Lecture 2
Parallelism, Concurrency, and Performance

Christopher Mitchell, Ph.D.
cmitchell@cs.nyu.edu || http://z80.me
Lecture 2 Outline

➢ GDB Overview/Review
➢ Parallelism and Concurrency
➢ Parallel Programming Models and Performance
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

Parallelism and Concurrency
Same Meaning?

- **Concurrency**: At least two tasks are making progress in the **same time frame**.
  - Not necessarily at the same time
  - Include techniques like time-slicing
  - Can be implemented on a single processing unit
  - Concept more general than parallelism

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

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.
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

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 $\frac{1}{F + \frac{(1-F)}{P}}$.

Sequential portion = 40% $\rightarrow$ $F = 0.4$
Execution time = 1.0 units
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.
DAG Model for Multithreading

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

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

**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:** \( \frac{T_1}{T_\infty} = 6.25 \)
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
That Was The Software
How about the Hardware?

Latency Vs Throughput

Symmetric Multiprocessing:
Multiple CPUS

Chip-Level Multiprocessing:
Multiple Cores

Granularity (instructions)

Latency Vs Throughput
Flynn Classification

- A taxonomy of computer architecture
- Proposed by Michael Flynn in 1966
- Classifies by:
  - Instruction parallelism
  - Data parallelism

<table>
<thead>
<tr>
<th></th>
<th>Single instruction</th>
<th>Multiple instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>
Single Data Architectures

PU = Processing Unit
Multiple Data Architectures

PU = Processing Unit
MIMD Memory Models: Shared Memory
MIMD Memory Models: Distributed Memory
MIMD Memory Models: Hybrid
Multicore and Manycore

“We have arrived at many-core solutions not because of the success of our parallel software but because of our failure to keep increasing CPU frequency.”

-Tim Mattson
Parallel Computing @ Intel

• Dilemma:
  • Parallel hardware is ubiquitous
  • Parallel software is not!

• After more than 25 years of research, we are not closer to solving the parallel programming model!
| ABCPL | ACE | ACT++ | Active messages | Adl | Adlsmith | ADDAP | AFAPI | ALWAN | AM | AMDC | AppLeS | Amoeba | ARTS | Athapascan-0b | Aurora | Automap | bb_threads | Blaze | BSP | BlockComm | C* | "C* in C" | C++ | CarlOS | Cashmere | C4 | CC++ | Chu | Charlotte | Charm | Charm++ | Cid | Cilk | CM-Fortran | Converse | Code | COOL | CORRELATE | CPS | CRL | CSP | Cthreads | CUMULVS | DAGGER | DAPPLE | Data Parallel C | DC++ | DCE++ | DDD | DICE | DIPC | DOLIB | DOME | DOSMOS. | DRL | DSM-Threads | Ease | ECO | Eiffel | Eilean | Emerald | EPL | Excalibur | Express | Falcon | Filaments | FM | FLASH | The FORCE | Fork | Fortran-M | FX | GA | GAMMA | Glenda | GLU | GUARd | HASL | Haskell | HPC++ | JAVA$\mathrm{R}$ | HURUS | HPC | IMPACT | ISiS | JAVA$\mathrm{R}$ | JADE | Java RMI | javaPG | JavaSpace | JIDL | Joyce | Khoros | Karma | KOAN/ForTran-S | LAM | Lilac | Linda | JADA | WWWinda | JSETL-Linda | ParLin | Eilean | P4-Linda | POSTYL | Objective-Linda | LIPS | Locust | Lpars | Lucid | Maisie | Manifold | Mentat | Legion | Meta Chaos | Midway | Millipede | CparPar | Mirage | Mpc | MOSIX | Modula-P | Modula-2* | Multipol | MPI | MPC++ | Mumin | Nano-Threads | NESL | NetClasses++ | Nexus | Nimbled | NOW | Objective Linda | Occam | Omega | OpenMP | Orca | OOF90 | P++ | P3L | Pablo | PADE | PADE | Penda | Papers | AFAPI | Para++ | Paradigm | Parafrase2 | Parallel | Parallel-C++ | Parallelaxis | ParC | ParLib++ | ParLin | Parmacs | Parti | pC | PCN | PCP | PH | PEACE | PCU | PET | PENNY | Phosphorus | POET | Polaris | POOMA | POOL-T | PRESTO | P-RIO | Prospero | Proteus | QPC++ | PVM | PSI | PSDM | Quake | Quark | Quick Threads | Sage++ | SCANDAL | SAM | pC++ | SCHEDULE | SciTL | SDDA | SHMEM | SIMPLE | Sina | SISAL | distributed smalltalk | SMI | SONiC | Split-C | SR | Stthreads | Strand | SUIF | Synergy | Telephos | SuperPascal | TCGMSG | Threads.h++ | TreadMarks | TRAPPER | uC++ | UNITY | UC | V | ViC* | Visifold V-NUS | VPE | Win32 threads | WinPar | XENOOPS | XPC | Zounds | ZPL |
The Mentality of Yet Another Programming Language

