Plan

Last time

- power wall - difficult to shorten clock cycle
- memory wall - CPU speed versus memory latency
- analyzed $A \cdot B$ on a single core with large slow and small fast memories.
  - **reuse** of local data needed for high performance
  - **blocking** helps to achieve reuse

Today

- some terminology
- parallel computing models

Some loose terminology

PE: any physical computational device

Task: is a set of instructions

Stream: is a set of instructions executed in sequence

Paralel Task: (parallel stream) is a set of tasks (streams) that are executed concurrently by multiple PEs

Shared memory: logical and physical addresses are the same for all PEs (global address space).

Some loose terminology

Communications: data exchange among tasks.

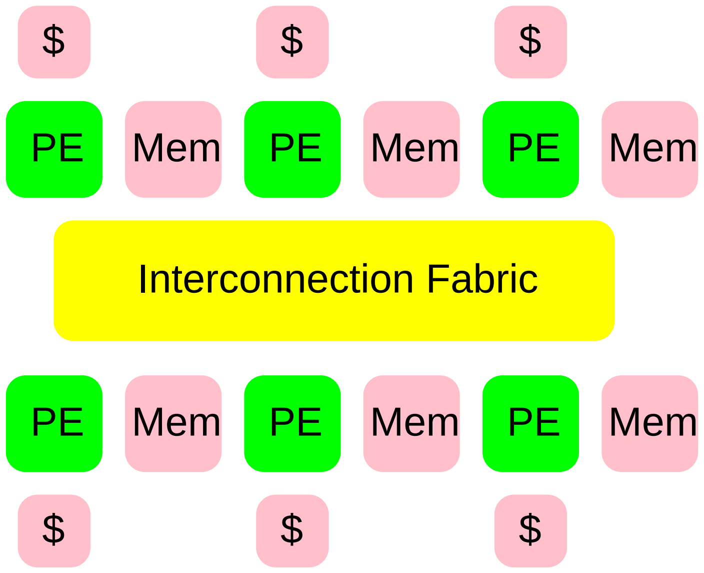
Distributed memory: PEs can directly access only their own local data

- PEs must use explicit communication mechanism to access data they do not own and which is local to other PEs.

Synchronization: coordination of parallel tasks

- implemented by establishing a synchronization point where a task must wait until other tasks reach the same execution point.

Hardware models



Several choices to make,

- hardware choices - # PEs, how powerfull, conectivity, etc.
- software choices

Parallel programming models

Software choices

- **Control**
  - How is parallel execution created?
- **Data sharing**
  - What is private and what is shared?
  - How is the data accessed?
- **Synchronization**
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?
- **Cost** of each of the above

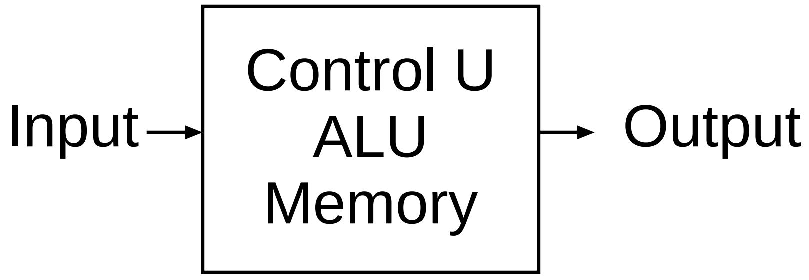
Parallel machine models

Flynn's taxonomy for instruction and data streams:

- **SISD** - Single Instruction, Single Data
- **SIMD** - Single Instruction, Multiple Data
- **MISD** - Multiple Instruction, Single Data **(?)**
- **MIMD** - Multiple Instruction, Multiple Data

