COMP 633 Parallel Computing
Fall 2021

http://www.cs.unc.edu/~prins/Classes/633/

Parallel computing

- What is it?
  - multiple processors cooperating to solve a single problem
  - hopefully faster than using a single processor!

- Why is it needed?
  - greater compute performance

Where is performance needed?

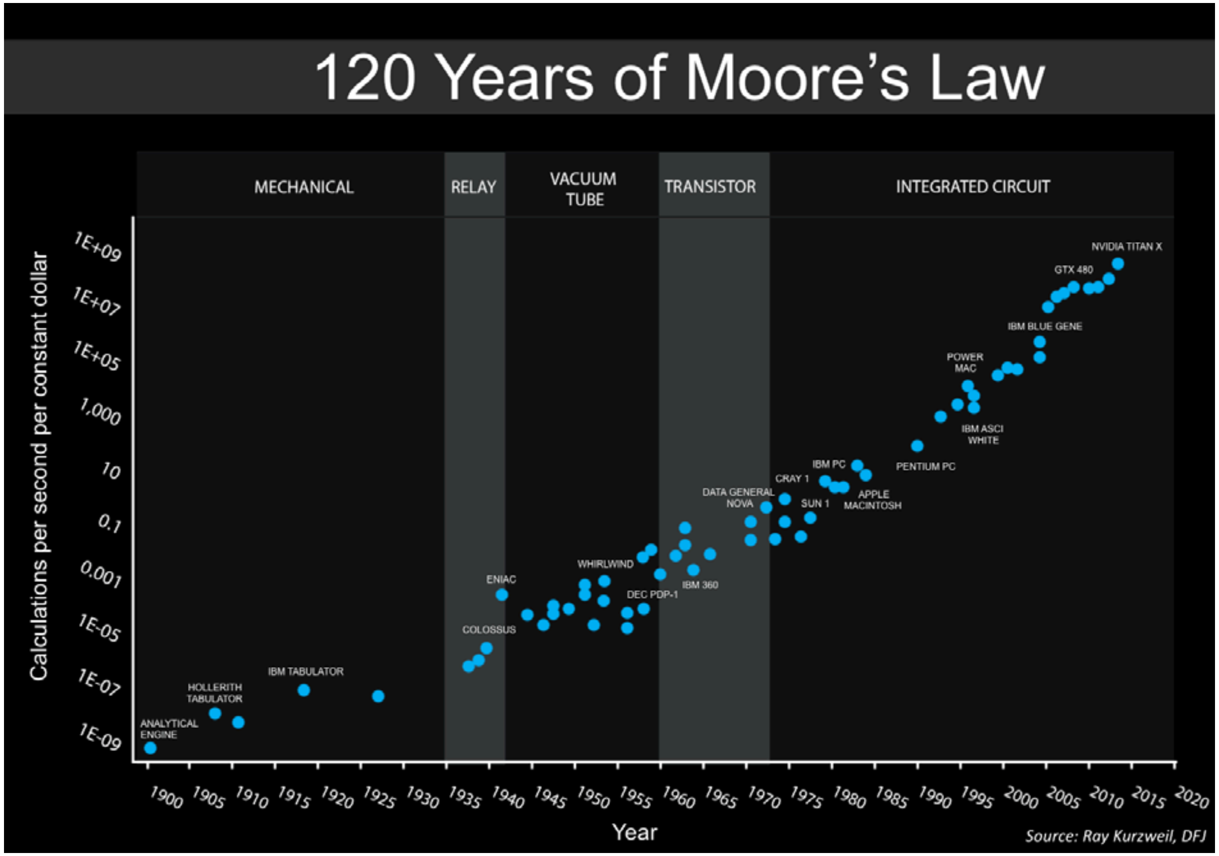
- sometimes performance is required in time-critical tasks
  - timely and accurate weather forecast
  - obstacle detection for self driving cars

- sometimes performance gives a competitive advantage
  - from Walmart to Wall Street
    - data mining of trends
    - delivery logistics
    - real-time analytics (high frequency trading)
  - engineering, manufacturing, and pharmaceuticals
    - vehicle crash simulations, material properties prediction, drug design

- sometimes performance is the only way to answer a question
  - scientific progress using mathematical modeling and numerical simulation
    - human genome assembly
    - computational science and the timely Nobel prize

Why can't we just build a faster single processor ?

- Moore's "Law"
  - processor performance per $ doubles every two years !

