Parallel Programming

Why Bother with Programs?

They are what runs on the machines we design

- Helps make design decisions
- Helps evaluate systems tradeoffs

Led to the key advances in uniprocessor architecture

- Caches and instruction set design

More important in multiprocessors

- New degrees of freedom
- Greater penalties for mismatch between program and architecture

Important for Whom?

Algorithm designers

- Designing algorithms that will run well on real systems

Programmers

- Understanding key issues and obtaining best performance

Architects

- Understand workloads, interactions, important degrees of freedom
- Valuable for design and for evaluation

Outline

Motivating Problems (application case studies)

Steps in creating a parallel program

What a simple parallel program looks like

- In the three major programming models
- What primitives must a system support?

Later: Performance issues and architectural interactions

Creating a Parallel Program

Assumption: Sequential algorithm is given

- Sometimes need very different algorithm, but beyond scope

Pieces of the job:

- Identify work that can be done in parallel
- Partition work and perhaps data among processes
- Manage data access, communication and synchronization
- *Note*: work includes computation, data access and I/O

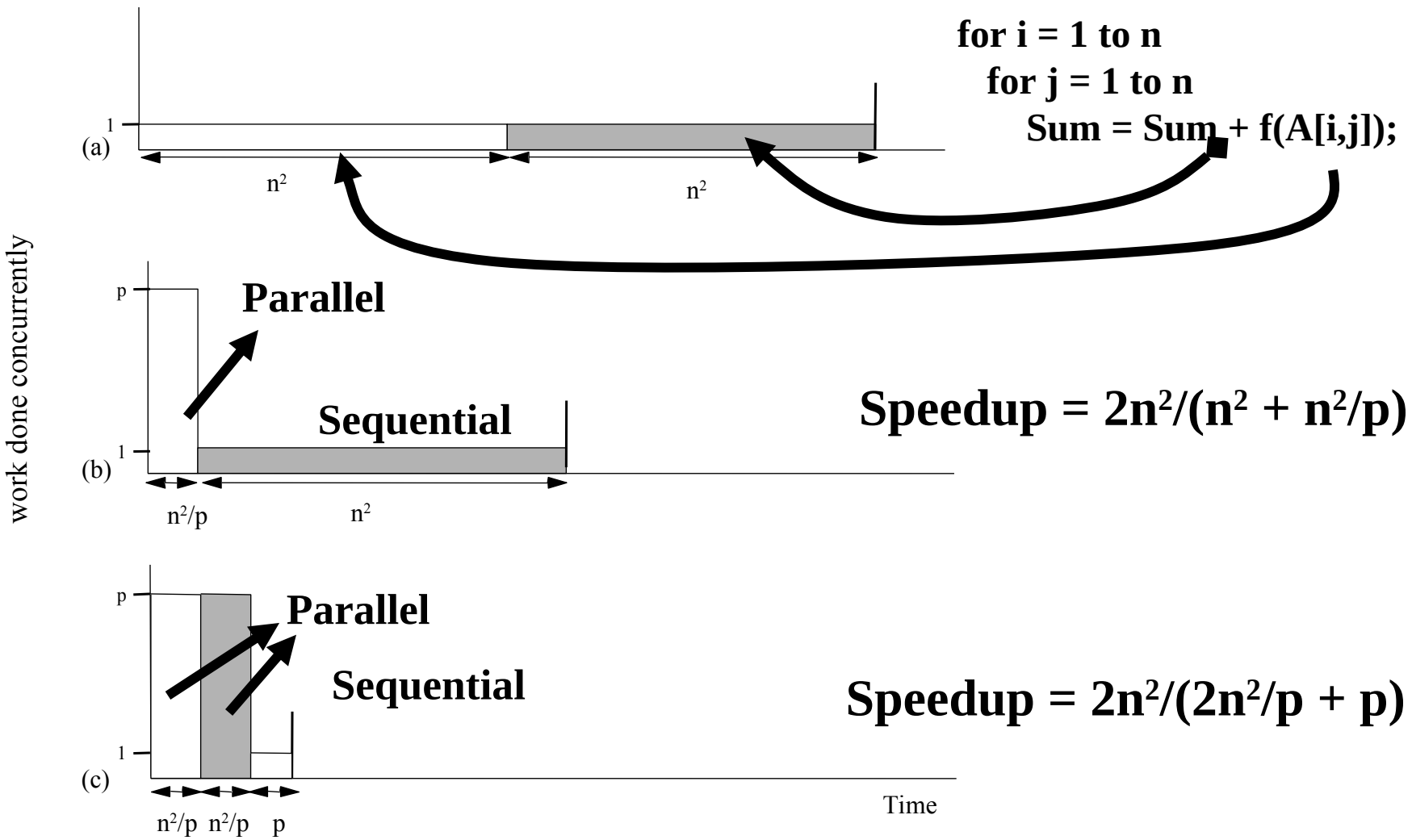
Main goal: Speedup (plus low prog. effort and resource needs)

$$Speedup\ (p) = \frac{Performance(p)}{Performance(1)}$$

For a fixed problem:

$$Speedup\ (p) = \frac{Time(1)}{Time(p)}$$

Pictorial Depiction



What does this mean?