SISD

Classical Von Neumann machine:



load	a
load	b
c	$= a + b$
store	c
d	$= c + c$

One instruction stream is acted on by a single CPU

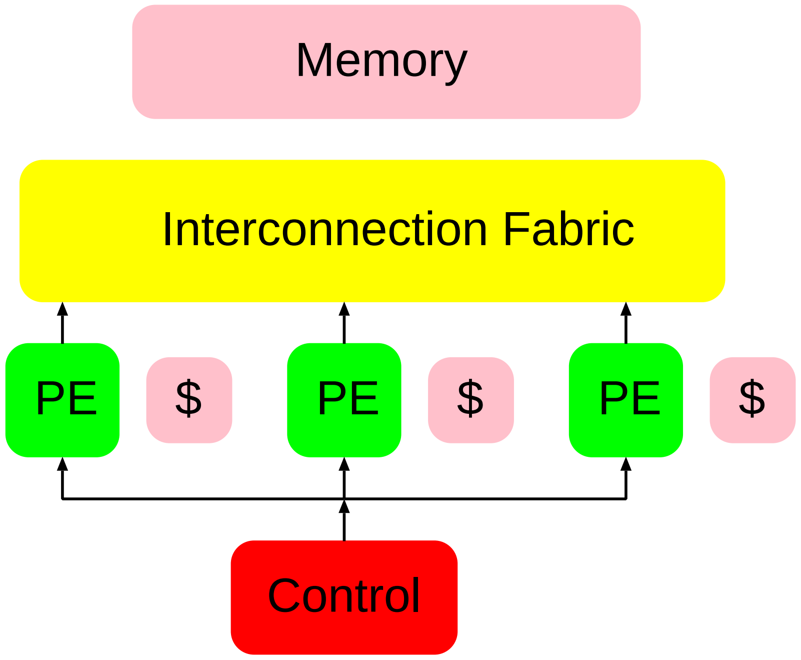
Single data is used as input at any clock cycle

Deterministic execution

Oldest type of a computer

(Later additions of pipelining and multiple functional units)

SIMD



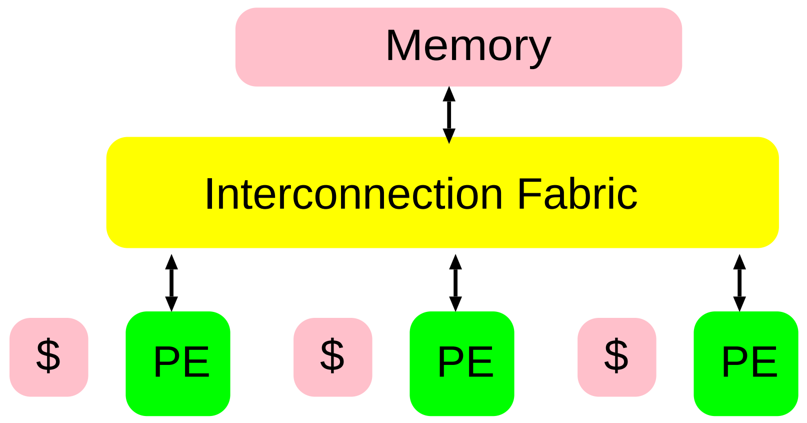
Each element of an array of data is acted upon by the same single instruction.

SIMD

P_1		P_2		P_3	
load	a(1)	load	a(2)	load	a(3)
load	b(1)	load	b(2)	load	b(3)
c(1)	= a(1) + b(1)	c(2)	= a(2) + b(2)	c(3)	= a(3) + b(3)
store	c(1)	store	c(2)	store	c(3)

- PEs execute same instructions at same cycles (lockstep execution).
- PEs operate on different data sets organized into a common structure
- Specialized for problems with high degree of regularity.
- Connection Machine, MasPar, ILLIAC IV (by now long gone)
- GPUs (Nvidia, AMD), Cell (Sony Play Station), etc.

Shared memory MIMD



Every PE may be executing a different instruction.

Shared memory MIMD

P_1		P_2		P_3	
load	a(1)	load	a(2)	store	a(3)
load	b(1)	load	b(2)	load	a(2)
c(1)	= a(1) + b(1)	c(2)	= a(2) * b(2)	a(3)	= a(2) - b(3)
store	c(1)	c(2)	=c(1) + c(2)	store	a(3)

- Synchronous or asynchronous execution.
- SMPs, multi-core PCs.