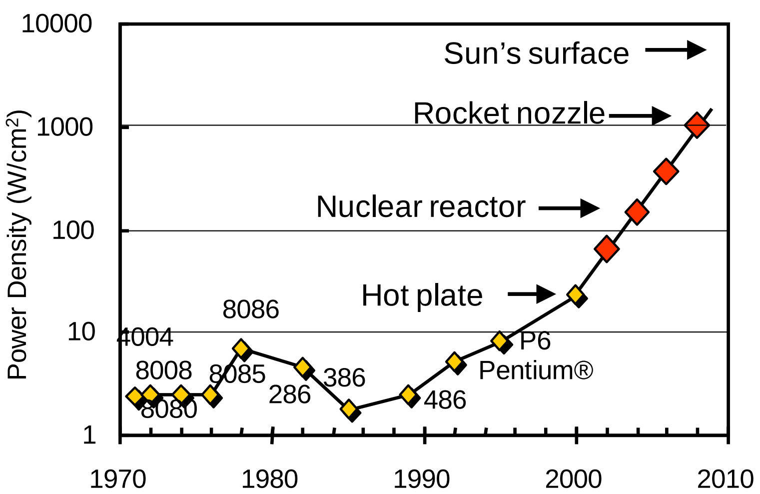


Transistor miniaturization and performance

- Dennard scaling
  - transistor switching power ∝ transistor size
  - shrinking transistor size
    - decreases switching power
    - decreases switching time (higher clock frequency)
    - increases number of transistors per unit area
  - so for the same power and space budget we get
    - faster arithmetic operations
    - pipelined arithmetic
    - more and larger caches

    ⇒ increased performance

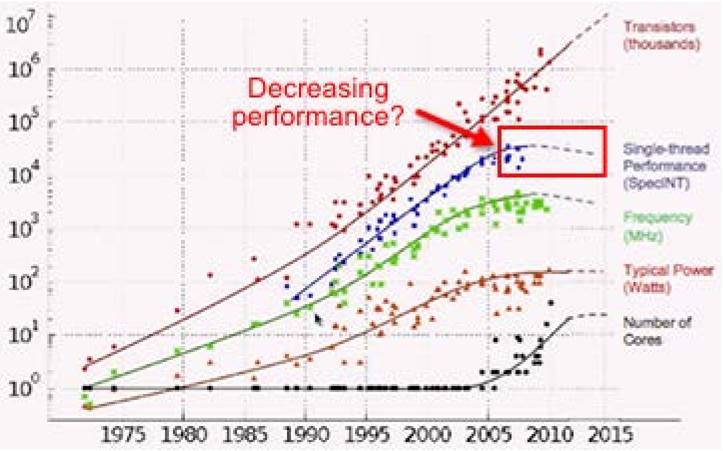
- Limits to Dennard Scaling
  - as transistor size approaches quantum mechanical limits
    - increasing leakage current
    - exponential power increase!



Source: Patrick Gelsinger, Intel®

Parallelism is now the principal source of performance

- Processor evolution after 2004 (Intel)
  - **multiple cores per socket**
  - lower per-core performance
  - similar power per chip
    - per-core “turbo” mode
  - vector units and larger caches
  - multiple and higher performance off-chip memory interfaces



