Parallel Programming

OpenMP – Pt. 1

William McDoniel and Prof. Paolo Bientinesi

HPAC, RWTH Aachen
mcdoniel@aices.rwth-aachen.de

WS 17/18

Why Parallelism?

We program in parallel because of our architecture:

- One big core would be great!
- A multi-core CPU with shared memory (like in a laptop) needs to be told how to split up the work.
- Many CPUs with distributed memory (like in a cluster) need to split up the work *and* talk to each other.

OpenMP and MPI

The two major standards for parallelism are OpenMP and MPI.

OpenMP (Open Multi-Processing)

- Higher-level interface based on:
  - compiler directives
  - library routines
  - runtime
- Shared memory

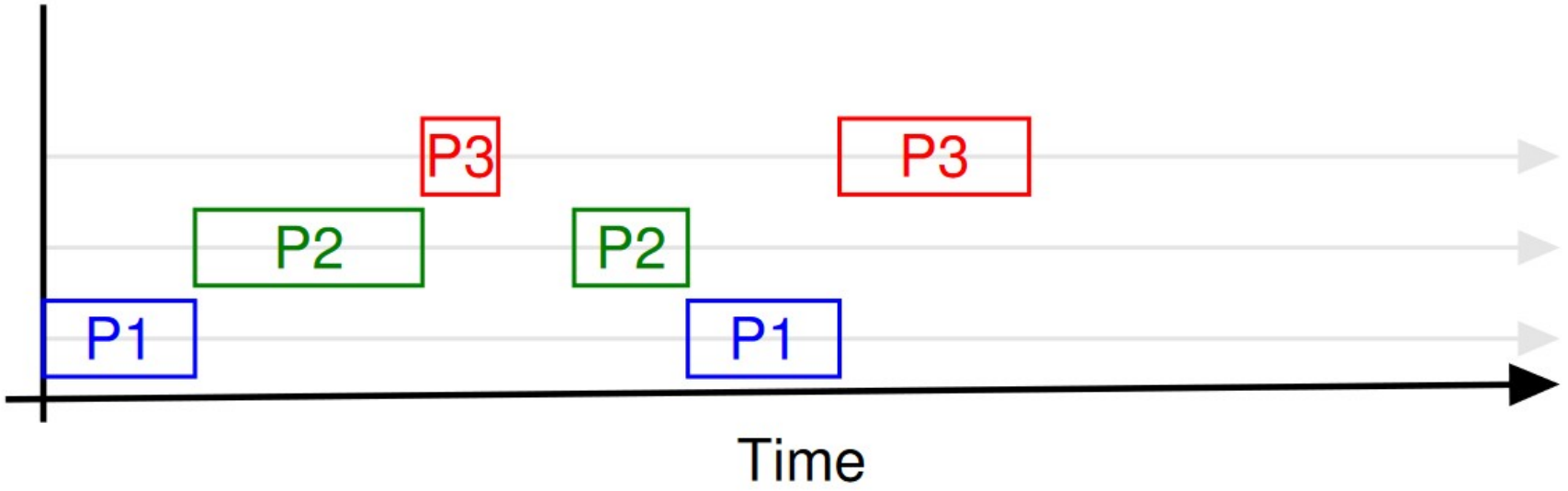
MPI (Message Passing Interface)

- Relatively low-level programming model
  - API provides communication primitives for languages
  - Data distribution and communication must be done manually
  - Primitives are easy to use, but designing parallel programs is hard
- Distributed memory

