Introduction to High Performance Computing

CS 1645 | CS 2045

Spring 2017

Administrivia

Me:

Dr. Bryan Mills, bmills@cs.pitt.edu

My Office:

Sennott Square 6148 - Tuesday 6-7, Wednesday 4-5

Course Website:

http://people.cs.pitt.edu/~bmills/pages/cs1645.html

Teaching Assistant:

Fan Zhang, zhenjiangfan@cs.pitt.edu

Some images from wikipedia. Special thanks to Rami Melhem and Meneses Rojas for both slides and content.

History of Parallel Computing

- I/O Channels and DMA
- Instruction Pipelining
- Supercomputers!
  - Massively parallel processors (MPPs)
- Distributed Computing
  - Internet, Clusters, Cloud
- Multicore Technology
- GPUs

What is a Supercomputer?

“A **supercomputer** is a computer with a high-level computational capacity compared to a general-purpose computer.” - Wikipedia

“a large very fast mainframe used especially for scientific computations” - Webster Dictionary

Lets just say a supercomputer is…

- is fast (measured in FLOPS)
- is expensive (TaihuLight cost $273 Million)
- is shortlived (~5 years)
- introduces massive leap in computational ability

FLoating-point **O**perations **P**er **S**econd (FLOPS)

Name	Abbrevation	FLOPS
kiloFLOPS	kFLOPS	10^{3}
megaFLOPS	MFLOPS	10^{6}
gigaFLOPS	GFLOPS	10^{9}
teraFLOPS	TFLOPS	10^{12}
petaFLOPS	PFLOPS	10^{15}
exaFLOPS	EFLOPS	10^{18}
zettaFLOPS	ZFLOPS	10^{21}
yottaFLOPS	YFLOPS	10^{24}

FLoating-point **O**perations **P**er **S**econd (FLOPS)

Name	Abbrevation	FLOPS
kiloFLOPS	kFLOPS	10^3
megaFLOPS	MFLOPS	10^6
gigaFLOPS	GFLOPS	10^9
teraFLOPS	TFLOPS	10^{12}
petaFLOPS	PFLOPS	10^{15}
exaFLOPS	EFLOPS	10^{18}
zettaFLOPS	ZFLOPS	10^{21}
yottaFLOPS	YFLOPS	10^{24}

Intel i7 Core, 980 XE clocks in at 109 MFLOPS

Nvidia Tesla C2050 GPU 515 GFLOPS

Sunway TaihuLight, fastest computer in the world clocks in at 93.01 PFLOPS

Bigger = Better?