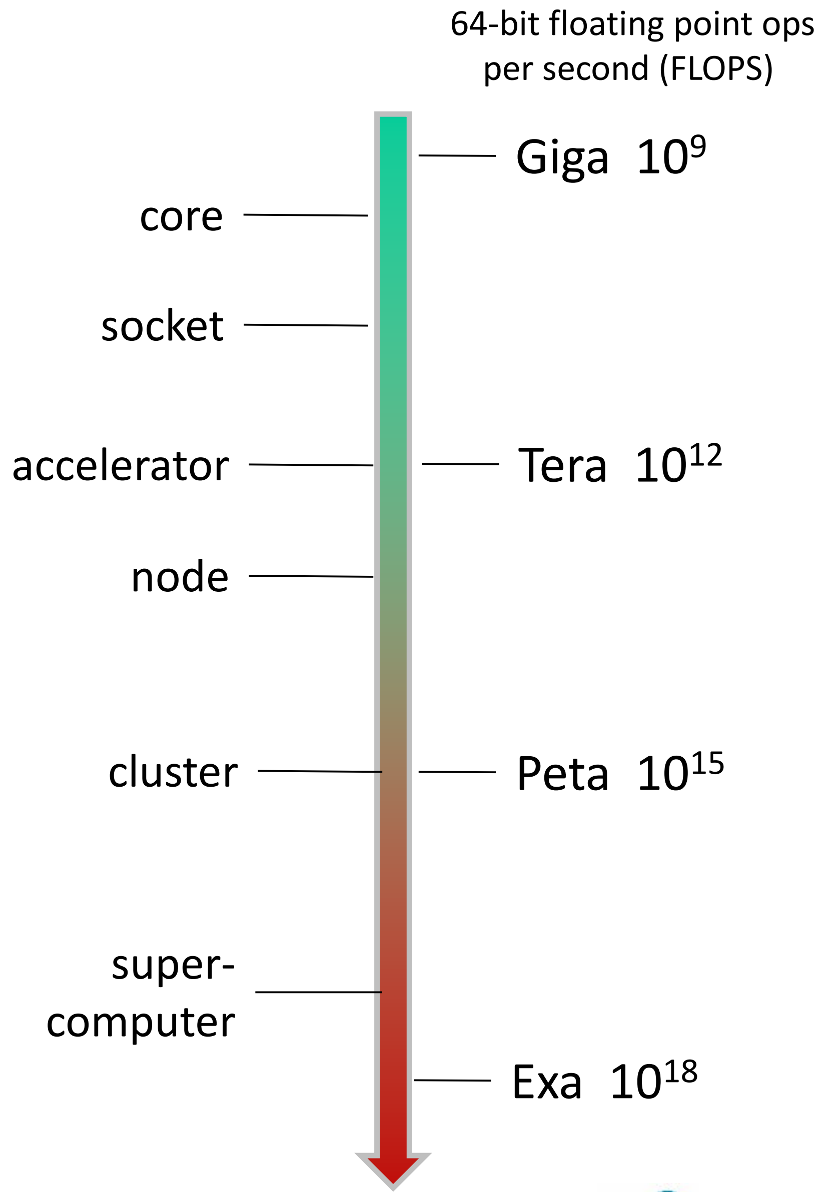
processor performance characteristics

- Moore’s “law”
  - performance per socket is still increasing but no longer exponentially
  - power/cooling per socket is the limiting factor

- Factors limiting parallel computing
  - overall system power
  - inconveniently slow speed of signal propagation!

Parallel computing at various scales

- Modern processor core
  - pipelined, superscalar, multiword ALUs
  - L1 and L2 caches
- Socket
  - multiple cores (4 – 64)
  - L3 cache
- Accelerators
  - Nvidia V100 GPU (2560 arithmetic units)
- Node
  - up to 4 sockets
  - up to 8 accelerators
  - fast local interconnect
- Cluster
  - tens to thousands of nodes
  - high speed interconnection network



Top supercomputers (2020)

Sunway TaihuLight
National Research Center for Parallel Computer Engineering and Technology in Wuxi, CN

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
4	**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

What are the parallel computing challenges?

- Parallel computing involves many aspect of computer science
  - new algorithms must be designed
  - new algorithm analysis techniques must be used
  - new programming models and languages must be learned
  - memory operation and performance must be understood
  - communication costs and network behavior must be considered
  - different operating systems, services, and I/O
  - different debugging and performance monitoring
  - novel and continuously changing hardware
  - …

Summary: Why study parallel computing?

- It is useful and it is used
- It involves new algorithms and analytic techniques
- Future computing will increasingly be predicated on the use of parallelism
- To understand what is feasible and what is not

How else is parallelism used?

- Parallelism may improve reliability
  - high availability
  - high assurance
- Parallelism may be inherent in the problem
  - (G)UIs
  - distributed systems
    - >80 processors in a modern luxury car
- Parallelism is a simple load scaling approach
  - server farms

… but these are not the focus of this course!

Parallel Computing vs. Distributed Computing

- Parallel Computing (COMP 633)
  - Multiple processors cooperating to solve a single problem
  - Key concepts
    - Design and analysis of scalable parallel algorithms
    - Programming models
    - Systems architecture and hardware characteristics
    - Performance analysis, prediction, and measurement

- Distributed Systems (COMP 734)
  - Providing reliable services to multiple users via a system consisting of multiple processors and a network
  - Key concepts
    - Services & protocols
    - Reliability
    - Security
    - Scalability

Parallel Computing vs. Concurrent Algorithms

- Parallel Computing (COMP 633)
  - Multiple processors cooperating to solve a single problem
  - Key concepts
    - Design and analysis of scalable parallel algorithms
    - Programming models
    - Systems architecture and hardware characteristics
    - Performance analysis, prediction, and measurement
- Distributed and Concurrent Algorithms (COMP 735)
  - Specification of fundamental algorithms and proofs of their correctness and performance properties
    - Mutual exclusion
    - Readers and writers
  - Key concepts
    - Lower and upper bounds, impossibility proofs
    - Formal methods
    - Wait-free and lock-free methods

Course Introduction

- Organization and content of this course
  - prerequisites
  - source materials
  - course grading
  - what will be studied

- Introductory examples

Organization of the course

- Course web page
  - Syllabus
    - Prerequisites
    - Learning Objectives
    - Honor Code
    - Topics
  - Online discussion - Piazza
  - Source materials – reading assignments
  - Assignments and grading
  - Computer usage

- Reading assignment for next time
  - Parallel Random Access Machine (PRAM) model and algorithms
    - sections 1, 2, 3.1 (pp 1-8)

- Sign up for Piazza
  - using link on web page

What will we study?

