CSC2/458 Parallel and Distributed Systems
PPMI: Basic Building Blocks

Sreepathi Pai

February 13, 2018

URCS

Outline

Multiprocessor Machines

Archetypes of Work Distribution

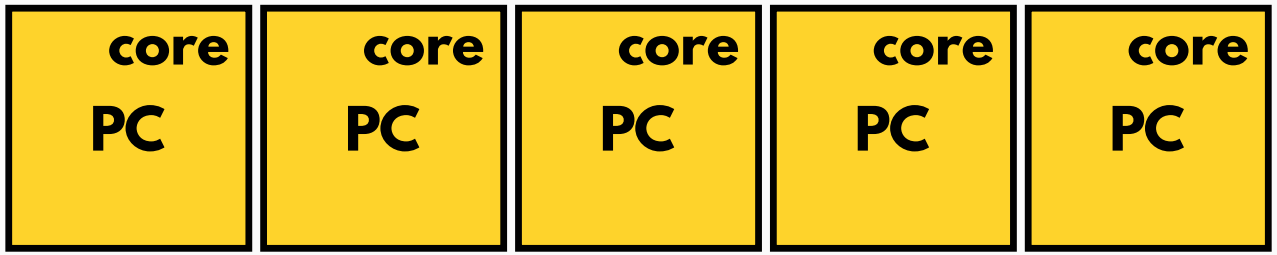
Multiprocessing

Multithreading and POSIX Threads

Non-blocking I/O or ‘Asynchronous’ Execution

Outline

Very Simplified Programmer's View of Multicore



- Multiple program counters (PC)
- MIMD machine
- To what do we set these PCs?
  - Can the hardware do this automatically for us?

Automatic Parallelization to the Rescue?

```
for(i = 0; i < N; i++) {
   for(j = 0; j < i; j++) {
      // something(i, j)
   }
}
```

- Assume a stream of instructions from a single-threaded program
- How do we split this stream into pieces?

Thread-Level Parallelism

- Break stream into long continuous streams of instructions
  - Much bigger than issue window on superscalars
  - 8 instructions vs hundreds
- Streams are largely independent
  - Best performance on current hardware
- "Thread-Level Parallelism"
  - ILP
  - DLP
  - MLP

Parallelization Issues

- Assume we have a parallel algorithm
- Work Distribution
  - How to split up work to be performed among threads?
- Communication
  - How to send and receive data between threads?
- Synchronization
  - How to coordinate different threads?
  - A form of communication

Outline

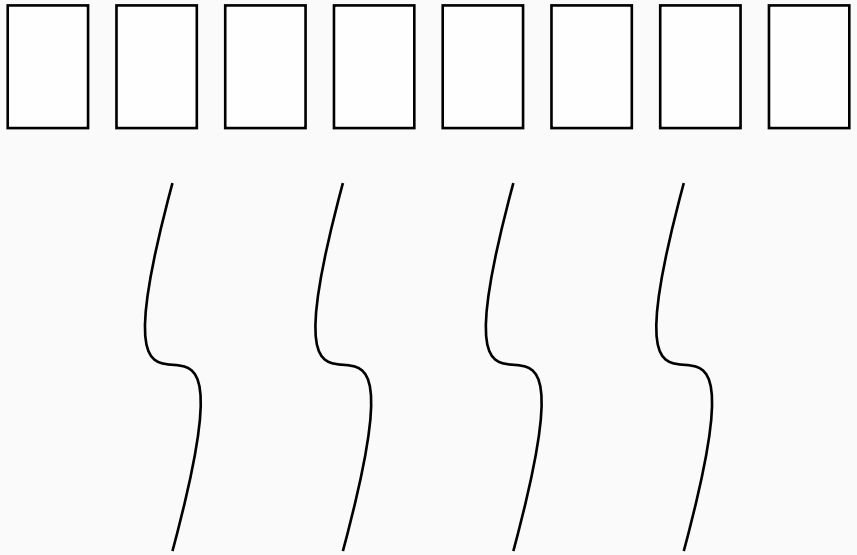
Types of Parallel Programs (Simplified)

Let's assume all parallel programs consist of "atomic" tasks.

