CoE 163

Computing Architectures and Algorithms

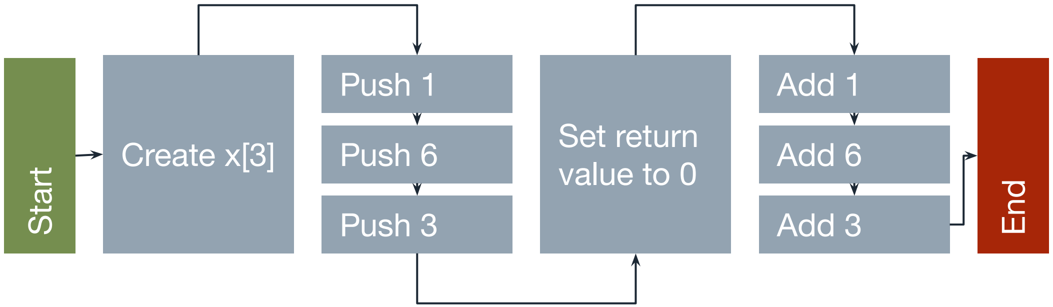
11a: Parallel Programming Introduction

SEQUENTIAL PROGRAMMING

So far, you have been taught that each line of your code is executed *sequentially*. It’s like a series of commands the computer just executes one after another.

SEQUENTIAL PROGRAMMING

```
int main() {
    int x[] = {1, 6, 3};
    return x[0] + x[1] + x[2];
}
```



Start

Create x[3]

Push 1

Push 6

Push 3

Set return value to 0

Add 1

Add 6

Add 3

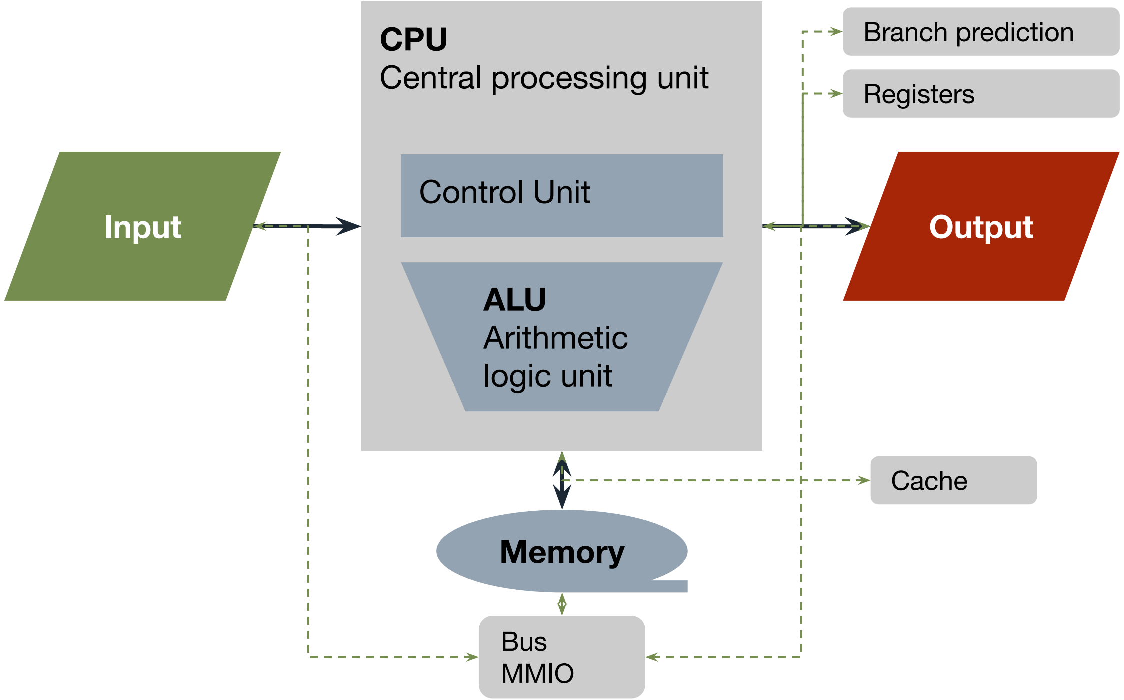
End

VON NEUMANN ARCHITECTURE

Most common form of computer architecture - discovered in around 1940s.