Concurrency

Time-sharing or Multitasking systems

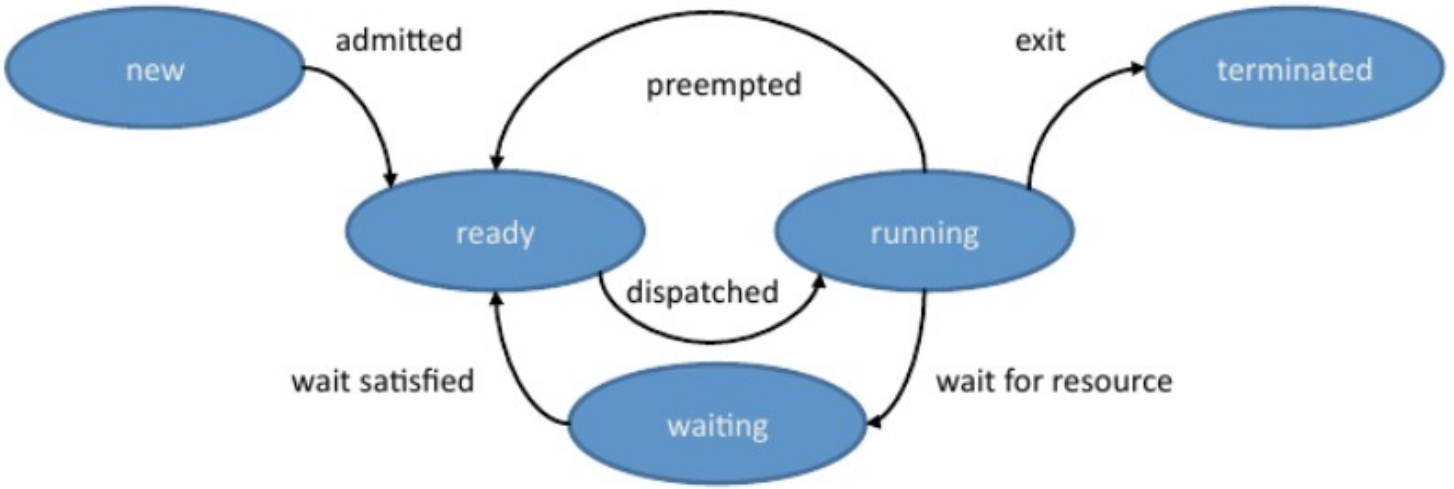
- CPU executes multiple processes by switching among them
- Switching occurs often enough for users to interact with each program while running
- In multi-core / multi-computer, processes may run in parallel



Process

- A process is an instance of a program in execution

- States of a process
  - New: the process is being created
  - Ready: waiting to be assigned to a processor
  - Running: instructions are being executed
  - Waiting: waiting for some event to occur (e.g., I/O completion)
  - Terminated: has finished execution



Thread

- Basic unit of CPU utilization
- Shares memory with other threads within the same process
- Usually a process will use 1 or 2 threads per core

Processes vs Threads

- Processes:
  - Independent
  - Have separate address spaces
  - Creation, context-switching, etc. is more expensive
  - Communicate via system-provided inter-process communication mechanisms

- Threads:
  - Exist within processes
  - Share address spaces
  - Lighter (faster creation, context switching, etc.)
  - Communicate via shared variables

OpenMP

- API for shared-memory parallelism
- Steered by the OpenMP ARB (industry, research)
- Supported by compilers on most platforms
- Not a programming language. Mainly annotations to the (sequential) code.
- OpenMP API consists of:
  - Compiler directives
  - Library routines
  - Environment variables
- Simple to use, high-level, incremental parallelism
- Performance oriented
- Data (and task) parallelism

Main ideas

- User gives a high-level specification of the portions of code to be executed in parallel

```
int main( ... )
{
    ...
    #pragma omp parallel
    {
        <region executed by multiple threads>
    }
    ...
}
```

`pragma` (pragmatic): tell the compiler to use some compiler-dependent feature/extension.

Main ideas (II)

- User may provide additional information on how to parallelize
  - `#pragma omp parallel num_threads(4)`
  - `omp_set_schedule( static | dynamic | ... );`
- OpenMP takes care of the low level details of creating threads, execution, assigning work, ...
- Provides relatively easy variable scoping, synchronization and primitives to avoid data races.
- Usage:
  - `#include "omp.h"`
  - `[gcc|icc] -fopenmp <source.c> -o <executable.x>`