- Course is organized around different models of parallel computation
  - shared memory models [main focus]
    - PRAM
    - Loop-level parallelism, threads, tasks (OpenMP, Cilk)
    - Accelerators (Cuda)
  - distributed memory models [secondary focus]
    - bulk-synchronous processing (BSP, UPC), message passing (MPI)
  - data-intensive models [cursory treatment]
    - MapReduce/Hadoop, spark

- For each model we examine
  - algorithm design techniques
  - cost model and performance prediction
  - how to express programs
  - hardware and software support
  - performance analysis
  - advantages and limitations of the model including realism, applicability and tractability

  by studying some examples in detail

Let's try it right now!

- Vector summation
  - given vector V[1..n] compute $s = \sum_{i=1}^{n} V_i$

    e.g. for n = 8

    $s = V_1 + V_2 + ... + V_7 + V_8$

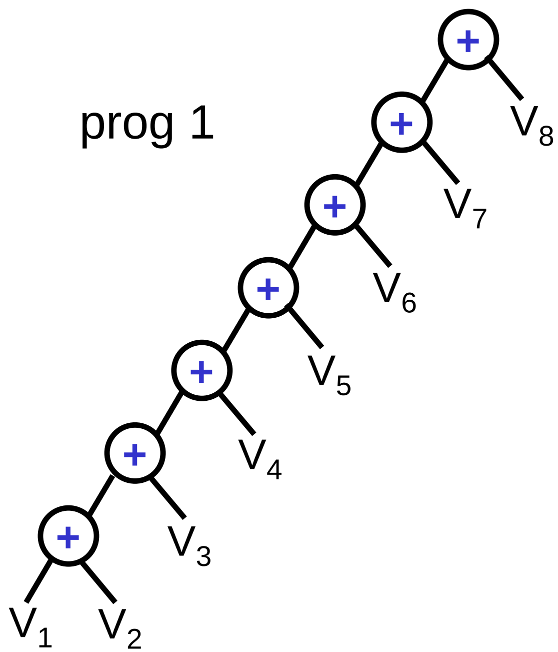
- sequential algorithm
  - n-1 additions: optimal
    - e.g. sum from left to right
  - sequential running time
    - $T(n) = O(n)$

Example 1: DAG model of parallel computation

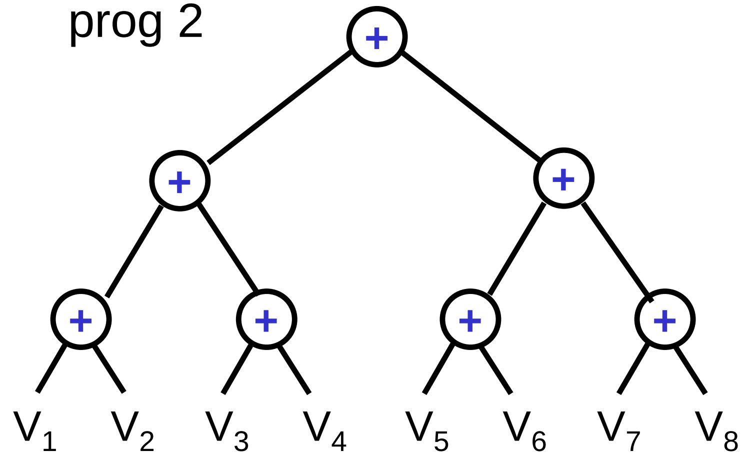
- A program P = (V, E) is a tree where

leaf vertices in V	~	values
interior vertices in V	~	operations
edges E	~	evaluation dependences

