Lecture 3
Parallelism, Concurrency, and Performance

Christopher Mitchell, Ph.D.
cmitcHELL@cs.nyu.edu || http://z80.me
Prelude: A Ride to the Train

- Algorithm design: get everyone into the car
- Placement constraints?
- Ordering constraints?
Constraints and Dependencies

- **Placement**
  - Luggage can’t go in the driver’s seat
  - The driver’s seat must have a licensed driver
  - Humans cannot go in the trunk
  - Only one human per seat (seatbelts!)

- **Partial** Ordering
  - A non-baby must place the baby in its seat before boarding
  - A non-baby must stow the luggage before boarding
Parallelism Opportunities

• Breaking down the problem
  • Dependencies
    • Effect on what can be parallelized
  • Invariants
  • Some operations are truly atomic

• Don’t start coding:
  • If you don’t know the constraints and the algorithm design
  • If you don’t know the hardware/OS/library resources available

• Theme of next 2 classes:
  Match the algorithm and the underlying resources.
Lecture 2 Outline

➢ Lab 0 Questions?
➢ Parallelism and Concurrency
➢ Amdahl’s Law
➢ Defining Parallelism
➢ GDB Overview/Review
Starting from Serial Programs

• What’s a program? From last time:

\[ n \quad , \quad , \quad 3 \quad 2 \quad 1 \]

Sequence of instructions

• How can we make this faster?
Step Back: Time

• “Absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external.” (I. Newton, 1689)

• “Time is, like, Nature’s way of making sure that everything doesn’t happen all at once.” (Anonymous, circa 1968)

• No global wallclock
Instruction

• We made assumptions:
  • Instantaneous (atomic): our instructions are discrete units
  • Isolated (nothing running at the same time): only one instruction runs at a time
Program

- A sequence of instructions is a series of events in time.
- Each instruction finishes before the next one begins (except of course, that’s not true).
Concurrency

- Programs run (roughly) simultaneously
Concurrency: Interleavings

• Programs run (roughly) simultaneously
Formalizing Concurrency

- Intervals
- Precedence
- Partial Orders
- Total Orders
- Serializability and Linearizability
Intervals

- Can’t think of operations as instantaneous any more
- These *happen* to be **disjoint**
Intervals

• Can be **overlapping** or **disjoint**

---

time

\[ t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \]
Precedence

• C happens before D (D precedes C)
• A is concurrent with B
• Formal definition: C precedes D iff end of C happens before start of D.
• Notation: C → D
Precedence Rules

- Always false: $A \rightarrow A$
- If $C \rightarrow D$, impossible for $D \rightarrow C$
- If $A \rightarrow C$ and $C \rightarrow D$, then $A \rightarrow D$
- With overlaps, $A \rightarrow B$ and $B \rightarrow A$ can both be false
Ordering: Partial Orders

- Partial orders can be used to build total orders
- Properties:
  - Irreflexive: $A \rightarrow A$ is always false
  - Antisymmetric: If $A \rightarrow B$, then never $B \rightarrow A$
  - Transitive: If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$
Ordering: Total Orders

• Add concurrent (overlapping) operations into order

• Redefine $A \rightarrow B$ to be true if **start** of $A$ precedes start of $B$

• New Properties:
  • **Irreflexive**: $A \rightarrow A$ is always false
  • **Antisymmetric**: If $A \rightarrow B$, then never $B \rightarrow A$
  • **Transitive**: If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$
  • **Ordering**: Exactly one of $A \rightarrow B$ or $B \rightarrow A$ must be true

• Problem with total orders: truly parallel operations
Linearizability

• **Local** property of a single data structure or object

• Operations (potentially parallel operations) can be put into a total order, with results identical to the result if the operations are run one-by-one

• Operations appear **atomic**: no operation can see an intermediate state of another operation
  * Apparent end of A must precede apparent start of B
Serializability

• **Global** property of operations on many pieces of data

• There is *some* serial order of all operations from one or more concurrent or parallel sources (threads, processes, programs) that produces the same results as every operation happening one at a time

• Doesn’t say *what* that order is
  • Weaker than linearizability
  • Does not require end of A to precede start of B

• Database world...
Formal to Practical: Concurrency vs. Parallelism

- **Concurrency**: At least two tasks are making progress in the *same time frame*.
  - Appear overlapping from timeline; may not be parallel
  - Include techniques like time-slicing
  - Can be implemented on a single processing unit
  - More general than parallelism

- **Parallelism**: At least two tasks execute literally at the *same time*.
  - Requires hardware with multiple processing units
Example: Request Processing

2 time units of work

Request 2

6 time units of work

Request 1

Server

Now: t = 0

How will this server react: (1) if it is serial, (2) if it is concurrent but not parallel, and (3) if it is parallel?

[Hint: look at total completion time and average completion time]
Example: Request Processing

Serial:
• Average completion time (be careful)?
• Total completion time?
• Resource utilization?
Example: Request Processing

Concurrent (Without Parallelism):
- Average completion time (be careful)?
- Total completion time?
- Resource utilization?
Example: Request Processing

Parallel:
- Average completion time?
- Total completion time?
- Resource utilization?
Parallel and Concurrent Programs

All Programs

Concurrent Programs

Parallel Programs
Simply Speaking...

Concurrency + Parallelism

= Performance

(but how much?)
Questions!