Top500

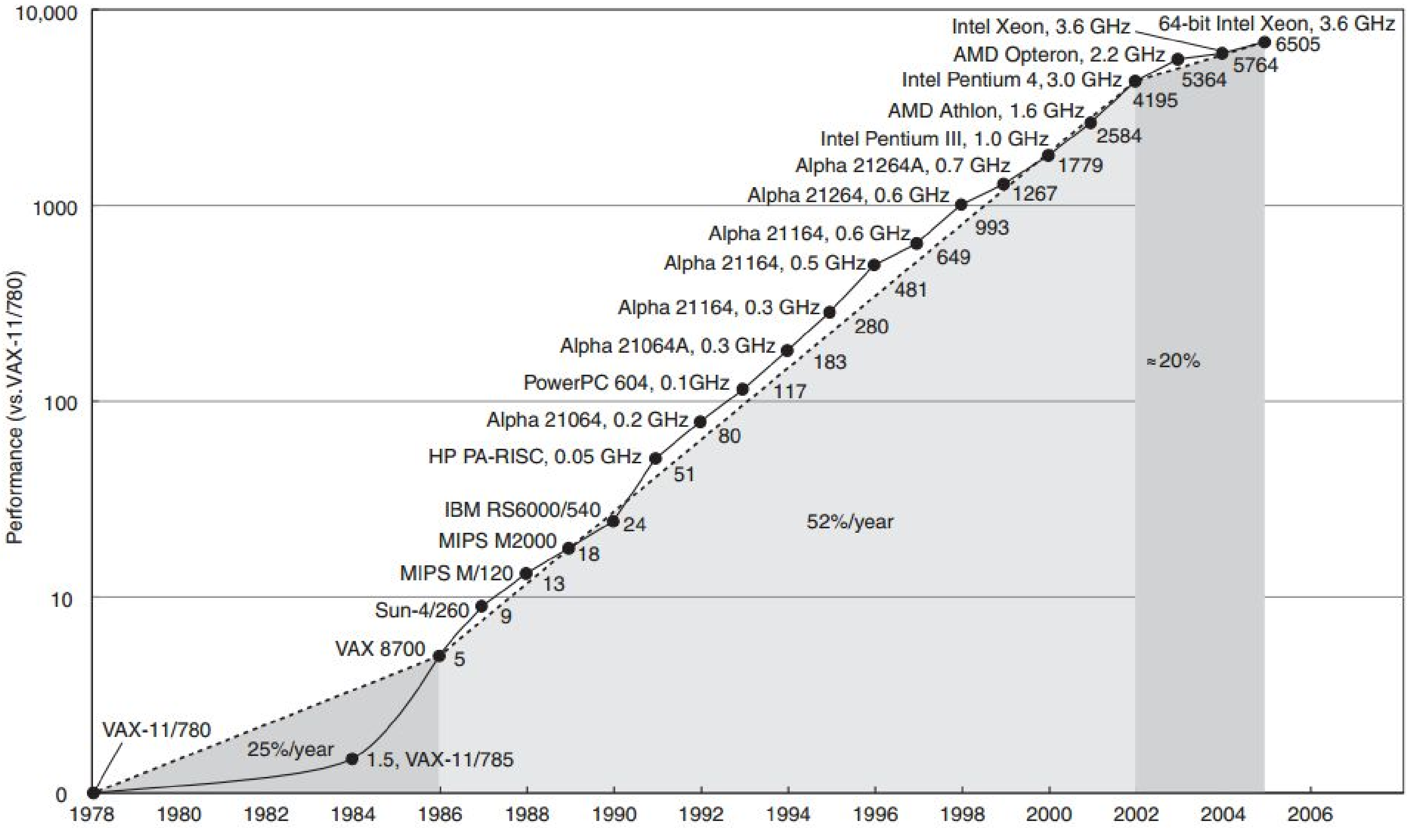
- Ranks the world's fastest supercomputers
- Uses LINPACK, a linear algebra benchmark, to measure max number of FLOP/s.

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
2	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P , NUDT National Super Computer Center in Guangzhou China	3,120,000	33,862.7	54,902.4	17,808
3	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
4	**Gyoukou** - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz , ExaScaler Japan Agency for Marine-Earth Science and Technology Japan	19,860,000	19,135.8	28,192.0	1,350
5	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209

Who Cares?

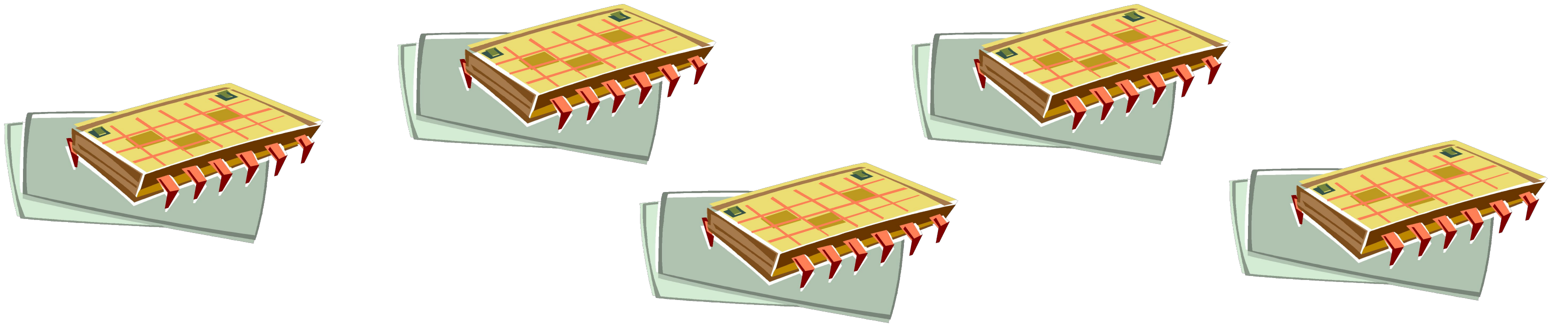
- Supercomputers lead the way but all of computing is moving to parallel processing
- Serial programs work fine?
  - Faster is always better
    - To go faster you must “think in parallel”
- Web Programming?
  - AJAX, Dependency Injection, Parallel Page Loads, WebSockets, NodeJS, Go, …………

Single Processor Performance



Multicore Design

- Instead of designing and building faster microprocessors, put multiple processors on a single integrated circuit.