<table>
<thead>
<tr>
<th>ABCPL</th>
<th>CORRELATE</th>
<th>GLU</th>
<th>GUARD</th>
<th>Mentat</th>
<th>Parafrase2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>CPS</td>
<td>GUARD</td>
<td>HA-SL</td>
<td>Legion</td>
<td>pC++</td>
</tr>
<tr>
<td>ACT++</td>
<td>CRL</td>
<td>Haskell</td>
<td>HPC++</td>
<td>Meta Chaos</td>
<td>PARALLEL-C++</td>
</tr>
<tr>
<td>Active messages</td>
<td>CSP</td>
<td>HPC++</td>
<td>Millipede</td>
<td>ParC</td>
<td></td>
</tr>
<tr>
<td>Adh</td>
<td>Cthreads</td>
<td>JAVA</td>
<td>CparPar</td>
<td>ParLib++</td>
<td></td>
</tr>
<tr>
<td>Admuth</td>
<td>CUMULVS</td>
<td>HORUS</td>
<td>Mirage</td>
<td>ParLib</td>
<td></td>
</tr>
<tr>
<td>ADDAP</td>
<td>DAGGER</td>
<td>HPC</td>
<td>MpC</td>
<td>Params</td>
<td></td>
</tr>
<tr>
<td>AFAPI</td>
<td>DAPPLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Blaze</th>
<th>DSM-Threads</th>
<th>KOAN/ForTran-S</th>
<th>LAM</th>
<th>NOW</th>
<th>Nimrod</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSP</td>
<td>Ease</td>
<td>LilaC</td>
<td>Linda</td>
<td>JADA</td>
<td>Objectiive Linda</td>
</tr>
<tr>
<td>BlockComm</td>
<td>ECO</td>
<td>WWWinda</td>
<td>LAM</td>
<td>LAM</td>
<td>Occam</td>
</tr>
<tr>
<td>C++</td>
<td>Eiffel</td>
<td>LAM</td>
<td>LAM</td>
<td>Omega</td>
<td>OpenMP</td>
</tr>
<tr>
<td>&quot;C* in C</td>
<td>Eilean</td>
<td>LAM</td>
<td>LAM</td>
<td>Orca</td>
<td>OOF90</td>
</tr>
<tr>
<td>C++</td>
<td>Emerald</td>
<td>LAM</td>
<td>LAM</td>
<td>Parlin</td>
<td>++</td>
</tr>
<tr>
<td>CarIoS</td>
<td>EPL</td>
<td>LAM</td>
<td>LAM</td>
<td>P4-Linda</td>
<td>3L</td>
</tr>
<tr>
<td>Cashmere</td>
<td>Excalibur</td>
<td>LAM</td>
<td>LAM</td>
<td>POSTBL</td>
<td>Pablo</td>
</tr>
<tr>
<td>C4</td>
<td>Express</td>
<td>LAM</td>
<td>LAM</td>
<td>Objective-Linda</td>
<td>PADE</td>
</tr>
<tr>
<td>CC++</td>
<td>Falcon</td>
<td>LAM</td>
<td>LAM</td>
<td>LiPS</td>
<td>PADE</td>
</tr>
<tr>
<td>Chu</td>
<td>Filaments</td>
<td>LAM</td>
<td>LAM</td>
<td>Locust</td>
<td>PADRE</td>
</tr>
<tr>
<td>Charlotte</td>
<td>FM</td>
<td>LAM</td>
<td>LAM</td>
<td>Lux</td>
<td>Panda</td>
</tr>
<tr>
<td>Charm</td>
<td>FLASH</td>
<td>LAM</td>
<td>LAM</td>
<td>Maisie</td>
<td>Papers</td>
</tr>
<tr>
<td>Charm++</td>
<td>The FORCE</td>
<td>LAM</td>
<td>LAM</td>
<td>Manifold</td>
<td>AFAPI</td>
</tr>
<tr>
<td>Cid</td>
<td>Fork</td>
<td>LAM</td>
<td>LAM</td>
<td>Paradigm</td>
<td>Para++</td>
</tr>
<tr>
<td>Cilk</td>
<td>Fortran-M</td>
<td>LAM</td>
<td>LAM</td>
<td>Paradigm</td>
<td>Paradigm</td>
</tr>
<tr>
<td>CM-ForTran</td>
<td>FX</td>
<td>LAM</td>
<td>LAM</td>
<td>Paradigm</td>
<td>Paradigm</td>
</tr>
<tr>
<td>Converse</td>
<td>GA</td>
<td>LAM</td>
<td>LAM</td>
<td>Paradigm</td>
<td>Paradigm</td>
</tr>
<tr>
<td>Code</td>
<td>GAMMA</td>
<td>LAM</td>
<td>LAM</td>
<td>Paradigm</td>
<td>Paradigm</td>
</tr>
<tr>
<td>COOL</td>
<td>Glenda</td>
<td>LAM</td>
<td>LAM</td>
<td>Paradigm</td>
<td>Paradigm</td>
</tr>
</tbody>
</table>