Ws = Fraction of work that is serial

Wp = Fraction of work that is parallel

Execn Time with 1 processor = Ws + Wp

Execn Time with p processors = Ws + Wp/p

Speedup = Ws + Wp / (Ws + Wp/p)

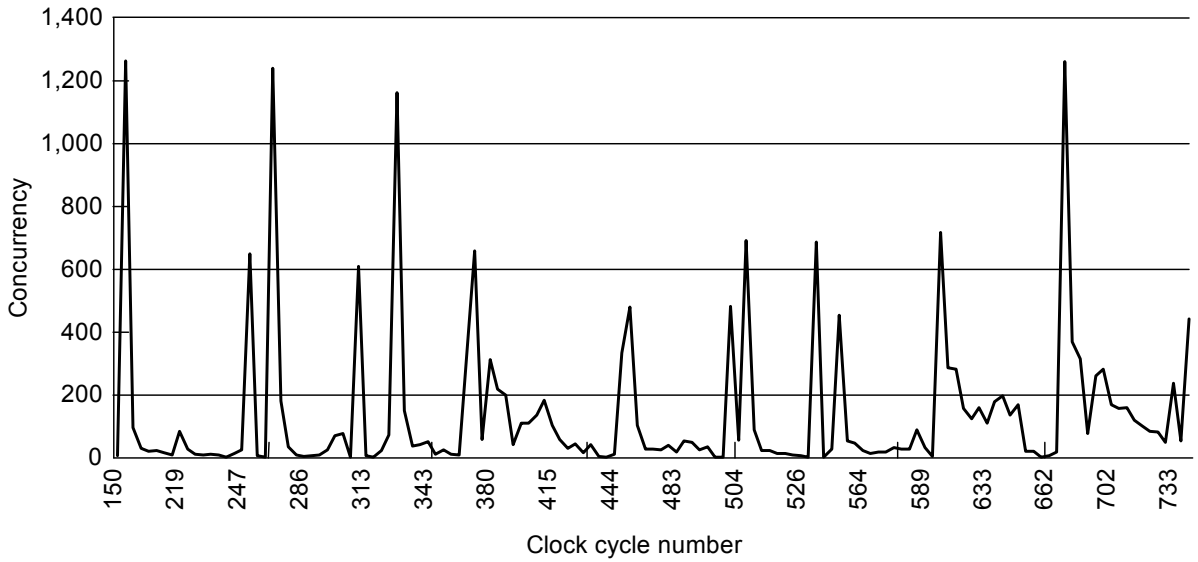
= 1 / (Ws / (Ws + Wp) + Wp / p*(Ws + Wp))

Note that Ws / (Ws + Wp) = s (the serial fraction of the work)

Speedup < 1 / s, i.e. regardless of how many processors you give it, the speedup is limited by the serial fraction => Amdahl's Law

Concurrency Profiles

- Usually time varying



- Area under curve is total work done, or time with 1 processor
- Horizontal extent is lower bound on time (infinite processors)
- Speedup is the ratio: $\frac{\sum_{k=1}^{\infty} f_k k}{\sum_{k=1}^{\infty} f_k \left\lceil \frac{k}{p} \right\rceil}$, base case: $\frac{1}{s + \frac{1-s}{p}}$
- Amdahl's law applies to any overhead, not just limited concurrency

