Part III: Concepts

Goal: Understand basic concepts and trade-offs of parallelism

Interesting Question

- Question on topic of “no standard parallel model”: Sequential computers were quite different originally, before one machine (IBM 701) gained widespread use. Won’t the widespread use of Intel (or AMD) CMPs have that same effect for parallelism?
O/E - E/O Sort

- The array is assigned to memories

```
| P0 | P1 | P2 | P3 |
```

One Step:
- get end neighbor values: $\lambda$
- O/E half step: $(n/P)c$
- get end neighbor values: $\lambda$
- E/O half step: $(n/P)c$
- And-reduce over done?: $\lambda \log P$

No. Steps: n/2 in worst case

Threads

- A thread consists of program code, a program counter, call stack, and a small amount of thread-specific data

- Threads share access to memory (and the file system) with other threads
- Threads communicate through the shared memory
- Though it may seem odd, apply the CTA model to thread programming -- emphasize locality, expect sharing to cost plenty

Threads are familiar, but don’t use std model
Processes

- A process is a thread in its own private address space
  - Processes do not communicate through shared memory, but need another mechanism like message passing; shared address space another possibility
  - Key issue: How is the problem divided among the processes, which includes data and work
  - Processes (logically subsume) threads

Compare Threads & Processes

- Both have code, PC, call stack, local data
  - Threads -- One address space
  - Processes -- Separate address spaces

- Weight and Agility
  - Threads: lighter weight, faster to setup, tear down, more dynamic
  - Processes: heavier weight, setup and tear down more time consuming, communication is slower

--- Mostly we use ‘thread’ & ‘process’ interchangeably ---
For Monday

- Consider the Red/Blue Simulation: 2D torus array randomly half filled with red, blue cells; unoccupied is white. In 1st half step, red can move right into unoccupied cell; in 2nd half step, blue can move down into unoccupied cell; both happening is OK; terminate if any 10x10 tile is outside [45%,55%]

- Write a parallel program for the Red/Blue problem for a multicore or SMP machine using Pthreads; apply CTA-type analysis, trying to increase locality.

Terminology

- Terms used to refer to a unit of parallel computation include: thread, process, processor, …
  - Technically, thread and process are SW, processor is HW
  - Usually, it doesn’t matter
  - I will (try to) use “thread/process” for logical parallelism, and “processor” when I mean physical parallelism
Parallelism vs Performance

- Naïvely, many people think that applying $P$ processors to a $T$ time computation will result in $T/P$ time performance
- Generally wrong
  - For a few problems (Monte Carlo) it is possible to apply more processors directly to the solution
  - For most problems, using $P$ processors requires a paradigm shift
- Assume “$P$ processors => $T/P$ time” to be the best case possible

Better Intuition

- (Because of the presumed paradigm shift) the sequential and parallel solutions differ so we don’t expect a simple performance relationship between the two
  - More or fewer instructions must be executed

Examples of other differences
- The hardware is different
- Parallel solution has difficult-to-quantify costs that the serial solution does not have, etc.
More Instructions Needed

- To implement parallel computations requires overhead that sequential computations do not need
  - All costs associated with communication are overhead: locks, cache flushes, coherency, message passing protocols, etc.
  - All costs associated with thread/process setup
  - Lost optimizations -- many compiler optimizations not available in parallel setting
    - Instruction reordering

Performance Loss: Overhead

- Threads and processes incur overhead

![Diagram of thread and process setup and tear down]

- Obviously, the cost of creating a thread or process must be recovered through parallel performance:

\[
\frac{(t + o_s + o_{td} + \text{cost}(t))/2}{t} < 1 \\
\therefore \quad o_s + o_{td} + \text{cost}(t) < t
\]

- \(t = \) execution time
- \(o_s = \) setup, \(o_{td} = \) tear down
- \(\text{cost}(t) = \) all other || costs
More Instructions (Continued)

- Redundant execution can avoid communication -- a parallel optimization

**New random number needed for loop iteration:**
(a) Generate one copy, have all threads ref it … requires communication
(b) Communicate seed once, then each thread generates its own random number … removes communication and gets parallelism, but by increasing instruction load

---

A common (and recommended) programming trick

---

Fewer Instructions

- Searches illustrate the possibility of parallelism requiring fewer instructions

- Independently searching subtrees means an item is likely to be found faster than sequential
Parallelism vs Performance

- Sequential hardware ≠ parallel hardware
  - There is more parallel hardware, e.g. memory
  - There is more cache on parallel machines
  - Sequential computer ≠ 1 processor of parallel computer, because of coherence hw, power, etc.
    - Important in multicore context
  - Parallel channels to disk, possibly

These differences tend to favor parallel machine

Superlinear Speed up

- Additional cache is an advantage of parallelism

The effect is to make execution time < \( T/P \) because data (& program) memory reference are faster
- Cache-effects help mitigate other parallel costs
Other Parallel Costs