Shared memory architecture

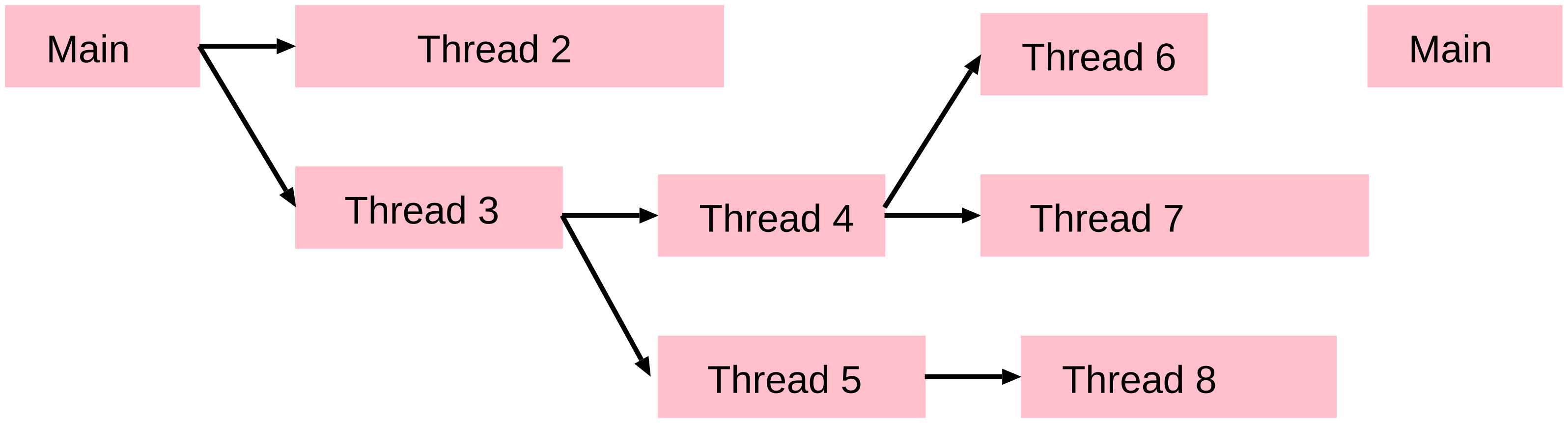
- Common global address space visible to all PEs.
- Communication via shared data.
- Shared data needs to be protected by ”locks” to avoid **race conditions**.
- Cache coherence mechanism required.

Shared memory threads model

A program is divided into multiple concurrent streams (also called threads).

- **Main program** (master thread) asks the OS for all resources it needs to run.
- Master may perform some serial work, and then creates a number of tasks (slave threads) that the OS can run concurrently.
- Each thread may create new threads (its slaves).
- Each thread has local data but also shares resources of its masters.
- Threads communicate through global memory.
- Threads can be terminated and created until the master completes its job.

Shared memory threads model



Things may go wrong if not treated properly.

Simple example - race condition

global int s = 0; global int a[0:3] = {1,2,3,4}	
thread 1	thread 2
for i = 0:1	for i = 2:3
s = s + a[i]	s = s + a[i]

- load and store are atomic, cannot be interrupted
- computations occur in local (to thread) registers

Sample trace:

Reg0 = a[0]	Reg0 = a[2]
Reg1 = s	Reg1 = s
Reg1 = Reg0 + Reg1	Reg1 = Reg0 + Reg1
s = Reg1	**s = Reg1**

Shared memory

Reg0 = a[0]
Reg1 = s
Reg1 = Reg0 + Reg1
s = Reg1
	Reg0 = a[2]
	Reg1 = s
	Reg1 = Reg0 + Reg1
	s = Reg1

Reg0 = a[1]
Reg1 = s
$\boxed{\text{Reg1} = \text{a[0] } \textbf{or} \text{ a[2]}}$
Reg1 = Reg0 + Reg1
s = Reg1
	Reg0 = a[3]
	Reg1 = s
	$\boxed{\text{Reg1} = \text{a[2] } \textbf{or} \text{ a[0]}}$
	Reg1 = Reg0 + Reg1
	s = Reg1

Need additional constructs to enforce correctness.

