NYU
New York University

Multicore Processors: Architecture & Programming

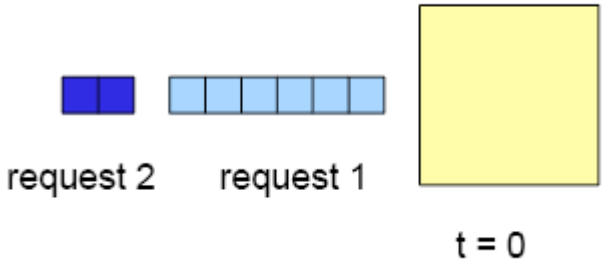
Concurrency and Parallelism

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

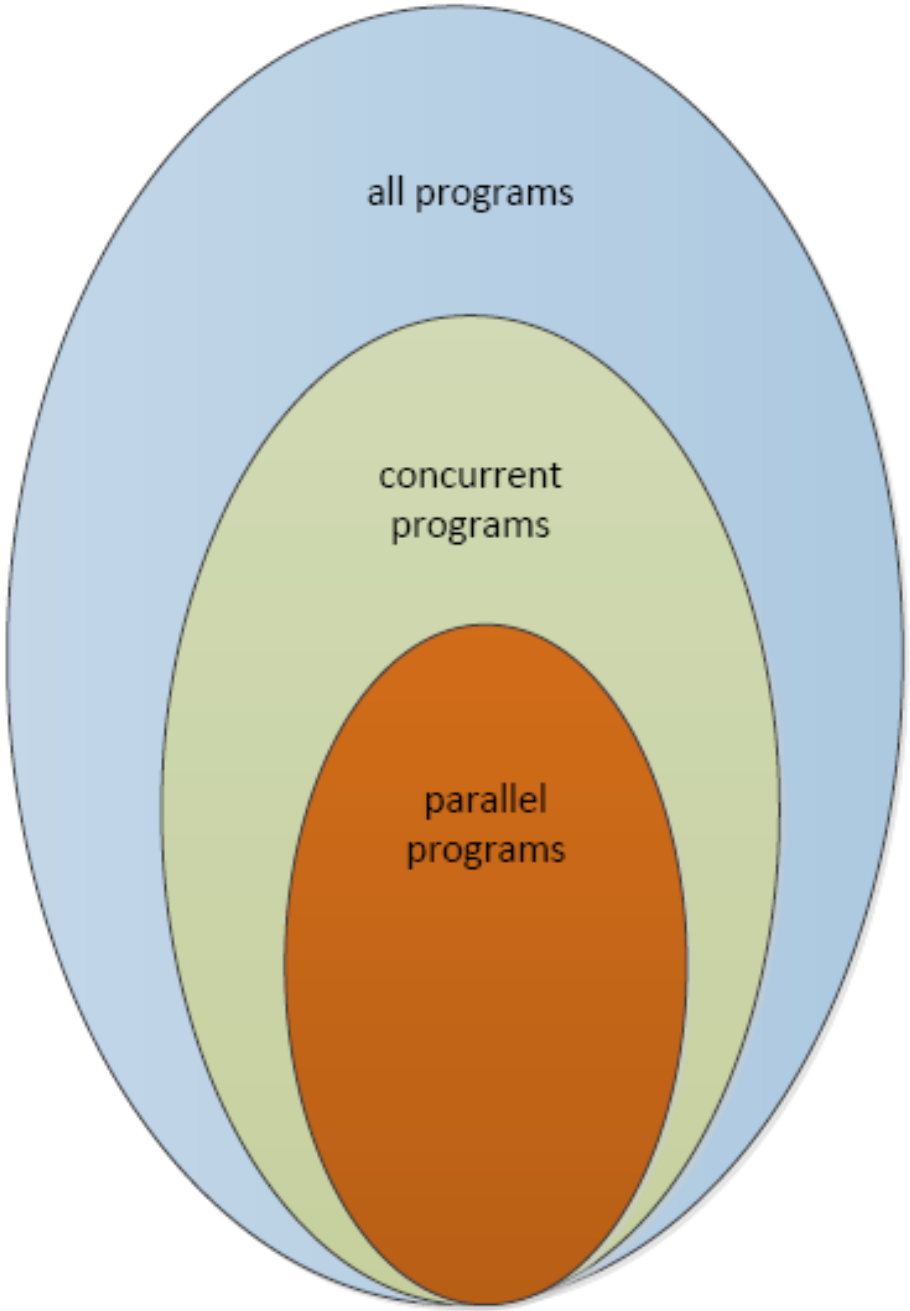
Same Meaning?

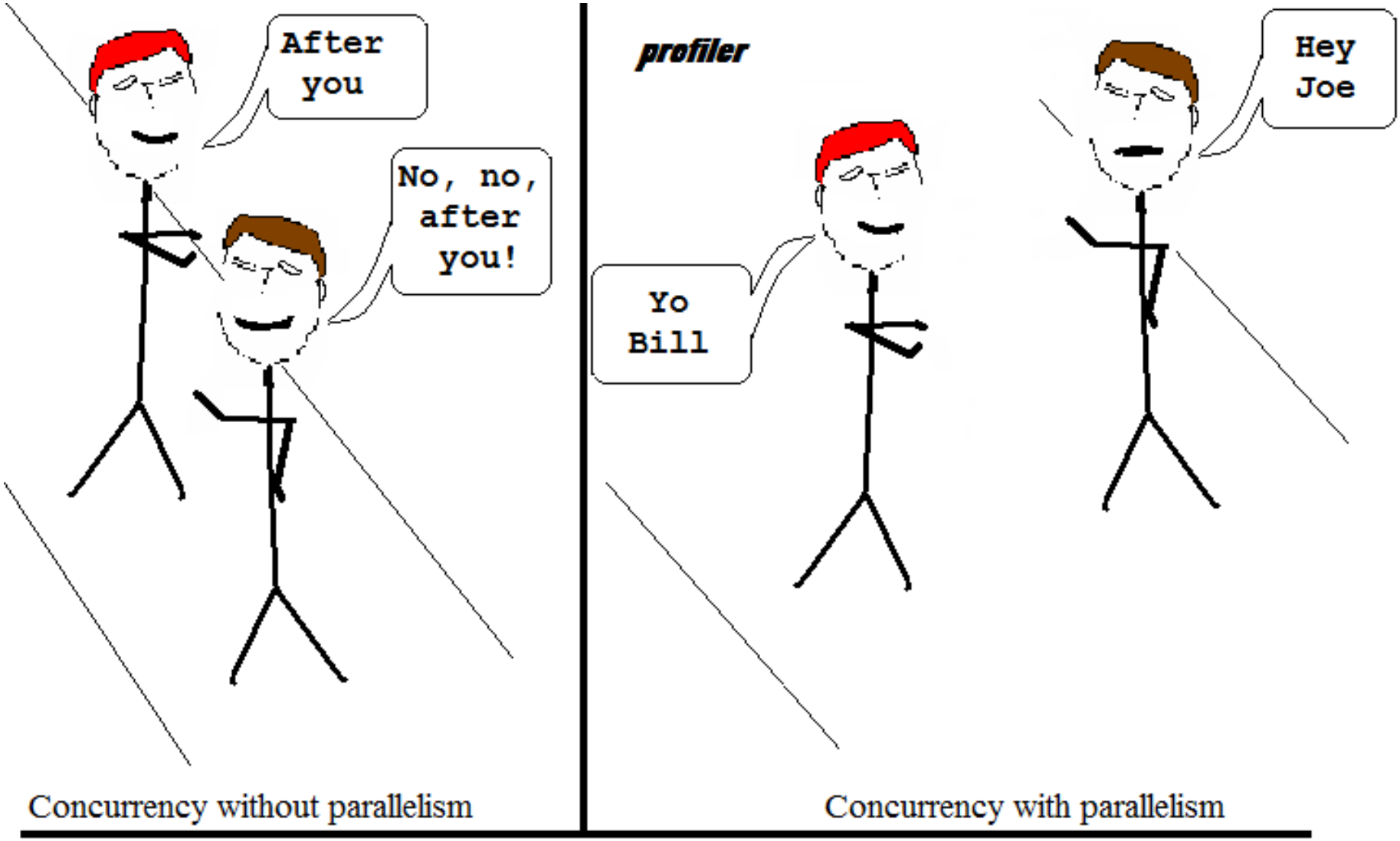
- Concurrency: At least two tasks are making progress at the same time frame.
  - Not necessarily at the same time
  - Include techniques like time-slicing
  - Can be implemented on a single processing unit
  - Concept more general than parallelism
- Parallelism: At least two tasks execute *literally* at the same time.
  - Requires hardware with multiple processing units