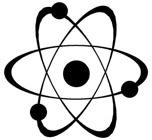


Why we're building parallel systems

- Up to now, performance increases have been attributable to increasing density of transistors.

- But there are inherent problems.

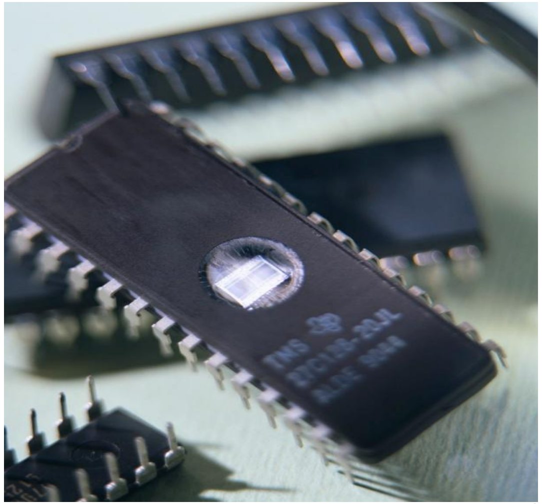
A little physics lesson



- Smaller transistors = faster processors.
- Faster processors = increased power consumption.
- Increased power consumption = increased heat.
- Increased heat = unreliable processors.

Solution

- Move away from single-core systems to multicore processors.
- "core" = central processing unit (CPU)



Parallelism for all!

Supercomputer in your Pocket?

iPhone X uses A11 processor has 4 cores

Pixel 2 uses Snapdragon 835 has 8 cores

Need to write parallel programs?

- Running multiple instances of a serial program often isn't very useful.
- Think of running multiple instances of your favorite game.
- What you really want is for it to run faster.

Approaches to the serial problem

- Rewrite serial programs so that they're parallel.
- Write translation programs that automatically convert serial programs into parallel programs.
  - This is very difficult to do.
  - Success has been limited.

More problems

- Some coding constructs can be recognized by an automatic program generator, and converted to a parallel construct.
- However, it’s likely that the result will be a very inefficient program.
- Sometimes the best parallel solution is to step back and devise an entirely new algorithm.

Example

- Compute n values and add them together.
- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

Example (cont.)

- We have p cores, p much smaller than n.
- Each core performs a partial sum of approximately n/p values.

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value( . . .);
   my_sum += my_x;
}
```

Each core uses it's own private variables and executes this block of code independently of the other cores

Example (cont.)

- After each core completes execution of the code, it's private variable my_sum contains the sum of the values computed by its calls to Compute_next_value.
- Ex., 8 cores, n = 24, then the calls to Compute_next_value return:

1,4,3,   9,2,8,   5,1,1,   5,2,7,   2,5,0,   4,1,8,   6,5,1,   2,3,9

Example (cont.)

- Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated "master" core which adds the final result.

Example (cont.)