- Wait: All computations must wait at points, but serial computation waits are well known
- Parallel waiting …
  - For serialization to assure correctness
  - Congestion in communication facilities
    - Bus contention; network congestion; etc.
  - Stalls: data not available/recipient busy
- These costs are generally time-dependent, implying that they are highly variable

Bottom Line …

- Applying $P$ processors to a problem with a time $T$ (serial) solution can be either … better or worse …
  - it’s up to programmers to exploit the advantages and avoid the disadvantages
Amdahl’s Law

- If $1/S$ of a computation is inherently sequential, then the maximum performance improvement is limited to a factor of $S$

\[
T_P = \frac{1}{S} \times T_S + \frac{(1-1/S)}{P} \times T_S
\]

- Amdahl’s Law, like the Law of Supply and Demand, is a fact

---

Gene Amdahl -- IBM Mainframe Architect

Interpreting Amdahl’s Law

- Consider the equation

\[
T_P = \frac{1}{S} \times T_S + \frac{(1-1/S)}{P} \times T_S
\]

- With no charge for || costs, let $P \rightarrow \infty$ then $T_P \rightarrow \frac{1}{S} \times T_S$

The best parallelism can do is to eliminate the parallelizable work; the sequential remains

- Amdahl’s Law applies to problem instances

Parallelism seemingly has little potential
More On Amdahl’s Law

- Amdahl’s Law assumes a fixed problem instance: Fixed $n$, fixed input, perfect speedup
  - The algorithm can change to become more parallel
  - Problem instances grow implying proportion of work that is sequential may reduce
  - … Many, many realities including parallelism in ‘sequential’ execution imply analysis is simplistic

- Amdahl is a fact; it’s not a show-stopper

Digress: Inherently Sequential

- As an artifact of $P$-completeness theory, we have the idea of Inherently Sequential — computations not appreciably improved by parallelism

Circuit Value Problem: Given a circuit $\alpha$ over Boolean inputs, values $b_1, \ldots, b_n$ and designated output value $y$, is the circuit true for $y$?

- Probably not much of a limitation
Two kinds of performance

- **Latency** -- time required before a requested value is available
  - Latency, measured in seconds; called *transmit time* or *execution time* or just *time*
- **Throughput** -- amount of work completed in a given amount of time
  - Throughput, measured in "work/sec", where "work" can be bits, instructions, jobs, etc.; also called *bandwidth* in communication

Both terms apply to computing and communications

Latency

- Reducing latency (execution time) is a principal goal of parallelism
- There is upper limit on reducing latency
  - Speed of light, esp. for bit transmissions
  - In networks, switching time (node latency)
  - (Clock rate) x (issue width), for instructions
  - Diminishing returns (overhead) for problem instances

Hitting the upper limit is rarely a worry
Throughput

- Throughput improvements are often easier to achieve by adding hardware
  - More wires improve bits/second
  - Use processors to run separate jobs
  - Pipelining is a powerful technique to execute more (serial) operations in unit time

Latency Hiding

- Reduce wait times by switching to work on different operation
  - Old idea, dating back to Multics
  - In parallel computing it’s called latency hiding

- Idea most often used to lower $\lambda$ costs
  - Have many threads ready to go …
  - Execute a thread until it makes nonlocal ref
  - Switch to next thread
  - When nonlocal ref is filled, add to ready list

---

Better throughput often hyped as if better latency

---

Tera MTA did this at instruction level
Latency Hiding (Continued)

- Latency hiding requires ...
  - Consistently large supply of threads $\sim \frac{\lambda}{e}$
  - Where $e = \text{average # cycles between nonlocal refs}$
  - Enough network throughput to have many requests in the air at once

- Latency hiding has been claimed to make shared memory feasible in the presence of large $\lambda$.

There are difficulties

Latency Hiding (Continued)

- Challenges to supporting shared memory
  - Threads must be numerous, and the shorter the interval between nonlocal refs, the more
  - Running out of threads stalls the processor
  - Context switching to next thread has overhead
    - Many hardware contexts -- or --
    - Waste time storing and reloading context
  - Tension between latency hiding & caching
    - Shared data must still be protected somehow
  - Other technical issues
Performance Loss: Contention

- Contention -- the action of one processor interferes with another processor’s actions -- is an elusive quantity
  - Lock contention: One processor’s lock stops other processors from referencing; they must wait
  - Bus contention: Bus wires are in use by one processor’s memory reference
  - Network contention: Wires are in use by one packet, blocking other packets
  - Bank contention: Multiple processors try to access different locations on one memory chip simultaneously