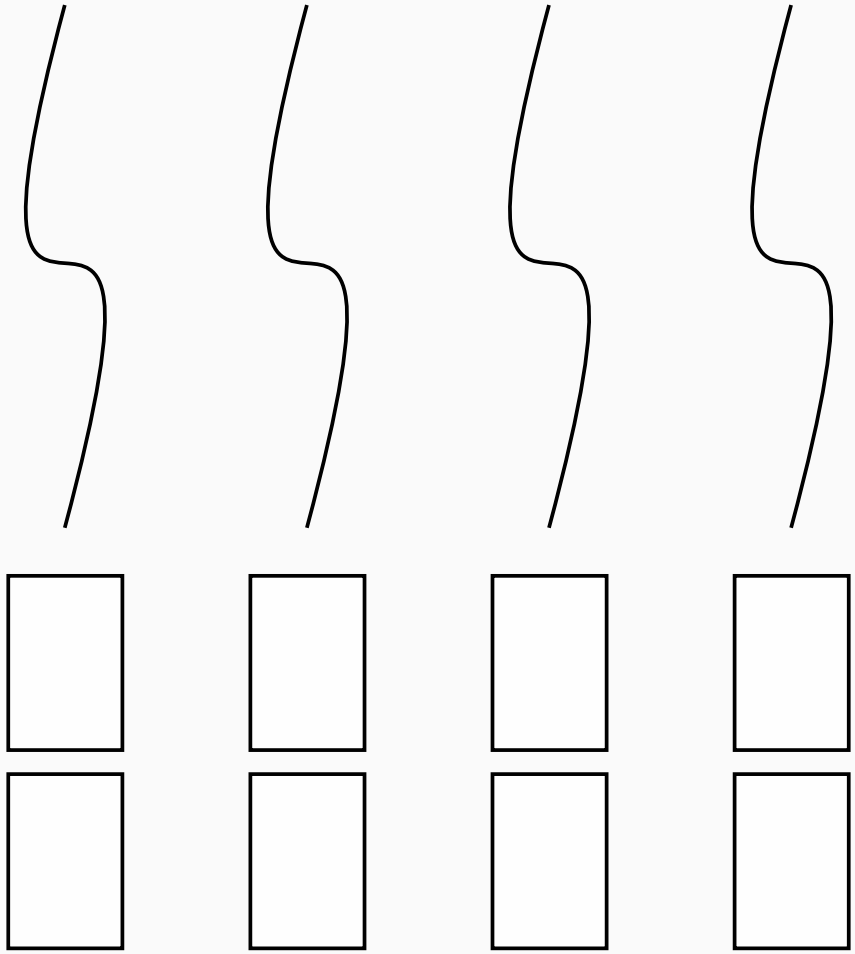
- All tasks identical, all perform same amount of work
  - Count words per page (many pages)
  - Matrix Multiply, 2D-convolution, most "regular" programs
- All tasks identical, but perform different amounts of work
  - Count words per chapter (many chapters)
  - Graph analytics, most "irregular" programs
- Different tasks
  - Pipelines
  - Servers (Tasks: Receive Request, Process Request, Respond to Request)

Scheme 1: One task per thread, same work

Count words per page of a book.



Work assigned once to threads (Static)



How many threads?

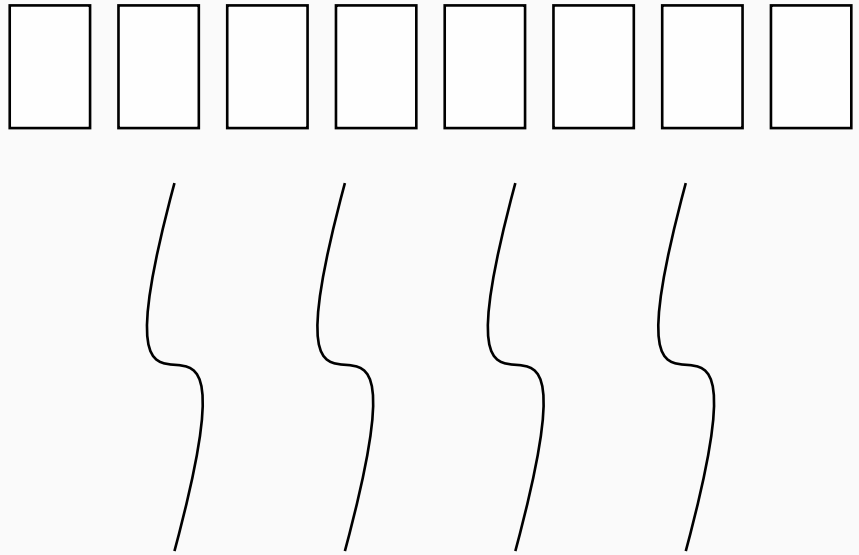
- As many as tasks
- As many as cores
- Less than cores
- More than cores