Executes instructions *sequentially* through a central processing unit (CPU) attached to input, output, and memory streams.

VON NEUMANN ARCHITECTURE



FLYNN'S TAXONOMY

Classify computer architectures based on number of instruction and data streams available.

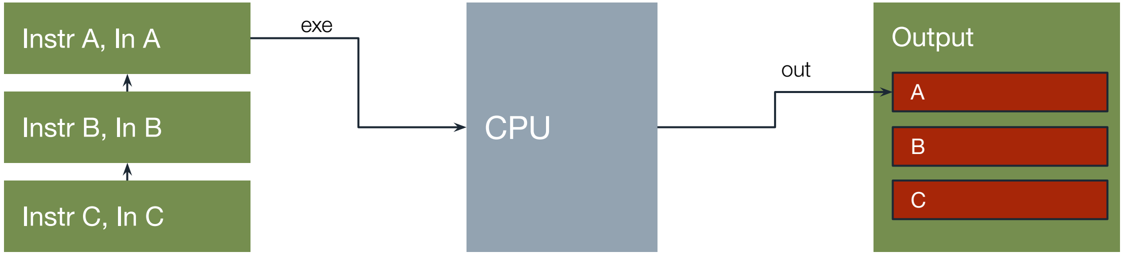
Most PCs are only SISD until around 2010s, when multiple-core CPUs became possible.

FLYNN’S TAXONOMY

	Data stream count	
instruction streams count	**SISD** single instruction, single data	**SIMD** single instruction, multiple data
	MISD multiple instruction, single data	**MIMD** multiple instruction, multiple data

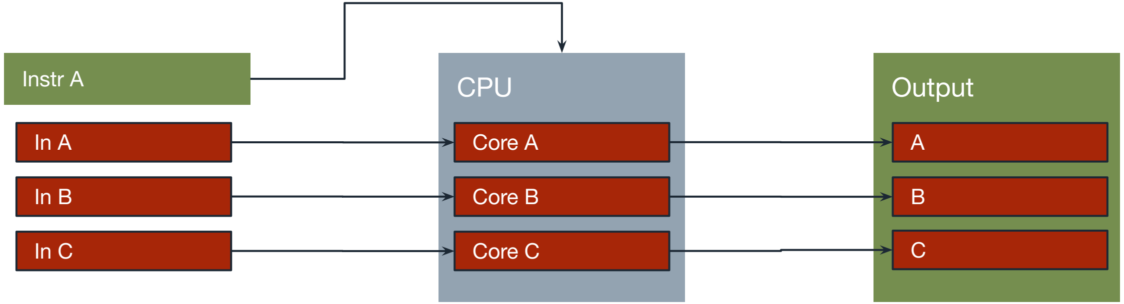
FLYNN'S TAXONOMY: SISD

- Can only process one instruction at a time, and output one data at a time
- Only has one input stream and one output stream - queueing is needed
- Found in older single-core PCs and mainframes



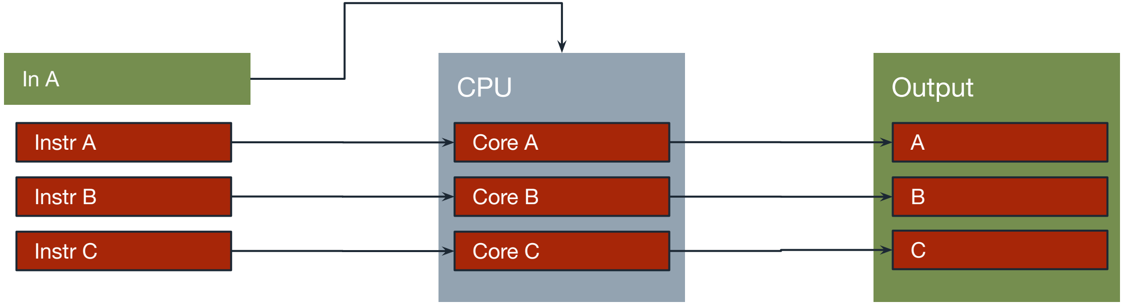
FLYNN’S TAXONOMY: SIMD

- Multiple processors are loaded with the same instructions, but working on different data units
- Usually used to process smaller outputs to build a larger output
- Found in GPUs, which usually work on repetitive units of data



