Hello!

Prof. Kayvon

Prof. Olukotun

James

Keshav

Raj

Arden

Drew

Pranil
One common definition

A parallel computer is a collection of processing elements that cooperate to solve problems quickly.

We care about performance and we care about efficiency.

We're going to use multiple processors to get it.
DEMO 1

(CS149 Fall 2022’s first parallel program)
Speedup

One major motivation of using parallel processing: achieve a speedup

For a given problem:

\[
\text{speedup (using } P \text{ processors)} = \frac{\text{execution time (using 1 processor)}}{\text{execution time (using } P \text{ processors)}}
\]
Class observations from demo 1

- Communication limited the maximum speedup achieved
  - In the demo, the communication was telling each other the partial sums

- Minimizing the cost of communication improved speedup
  - Moved students ("processors") closer together (or let them shout)
DEMO 2

(scaling up to four “processors”)

Stanford CS149, Fall 2022
Class observations from demo 2

- Imbalance in work assignment limited speedup
  - Some students ("processors") ran out work to do (went idle), while others were still working on their assigned task

- Improving the distribution of work improved speedup
DEMO 3

(massively parallel execution)
Class observations from demo 3

- The problem I just gave you has a significant amount of communication compared to computation

- Communication costs can dominate a parallel computation, severely limiting speedup
Course theme 1: Designing and writing parallel programs ... that scale!

- Parallel thinking
  1. Decomposing work into pieces that can safely be performed in parallel
  2. Assigning work to processors
  3. Managing communication/synchronization between the processors so that it does not limit speedup

- Abstractions/mechanisms for performing the above tasks
  - Writing code in popular parallel programming languages
Course theme 2:
Parallel computer hardware implementation: how parallel computers work

- Mechanisms used to implement abstractions efficiently
  - Performance characteristics of implementations
  - Design trade-offs: performance vs. convenience vs. cost

- Why do I need to know about hardware?
  - Because the characteristics of the machine really matter
    (recall speed of communication issues in earlier demos)
  - Because you care about efficiency and performance
    (you are writing parallel programs after all!)
Course theme 3: Thinking about efficiency

- FAST != EFFICIENT

- Just because your program runs faster on a parallel computer, it does not mean it is using the hardware efficiently
  - Is 2x speedup on computer with 10 processors a good result?

- Programmer’s perspective: make use of provided machine capabilities

- HW designer’s perspective: choosing the right capabilities to put in system (performance/cost, cost = silicon area?, power?, etc.)
Course logistics
Getting started

- **The course web site**
  - https://cs149.stanford.edu

- **Textbook**
  - There is no course textbook (the internet is plenty good these days), also see the course web site for suggested references
Four programming assignments

Assignment 1: ISPC programming on multi-core CPUs
Assignment 2: scheduler for a task graph
Assignment 3: Writing a renderer in CUDA on NVIDIA GPUs
Assignment 4: parallel large graph algorithms on a multi-core CPU
Optional assignment 5: (will boost some prior grade)

Programming assignments can (optionally) be done with a partner.

We realize finding a partner can be hard and/or stressful. 😞 😞

Fill out our partner request form by Friday noon and we will find you a partner! 😊 😊

Plus a few optional extra credit challenges… ;-)
Written assignments

- Every two-weeks we will have a take-home written assignment

- Written assignments contain modified versions of previous exam questions, so they:
  - Give you practice with key course concepts
  - Provide practice for the style of questions you will see on an exam
Commenting and contributing to lectures

Why Parallelism? Why Efficiency?

Instruction level parallelism (ILP)

- Processors did in fact leverage parallel execution to make programs run faster, it was just invisible to the programmer.

- Instruction level parallelism (ILP)
  - Idea: Instructions must appear to be executed in program order. BUT independent instructions can be executed simultaneously by a processor without impacting program correctness.
  - Superscalar execution: processor dynamically finds independent instructions in an instruction sequence and executes them in parallel.

It is computationally expensive for the processor to determine dependencies between instructions. The following PPT (slides 9/10) provides an example of how the number of checks grows with the number of instructions that are simultaneously dispatched:

This additional cost is likely one of the predominant reasons that ILP has plateaued at 4 simultaneous instructions. To circumvent this issue, architects have tried to force the compiler to solve the dependency issue using VLIW (very long instruction word). To summarize VLIW, if a processor contains 5 independent execution units, the compiler will have 5 operations in the "very long instruction word" that the processor will map to the 5 execution units:
https://en.wikipedia.org/wiki/Very_long_instruction_word. This way dependency checking is the responsibility of software and not hardware.