A Quick Example



This Figure shows a server receiving two requests at time 0. How will this server react if it is serial and if it is concurrent? [Hint: average completion time]


all programs
concurrent programs
parallel programs



Performance tuning technique number 106: Concurrency vs. Parallelism

Simply Speaking

Concurrency + Parallelism

=

High Performance

Very Important Paper

Read every single word of it!

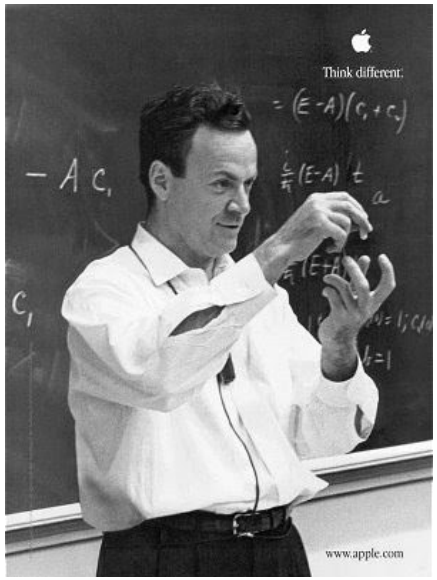
REVIEW SUMMARY

COMPUTER SCIENCE

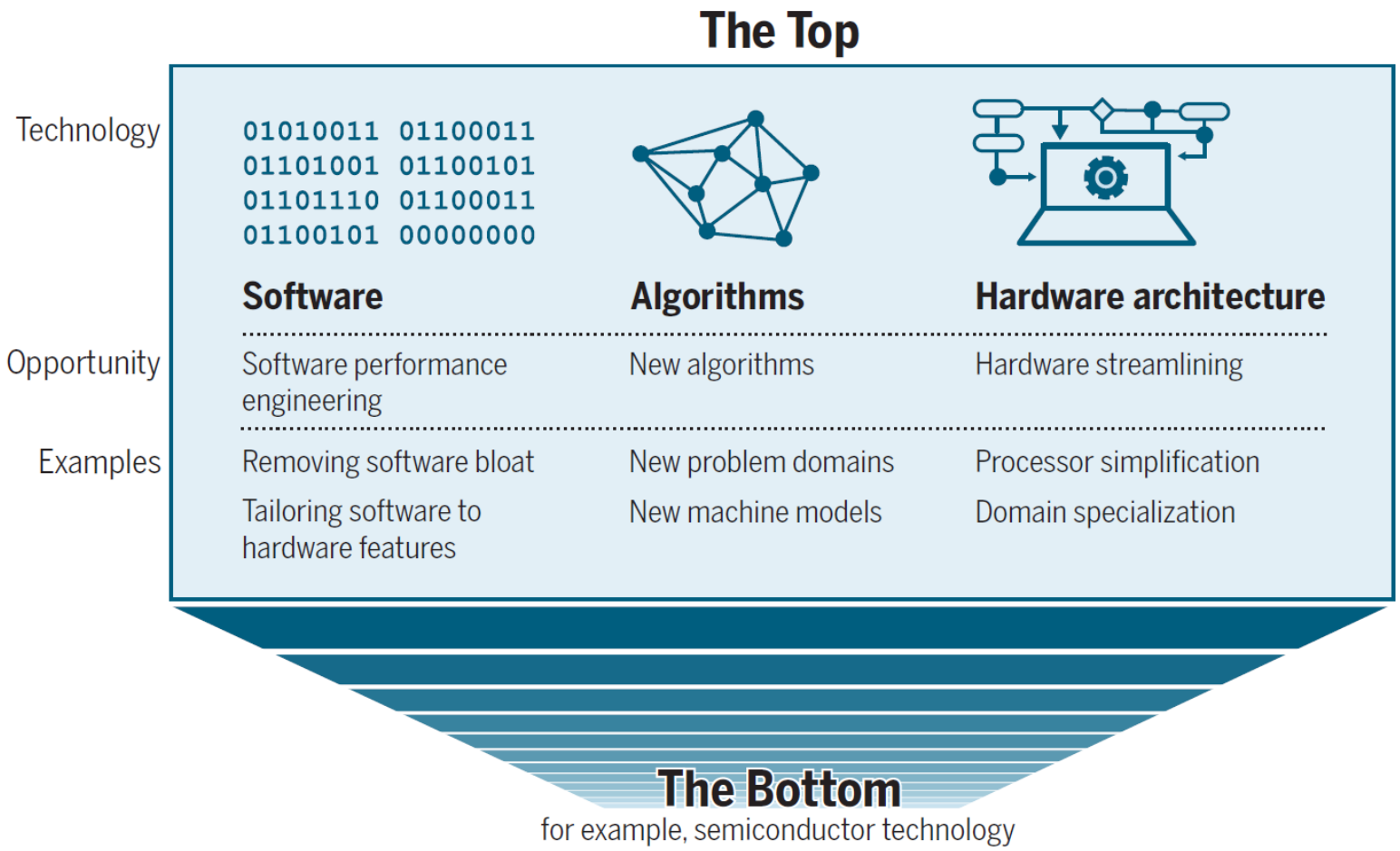
There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson, Neil C. Thompson*, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, Tao B. Schardl

Leiserson *et al.*, *Science* **368**, 1079 (2020) 5 June 2020



Richard Feynman:
"There is plenty of room at the bottom"



Performance gains after Moore's law ends. In the post-Moore era, improvements in computing power will increasingly come from technologies at the "Top" of the computing stack, not from those at the "Bottom", reversing the historical trend.

Main Points

- Unlike the historical gains at the bottom (Moore's law), gains from the top will be:
  - Opportunistic
  - Uneven
  - Sporadic
  - Subject to diminishing return

Main Points

- Most modern software systems contain a lot of opportunities for performance enhancement.
- To improve performance, programs will need to expose more parallelism and locality for the hardware to exploit.
- We need software performance programmers, not just programmers.
- Software performance engineers will need to collaborate with hardware architects.