Hardware and OS Limitations

- As many as tasks
  - Too many, OS scheduler limitations
- As many as cores
  - Reasonable default
- Less than cores
  - If hardware bottleneck is saturated
- More than cores
  - May help to cope with lack of ILP

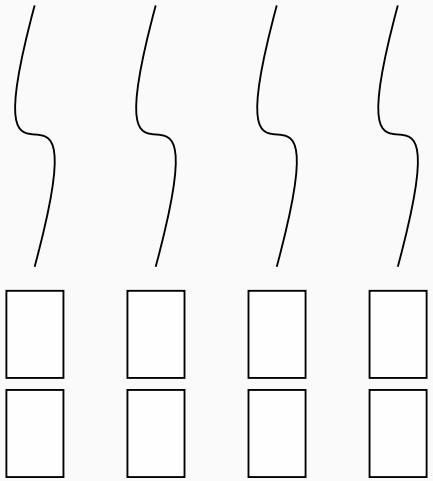
Scheme 2: Multiple tasks per thread, differing work

Count words per chapter of a book.



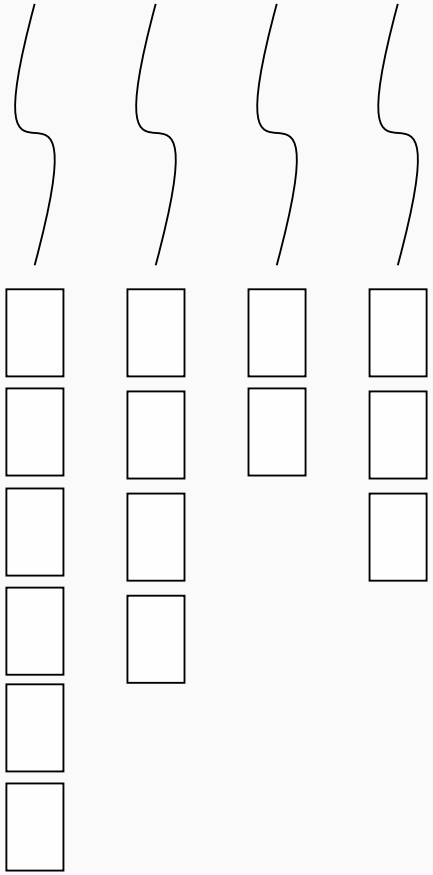
Static Work Assignment

Assign chapters evenly.



