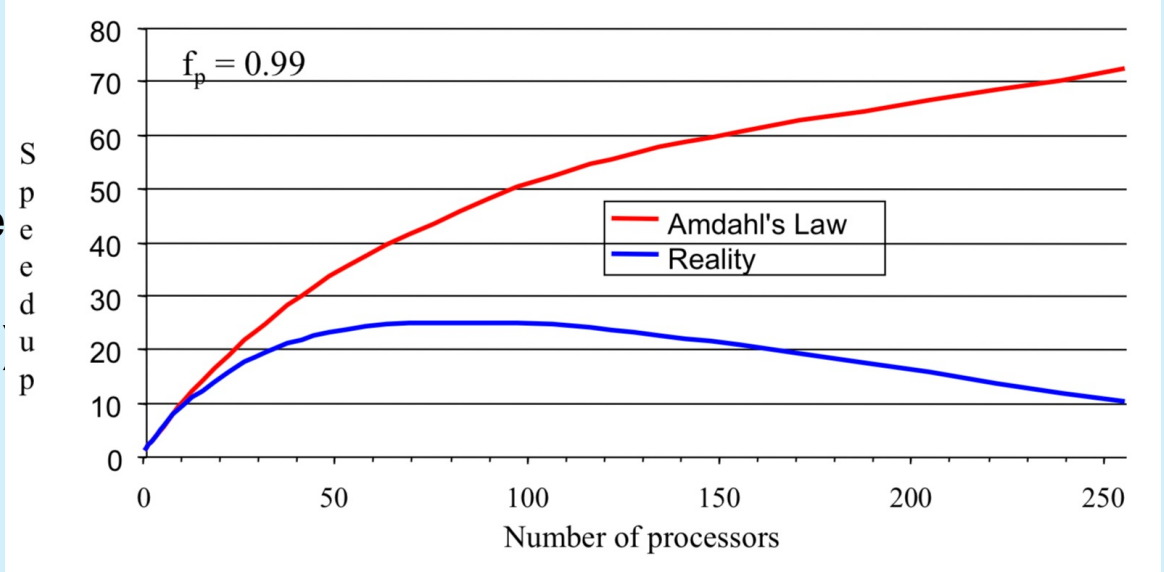
- **parallel overhead**: the amount of time required to coordinate parallel tasks, as opposed to doing useful work, including time to start and terminate tasks, communication, move data.

- **granularity**: a measure of the ratio of the amount of computation done in a parallel task to the amount of communication.
  - fine-grained (very little computation per communication-byte)
  - coarse-grained (extensive computation per communication-byte).

Imperial College London

Practical Limits: Amdahl's Law vs. Reality

- Amdahl's Law shows a theoretical upper limit for speedup
- In reality, the situation is even worse than predicte by Amdahl's Law due to:
  - Load balancing (waiting)
  - Scheduling (shared processors or memory)
  - Communications
  - I/O



Imperial College London

Shared memory systems

Most computers are now shared memory machines due to multicore

- Some are true SMP architectures... *e.g.* BlueGene/Q nodes
- ...but most are NUMA [what is “NUMA”? (see slides 10-11)]
- Program NUMA as if they are SMP – details are hidden from the user.

Difficult to build shared-memory systems with large core numbers (> 1024 cores)

- Expensive and power intensive
- Some systems manage by using software to provide shared-memory capability

Imperial College London

Memory (Reminders)

- Memory allocation in terms of stack and heap is not specified in the C++ standard.
- Instead, the standard distinguishes *automatic* and *dynamic s*torage duration.
- **Local variables** have automatic storage duration and compilers store them on the **stack**.
- Objects with dynamic memory allocation (created with new) are stored on the free store, conventionally referred to as the heap. In languages that are not garbage-collected, objects on the heap lead to memory leaks if they are not freed.

Imperial College London

When threads are created

- When threads are created they are assigned their own **Stack**
- When threads are created they share the **Heap**

- Remember: **Stack** is used for static memory allocation **and Heap** for dynamic memory allocation, both stored in the computer's RAM.
- Remember: **Stack** is faster than the **Heap**
- Remember: the sizes of the **Stack** and **Heap** depend on many factors
- **Stack** is accessed through a last-in, first-out (LIFO) memory allocation system. **Heap** Space exists as long as the application runs and is **larger** than **Stack**, which is temporary, but faster.

Imperial College London

Two patterns for parallel programming

MPI

- Implementation of the “Message Passing Interface” standard
  - Focuses on message passing
  - Ideal for writing distributed computing applications (multiple cores that do not share memory)
  - MPI is a library for message-passing between “shared-nothing” processes.

OpenMP

- Implementation of the “Multi-Processing” interface
  - OpenMP is a language-extension for expressing data-parallel operations
  - Focuses on shared memory
  - Ideal for writing shared memory multi-process applications.