Main Points

```
for i in xrange(4096):
  for j in xrange(4096):
    for k in xrange(4096):
      C[i][j] += A[i][k] * B[k][j]
```

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

But: code of #7 is 20x longer than code in #1

How about the Algorithms?

- Algorithm design requires human ingenuity → It is hard to anticipate advances.
- Much progress in algorithms will come from:
  - Attacking new problem domains
  - Addressing scalability concerns
  - Tailoring algorithms to take advantage of modern hardware.
- In post Moore's era, it will be essential for algorithm designers and hardware architects to work together to find simple abstractions that designers can understand, and that architecture can implement efficiently.

Problems with current software

- Trading off efficiency for other traits such as coding ease.
- Failure to tailor code to the underlying architecture.

Simple code tends to be slow. Fast code tends to be complicated.

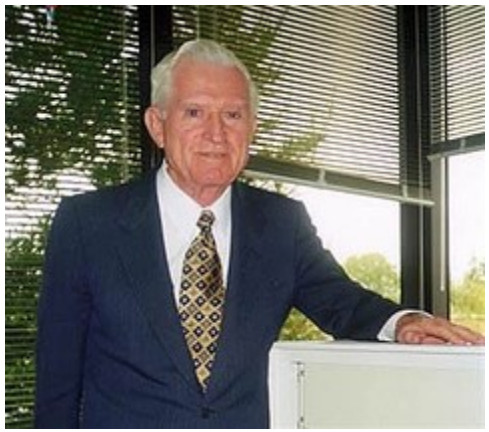
- In a world where it is easy to write fast code:
  - Application programmers must be equipped with the knowledge and skills to performance engineer their code.
  - Productivity tools to assist must be improved considerably.

Questions!

If we have as much hardware as we want, do we get as much parallelism as we wish?

For example:
If we have 2 cores, do we get 2x speedup?

Amdahl's Law

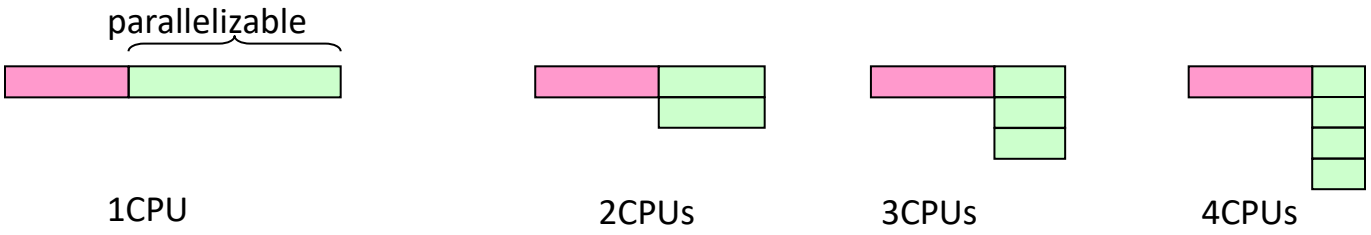


Gene M. Amdahl

- How much of a speedup one could get for a given parallelized task?

If F is the fraction of a calculation (or program) that is sequential, then the maximum speed-up that can be achieved by using P processors is

1 / [F+(1-F)/P]



Derivation of Amdahl's Law

- **Tseq** = Time taken to execute the program on a sequential machine.
  - Depends on the architecture of the hardware, the problem size, and the algorithm.
- **Tpar** = Time taken to execute the program on a machine with p processors.
  - Depends on the parallel architecture of the hardware, the problem size, and the parallel algorithm.
- **F** = Fraction of the program that is sequential.
  - Tpar = (F*Tseq) + (1-F)(Tseq/p)
- **Speedup** = Tseq/Tpar = 1/[F + (1-F)/p]

What Was Amdahl Trying to Say?

- Don't invest *blindly* on large number of processors.
- Sometimes having faster cores, to finish the sequential part fast, makes more sense than having many cores.

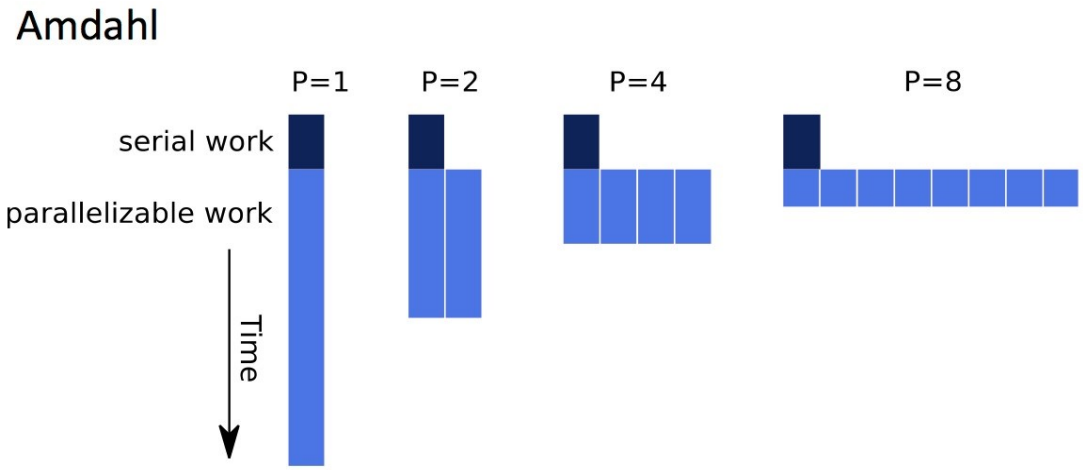
Was he right?

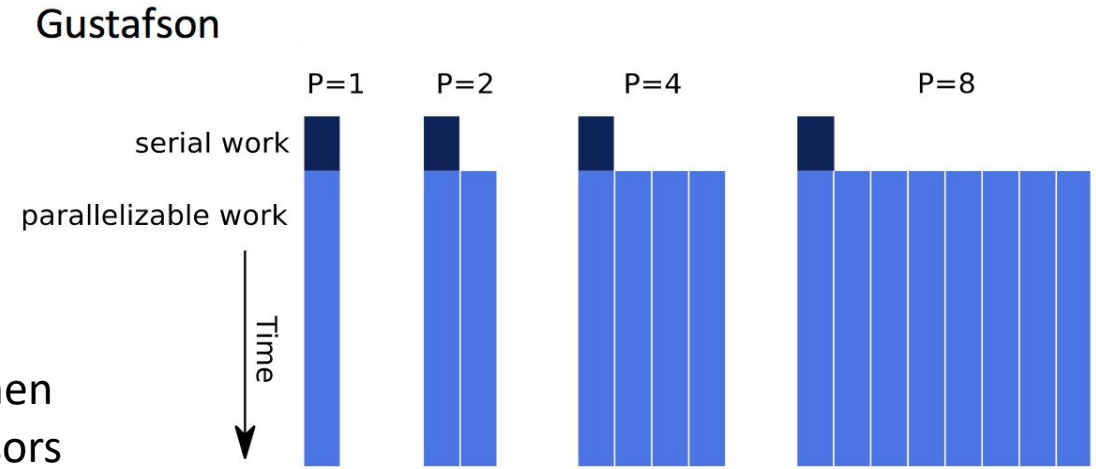
- At his days (the law appeared 1967) many programs had long sequential parts. This is not necessarily the case nowadays.
- It is not very easy to find F (sequential portion)

So …

- Decreasing the sequential portion is of greater importance than adding more cores blindly.
- Only when a program is mostly parallelized, does adding more processors help more than parallelizing the remaining rest.
- **Gustafson's law**: computations involving arbitrarily large data sets can be efficiently parallelized
- Both Amdahl and Gustafson do not take into account:
  - The overhead of synchronization, communication, OS, etc.
  - Load may not be balanced among cores
- So you have to use these laws as guideline and theoretical bounds only.