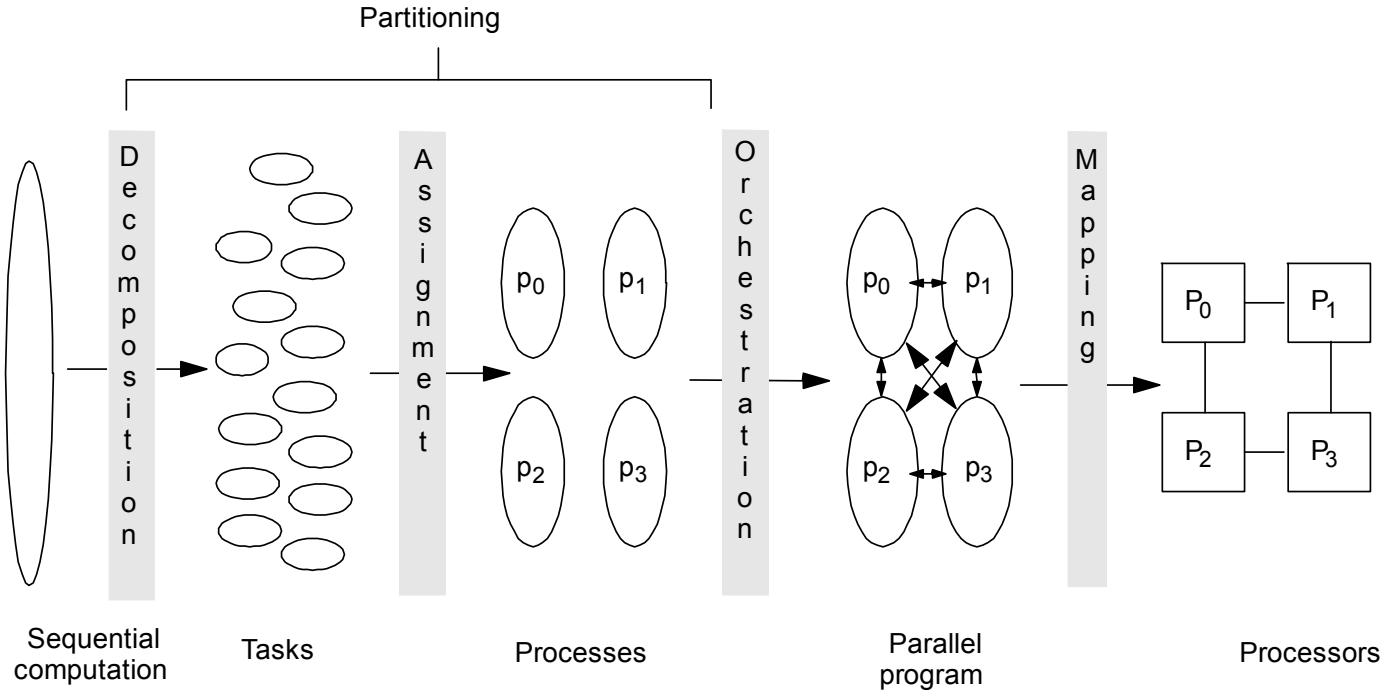
Key Insights

Try to decompose problem as much as possible to reduce serial fraction.

At the same time, overheads caused by such decomposition (communication and synchronization) are also a serial component.

Need to find a good trade-off point!

Steps in Creating a Parallel Program



4 steps: Decomposition, Assignment, Orchestration, Mapping

- Done by programmer or system software (compiler, runtime, ...)
- Issues are the same, so assume programmer does it all explicitly

Some Important Concepts

Task:

- Arbitrary piece of undecomposed work in parallel computation
- Executed sequentially; concurrency is only across tasks
- E.g. computation for each element of the 2-D array, grouping computations of a bunch of rows, etc.
- Fine-grained versus coarse-grained tasks

Process (thread):

- Abstract entity that performs the tasks assigned to processes
- Processes communicate and synchronize to perform their tasks

Processor:

- Physical engine on which process executes
- Processes virtualize machine to programmer
  - first write program in terms of processes, then map to processors

Decomposition

Break up computation into tasks to be divided among processes

- Static vs. Dynamic
- No. of available tasks may vary with time

i.e. identify concurrency and decide level at which to exploit it

Goal: Enough tasks to keep processes busy, but not too many

- No. of tasks available at a time is upper bound on achievable speedup

Assignment

Specifying mechanism to divide work up among processes

- Balance workload, reduce communication and management cost

Structured approaches usually work well

- Code inspection (parallel loops) or understanding of application
- Well-known heuristics
- *Static* versus *dynamic* assignment

As programmers, we worry about partitioning first

- *Usually* independent of architecture or prog model
- But cost and complexity of using primitives may affect decisions

As architects, we assume program does reasonable job of it

Orchestration

- Naming data
- Structuring communication
- Synchronization
- Organizing data structures and scheduling tasks temporally

Goals

- Reduce cost of communication and synch. as seen by processors
- Reserve locality of data reference (incl. data structure organization)
- Schedule tasks to satisfy dependences early
- Reduce overhead of parallelism management

Closest to architecture (and programming model & language)

- Choices depend a lot on comm. abstraction, efficiency of primitives
- Architects should provide appropriate primitives efficiently

Mapping

After orchestration, already have parallel program

Two aspects of mapping:

- Which processes will run on same processor, if necessary
- Which process runs on which particular processor
  - mapping to a network topology

One extreme: *space-sharing*

- Machine divided into subsets, only one app at a time in a subset
- Processes can be pinned to processors, or left to OS

Another extreme: complete resource management control to OS

- OS uses the performance techniques we will discuss later

Real world is between the two

- User specifies desires in some aspects, system may ignore

Usually adopt the view: process <-> processor