OpenMP is a higher-level of abstraction, since its purpose is to expose the program's concurrency and dataflow to the compiler. By contrast, MPI concurrency is implicit (all processes are parallel), and the *messages* establish the dataflow structure of the computation.

Shared Memory Programming: OpenMP

- Shared memory systems (SMPs and cc-NUMAs) have a single address space:
  - applications can be developed in which loop iterations (with no dependencies) are executed by different processors
  - shared memory codes are mostly data parallel, ‘SIMD’ kinds of codes
  - OpenMP is the new standard for shared memory programming (compiler directives)
  - Vendors offer native compiler directives

Imperial College London

OpenMP

- Model for shared memory parallel programming
- Portable across shared-memory architectures
- **Simple**
- Incremental parallelization
  - Parallelize individual computations in a programme while leaving the rest of the programme sequential
- Compiler based
  - Compiler generates thread programmes and synchronisation
- Extension to existing languages (Fortran, C, C++)
  - Mainly using directives (#pragma omp …)
  - Few library routines

Imperial College London

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming specification with "light" syntax
  - Requires compiler support (C/C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into serial regions and parallel regions
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Avoid data races

Imperial College London

How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - Race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

Most OpenMP programs only use these items

OpenMP pragma, function, or clause	Concepts
`#pragma omp parallel`	Parallel region, teams of threads, structured block, interleaved execution across threads
`int omp_get_thread_num()` `int omp_get_num_threads()`	Create threads with a parallel region and split up the work using the number of threads and thread ID
`double omp_get_wtime()`	Timing blocks of code
`setenv OMP_NUM_THREADS N` `export OMP_NUM_THREADS=N`	Set the default number of threads with an environment variable

`#pragma omp barrier`
`#pragma omp critical`
`#pragma omp atomic`
	Synchronization, critical sections

`#pragma omp for` `#pragma omp parallel for`	Worksharing, parallel loops
`reduction(op:list)`	Reductions of values across a team of threads
`schedule(dynamic [,chunk])` `schedule(static [,chunk])`	Loop schedules
`private(list), shared(list),` `firstprivate(list)`	Data environment
`#pragma omp master` `#pragma omp single`	Worksharing with a single thread
`#pragma omp task` `#pragma omp taskwait`	Tasks including the data environment for tasks.

Credits: Tim Mattson

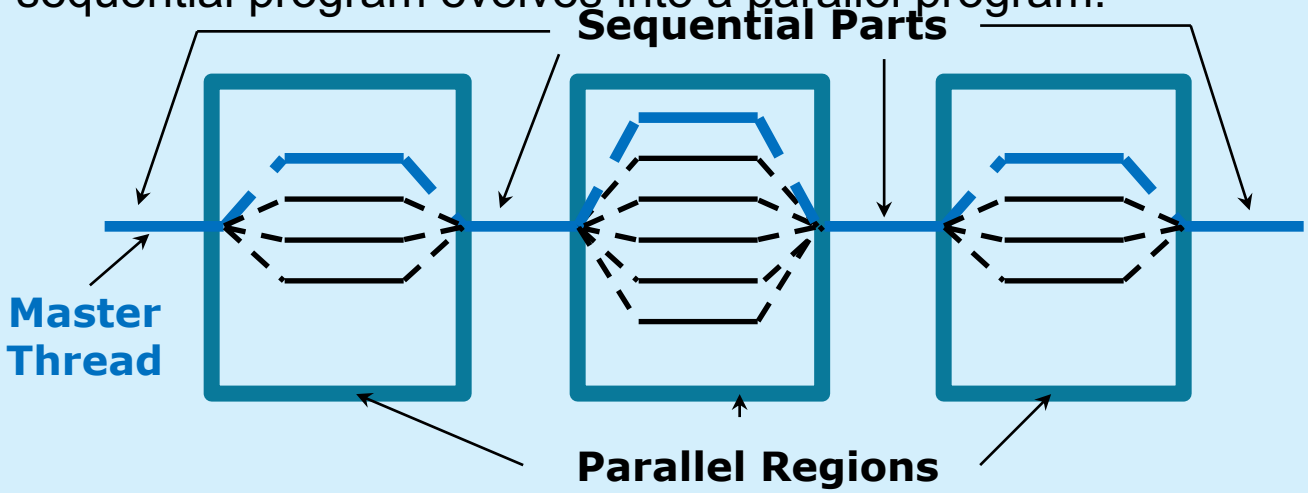
Scope

- In serial programming, the scope of a variable consists of the parts of a program where the variable can be used
- In OpenMP, the scope of a variable refers to the set of threads that can access the variable
- By default:
  - All variables that are visible at the beginning of a parallel block are shared across threads
  - All variables defined inside a parallel block are private to each thread