Amdahl's law vs Gustafson's law





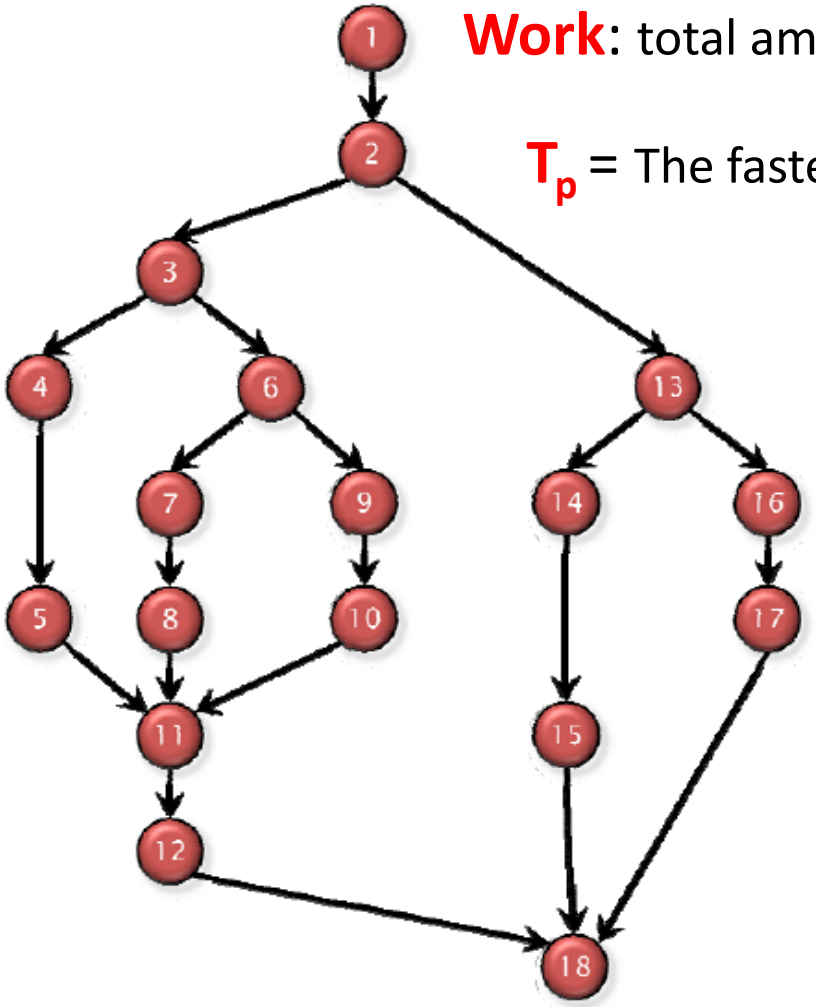
Amdahl's assumes fixed problem size and looked at speedup as we increase #processors.

Gustafson looks at what happen when both the problem size and #processors Increase.

Think of a program as a Directed Acyclic Graph (DAG)

- Vertices can represent:
  - A task (e.g. a function, a group of instructions, etc)
  - An instruction
- Directed arrows indicate dependencies among tasks (or instructions).
  - Source must finish before destination