• If we have as much hardware as we want, do we get as much parallelism as we wish?
• If we have 2 cores, do we get 2x speedup?
  • Think back to the “resource utilization” question.
Quantifying Parallelism
Amdahl’s Law

- How much of a speedup one could get for a given parallelized task?
- Amdahl’s Law (1967):

If $F$ is the fraction of a calculation that is sequential then the maximum speed-up that can be achieved by using $P$ processors is $\frac{1}{F + \frac{1-F}{P}}$
Amdahl’s Law

\[ \text{Speedup} = \frac{1}{(F + (1-F)/P)} \]

Sequential portion = 40% → \( F = 0.4 \)
Execution time = 1.0 units

Time = 0.4 + (0.6 / 2) = 0.7
Speedup = \( \frac{1}{0.4 + (0.6 / 2)} \) = 1.43

Time = 0.4 + (0.6 / 3) = 0.6
Speedup = \( \frac{1}{0.4 + (0.6 / 3)} \) = 1.67

Time = 0.4 + (0.6 / 4) = 0.55
Speedup = \( \frac{1}{0.4 + (0.6 / 4)} \) = 1.82
Amdahl’s Law

Speedup = \frac{1}{(F+(1-F)/P)}

- **2 CPUs**
  - Time = 0.4 + \frac{0.6}{2} = 0.7
  - Speedup = \frac{1}{0.4 + \frac{0.6}{2}} = 1.43

- **8 CPUs**
  - Time = 0.4 + \frac{0.6}{8} = 0.475
  - Speedup = \frac{1}{0.4 + \frac{0.6}{8}} = 2.11

- **∞ CPUs**
  - Time = 0.4 + \frac{0.6}{∞} = 0.4
  - Speedup = \frac{1}{0.4 + \frac{0.6}{∞}} = 2.5
Amdahl’s Law

\[ \text{Speedup}(P) = \frac{\text{Time}(1)}{\text{Time}(P)} \]
What Was Amdahl Saying?

1. Don’t invest blindly on large number of processors.
2. Having faster cores (or processor at his time) makes more sense than having many cores.

Was he right?

• In 1967, many programs had long sequential parts.
• This is not necessarily the case nowadays.
• It is not very easy to find F (the sequential portion)
So …

• Decreasing the serialized portion is of greater importance than adding more cores.

• Only when a program is mostly parallelized, does adding more processors help more than parallelizing the remainder.

• Gustafson’s law: computations involving arbitrarily large data sets can be efficiently parallelized.
So ...

- Both Amdahl and Gustafson do not take into account:
  - The overhead of synchronization, communication, OS, etc.
  - Load may not be balanced among cores
- So you have to use these laws as guideline and theoretical bounds only.
Most real programs have complex, interleaved parallel and serial sections.

Visualization tool: DAG
- Possible schedule(s)
- Node = unit of work
- Arrow = dependency
DAG Model for Multithreading

**Work**: total amount of time spent on all instructions

\[ T_p = \text{The fastest possible execution time on } P \text{ processors} \]

**Work Law**: \[ T_p \geq T_{1/P} \]
DAG Model for Multithreading

**Span:** The longest path of dependence in the DAG = $T_\infty$

**Span Law:** $T_p \geq T_\infty$
Can We Define Parallelism Now?

How about? $\frac{T_1}{T_\infty}$

Ratio of work to span
Can We Define Parallelism Now?

**Work:** \( T_1 = 50 \)

**Span:** \( T_\infty = 8 \)

**Parallelism:** \( T_1 / T_\infty = 6.25 \)
Reasoning about Parallelism

• At what level can we reason about parallelism?
  • Algorithm?
  • High-level language (eg, C++)?
  • Assembly?
  • Individual instructions?
Is Thread The Only Parallelism Granularity?

- **Instruction level parallelism (ILP)**
  - Superscalar
  - Out-of-order execution
  - Speculative execution

- **Thread level parallelism**
  - Hyperthreading technology (aka SMT)
  - Multicore

- **Process level parallelism**
  - Multiprocessor system
  - Hyperthreading technology (aka SMT)
  - Multicore
GDB Overview/Review
GDB

- GNU Debugger
- Useful for debugging your C++ and C programs

Prerequisites:
- GDB installed (duh)
- Compile with -g option
- Run the program using GDB (or attach to it)
Using GDB

1. Compile program with -g flag
2. Launch the program with gdb
   a) `gdb --args ./program arg1 arg2 arg3`
   b) Or: `gdb program, then run arg1 arg2 arg3`
3. Optional: set breakpoint
   a) `break functionname`
   b) Or: `break file.cc:line`
4. Run the program, wait for breakpoint or crash
5. Use `x`, `print`, `step`, `disas`, `info` to probe state
GDB Overview/Review

- `gdb --args ./binary argument argument`
- `run` - Start the program
- `break` - Set breakpoint
- `continue` - Pick up where execution stopped
- `print` - Print variable contents
- `x` - Inspect memory
- `list` - Display lines of code
- `step` - Execute the next line of code
- `info` - Get list of breakpoints, sources, etc
- `help/apropos` - Get help with commands

Conclusions

• Prelude: understand resources (and effects) of hardware and software before writing your application
• Formally defining ordering, linearizability, serializability
• Concurrency and parallelism are not the same thing.
• Amdahl’s law: intuit where effort should be spent
• Visualizing scheduling: DAGs
• GDB