COMP 364 / 464
High Performance Computing

Thread and Task Parallelism:

Shared-memory parallel computing

Processes on an SMP System

- The OS starts a process
  - One instance of your computer program … “program.exe”
- Many processes may be executed on a single core through “time sharing” (time slicing)
  - The OS allows each process to run for awhile and then swap it out.
- The OS may run multiple processes concurrently on different cores. This is the easiest mechanism for parallelism.
- Security considerations
  - Independent processes have no direct communication (exchange of data) and are not able to read another process’s memory
- Speed considerations
  - Time sharing among processes has a large overhead … ‘context switching’

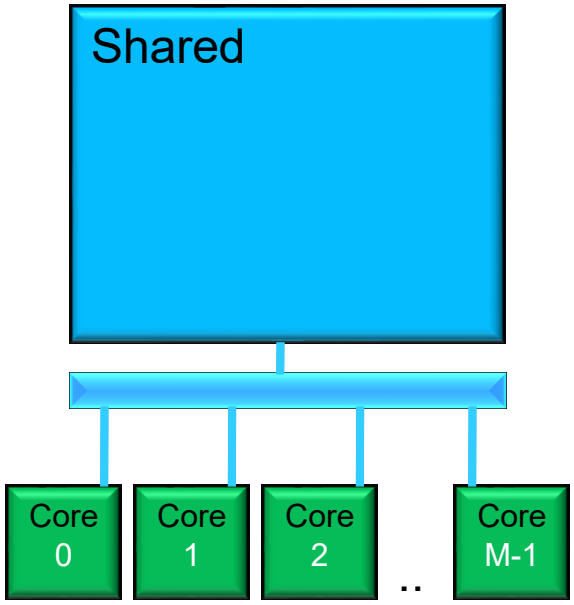
Threads

- Threads are instantiated (forked) in a program
- Threads run concurrently*
- All threads (forked from the same process) can read the memory allocated by the master thread of the process
- Each thread is given some private memory only seen by the thread (i.e., a private thread stack)
- Implementation of threads differs from one OS to another

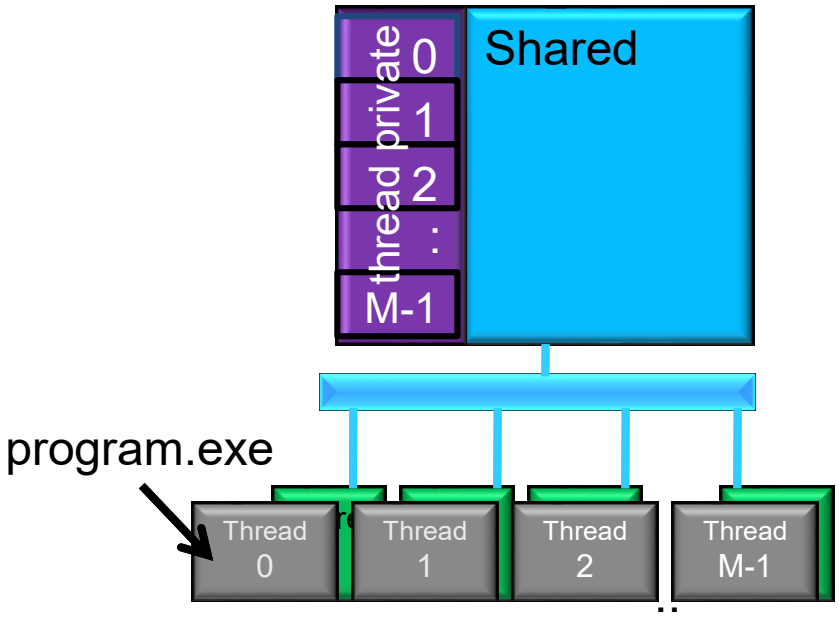
*When the # of threads forked exceeds the # of cores, time sharing (TS) will occur. Usually a bad idea. But TS with *user threads* is much less expensive than TS with *processes* … light weight context switching.

Programming with Threads on Shared Memory Systems

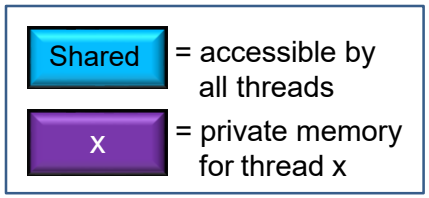
Hardware Model: Multiple Cores



Software Model: Threads in Parallel Region



M threads are usually mapped to M cores.



What is Thread Parallel Computing

- Concurrent execution of computational work (tasks).
  - Threads execute tasks concurrently from the same program.
  - Shared variable must be updated mutually exclusive (i.e., one at a time)
    - Threads can overwrite each other since data is shared.
  - Synchronization through *barriers* or other mutex objects.