DAG Model

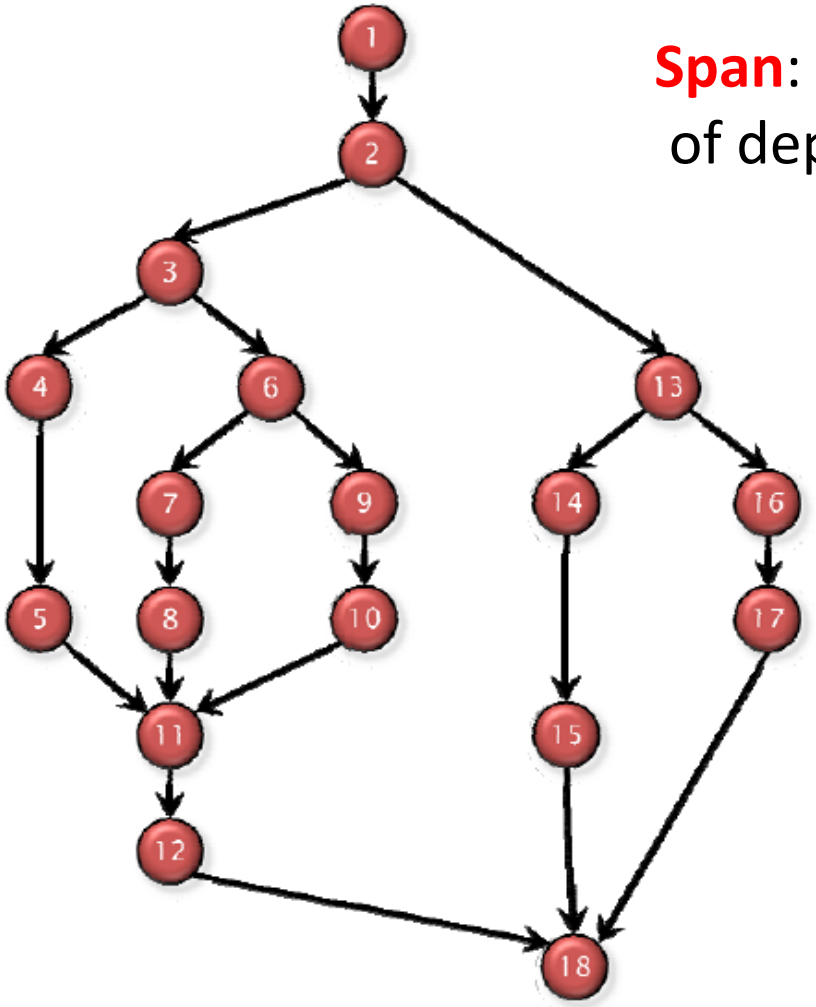


Work: total amount of time spent on all instructions

T_p = The fastest possible execution time on P processors

Work Law: $T_P \geq T_1/P$

DAG Model



Span: The longest path (in terms of time) of dependence in the DAG = T_∞

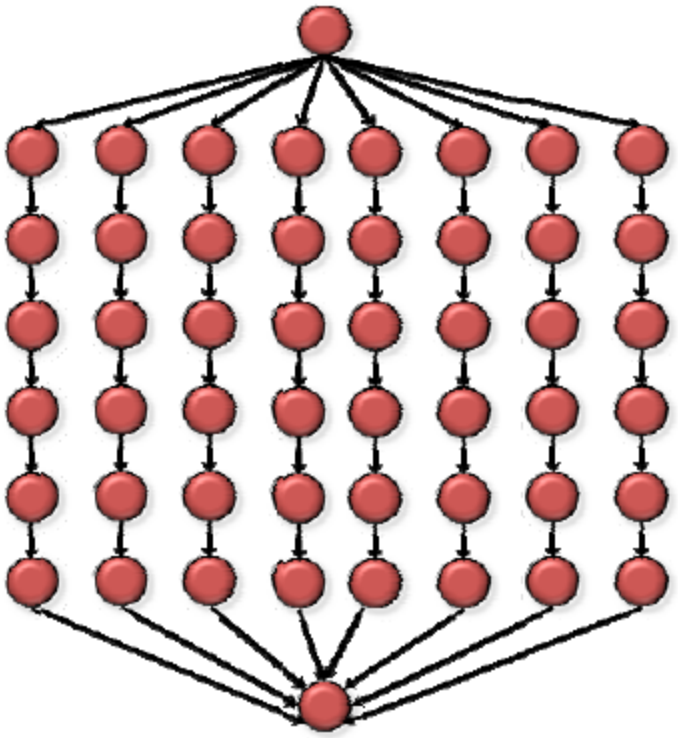
Span Law: $T_P \geq T_\infty$

Can We Define Parallelism Now?

How about? T_1/T_∞

Ratio of work to span

Can We Define Parallelism Now?



Work: $T_1 = 50$

Span: $T_\infty = 8$

Parallelism: $T_1/T_\infty = 6.25$

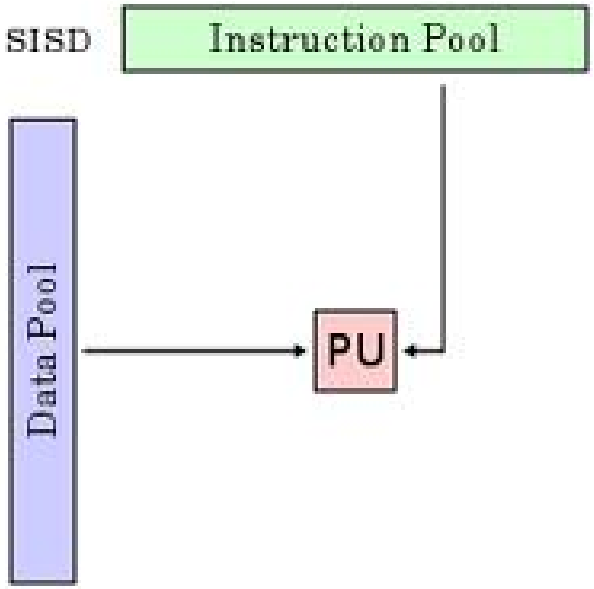
Parallelism Granularity

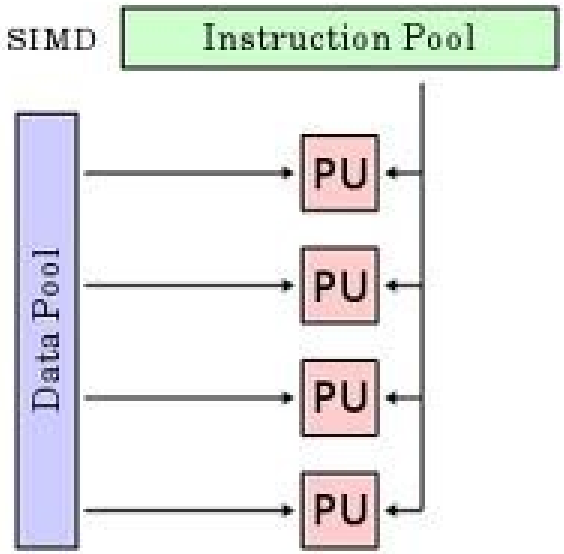
- Instruction level parallelism (ILP), exploited by:
  - Pipelining
  - Superscalar
  - Out-of-order execution
  - Speculative execution
- Thread level parallelism, exploited by:
  - Hyperthereading technology (aka SMT)
  - Multicore
- Process level parallelism, exploited by:
  - Multiprocessor system
  - Hyperthereading technology (aka SMT)
  - Multicore

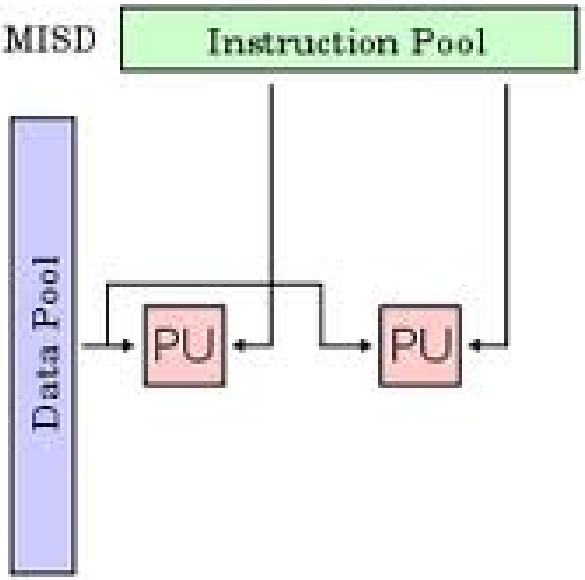
Flynn Classification

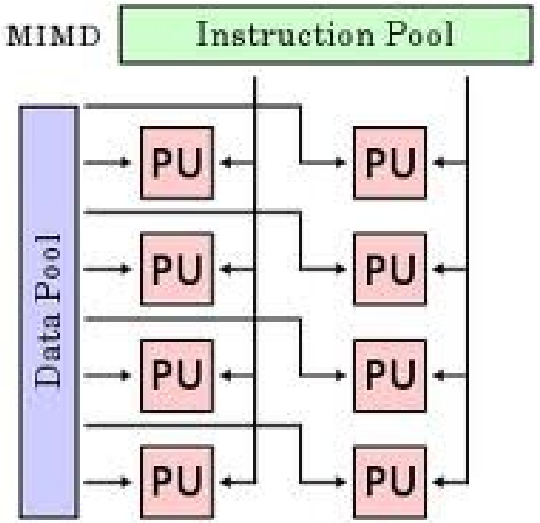
- A taxonomy of computer architecture
- Proposed by Michael Flynn in 1966
- It is based on two things:
  - Instructions
  - Data