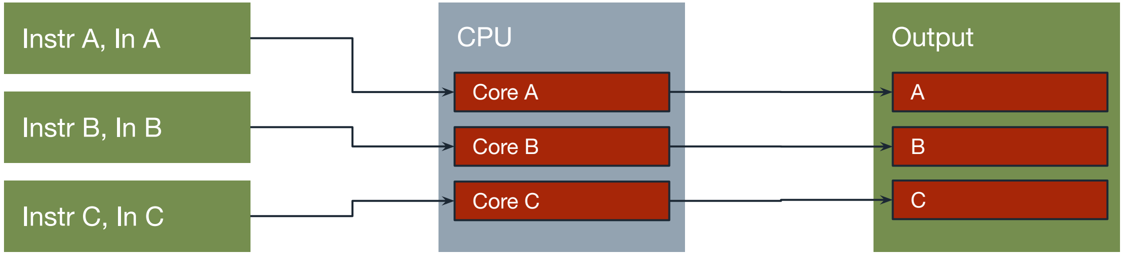
FLYNN'S TAXONOMY: MISD

- Multiple processors are loaded with different instructions, but working on the same data
- The architecture is not used a lot
- Found in fault tolerance systems and the US Space Shuttle computer - but nothing majorly available



FLYNN'S TAXONOMY: MIMD

- Multiple processors are loaded with different instructions, and working on different data
- This architecture saves time since tasks can now be executed in *parallel*
- Found in modern computing systems



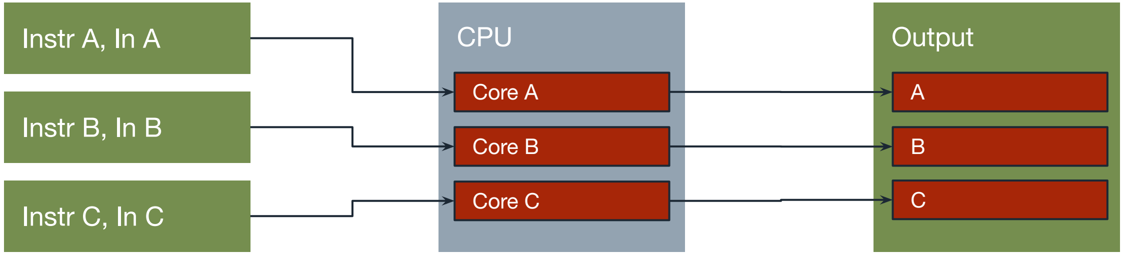
BEYOND SEQUENTIAL PROGRAMMING

With a queue in place, it takes time to execute a long list of instructions. A single CPU is too limiting!

What if we split our instructions such that we can maximize our time and resources?

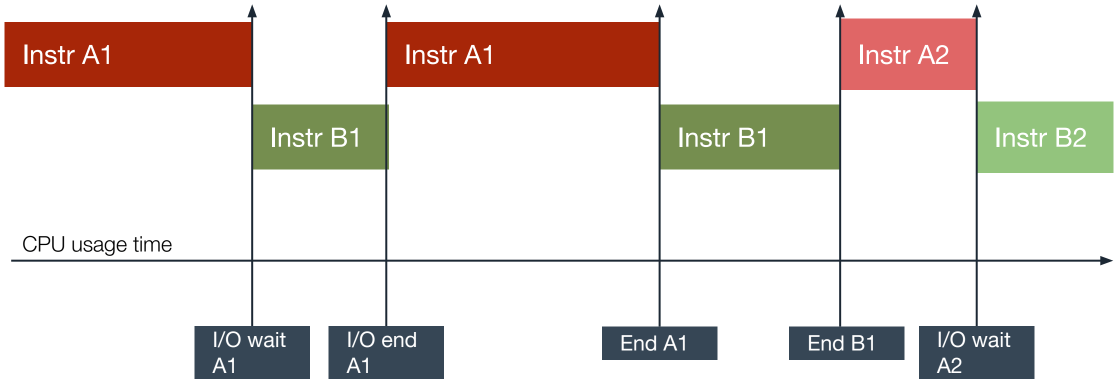
PARALLELISM

- Execute instructions at the same time
- Used to perform more work for less time by decomposing a problem such that each piece can run independently of each other *simultaneously*



CONCURRENCY

- Split a problem into instructions that can be executed independently
- Used to improve CPU utilization by getting a piece of instruction from a pool if it becomes idle due to various reasons (I/O wait, locks, etc.)