Shared memory

Advantages to programmer

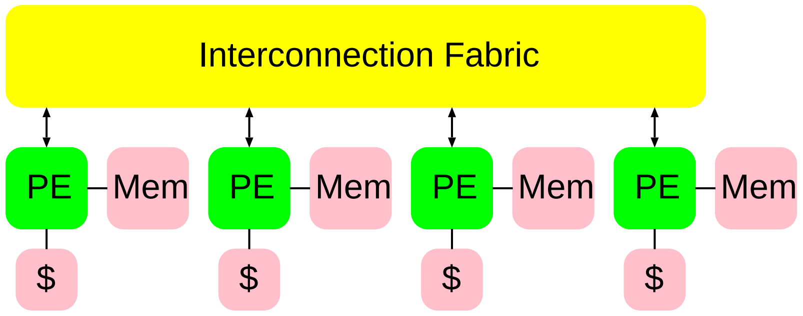
- global address space
- communication by shared variables

Disadvantages

- not scalable, more CPUs increase traffic on the memory-CPU path
- increased cache/memory management time
- programmer responsible for maintaining correct access of shared data

Expensive for a large number of processors.

Distributed memory MIMD



- local memories only, no global address space
- local memory store has no effect on another memory (no cache coherency required)
- programmer explicitly controls how and when remote data is communicated (needs to know `id`s of all PEs).
- complex networks are used but can be as simple as Ethernet

Message passing MIMD

Data is moved to and from PEs by sending and receiving **messages**,

- blocking - sender waits for acknowledgment
- or unblocking - can proceed with processing.

MPI is the standard for Message Passing Interface.

Distributed memory

Advantages:

- memory scale with number of PEs
- no overhead for maintaining cache coherency

Disadvantages:

- programmer is responsible how and when data is communicated between PEs
- difficult to map efficiently existing data structures to distributed memory
- NUMA access times

More definitions

Granularity: Task's computation to communication time ratio.

- **Coarse:** "large" amount of computational work is done between communication events - often distributed memory
- **Fine:** "small" amounts of computational work is done between communication events - often shared memory

Which one is better?

Performance metrics

Speed-up: Ratio of sequential to parallel execution times with n PEs,

$$s_n = \frac{t_{seq}}{t_{par}}$$

Theoretically, ideal speed-up for n PEs is bounded by n.

Can $s_n > n$?

Performance metrics

$s_n > n$ in some situations like

- parallel task may follow different execution path
- cache effects
  - for large data set a single PE may miss on cache often
  - if there are many PEs with own caches misses occur less often

Efficiency: how well the resources are utilized, $e_n = \frac{t_{seq}}{n \cdot t_{par}}$.

Scalability: tells whether the algorithm can handle a growing amount of work or PEs efficiently.

We can increase either the number of processors, the size of the problem or both, and ask questions like

- can increase of the number of processors increase the speed-up?
- can increase of the problem size increase the speed-up?
- can a parallel system keep efficiency by increasing the number of processors and the problem size simultaneously?
- at what rate do we need to increase the problem size and the number of processors to maintain constant efficiency?

Amdahl's law

Estimate of possible speed-up on n PEs.

Assumptions

- a program consists of T "elementary" ops
- "elementary" ops all require one clock cycle to execute
- $\alpha \cdot T$ ops can be executed concurrently on n PEs
- $(1 - \alpha) \cdot T$ ops must be executed sequentially
- there is no overhead for memory access or communication

Derive upper bound on s_n.

Amdahl's law

$$\begin{aligned} t_{seq} &= T, \;\; t_{par}(n) = \alpha \frac{T}{n} + (1-\alpha)T \\ \lim_{n\to\infty} s_n &= \lim_{n\to\infty} \frac{1}{\frac{\alpha}{n} + (1-\alpha)} = \frac{1}{1-\alpha} \end{aligned}$$

Amdahl's Law: The speedup is bounded by the fraction of sequential code.

	$n = 2$	$n = 4$	$n = 16$
$\alpha = 0.5$	1.33	1.6	1.882
$\alpha = 0.7$	1.538	2.105	2.909
$\alpha = 0.9$	1.818	3.076	6.4

Need small α

- analogues to small miss rate for caches
- bad news ?