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD



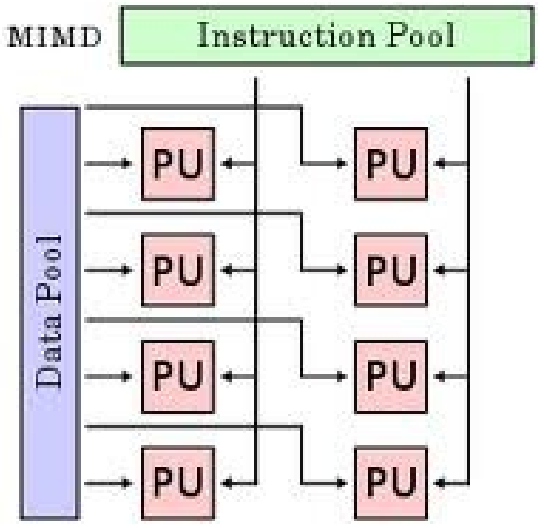






PU = Processing Unit

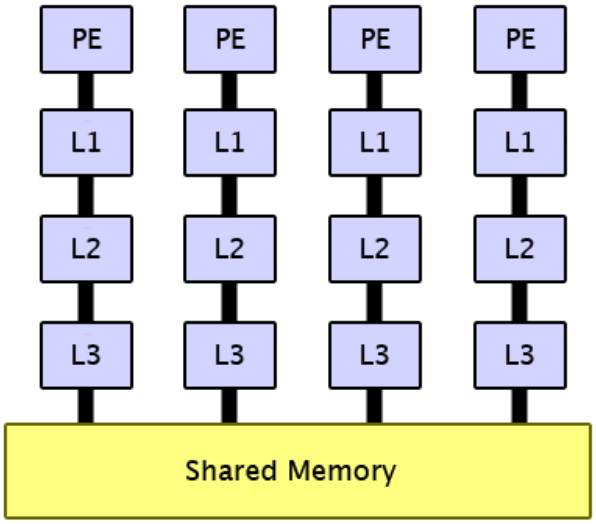
More About MIMD



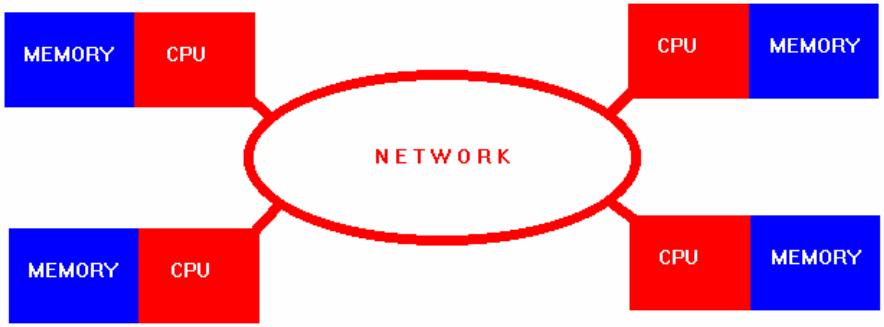
Shared-Memory **Distributed-Memory**

Or hybrid

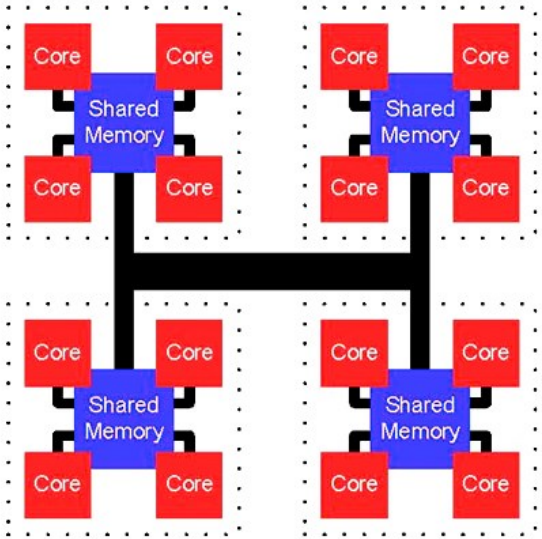
Shared Memory



Distributed Memory



Hybrid



Multicore and Manycore

> *We have arrived at many-core solutions not because of the success of our parallel software but because of our failure to keep increasing CPU frequency*.*
>
> **Tim Mattson**

Dilemma:

- Parallel hardware is ubiquitous
- Parallel software is not!

After more than 25 years of research, we are not closer to solving the parallel programming model!

The Mentality of Yet Another Programming Language … Doesn't work!

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
PCN
PCP:
PH
PEACE
PCU
PET
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER.
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
XENOOPS
XPC
Zounds
ZPL

The Mentality of Yet Another Programming Language … Doesn't work!

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
PCN
PCP:
pC++
SCHEDULE
SciTL
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC

We don't want to scare away the programmers … Only add a new API/language if we can't get the job done by fixing an existing approach.

"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
XENOOPS
XPC
Zounds
ZPL

Programming Model

- **Definition:** the languages and libraries that create an abstract view of the machine
- Control
  - How is parallelism created?
  - How are dependencies enforced?
- Data
  - Shared or private?
  - How is shared data accessed or private data communicated?
- Synchronization
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?

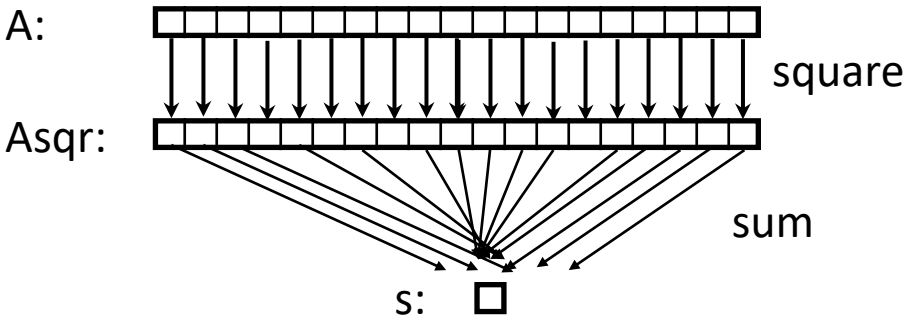
Important!

- You can run any paradigm on any hardware (e.g. an MPI on shared-memory)
- The hardware itself can be heterogeneous

The whole challenge of parallel programming is to make the best use of the underlying hardware to exploit the different types of parallelism

Example

We have a matrix A. We need to form another matrix Asqr that contains the square of each element of A. Then we need to calculate S, which is the sum of the elements in Asqr.



- **How can we parallelize this?**
- **How long will it take if we have unlimited number of processors?**