```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
} else {
   send my_x to the master;
}
```

Example (cont.)

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Global sum

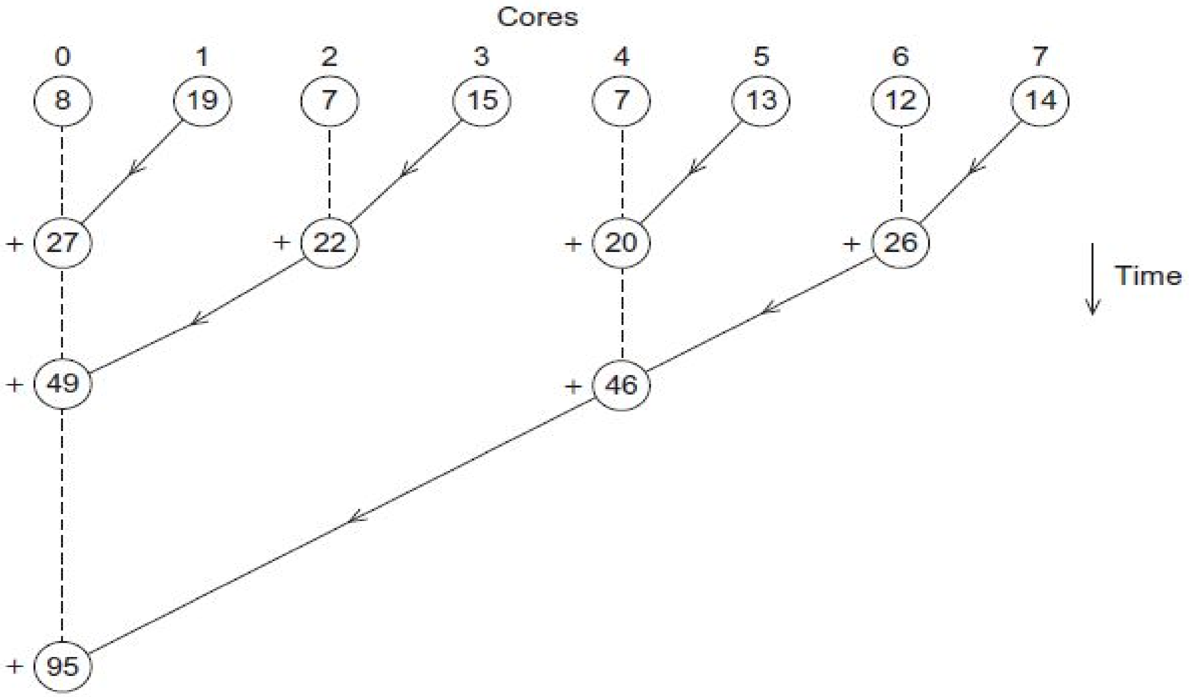
8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

Better parallel algorithm

- Don’t make the master core do all the work; share it among the other cores.
- Pair the cores; core 0 adds its result with core 1’s result.
- Core 2 adds its result with core 3’s result, etc.
- Work with odd and even numbered pairs of cores.
- Repeat the process now with the evenly ranked cores.
- Core 0 adds result from core 2.
- Core 4 adds the result from core 6, etc.
- Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result.

Tree-based Parallel Sum



Analysis

- In the first example, the master core performs 7 receives and 7 additions.
- In the second example, the master core performs 3 receives and 3 additions.
- The improvement is more than a factor of 2.

Analysis (cont.)

- The difference is more dramatic with a larger number of cores.
- If we have 1000 cores:
  - The first example would require the master to perform 999 receives and 999 additions.
  - The second example would only require 10 receives and 10 additions.
- Improvement of almost a factor of 100.

Speedup and Efficiency (page 58)