Chickens agains ox

When there is need to process larger jobs more resources must be provided.

One can either build a larger machine or add more small machnes.

Which approach is better?

If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?

Seymour Cray, the father of supercomputing.

Amdahl's law

Consider program A with fraction α of concurrent ops, and fraction $(1-\alpha)$ of sequential ops. Say A executes in time

- t_{fast} on a fast PE P_{fast}
- t_{slow} on a slow PE P_{slow}

and that $\frac{t_{fast}}{t_{slow}} = f$.

Say $\alpha = 0.9$.

For what value of f and n, is it better to use a system with slow PEs over a single fast PE?

Amdahl's law

Possible speed-up with n slow PEs over the fast PE is

$$s_n = \frac{t_{fast}}{(1-\alpha)t_{slow} + t_{slow}\frac{\alpha}{n}} = \frac{1}{\frac{(1-\alpha)}{f} + \frac{\alpha}{f \cdot n}}$$

s_n is an increasing function of n, so

$$s_{n-1} < s_n < s_\infty = \frac{f}{1-\alpha} = \frac{f}{0.1}$$

We have

$$s_n < 1, \;\; \forall\, f = \frac{t_{fast}}{t_{slow}} < 0.1$$

so P_{fast} always better then n P_{slow} when $t_{fast} < 0.1 \cdot t_{slow}$.

So why massively parallel machines?

Parallel machines are for

- speeding-up a particular computation,
- solving larger and larger problems.

Invert the question:

How would a single PE perform if it were rquired to solve the problem solved by n PEs?

So why massively parallel machines?

Assumptions:

- time t_{par} on multiprocessor with n PEs
- α fraction of time spent running on all PEs
- $(1-\alpha)$ fraction of time running on a single PE

$$\begin{aligned} t_{seq} &= \alpha \cdot n \cdot t_{par} + (1-\alpha) \cdot t_{par} \\ s_n &= \frac{t_{seq}}{t_{par}} = (1-\alpha) + \alpha \cdot n \to \infty \end{aligned}$$

Good news ?

So why massively parallel machines?

Assuming there is no communication cost we established that:

- Amdahls Law - speed-up is determined by the fraction α of parallel code

$$s_n \leq \frac{1}{1-\alpha}$$

- The choice between "oxens" and "chickens" depends on the ratio $f = \frac{t_{fast}}{t_{slow}}$ and the value of α

$$s_n < \frac{f}{1-\alpha}$$

- If n PEs execute in time t_{par} with α fraction of parallel code then a single PE executes in time

$$t_{seq} = \alpha \cdot n \cdot t_{par} + (1-\alpha) \cdot t_{par}$$

PRAM

Parallel Random Access Memory

- P processors
- all memory access free
- all basic arithmetic operations cost 1 unit
- parallel step - number of ops performed simultanously
- cost of a parallel algorithm is the number of parallel steps