We don’t want to scare away the programmers … Only add a new API/language if we can’t get the job done by fixing an existing approach.
Programming Models and Performance
Parallel Programming Models

- Parallel Programming Models
  - Control Flow
    - Message-Passing
  - Data Parallel
    - Locking
    - Message Passing

MIMD

SPMD or SIMD (GPUs!)
Programming Model

- **Definition:** the languages and libraries that create an abstract view of the machine

- **Control**
  - How is parallelism created?
  - How are dependencies enforced?

- **Data**
  - Shared or private?
  - How is shared data accessed or private data communicated?

- **Synchronization**
  - What operations can be used to coordinate parallelism
  - What are the atomic (indivisible) operations?
Any Paradigm on Any Hardware

- You can run any paradigm on any hardware (e.g. an MPI on shared-memory)
- The hardware itself can be heterogeneous

The whole challenge of parallel programming is to make the best use of the underlying hardware to exploit the different types of parallelism
Example

We have a matrix $A$. We need to form another matrix $A_{sqr}$ that contains the square of each element of $A$. Then we need to calculate $S$, which is the sum of the elements in $A_{sqr}$.

Thanks to Katherine Yelick.
We have a matrix $A$. We need to form another matrix $A_{sqr}$ that contains the square of each element of $A$. Then we need to calculate $S$, which is the sum of the elements in $A_{sqr}$.

- How can we parallelize this?
- How long will it take if we have unlimited number of processors?
Example

• First, decompose your problem into a set of tasks
  • There are many ways of doing it.
  • Tasks can be of the same, different, or undetermined sizes.

• Draw a task-dependency graph (do you remember the DAG we saw earlier?)
  • A directed graph with nodes corresponding to tasks
  • Edges indicating dependencies, that the result of one task is required for processing the next.
Example

A:
sqr (A[0])

A_{sqr}:
sqr(A[1])
sqr(A[2])

\ldots

sqr(A[n])

\text{sum}

\text{square}
Writing a Parallel Program

Decomposition

Original Problem

Units of execution + new shared data for extracted dependencies

Algorithm strategy

Tasks, shared and local data

Implementation & building blocks

Corresponding source code

Source: “Many Core Processors ... Opportunities and Challenges” by Tim Mattson
• Does your knowledge of the underlying hardware change your task dependency graph? If yes, how?