Imperial College London

OpenMP Programming Model:

- Fork-Join Parallelism:
  - Master thread spawns a team of threads as needed.
  - Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



Imperial College London

OpenMP Pragma Syntax

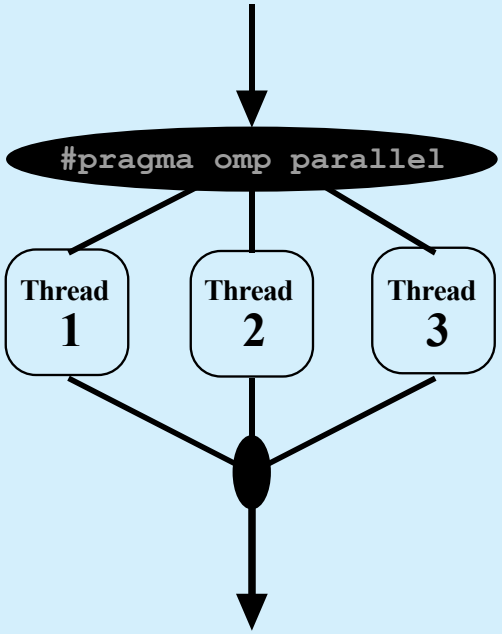
- Most constructs in OpenMP* are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:

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

Imperial College London

Parallel regions

- Defines parallel region over structured block of code
- Threads are created as **'parallel'** pragma is crossed
- Threads block at end of region
- Data is shared among threads unless specified otherwise



C/C++ :

```
#pragma omp parallel
{
        commands
}
```

Imperial College London

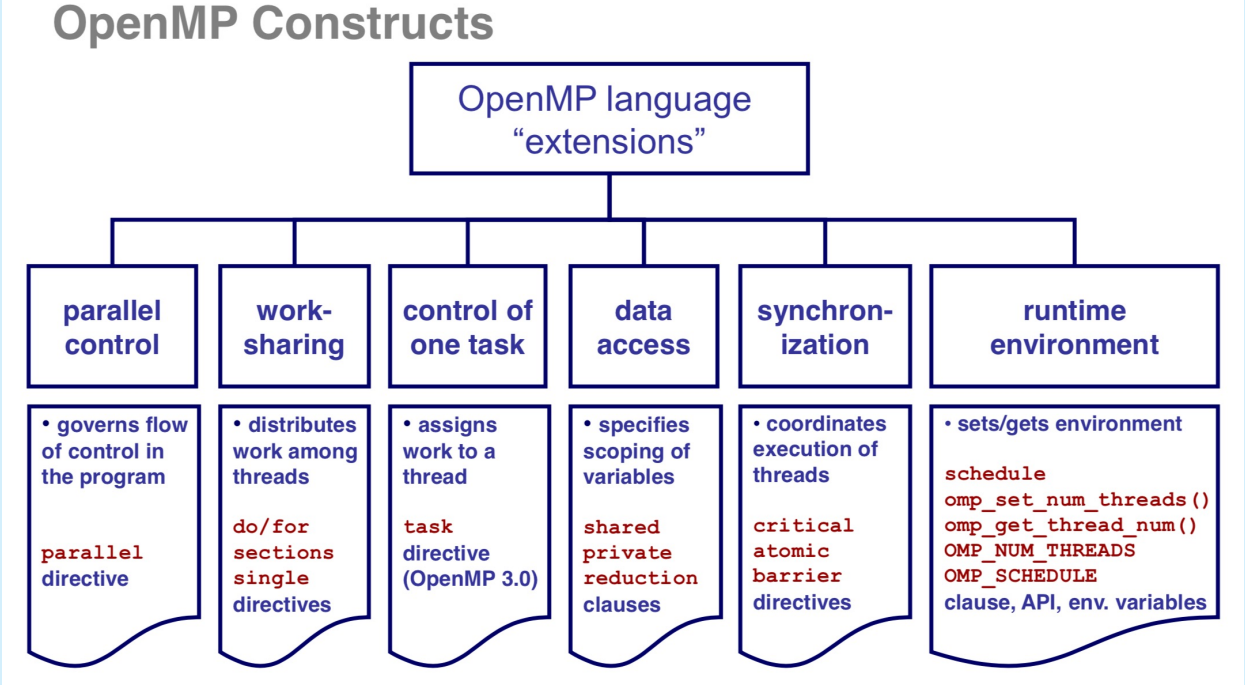
For loops

- Splits loop iterations into threads
- Must be in the parallel region
- Must precede the loop

```
#pragma omp parallel
#pragma omp for
    for (i = 0; i < n; i++)
    {
      call_function(i);
    }
```

Imperial College London

Overview of OpenMP constructs



[Cornell CompSci]