Static Work Assignment

Chapters are of different lengths leading to *load imbalance*



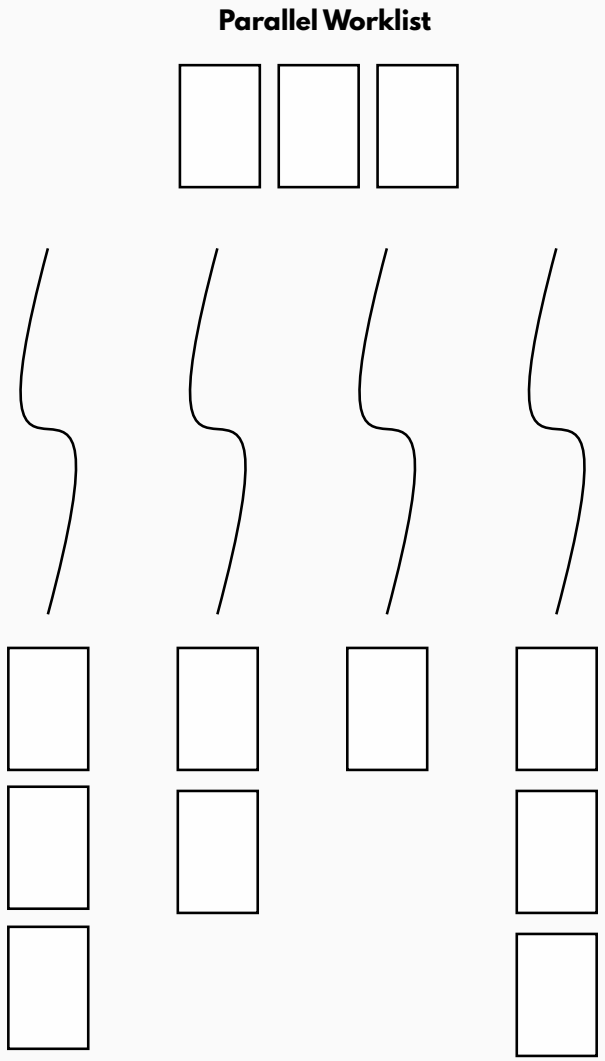
Assigning chapters by size of chapters

- Not always possible
  - May not know size of all chapters
- Bin-packing problem
  - NP-hard

Dynamic (Greedy) Balancing

- Create a set of worker threads (*thread pool*)
- Place work (i.e. chapters) into a parallel worklist
- Each worker thread pulls work off the worklist
- When it finishes a chapter, it pulls more work off the worklist

Dynamic Balancing



Generalized Parallel Programs

- Threads can create additional work ("tasks")
- Tasks may be dependent on each other
  - Form a dependence graph
- Same ideas as thread pool
  - Except only "ready" tasks are pulled off worklist
  - As tasks finish, their dependents are marked ready
- May have thread-specific worklists
  - To prevent contention on main worklist

Outline

Multiprocessing

- Simplest way to take advantage of multiple cores
  - Run multiple processes
  - `fork` and `wait`
- Traditional way in Unix
  - "Processes are cheap"
  - Not cheap in Windows
- Nothing-shared model
  - Child inherits some parent state
- Only viable model available in some programming languages
  - Python
- Shared nothing: Communication between processes?

Communication between processes

- Unix Interprocess Communication (IPC)
  - Filesystem
  - Pipes (anonymous and named)
  - Unix sockets
  - Semaphores
  - SysV Shared Memory

Outline

Multithreading

- One process
- Process creates threads ("lightweight processes")
  - How is a thread different from a process? [What minimum state does a thread require?]
- Everything shared model
- Communication
  - Read and write to memory
  - Relies on programmers to think *carefully* about access to shared data
  - Tricky

Multithreading Programming Models

Roughly in (decreasing) order of power and complexity:

- POSIX threads (pthreads)
  - C++11 threads may be simpler than this
- Thread Building Blocks from Intel
- Cilk
- OpenMP

POSIX Threads on Linux

- Processes == Threads for scheduler in Linux
  - 1:1 threading model
  - See OS textbook
- pthreads provided as a library
  - `gcc test.c -lpthread`
- OS scheduler can affect performance significantly
  - Especially with *user-level* threads

Multithreading Components

- Thread Management
  - Creation, death, waiting, etc.
- Communication
  - Shared variables (ordinary variables)
  - Condition Variables
- Synchronization
  - Mutexes (Mutual Exclusion)
  - Barriers
  - (Hardware) Read-Modify-Writes or “Atomics”

Outline

CPU and I/O devices

- CPU *compute*
- I/O devices perform I/O
- What should the CPU do when it wants to do I/O?

Parallelism

- I/O devices can usually operate in parallel with CPU
  - Read/write memory with DMA, for example
- I/O devices can inform CPU when they complete work
  - (Hardware) Interrupts
- How do we take advantage of this parallelism?
  - Even with a single-core CPU?
  - Hint: OS behaviour on I/O operations?

Non-blocking I/O within a program

- Default I/O programming model: block until request is satisfied
- Non-blocking I/O model: don't block
  - also called "Asynchronous I/O"
  - also called "Overlapped I/O"
- Multiple I/O requests can be outstanding at the same time
  - How to handle completion?
  - How to handle data lifetimes?

General Non-blocking I/O Programming Style

- Operations don't block
  - Only succeed when guaranteed not to block
  - Or put request in a (logical) queue to be handled later
- Operation completion can be detected by:
  - Polling (e.g. `select`)
  - Notification (e.g. callbacks)

Programming Model Constructs for Asynchronous Programming

- Coroutines
- Futures/Promises