PARALLEL PROGRAMMING

In comparison to sequential programming, *parallel programming* uses multiple computing modules to solve a problem.

It saves time because it can now execute tasks at the same time - *multitasking*!

It enables *concurrency*!

PARALLEL PROGRAMMING MODELS

Parallel programming programs can be modeled in various ways with two broad categories

- Process interaction
    - Communication between different parallel processes
- Problem decomposition
    - Formulation of parallel processes

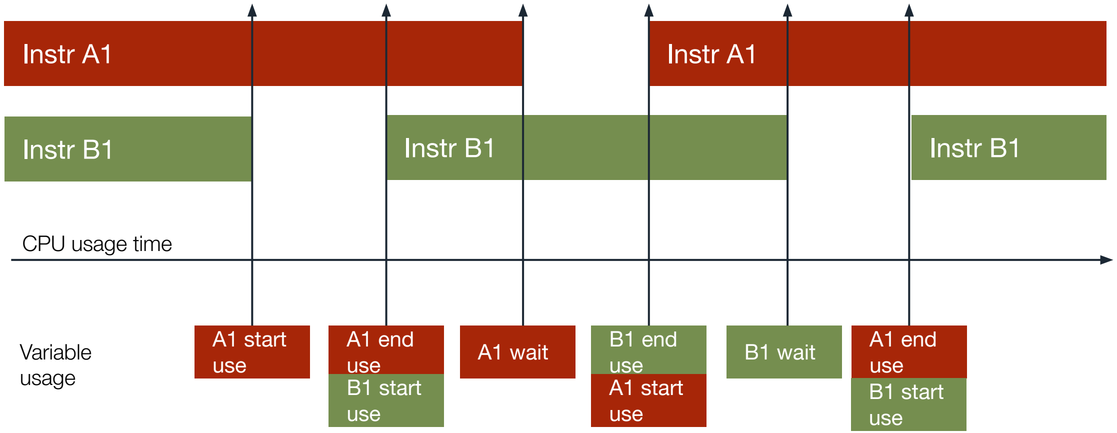
PROCESS INTERACTION

Programs can be further divided into different categories:

- Shared space
    - Like bulletin boards
- Message passing
    - Like postal mail

PROCESS INTERACTION: SHARED SPACE

- Multiple tasks share a single address space
- Locks and semaphores are used to “synchronize” and control access to the memory and prevent data conflicts



Instr A1

Instr B1

Instr B1

Instr A1

Instr B1

CPU usage time

Variable usage

A1 start use

A1 end use

B1 start use

A1 wait

B1 end use

A1 start use

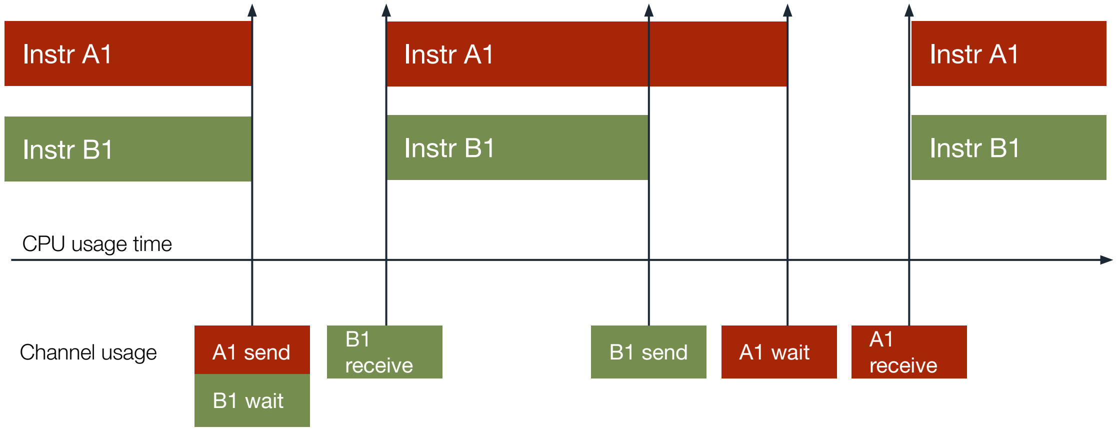
B1 wait

A1 end use

B1 start use

PROCESS INTERACTION: MESSAGE PASSING

- Multiple tasks communicate with each other through some channel
- Blocking channels are used to “synchronize” and control access to the memory and prevent data conflicts