For a problem A of size n, assume to it takes:

- $T_s(n)$ time to execute in serial
- $T_p(n)$ time to execute with P processors

Speedup is, $S = T_s(n) / T_p(n)$

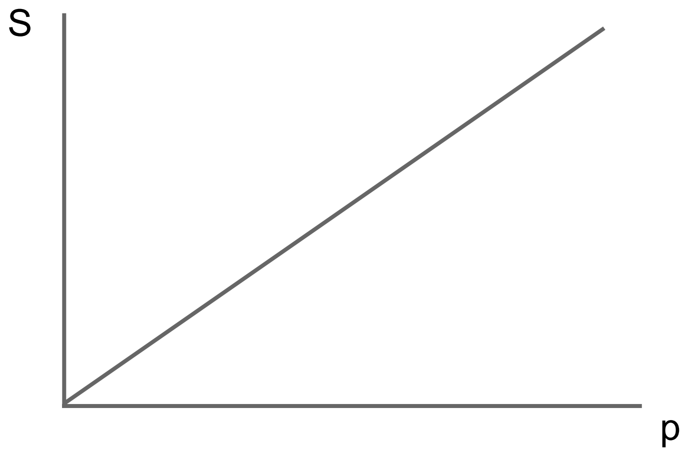
Efficiency is, $E = S / P$

Speedup is between 0 and p; Efficiency is between 0 and 1

Speedup

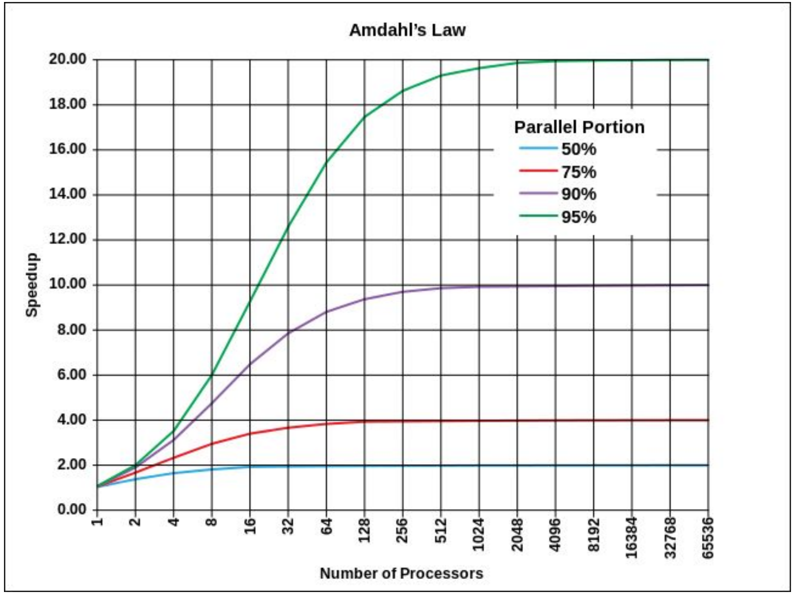
Linear speedup assumes that as we apply more processors we can always go faster.

Program is perfectly scalable if the speedup is independent of the problem size.



Amdahl's law

Unless "all" of a serial program is parallelized, the possible speedup is going to be very limited.



Amdahl's law - Example

- We can parallelize 90% of a serial program.
- Parallelization is "perfect" regardless of the number of cores *p* we use.

- T_s = 20 seconds
- $T_p = (0.9 * T_s)/p + 0.1 * T_s = (18 / p) + 2$

Amdahl's law - Example

- We can parallelize 90% of a serial program.
- Parallelization is "perfect" regardless of the number of cores *p* we use.

T_s = 20 seconds

$T_p = (0.9*T_s)/p + 0.1 * T_s = (18 / p) + 2$

Parallel Part   Serial Part

Amdahl's law - Example

$$T_p = (0.9 * T_s)/p + 0.1 * T_s = (18 / p) + 2$$

$$S = T_s / Tp =$$

$$\frac{T_s}{(0.9 * T_s)/p + 0.1 * T_s} = \frac{20}{18/p + 2}$$

<u>What is the maximum Speedup?</u>

Gustafson’s Law