I am not sure if VLIW has helped significantly pushed the four simultaneous instruction threshold through. If somebody knows, please share.

Question: The key phrase on this slide is that a processor must execute instructions in a manner "appears" as if they were executed in program order. This is a key idea in this class.

What is program order?

And what does it mean for the results of a program's execution to appear as if instructions were executed in program order?

And finally... Why is the program order guarantee a useful one? (What if the results of execution were inconsistent with the results that would be obtained if the instructions were executed in program order?)

A programmer might write something like the code below.

```c
void

x = a * b
print(x);

y = c * d
print(y);
```
Participation (comments)

- You are asked to submit one well-thought-out comment per lecture
  - Only two comments per week
  - We expect you to submit “in the same week” as the lectures (no you cannot submit 18 comments at the end of the quarter when you are studying for the final)

- Why do we write?
  - Because writing is a way many good architects and systems designers force themselves to think (explaining clearly and thinking clearly are highly correlated!)

- But take it seriously, there is a participation component to the final grade
What we are looking for in comments

- Try to explain the slide (as if you were trying to teach your classmate while studying for an exam)
  - “The instructor said this, but if you think about it this way instead it makes much more sense...”

- Explain what is confusing to you:
  - “What I’m totally confused by here was...”

- Challenge classmates with a question
  - For example, make up a question you think might be on an exam.

- Provide a link to an alternate explanation
  - “This site has a really good description of how multi-threading works...”

- Mention real-world examples
  - For example, describe all the parallel hardware components in the PS5

- Constructively respond to another student’s comment or question
  - “@segfault21, are you sure that is correct? I thought that Prof. Kayvon said...”

- It is OKAY (and even encouraged) to address the same topic (or repeat someone else’s summary, explanation or idea) in your own words
  - “@funkysenior21’s point is that the overhead of communication...”
Grades

56%  Programming assignments (4)
10%  Written assignments (5)
16%  Midterm exam
    -  It will be an evening exam on Nov 15th
16%  Final exam
    -  During the university-assigned final exam slot for this class (TBD)
2%   Asynchronous participation (website comments)
Why parallelism?
Some historical context: why avoid parallel processing?

- Single-threaded CPU performance doubling ~ every 18 months
- Implication: working to parallelize your code was often not worth the time
  - Software developer does nothing, code gets faster next year. Woot!

Image credit: Olukutun and Hammond, ACM Queue 2005
Until ~15 years ago: two significant reasons for processor performance improvement

1. Exploiting instruction-level parallelism (superscalar execution)

2. Increasing CPU clock frequency
What is a computer program?
Here is a program written in C

```c
int main(int argc, char** argv) {
    int x = 1;
    for (int i=0; i<10; i++) {
        x = x + x;
    }
    printf("%d\n", x);
    return 0;
}
```
What is a program? (from a processor’s perspective)

A program is just a list of processor instructions!

```c
int main(int argc, char** argv) {
    int x = 1;
    for (int i=0; i<10; i++) {
        x = x + x;
    }
    printf("%d\n", x);
    return 0;
}
```

Compile code

```
_main:
100000f10:  pushq  %rbp
100000f11:  movq  %rsp, %rbp
100000f14:  subq  $32, %rsp
100000f18:  movl  $0, -4(%rbp)
100000f1f:  movl  %edi, -8(%rbp)
100000f22:  movq  %rsi, -16(%rbp)
100000f26:  movl  $1, -20(%rbp)
100000f2d:  movl  $0, -24(%rbp)
100000f34:  cmpl  $10, -24(%rbp)
100000f38:  jge  23 <_main+0x45>
100000f3e:  movl  -20(%rbp), %eax
100000f41:  addl  -20(%rbp), %eax
100000f44:  movl  %eax, -20(%rbp)
100000f47:  movl  -24(%rbp), %eax
100000f4a:  addl  $1, %eax
100000f4d:  movl  %eax, -24(%rbp)
100000f50:  jmp  -33 <_main+0x24>
100000f55:  leaq  58(%rip), %rdi
100000f5c:  movl  -20(%rbp), %esi
100000f5f:  movb  $0, %al
100000f61:  callq  14
100000f66:  xorl  %esi, %esi
100000f68:  movl  %eax, -28(%rbp)
100000f6b:  movl  %esi, %eax
100000f6d:  addq  $32, %rsp
100000f71:  popq  %rbp
100000f72:  rets
```