Instr A1

Instr B1

Instr A1

Instr B1

Instr A1

Instr B1

CPU usage time

Channel usage

A1 send

B1 wait

B1 receive

B1 send

A1 wait

A1 receive

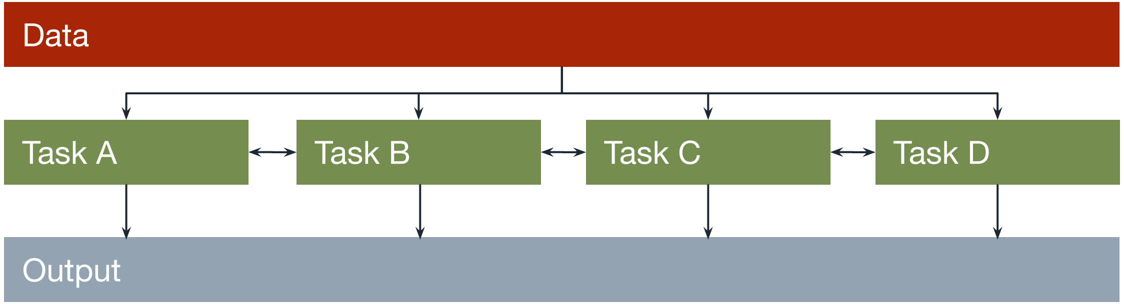
PROCESS DECOMPOSITION

Programs can be further divided into different categories:

- Task parallelism
  - Split program into different specialized tasks
- Data parallelism
  - Split data for processing to tasks nodes

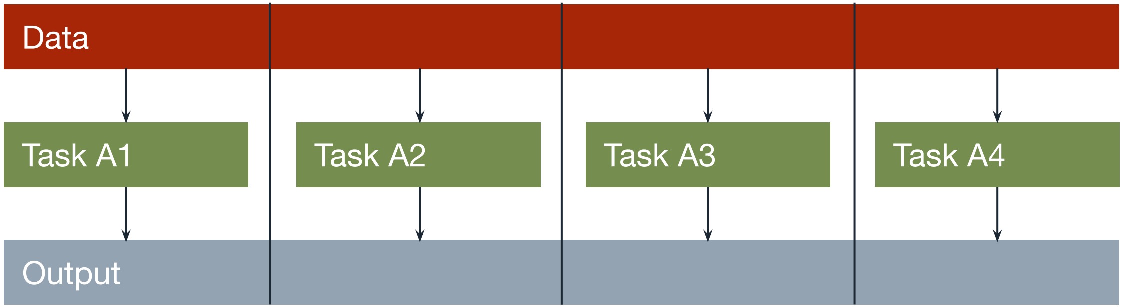
PROCESS DECOMPOSITION: TASKS

- Program split into several tasks
- An MIMD/MISD architecture falls under this type
- Synchronization is explicit through mutex locks and semaphores
- Operating on private data



PROCESS DECOMPOSITION: DATA

- Data split into pieces for processing of copies of tasks
- An MIMD/SIMD architecture falls under this type
- Communication is usually through shared memory while synchronization is implicit through locksteps (atomic transactions)
- Operating on shared data



PARALLELIZING A PROGRAM

- Can the program be parallelized?
  - Does it have portions we can copy and execute over all the data repetitively?
- Is it worth it to parallelize?
  - Is this portion of the program doing the most work?
- Where are the data dependencies?
  - Do we need to execute this part before moving on?

PARALLELIZING A PROGRAM

- Are there any bottlenecks?
  - Where do we need to wait for the data to be available?
- How do we decompose the program?

CONSIDER…

We want to create our own pseudorandom number generation for our game rigging needs.

PSEUDORANDOM GENERATOR

- Set first a number S (“seed”) where the number generation will start
- Pick three constants a (large prime), c (large prime), and m that will influence the next values of the generator
- Use this value to generate the next value using the same equation!

$$x_0 = S$$
$$x_1 = (ax_0 + c)\ mod\ m$$
$$x_n = (ax_{n-1} + c)\ mod\ m$$

Generated numbers from 0 to m - 1!