Impossible to realize but

- good for analysis of algorithms
- provide upper bounds on possible speed-up

PRAM - binary summation

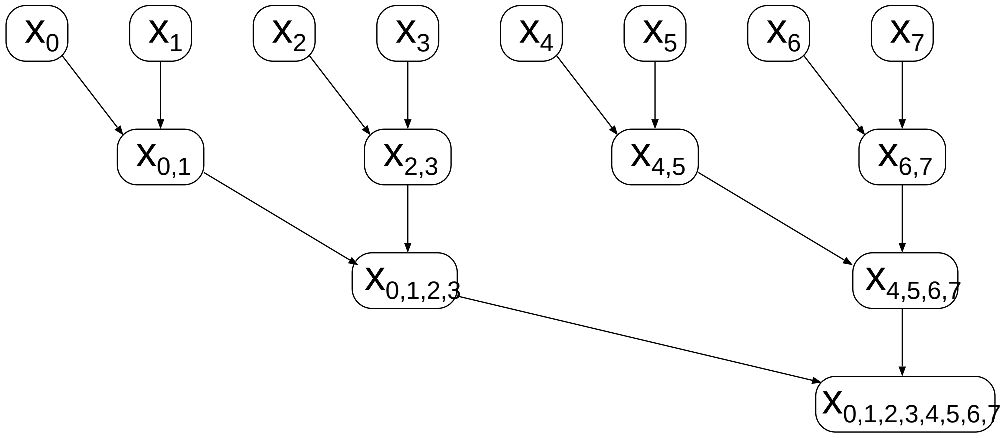
$$s = \sum_{i=1}^{2^n} F_i$$

# flops ?

PRAM - binary summation

$$s = \sum_{i=1}^{2^n} F_i$$

```
for $j = 1 : n$
    forall $k = 1 : 2^{n-j}$
        $F[2^j k] = F[2^j k - 2^{j-1}] + F[2^j k];$
```



# flops ?, speed-up s_N ?, speed-up s_P ?

Performance metrics

For adding $N = 2^n$ numbers on $N/2$ PEs

$$E_N = \frac{t_{seq}}{N \cdot t_{par}} \approx \frac{1}{\log_2 N}$$

This process is not scalable.

For adding n numbers on p processors

$$E_p = \frac{n-1}{p(\frac{n-p}{p} + \log_2 p)} \approx \frac{n}{n + p\log_2 p}$$

By choosing $n = p\log_2 p$ we keep efficiency constant.

PRAM - parallel prefix

$$s_k = \sum_{i=1}^{k} F_i \,, \;\; k = 1, ..., 2^n$$

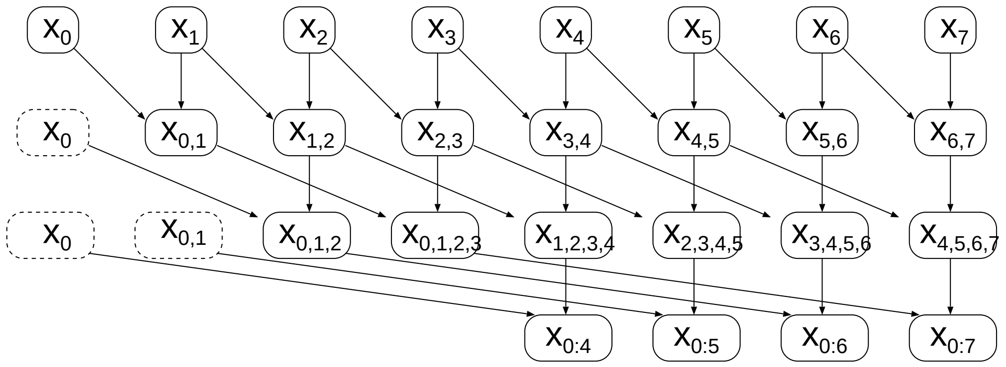
$$\begin{aligned}
s_1 &= F_1 \\
s_2 &= F_1 + F_2 \\
s_3 &= F_1 + F_2 + F_3 \\
&\vdots \\
s_m &= F_1 + F_2 + F_3 + \cdots + F_m
\end{aligned}$$

# flops ?

PRAM - parallel prefix

$$s_k = \sum_{i=1}^{k} F_i\,,\;\; k = 0, 1, ..., 2^n$$

```
for $j = 0 : n - 1$
    forall $k = 2^j : n$
        $F[j+1, k] = F[j, k] + F[j, k - 2^j];$
```



# flops ?, speed-up s_N ?, speed-up s_P ?

PRAM - parallel prefix

Binary tree and prefix computation apply to any associative operation like:

- $F_0 \cdot F_1 \cdot F_2 \cdots F_{2^n - 1}$
- $\max(F_0, F_1, F_2, \ldots, F_{2^n - 1})$

PRAM - linear recurrence

Fibonacci sequence:

$F_0 = 0,\ F_1 = 1,$

$$F_n = F_{n-1} + F_{n-2}$$

Can it be parallelized?

PRAM - linear recurrence

Another way of writing the recursion is

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$$

Now use the parallel-prefix scheme.

Is it true for any

- linear recurrence?
- nonlinear?

Next time

Scheduling parallel tasks