Stanford CS149, Fall 2022
Kind of like the instructions in a recipe for your favorite meals

Mmm, carne asada

Instructions

1. In a large mixing bowl combine orange juice, olive oil, cilantro, lime juice, lemon juice, white wine vinegar, cumin, salt and pepper, jalapeño, and garlic; whisk until well combined.
2. Reserve ½ cup of the marinade; cover the rest and refrigerate.
3. Combine remaining marinade and steak in a large resealable freezer bag; seal and refrigerate for at least 2 hours, or overnight.
4. Preheat grill to HIGH heat.
5. Remove steak from marinade and lightly pat dry with paper towels.
6. Add steak to the preheated grill and cook for another 6 to 8 minutes per side, or until desired doneness. **Note that flank steak tastes best when cooked to rare or medium rare because it’s a lean cut of steak.**
7. Remove from heat and let rest for 10 minutes. Thinly slice steak against the grain, garnish with reserved cilantro mixture, and serve.
What does a processor do?
A processor executes instructions

Professor Kayvon’s
Very Simple Processor

Fetch/Decode

ALU (Execution Unit)

Execution Context
- Register 0 (R0)
- Register 1 (R1)
- Register 2 (R2)
- Register 3 (R3)

Determine what instruction to run next

Execution unit: performs the operation described by an instruction, which may modify values in the processor’s registers or the computer’s memory

Registers: maintain program state: store value of variables used as inputs and outputs to operations
One example instruction: add two numbers

Step 1:
Processor gets next program instruction from memory (figure out what the processor should do next)

\[
\text{add } R0 \leftarrow R0, R1
\]

"Please add the contents of register R0 to the contents of register R1 and put the result of the addition into register R0"

Step 2:
Get operation inputs from registers
Contents of R0 input to execution unit: 32
Contents of R1 input to execution unit: 64

Step 3:
Perform addition operation:
Execution unit performs arithmetic, the result is: 96
One example instruction: add two numbers

Step 1:
Processor gets next program instruction from memory (figure out what the processor should do next)

\[
\text{add } R0 \leftarrow R0, R1
\]

"Please add the contents of register R0 to the contents of register R1 and put the result of the addition into register R0"

Step 2:
Get operation inputs from registers

- Contents of R0 input to execution unit: 32
- Contents of R1 input to execution unit: 64

Step 3:
Perform addition operation:
Execution unit performs arithmetic, the result is: 96

Step 4:
Store result back to register R0
Execute program

My very simple processor: executes one instruction per clock

```
ld  r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
...  
...  
...  
...  
...  
st  addr[r2], r0
```
Execute program

My very simple processor: executes one instruction per clock

Fetch/Decode

Execution Unit (ALU)

Execution Context

```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
...
st   addr[r2], r0
```
Execute program

My very simple processor: executes one instruction per clock

Fetch/Decode

Execution Unit (ALU)

Execution Context

ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
Execute program

My very simple processor: executes one instruction per clock

ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
...
...
...
...
...
...
st addr[r2], r0
Review of how computers work...

What is a computer program? (from a processor’s perspective)

*It is a list of instructions to execute!*

What is an instruction?

*It describes an operation for a processor to perform.*

*Executing an instruction typically modifies the computer’s state.*

What do I mean when I talk about a computer’s “state”?

*The values of program data, which are stored in a processor’s registers or in memory.*
Let's consider a very simple piece of code:

\[ a = x^2 + y^2 + z^2 \]

Assume register \( R0 = x \), \( R1 = y \), \( R2 = z \)

1. \( \text{mul } R0, R0, R0 \)
2. \( \text{mul } R1, R1, R1 \)
3. \( \text{mul } R2, R2, R2 \)
4. \( \text{add } R0, R0, R1 \)
5. \( \text{add } R3, R0, R2 \)

\( R3 \) now stores value of program variable ‘\( a \)’

This program has five instructions, so it will take five clocks to execute, correct? Can we do better?
What if up to two instructions can be performed at once?

\[ a = x^2 + y^2 + z^2 \]

Assume register
\[ R0 = x, \ R1 = y, \ R2 = z \]