prog 1



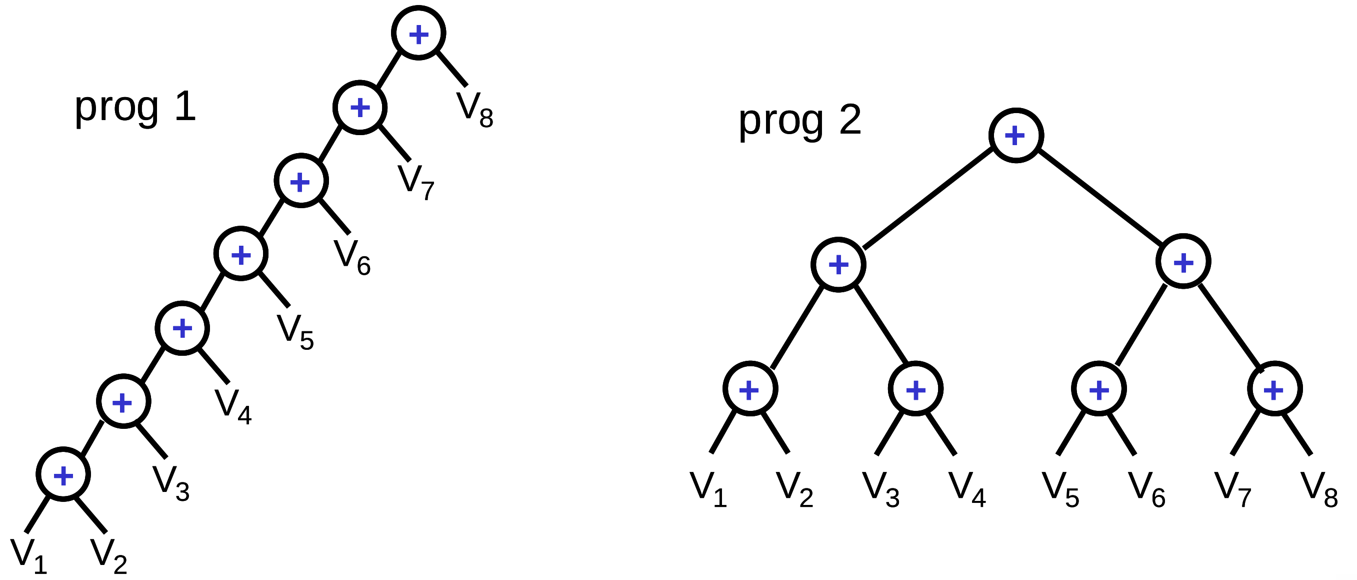
prog 2



$$V_1 + V_2 + V_3 + V_4 + V_5 + V_6 + V_7 + V_8$$

Execution of a DAG "program"

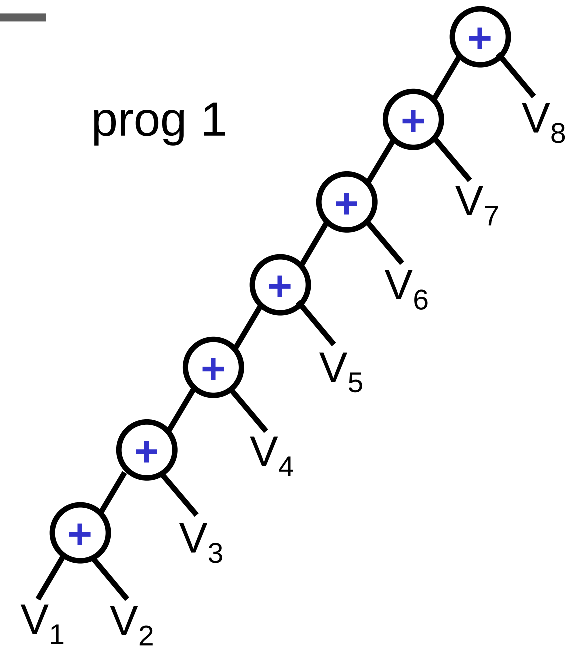
- definition
  - an operation is ready if all of its children are leaves
- parallel execution step
  - simultaneously evaluate all ready operations and replace each with its value
- program execution
  - perform parallel execution steps until no operations remain



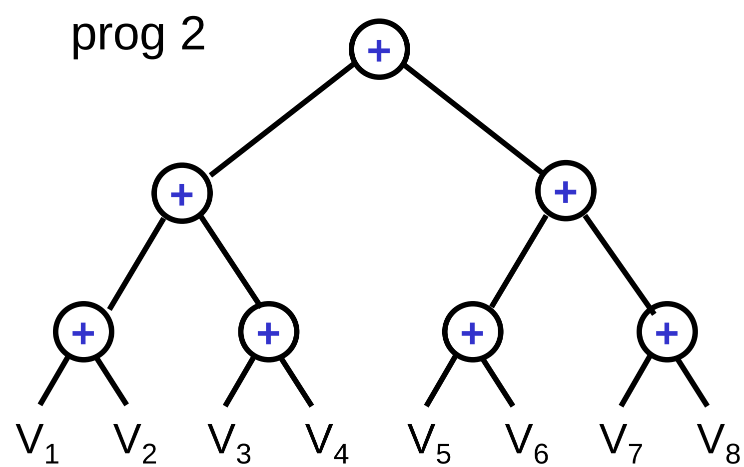
Complexity metrics for DAG model

- Work complexity of a DAG program
  - total number of operations performed
    - = # interior vertices in DAG