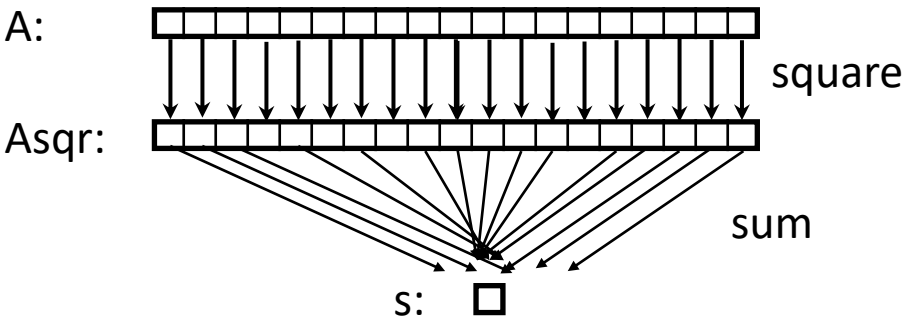
slide derived from Katherine Yelick

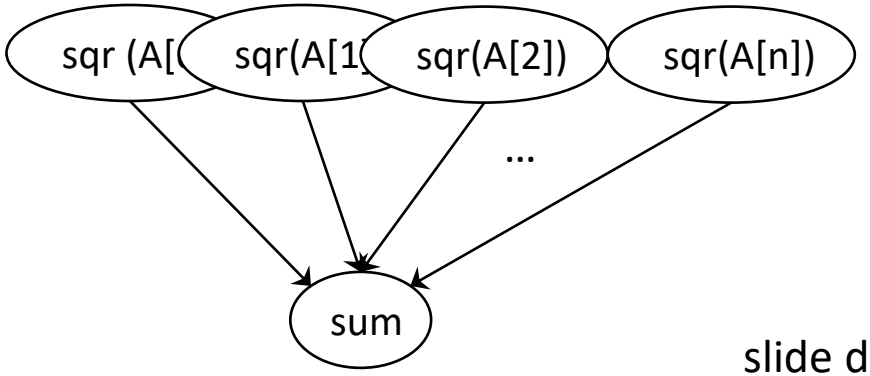
Example

- First, decompose your problem into a set of tasks
  - There are many ways of doing it.
  - Tasks can be of the same or different sizes.
- Draw a task-dependency graph (do you remember the DAG we saw earlier?)
  - A directed graph with Nodes corresponding to tasks
  - Edges indicating dependencies, that the result of one task is required for processing the next.

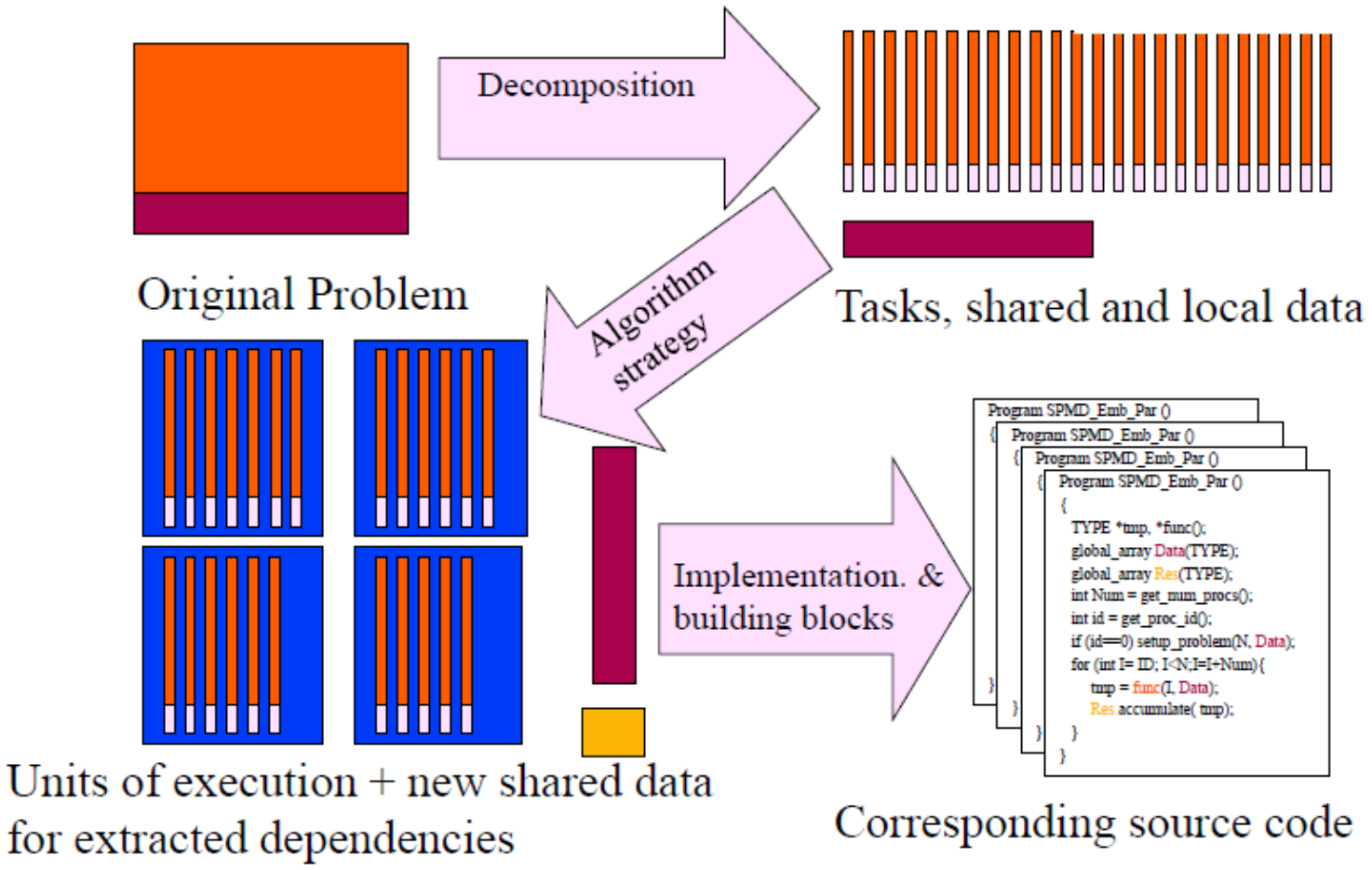
slide derived from Katherine Yelick

Example





slide derived from Katherine Yelick



Source: “Many Core Processors … Opportunities and Challenges” by Tim Mattson

Does your knowledge of the underlying hardware change your task dependency graph? If yes, how?

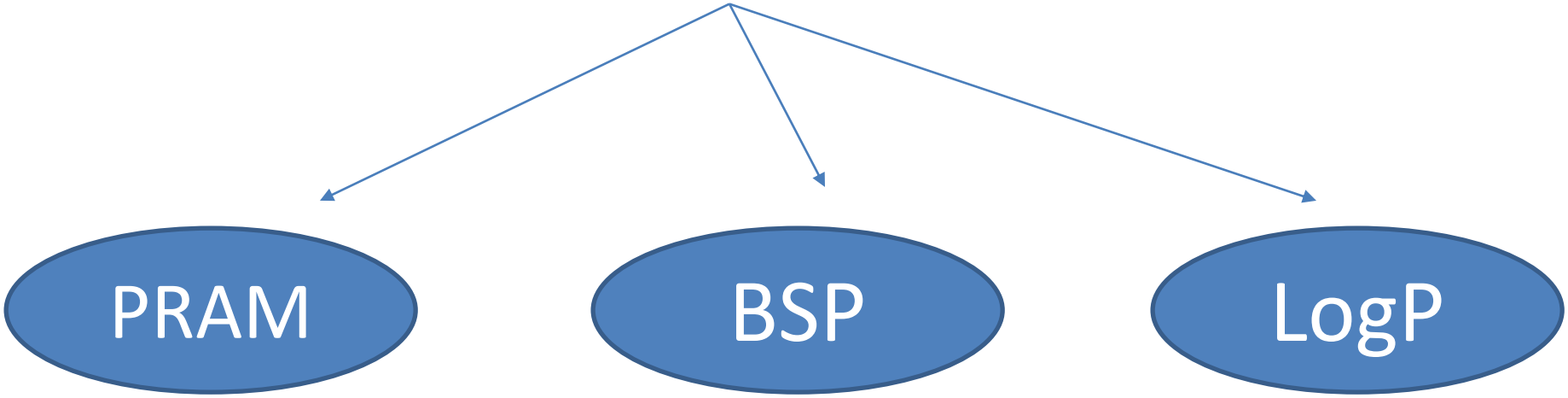
Suppose you have several candidate algorithms for solving a problem, how do you pick?