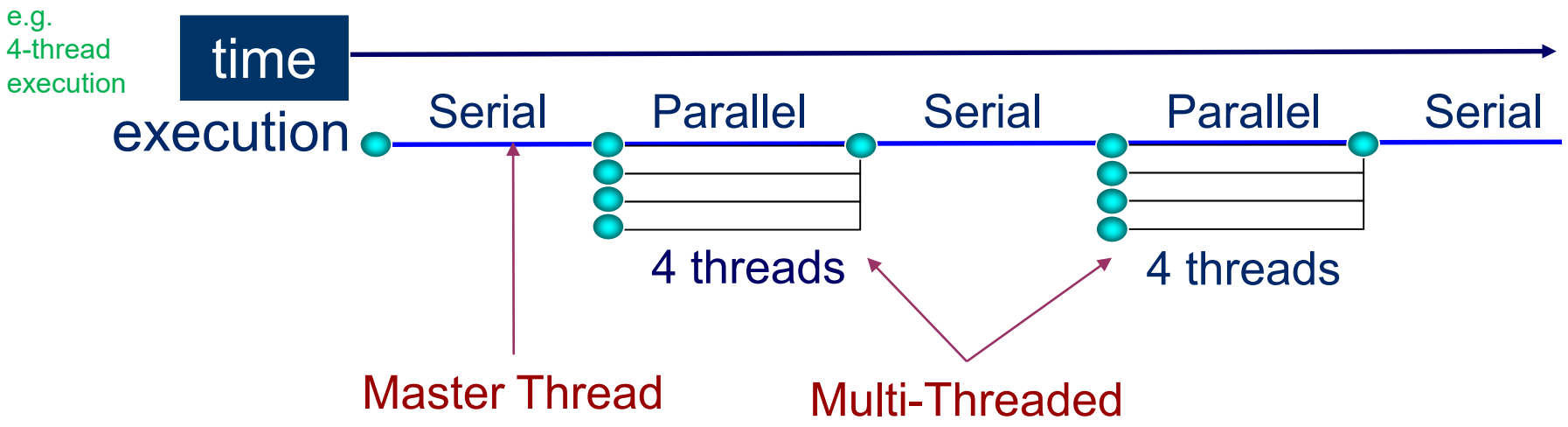
```
1  // Loops for
2  // repetitive tasks
3  #pragma omp parallel for
4  for (i = 0; i < N; i++){
5    a[i] = b[i] + c[i];
6  }
```

```
1  // Update shared
2  // variable(s)
3  #pragma omp parallel for \
4      reduction (+:prod)
5  for (i = 0; i < N; i++){
6    prod = prod + b[i]*c[i];
7  }
```

OpenMP Parallel Directives …

Fork-Join Parallelism

- Programs begin as a single process: master thread
- Master thread executes in serial mode until the parallel region is encountered (e.g., OpenMP `parallel {}` or `pthread_create()`)
- Master thread creates (forks) a team of parallel threads that simultaneously execute tasks in a parallel region
- After executing the statements in the parallel region, team threads synchronize and terminate (join) but master thread continues (serially).



Parallel Region

C/C++

```
1 ...
2 #pragma omp parallel
3 {
4   code statements
5   work(…)
6 }
```

- Line 2: Team of threads formed.
- Lines 3-6: This is the parallel region
  - Each thread executes code block and function call
- Line 6: All threads synchronize at end of parallel region (implied barrier)
- In example above, user must explicitly create independent work (tasks) in the code block and routine

Shared Data: race conditions

- Threads share “global” variables and other variables defined in the serial program before the parallel region.
  - Threads also have a private stack and have private variables.
- When multiple threads access the same data w/o mutex or other protection, the results are undefined.
  - This is a ‘*race condition*’ since you never know which thread will get there first!
- OpenMP provides basic access control primitives to avoid this.
  - All thread libraries have something similar.

```
1 int shared_int = 0;
2 #pragma omp parallel \
3     shared( shared_int )
4 {
5   // Undefined behavior.
6   shared_int++;
7 }
8 // Answer may be inconsistent.
9 printf("%d\n", shared_int);
```

How to safely share data?

- Locks: Define a special mutual exclusive (mutex) variable that is shared and only one thread can hold lock at a given time.
  - Very low-level! Avoid if you can.
- Critical regions: Define a region of code that only one thread can enter at a given time.
  - Usually just a wrapper over more complicated locks.
- Atomics: Only allow one thread to update a memory location at a time.
  - Hardware supported! No need for a shared lock anymore. Much faster (usually).
- Reductions: Implicit algorithms designed to accumulate result across all threads.
  - Avoids the need to actually share data. Example: sum result across threads.
- All are a form of synchronization and *serialize* data access. They are absolutely necessary but should be used sparingly. Why?
  - They decrease the parallel performance!
  - Any serialized execution lowers the parallel portion of the code and decreases our speed-up per Amdahl's law.

Performance Considerations

- Maximum your time in the parallel regions of the code.
- Avoid shared data structures where possible.
  - This avoids the need for critical sections or locks.
- Minimize the # of parallel regions needed.
  - Starting and stopping parallel thread teams is costly. Start them and keep them alive if possible.
- Recall that data is loaded by cacheline, not by the word. Multiple threads may try to r/w separate words but on the same cacheline. The cache subsystem will make this work but the performance can be degraded.
  - This is known as ‘false’ sharing and the performance hit gets worse as the # of threads increase … especially if the data has to go all the way to main memory.