Increase the size of the problem and the size of the serial portion decreases.

Just make the problem bigger and parallel algorithms will perform better.

This assumes that the serial work are things like setup, config, etc that doesn’t increase as the problem grows.

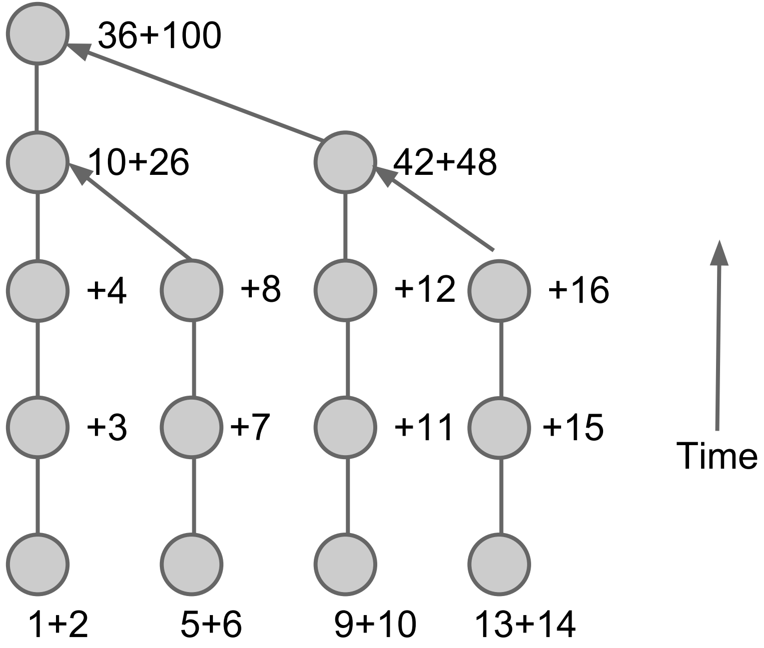
Efficient Parallel Sum (4 processors)

1. Divide work by 4.
2. Each processor works on their numbers
3. Then adds theirs and their neighbors

Computes 16 numbers in 5 steps.
  Speedup: 16/5 = 3.2
Computes 1024 numbers in 255 + 2 steps.
  Speedup: 1024/257 = 3.9

How long to compute n numbers?

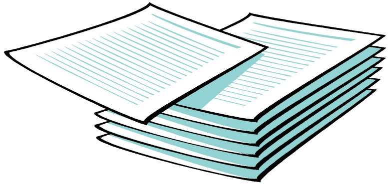
What is the speedup?



How do we write parallel programs?

- Task parallelism
  - Partition various tasks carried out solving the problem among the cores.
- Data parallelism
  - Partition the data used in solving the problem.
  - Each core carries out similar operations on it’s part of the data.

Professor P

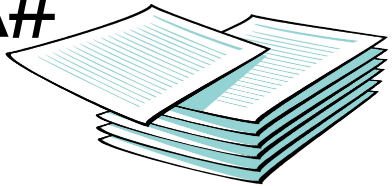


15 questions

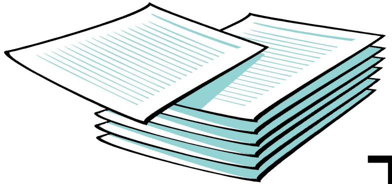
300 exams

Data Parallelism

TA#
1

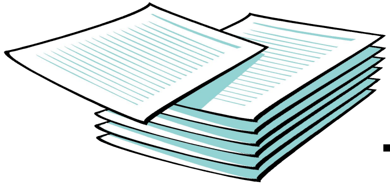


100 exams



100 exams

TA#
3



100 exams

TA#
2

Task Parallelism

TA#
1

Questions 1 - 5

Questions 11 - 15

TA#
3

Questions 6 - 10

TA#
2

Data Parallelism

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

Task Parallelism