  ---Contention is very time dependent, that is, variable---

Performance Loss: Load Imbalance

- Load imbalance, work not evenly assigned to the processors, underutilizes parallelism
  - The assignment of work, not data, is key
  - Static assignments, being rigid, are more prone to imbalance
  - Because dynamic assignment carries overhead, the quantum of work must be large enough to amortize the overhead
  - With flexible allocations, load balance can be solved late in the design programming cycle
The Best Parallel Programs …

- Performance is maximized if processors execute continuously on local data without interacting with other processors
  - To unify the ways in which processors could interact, we adopt the concept of dependence
  - A dependence is an ordering relationship between two computations
    - Dependencies are usually induced by read/write
    - Dependencies that cross process boundaries induce a need to synchronize the threads
      - Dependencies are well-studied in compilers

Dependences

- Dependencies are orderings that must be maintained to guarantee correctness
  - Flow-dependence: read after write  True
  - Anti-dependence: write after read  False
  - Output-dependence: write after write  False

- True dependences affect correctness
- False dependences arise from memory reuse
Example of Dependences

- Both true and false dependences

1. \( \text{sum} = a + 1; \)
2. \( \text{first\_term} = \text{sum} \times \text{scale1}; \)
3. \( \text{sum} = b + 1; \)
4. \( \text{second\_term} = \text{sum} \times \text{scale2}; \)

- Flow-dependence read after write; must be preserved for correctness
- Anti-dependence write after read; can be eliminated with additional memory
Removing Anti-dependence

- Change variable names

```plaintext
1. sum = a + 1;
2. first_term = sum * scale1;
3. sum = b + 1;
4. second_term = sum * scale2;
```

Granularity

- Granularity is used in many contexts...here *granularity* is the amount of work between cross-processor dependences
  - Important because interactions usually cost
  - Generally, larger grain is better
    - fewer interactions, more local work
    - can lead to load imbalance
  - Batching is an effective way to increase grain
Locality

- The CTA motivates us to maximize locality
  - Caching is the traditional way to exploit locality … but it doesn’t translate directly to jjism
  - Redesigning algorithms for parallel execution often means repartitioning to increase locality
  - Locality often requires redundant storage and redundant computation, but in limited quantities they help

Measuring Performance

- Execution time … what’s time?
  - ‘Wall clock’ time
  - Processor execution time
  - System time
- Paging and caching can affect time
  - Cold start vs warm start
- Conflicts w/ other users/system components
- Measure kernel or whole program
FLOPS

- Floating Point Operations Per Second is a common measurement for scientific pgms
  - Even scientific computations use many ints
  - Results can often be influenced by small, low-level tweaks having little generality: mult/add
  - Translates poorly across machines because it is hardware dependent
  - Limited application

Speedup and Efficiency

- Speedup is the factor of improvement for $P$ processors: $\frac{T_S}{T_P}$

![Graph showing Speedup and Efficiency](image)
Issues with Speedup, Efficiency

- Speedup is best applied when hardware is constant, or for family within a generation
  - Need to have computation, communication in same ratio
  - Great sensitivity to the $T_S$ value
    - $T_S$ should be time of best sequential program on 1 processor of ||-machine
    - $T_{P=1} \neq T_S$ Measures relative speedup

Scaled v. Fixed Speedup

- As $P$ increases, the amount of work per processor diminishes, often below the amt needed to amortize costs
- Speedup curves bend dn
- Scaled speedup keeps the work per processor constant, allowing other affects to be seen
- Both are important

If not stated, speedup is fixed speedup
“Cooking” The Speedup Numbers

- The sequential computation should not be charged for any || costs ... consider

- If referencing memory in other processors takes time (\(\lambda\)) and data is distributed, then one processor solving the problem results in greater \(t\) compared to true sequential

This complicates methodology for large problems

What If Problem Doesn’t Fit?

- Cases arise when sequential doesn't fit in 1 processor of parallel machine
- Best solution is relative speed-up
  - Measure \(T_p=\text{smallest possible}\)
  - Measure \(T_P\), compute \(T_p/T_P\) as having \(P/p\) potential improvement
We Will Return …

- Many issues regarding parallelism have been introduced, but they require further discussion … we will return to them when they are relevant.

Summary of Key Points

- Amdahl’s Law is a fact but it doesn’t impede us much.
- Inherently sequential problems (probably) exist, but they don’t impede us either.
- Latency hiding could hide the impact of $\lambda$ with sufficiently many threads and much (interconnection) bandwidth.
- Impediments to parallel speedup are numerous: overhead, contention, inherently sequential code, waiting time, etc.
Review Key Points (continued)

- Concerns while parallel programming are also numerous: locality, granularity, dependences (both true and false), load balance, etc.
- Happily: Parallel and sequential computers are different: More hardware means more fast memory (cache, RAM), implying the possibility of superlinear speedup
- Measuring improvement is complicated