1. \text{mul} \ R0, \ R0, \ R0
2. \text{mul} \ R1, \ R1, \ R1
3. \text{mul} \ R2, \ R2, \ R2
4. \text{add} \ R0, \ R0, \ R1
5. \text{add} \ R3, \ R0, \ R2

R3 now stores value of program variable 'a'
What if up to two instructions can be performed at once?

\[ a = x^2 + y^2 + z^2 \]

Assume register
\[ R0 = x, R1 = y, R2 = z \]

1. \text{mul} R0, R0, R0
2. \text{mul} R1, R1, R1
3. \text{mul} R2, R2, R2
4. \text{add} R0, R0, R1
5. \text{add} R3, R0, R2

R3 now stores value of program variable ‘a’
What does it mean for our parallel to scheduling to "respects program order"?
What about three instructions at once?

\[ a = x^2 + y^2 + z^2 \]

Assume register

\[ R0 = x, R1 = y, R2 = z \]

1. \texttt{mul R0, R0, R0}
2. \texttt{mul R1, R1, R1}
3. \texttt{mul R2, R2, R2}
4. \texttt{add R0, R0, R1}
5. \texttt{add R3, R0, R2}

\texttt{R3} now stores value of program variable ‘a’
What about three instructions at once?

\[ a = x^2 + y^2 + z^2 \]

Assume register 
R0 = x, R1 = y, R2 = z

1. mul R0, R0, R0
2. mul R1, R1, R1
3. mul R2, R2, R2
4. add R0, R0, R1
5. add R3, R0, R2

R3 now stores value of program variable ‘a’
Instruction level parallelism (ILP) example

- ILP = 3

\[ a = x \times x + y \times y + z \times z \]
Superscalar processor execution

\[ a = x^2 + y^2 + z^2 \]

Assume register:
\[ R0 = x, R1 = y, R2 = z \]

Idea #1:

**Superscalar execution:** processor automatically finds independent instructions in an instruction sequence and executes them in parallel on multiple execution units!

In this example: instructions 1, 2, and 3 can be executed in parallel without impacting program correctness (on a superscalar processor that determines that the lack of dependencies exists)

But instruction 4 must be executed after instructions 1 and 2

And instruction 5 must be executed after instruction 4

* Or the compiler finds independent instructions at compile time and explicitly encodes dependencies in the compiled binary.
Superscalar processor

This processor can decode and execute up to two instructions per clock
Aside:
Old Intel Pentium 4 CPU

A more complex example

Program (sequence of instructions)

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>a = 2</td>
</tr>
<tr>
<td>01</td>
<td>b = 4</td>
</tr>
<tr>
<td>02</td>
<td>tmp2 = a + b // 6</td>
</tr>
<tr>
<td>03</td>
<td>tmp3 = tmp2 + a // 8</td>
</tr>
<tr>
<td>04</td>
<td>tmp4 = b + b // 8</td>
</tr>
<tr>
<td>05</td>
<td>tmp5 = b * b // 16</td>
</tr>
<tr>
<td>06</td>
<td>tmp6 = tmp2 + tmp4 // 14</td>
</tr>
<tr>
<td>07</td>
<td>tmp7 = tmp5 + tmp6 // 30</td>
</tr>
<tr>
<td>08</td>
<td>if (tmp3 &gt; 7)</td>
</tr>
<tr>
<td>09</td>
<td>print tmp3</td>
</tr>
<tr>
<td>10</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>print tmp7</td>
</tr>
</tbody>
</table>

Instruction dependency graph

value during execution
Diminishing returns of superscalar execution

Most available ILP is exploited by a processor capable of issuing four instructions per clock (Little performance benefit from building a processor that can issue more)

Source: Culler & Singh (data from Johnson 1991)
Moore’s Law: The number of transistors on microchips doubles every two years. This advancement is important for other aspects of technological progress in computing such as processing speed or the price of computers.