```
if (I'm the master core) {
    sum = my_x;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my_x to the master;
}
```

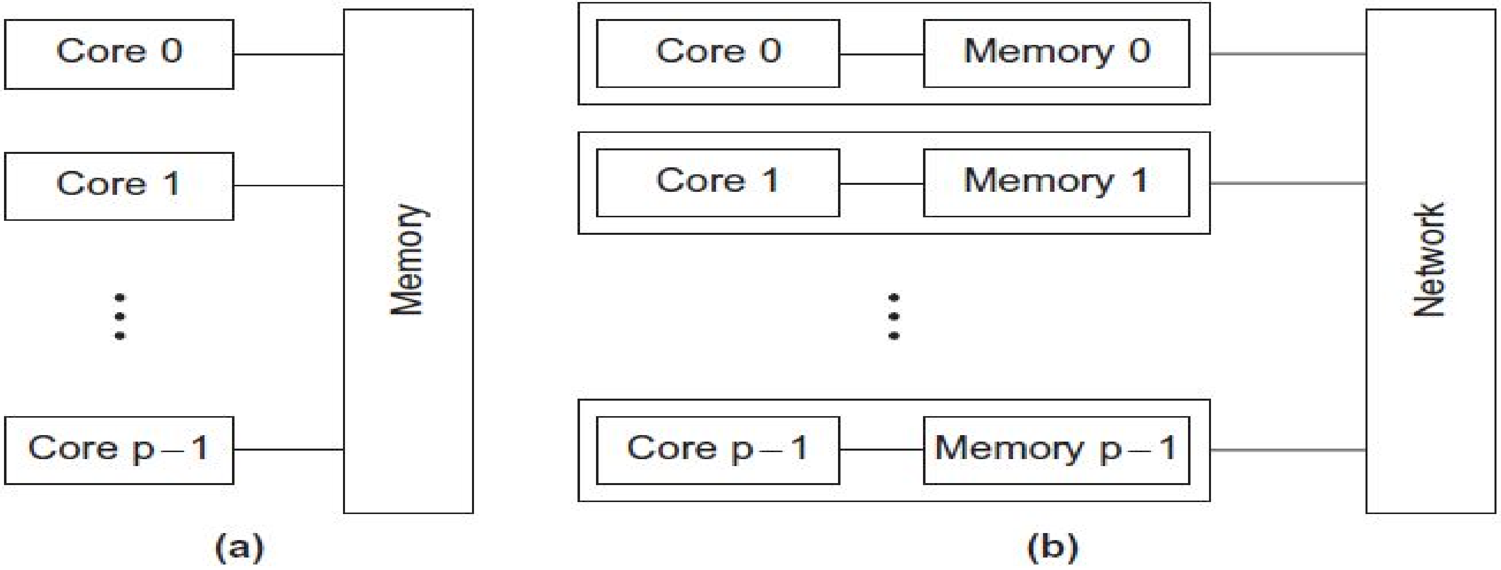
Tasks

1) Receiving
2) Addition

Coordination

- Cores usually need to coordinate their work.
- Communication – one or more cores send their current partial sums to another core.
- Load balancing – share the work evenly among the cores.
- Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.

Type of parallel systems



Shared-memory Distributed-memory

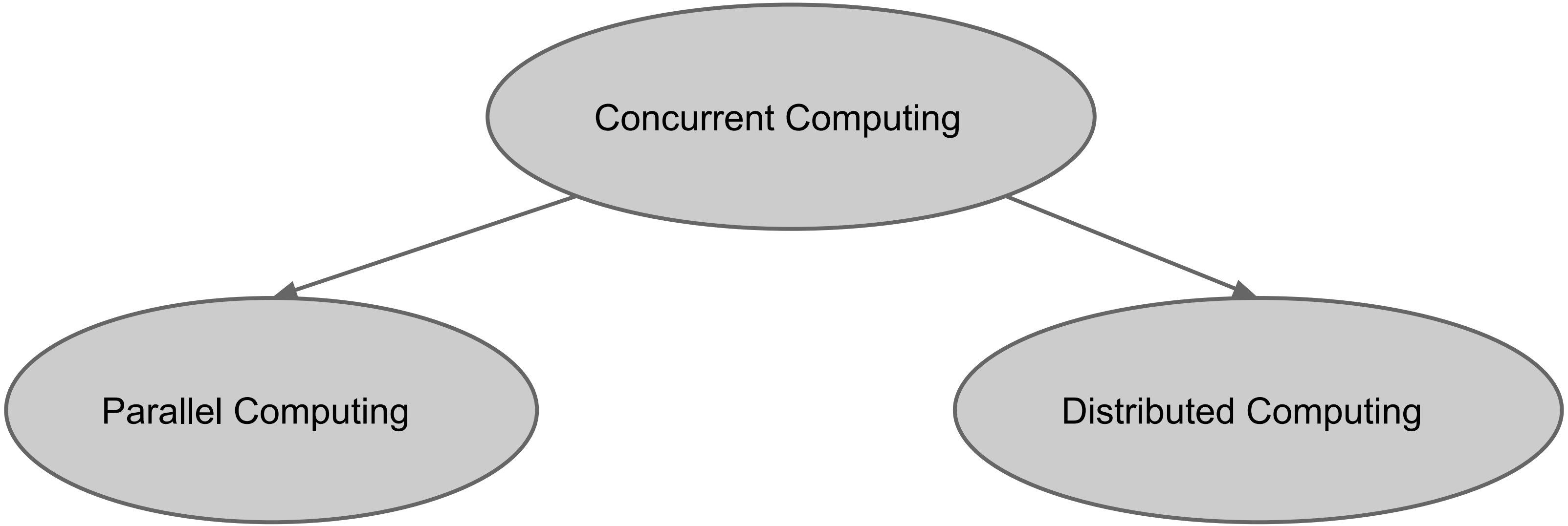
Type of parallel systems

- Shared-memory
  - The cores can share access to the computer's memory.
  - Coordinate the cores by having them examine and update shared memory locations.
- Distributed-memory
  - Each core has its own, private memory.
  - The cores must communicate explicitly by sending messages across a network.

What will we be doing?

- Learning how to write parallel algorithms.
- Writing parallel algorithms (primarily C)
  - Distributed Memory
    - MPI
  - Shared Memory
    - PThreads, OpenMP
  - GPUs
    - CUDA
  - Higher Level Constructs (if time permits)
    - Map/Reduce
    - Go
    - Dependency Injection

Terminology



Tightly Coupled.
Primary goal of performance.
Think supercomputers.

Loosely Coupled
Primary goal of ease of use:

- Reliability
- Accessibility
- Security

Think Internet