• Suppose you have several candidate algorithms for solving a problem, how do you pick?
Wish List for a Good Algorithm

1. Good performance
2. On a wide range of parallel machines
3. Minimal tuning to hardware in early stage

We need an analytical model that can predict the performance of our algorithm on a wide range of machines and must strike a balance between detail and simplicity.
Three Main Computational Models

PRAM

LogP

BSP
PRAM Model

- Parallel Random Access Machine
- Shared memory
- A synchronous MIMD
PRAM Model

- Can emulate a message-passing machine by partitioning memory into private memories.
- No communication cost (i.e. infinite bandwidth and zero latency).
- Infinite memory
- Different protocols can be used for reading and writing shared memory.
  - EREW - exclusive read, exclusive write: A program isn’t allowed to have two processors access the same memory location at the same time.
  - CREW - concurrent read, exclusive write
  - CRCW - concurrent read, concurrent write: Needs protocol for arbitrating write conflicts
  - CROW - concurrent read, owner write: Each memory location has an official "owner"
Pros/Cons of PRAM

+ Simple to use
- Unrealistic → performance prediction is inaccurate
LogP Model

- Distributed memory
- No specification of interconnection network
- Based on:
  - Latency of communication
  - Overhead in processing transmitted/received messages
  - Gap between consecutive transmissions (i.e., bandwidth limitation)
  - Processing power
LogP Model

Limited Volume ($L/g$) to or from a proc

Overhead ($o$)

Latency ($L$)

Interconnection Network

Processors ($P$)

Gaps ($g$)
Using the LogP Model

- One processor sends $n$ words to another processor

\[2o + L + g(n-1)\]
Pros/Cons of the LogP Model

+ Simple, 4 parameters
+ Can easily be used to guide the algorithm development
- Does not take contention into account → can sometimes underestimate communication time.

There are many variations to the LogP model, making it more accurate but more complex (e.g. LogGP, logGPC, pLogP, ...)

BSP Model

- Bulk Synchronous Parallel
- A BSP computer consists of
  - A set of processor-memory pairs
  - A communication network that delivers messages in a point-to-point manner
  - Mechanism for barrier synchronization for all or a subset of the processes
- BSP programs composed of supersteps
- Each superstep consists of three ordered stages:
BSP Model
BSP Model

• **Variables**
  • \textbf{p}: number of processors
  • \textbf{s}: processor computation speed (flops/s)
  • \textbf{h}: the maximum number of incoming or outgoing messages per processor
  • \textbf{g}: the cost of sending a message.
  • \textbf{l}: time to do a barrier synchronization

• Assume \( w_i \) is the computation time for work on processor \( p \) during a superstep.

• Cost of a superstep: \[ ? \]
BSP Model

- **Variables**
  - \( p \): number of processors
  - \( s \): processor computation speed (flops/s)
  - \( h \): the maximum number of incoming or outgoing messages per processor
  - \( g \): the cost of sending a message.
  - \( l \): time to do a barrier synchronization

- Assume \( w_i \) is the computation time for work on processor \( p \) during a superstep.

- Cost of a superstep: \( \max_{i=1}^{p} (w_i) + \max_{i=1}^{p} (h_i g) + l \)
Pros/Cons of BSP Model

+ Simple
+ Predictable performance
- Not very good if locality is important
- BSP does not distinguish between sending 1 message of length m, or m messages of length 1.
Be Careful!

- All these models are just approximations.
- They do not model memory which can greatly affect their predictions.
  - There are memory models though.
- An implementation of a good parallel algorithm on a specific machine will surely require tuning. But first, pick/design an algorithm based on one of the models discussed.
Conclusions

- Concurrency and parallelism are not exactly the same thing.
- There is parallelism at different granularities, with methods to exploit each parallelism granularity.
- You need to know the difference between: threads/processors/tasks.
- Knowing the hardware will help you generating a better task dependency graph.

- **Homework 1**