- Step complexity of a DAG program
  - number of execution steps
    - = length of longest path in DAG



	work	*steps*
Prog 1	7	7
Prog 2	7	3

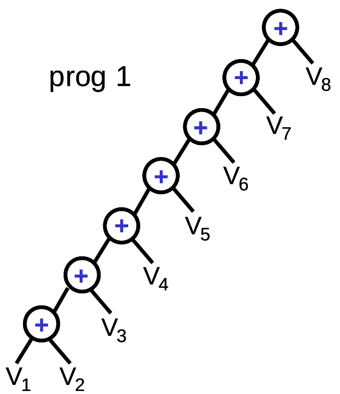


Asymptotic complexity metrics for DAG model

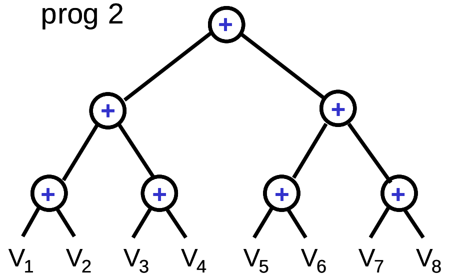
- Asymptotic complexity
  - problem size n
  - W(n) asymptotic work complexity
  - S(n) asymptotic step complexity
  - T*(n) optimal asymptotic sequential time complexity

- Definition
  - A DAG program is work efficient if W(n) = O(T*(n))

prog 1



prog 2

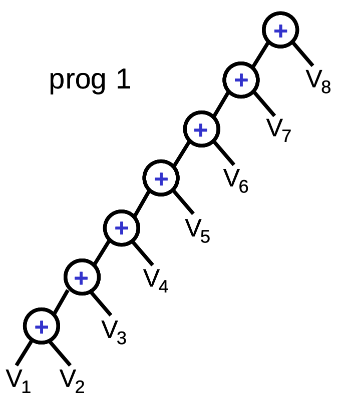


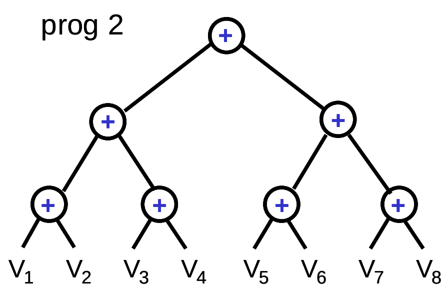
	W(n)	*S(n)*
Prog 1	O(n)	O(n)
Prog 2	O(n)	O(lg n)

Asymptotic complexity metrics for DAG model

- Asymptotic complexity
  - problem size n
  - W(n) asymptotic work complexity
  - S(n) asymptotic step complexity
  - T*(n) optimal asymptotic sequential time complexity

- Definition
  - A DAG program is work efficient if W(n) = O(T*(n))





	W(n)	***S(n)***
Prog 1	**O(n)**	**O(n)**
Prog 2	**O(n)**	**O(lg n)**

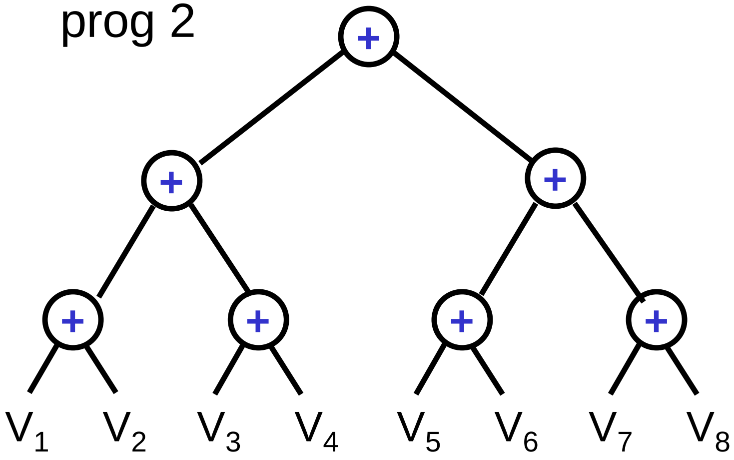
Execution of DAG programs with fixed resources