Transistor count

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world’s largest problems.
ILP tapped out + end of frequency scaling

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)

No further benefit from ILP

Processor clock rate stops increasing

Image credit: "The free Lunch is Over" by Herb Sutter, Dr. Dobbs 2005
The “power wall”

Power consumed by a transistor:

Dynamic power $\propto$ capacitive load $\times$ voltage$^2$ $\times$ frequency

Static power: transistors burn power even when inactive due to leakage

High power = high heat

Power is a critical design constraint in modern processors

<table>
<thead>
<tr>
<th></th>
<th>TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple M1 laptop:</td>
<td>13W</td>
</tr>
<tr>
<td>Intel Core i9 10900K (in desktop CPU):</td>
<td>95W</td>
</tr>
<tr>
<td>NVIDIA RTX 3080 GPU</td>
<td>320W</td>
</tr>
<tr>
<td>Mobile phone processor</td>
<td>$1/2$ - 2W</td>
</tr>
<tr>
<td>World’s fastest supercomputer</td>
<td>megawatts</td>
</tr>
<tr>
<td>Standard microwave oven</td>
<td>700W</td>
</tr>
</tbody>
</table>

Source: Intel, NVIDIA, Wikipedia, Top500.org
Power draw as a function of clock frequency

Dynamic power $\propto$ capacitive load $\times$ voltage$^2$ $\times$ frequency

Static power: transistors burn power even when inactive due to leakage

Maximum allowed frequency determined by processor’s core voltage
Single-core performance scaling

The rate of single-instruction stream performance scaling has decreased (almost to zero)

1. Frequency scaling limited by power
2. ILP scaling tapped out

Architects are now building faster processors by adding more execution units that run in parallel (or units that are specialized for a specific task (like graphics, or audio/video playback))

Software must be written to be parallel to see performance gains. No more free lunch for software developers!

Image credit: “The free Lunch is Over” by Herb Sutter, Dr. Dobbs 2005
Example: multi-core CPU

Intel “Comet Lake” 10th Generation Core i9 10-core CPU (2020)
One thing you will learn in this course

- How to write code that efficiently uses the resources in a modern multi-core CPU

- Example: assignment 1 (coming up!)
  - Running on a quad-core Intel CPU
  - Four CPU cores
  - AVX SIMD vector instructions + hyper-threading
  - Baseline: single-threaded C program compiled with -O3
  - Parallelized program that uses all parallel execution resources on this CPU...

~32-40x faster!

We’ll talk about these terms next time!
AMD Ryzen Threadripper 3990X
64 cores, 4.3 GHz

Four 8-core chiplets
NVIDIA Ampere GA102 GPU
GeForce RTX 3080 (2020)

17,408 fp32 multipliers organized in 68 major processing blocks.
Supercomputing

- Today: combinations of multi-core CPUs + GPUs
- Oak Ridge National Laboratory: Summit (currently #4 supercomputer in world)
  - 9,216 x 22-core IBM Power9 CPUs + 27,648 NVIDIA Volta GPUs
Mobile parallel processing

Raspberry Pi 3
Quad-core ARM A53 CPU
Mobile parallel processing

Power constraints heavily influence the design of mobile systems

Apple A13 Bionic
(in iPhone 11)

2 “big” CPU cores +
4 “small” CPU cores +
Apple-designed multi-core GPU +
Image processor +
Neural Engine for DNN acceleration +
Motion processor
Parallel + specialized HW

- Achieving high efficiency will be a key theme in this class

- We will discuss how modern systems are not only parallel, but also specialize processing units to achieve high levels of power efficiency
Another recent smartphone

Google Pixel 2 Phone:
Qualcomm Snapdragon 835 SoC + Google Visual Pixel Core
Datacenter-scale applications

Google TPU pods
TPU = Tensor Processing Unit: specialized processor for ML computations

Image Credit: TechInsights Inc.
Specialized hardware to accelerate DNN inference/training

- Google TPU3
- GraphCore IPU
- Huawei Kirin NPU
- Apple Neural Engine
- Intel Deep Learning Inference Accelerator
- SambaNova Cardinal SN10
- Cerebras Wafer Scale Engine
- Ampere GPU with Tensor Cores
Summary

- Today, single-thread-of-control performance is improving very slowly
  - To run programs significantly faster, programs must utilize multiple processing elements or specialized processing
  - Which means you need to know how to write parallel code

- Writing parallel programs can be challenging
  - Requires problem partitioning, communication, synchronization
  - Knowledge of machine characteristics is important

- I suspect you will find that modern computers have tremendously more processing power than you might realize, if you just use it!
Welcome to CS149!

- Get signed up on the website
- Find yourself a partner!
  (remember, we can help you)

Prof. Kayvon
Prof. Olukotun
James
Keshav
Raj
Arden
Drew
Pranil