RANDOMIZER: OBVIOUS SOLUTION

- Run the simple equation on a loop - save the previous iteration result and use it on the next
- This is sequential programming
- What if we want to get the millionth number in the sequence? Will it be fast enough?

```
x = S
k = 1000000

for n in range(0, k):
    x = (a * x + c) % m
```

RANDOMIZER: OBSERVATION

- Reform the equation (generator) to find the kth random number from the some nth random number
- From the nth number, we can generate k more random numbers
- Maybe we can leverage this observation to hasten the generation?

$$x_n = (ax_{n-1} + c)\ mod\ m$$

$$x_{n+k} = (a(a(a(...) + c) + c) + c)\ mod\ m$$

$$x_{n+k} = (a^k x_n + c\Sigma_{j=0}^{k-1} a^j)\ mod\ m$$

$$x_{n+k} = \left(a^k x_n + c\frac{a^k - 1}{a - 1}\right)\ mod\ m$$

$$x_{n+k} = (Ax_n + C)\ mod\ m$$

RANDOMIZER: PARALLEL SOLUTION

- Generate k random numbers on a loop sequentially
- Send out these k numbers to separate threads/processes to generate k numbers in parallel
- If we want to get the millionth random number, we only need to spend around 1000 steps for k=1000 compared to a million steps

```
init_arr = []
x = S
k = 1000

for n in range(1, k):
    x = (a * x + c) % m)
    init_arr.append(x)

# Send elements of x to different threads/processes
```

EMBARRASSINGLY PARALLEL PROGRAM

There's nothing shameful about it, but is instead an idiom for "overabundance". These programs are naturally "easy" and "simple".

Programs may need non-trivial data partition (input), data collection (output), and scheduling for the algorithm to work.

EMBARRASSINGLY PARALLEL PROGRAM

These programs have the following characteristics:

- Parallel processes working independently
- Almost no needed communication between processes

PARALLEL PYTHON: GIL

The *global interpreter lock* (GIL) is a lock that ensures that each thread runs one at a time.

This means that threading is a concurrency mechanism in Python, but we can still use multiple processors to achieve true parallelism.

RANDOMIZER: PARALLEL PYTHON

- The `multiprocessing` and `threading` libraries enable concurrency in Python
- `threading` sends function to different threads and is bounded by the GIL
- `multiprocessing` sends functions to different processors

```
def compute_kth_pool(a, c, m, k, ith, prev_item):
    a_pow = a ** k
    next_k = (a_pow * prev_item + c * ((a_pow - 1) //
(a - 1))) % m

    return (ith, next_k)
```

RANDOMIZER: MULTIPROCESS

```
init_arr_pool = [(a * S + c) % m]

for _ in range(1, k):
    init_arr_pool.append((a * init_arr_pool[-1] + c) % m)

pool = multiprocessing.Pool(processes=4)

for each_k in range(1, rand_num_idx // k):
    next_ans = []

    for z in range(k):
           next_ans.append(pool.apply_async(compute_kth_pool, (a, c,
m, k, z, init_arr_pool[z])))

    for each_ans in next_ans:
            i_prev_idx, next_val = each_ans.get()
            init_arr_pool[i_prev_idx] = next_val

pool.close()
```

RANDOMIZER: TIME PROFILE

Setting-up threads or processes usually have overhead time (to spawn and collect outputs), so it can sometimes be slower than serial programs.

Serial: ~10 ms

Multiprocess*: ~1000 ms

* Getting 10kth number, 4 processors with k = 10

TIPS

- Don’t confuse parallelism and concurrency!
- Parallelize a program only when necessary
- Consider overhead of setting-up each subprocess or thread when formulating parallel programs
- Practice decomposing problems into units

RESOURCES

- Short article on parallel programming
- Parallel programming tutorial from the Lawrence Livermore National Laboratory
- Short lecture on parallel programming from Cornell University

RESOURCES

- Parallel programming models from the Florida State University
- Article on Python threading
- Article on Python multiprocessing

CoE 163

Computing Architectures and Algorithms

11a: Parallel Programming Introduction