- At most p operations evaluated simultaneously in a DAG program H
  - models execution using p "processors"

- Definition
  - $T_p(n)$ is the time to execute H using p processors
    - n - problem size
    - p - maximum number of nodes that may be evaluated concurrently in each timestep
  - $T_1(n) = W(n)$
  - $T_\infty(n) = S(n)$

But what is $T_2(8)$ for prog 2?

prog 2



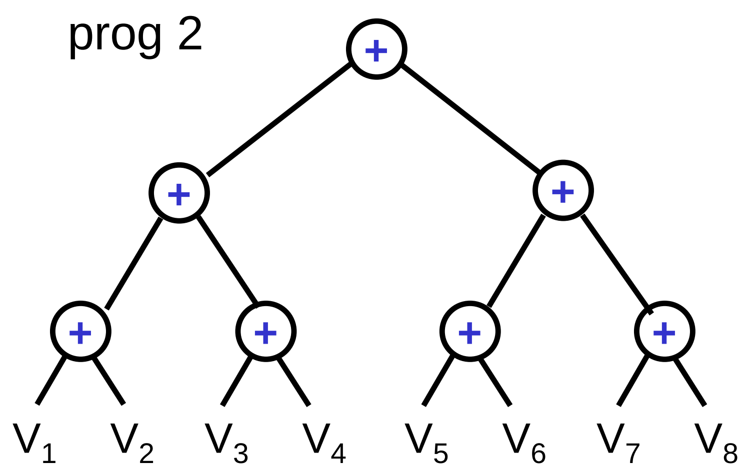
Evaluation order

- Determining evaluation order to minimize $T_p(n)$ is NP-hard!

- Simple non-optimal greedy evaluation order
  - at each step
    - p or fewer operations ready $\Rightarrow$ evaluate all ready nodes
    - more than p operations ready $\Rightarrow$ evaluate *any* p ready nodes

- Running time using greedy strategy can be bounded

$$\left\lceil \frac{W(n)}{p} \right\rceil \le T_p(n) \le \left\lfloor \frac{W(n)}{p} \right\rfloor + S(n)$$



"fast" parallel programs give good speedup

- Definition
  - a *fast* parallel program has step complexity S(n) that is asymptotically smaller than work complexity W(n)

$$S(n) = o(W(n)) \quad \text{means} \quad \lim_{n\to\infty} \frac{S(n)}{W(n)} = 0$$

- For a fixed number of processors *p*, a *fast* parallel program gives better *speedup* as problem size *n* is increased

$$\left\lceil \frac{W(n)}{p} \right\rceil \le T_p(n) \le \left\lfloor \frac{W(n)}{p} \right\rfloor + S(n)$$

$$\lim_{n\to\infty} T_p(n) = O\left(\frac{W(n)}{p}\right)$$

  - asymptotically *optimal speedup* on large problems!

But can't speedup indefinitely

- You can't speed up a parallel algorithm indefinitely using more processors
  - for a fixed problem size n, step complexity limits speedup

$$T_p(n) = O\left(\left\lfloor \frac{W(n)}{p} \right\rfloor + S(n) \right)$$

- prog 1 cannot be sped up at all using more processors!
  - W(n) = Θ(n)
  - S(n) = Θ(n)

- prog 2 requires Ω(lg n) steps regardless of the number of processors
  - W(n) = Θ(n)
  - S(n) = Θ(lg n)

Consequences: work efficiency is paramount

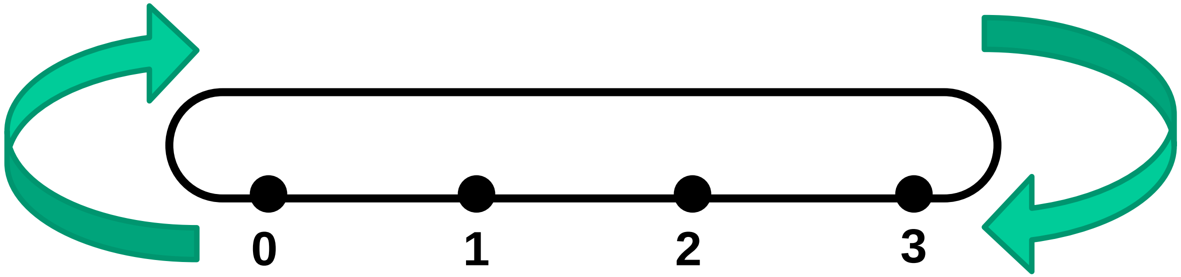
- A parallel program H that is *not* work efficient loses asymptotically!
  - for any given p, there exists a problem size n_0 such that
    - an efficient sequential program using one processor on problems of size $n > n_0$ is faster than the parallel program H using p processors!
  - it doesn't help if H is *fast*
  - worst results on large problems!

$$T_p(n) = O\left(\left\lfloor \frac{W(n)}{p} \right\rfloor + S(n)\right)$$

Example 2: Message-passing model

- p processors connected in a ring
  - each processor
    - runs the same program
    - has a unique processor id $0 \leq i < p$
    - can send a value to its left neighbor

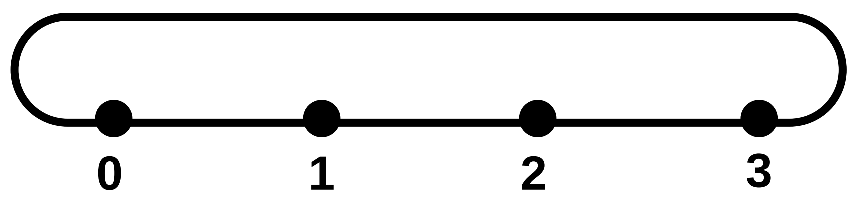
- summation of V[0..p-1] using p processors
  - assume V_i is in s on processor i at start
  - program terminates with $s = \sum_{j \in 0..p-1} V_j$ on processor 0



Summation program

```
for h := 1 to (lg p)
    x := s
    for j := 1 to 2^(h-1) do
        send value of x to left and receive new value for x from right
    s := s + x
```

Example: p = 4



s = V_0 V_1 V_2 V_3

h = 1, s = $V_0 + V_1$ $V_2 + V_3$

h = 2, s = $V_0 + V_1 + V_2 + V_3$

Analysis of summation program

```
for h := 1 to (lg p)
    x := s
    for j := 1 to 2^(h-1) do
        send value of x to left and receive new value for x from right
    s := s + x
```

- Let
  - t_a time to perform addition
  - t_c time to perform communication

$$T_p(n) = \sum_{h=1}^{\lg p} (t_a + 2^{h-1} t_c)$$
$$= (\lg p) \cdot t_a + (p-1) \cdot t_c$$

- Is this good performance?

What’s wrong?

- poor network?
  - network *diameter* is large thus values have to travel far
  - so communication time is huge compared to addition time
  - a smaller diameter network might do better
- bad communication strategy?
  - “cut-through” routing would be superior
- poor utilization of the processors?
  - only a few processors are performing useful additions!
- problem size too small?
  - this is the real problem!

Summation of n values with p processors

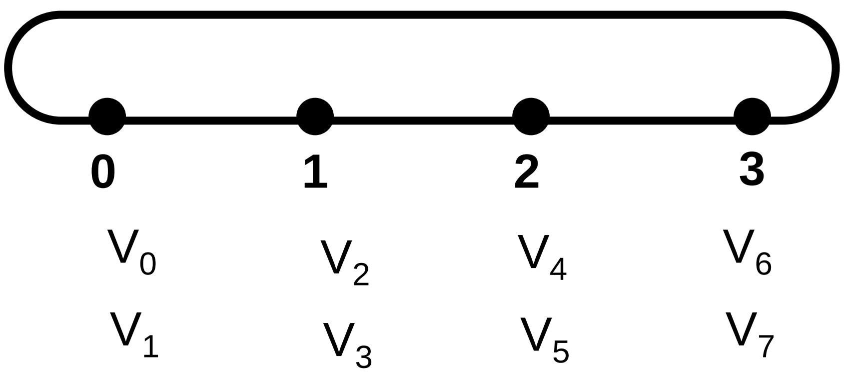
- Each processor holds n/p values

```
s := sum of n/p values in this processor
for h := 1 to (lg p)
    x := s
    for j := 1 to 2^(h-1) do
        send value of x to left and receive new value for x from right
    s := s + x
```

Example:

$n = 8$

$p = 4$



Summation of n values using p processors

- Analysis

$$T_p(n) = \left(\frac{n}{p} - 1\right)\cdot t_a + (\lg p)\cdot t_a + (p-1)\cdot t_c$$

$$\approx \underbrace{\left(\frac{n}{p}\right)\cdot t_a}_{\text{speedup}} + \underbrace{(\lg p)\cdot t_a + p\cdot t_c}_{\text{overhead}}$$

- excellent performance can be achieved
  - for arbitrary p, t_a, t_c
  - asymptotically optimal speedup with sufficiently large n
    - overheads and inefficiencies can be amortized!

For next week Tuesday

- read the PRAM handout
  - secns 1,2, 3.1 (pp 1-8)