Wish List for a Good Algorithm for Parallelism

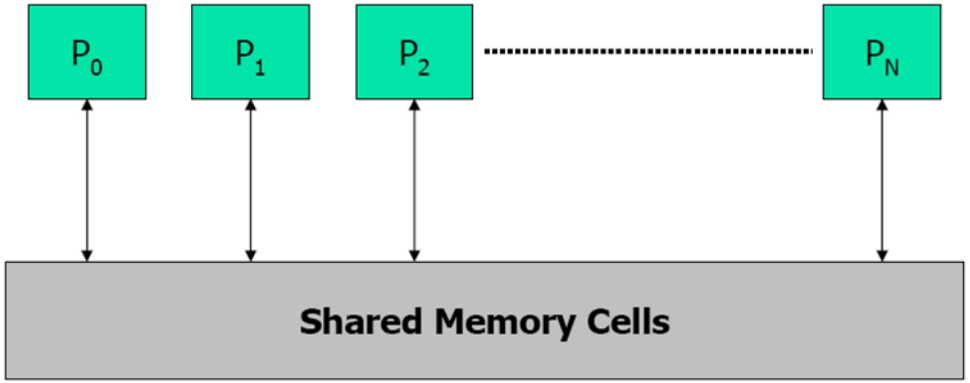
- Good performance
- On a wide range of parallel machines
- Without going into many details about the hardware at that early stage
- Scalable

This means: We need a computational model that can predict the performance of our algorithm on a wide range of machines and must strike a balance between details and simplicity.

Three Main Computational Models



PRAM Model

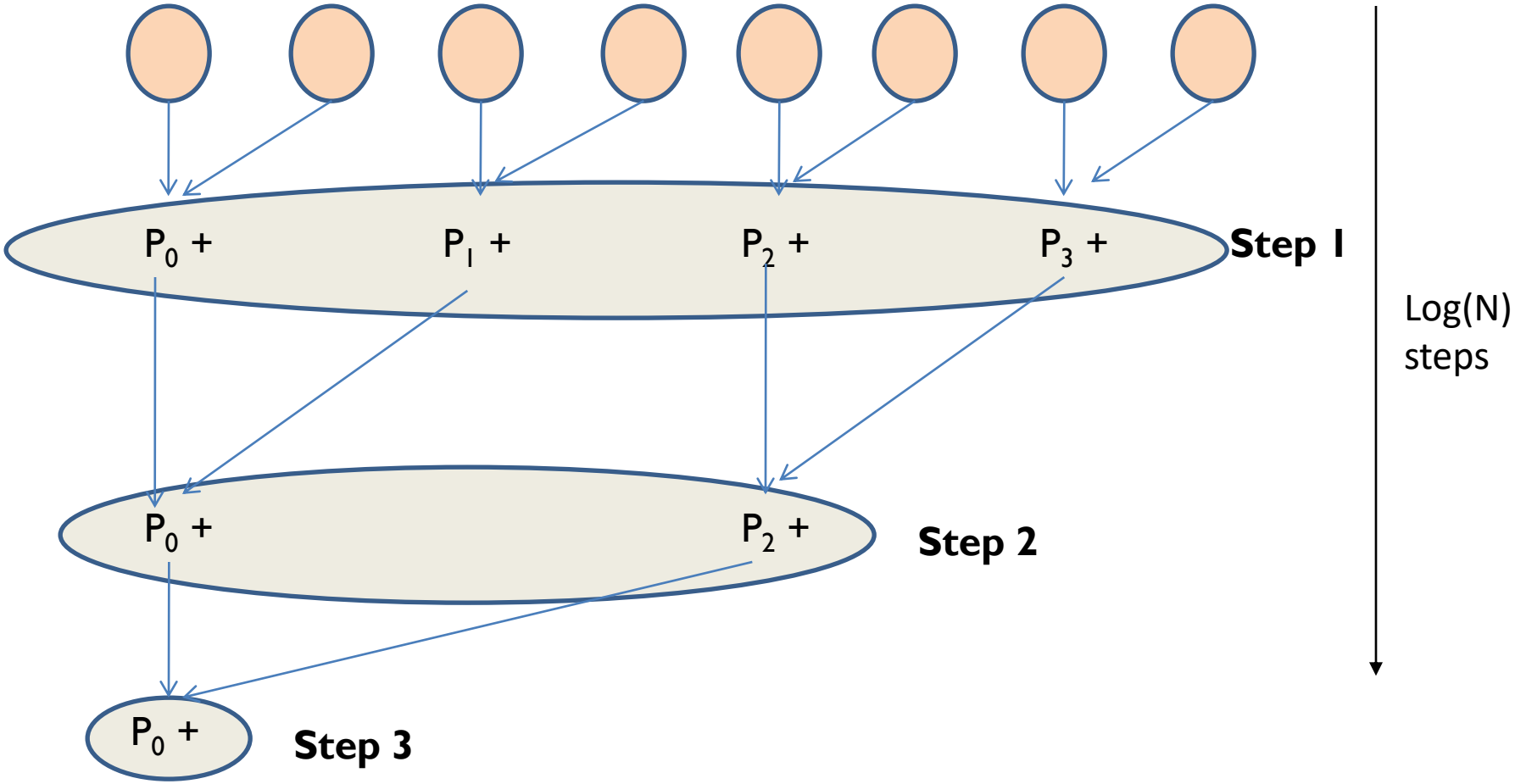


- Parallel Random Access Machine
- Shared memory
- A synchronous MIMD

PRAM Model

- No communication cost (i.e. infinite bandwidth and zero latency).
- Infinite memory
- Any operation takes one unit of time.
- Different protocols can be used for reading and writing shared memory.
  - **EREW** - exclusive read, exclusive write: A program isn't allowed to have two processors access the same memory location at the same time.
  - **CREW** - concurrent read, exclusive write
  - **CRCW** - concurrent read, concurrent write: Needs protocol for arbitrating write conflicts
  - **CROW** – concurrent read, owner write: Each memory location has an official "owner"

PRAM Example: Adding N Numbers



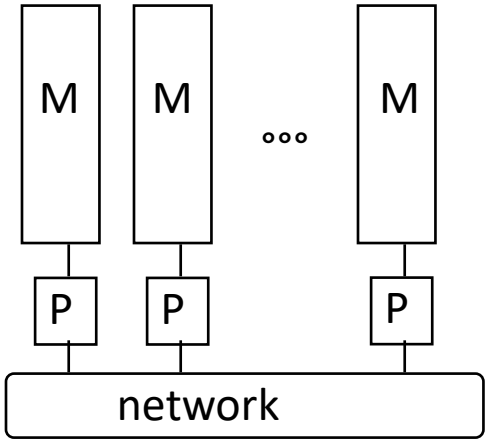
PRAM Example: Adding N Numbers

- time needed = Log (N) steps
  - Don't forget that the processors are executing in parallel.
- N / 2 processors needed to add N numbers

Pros/Cons of PRAM

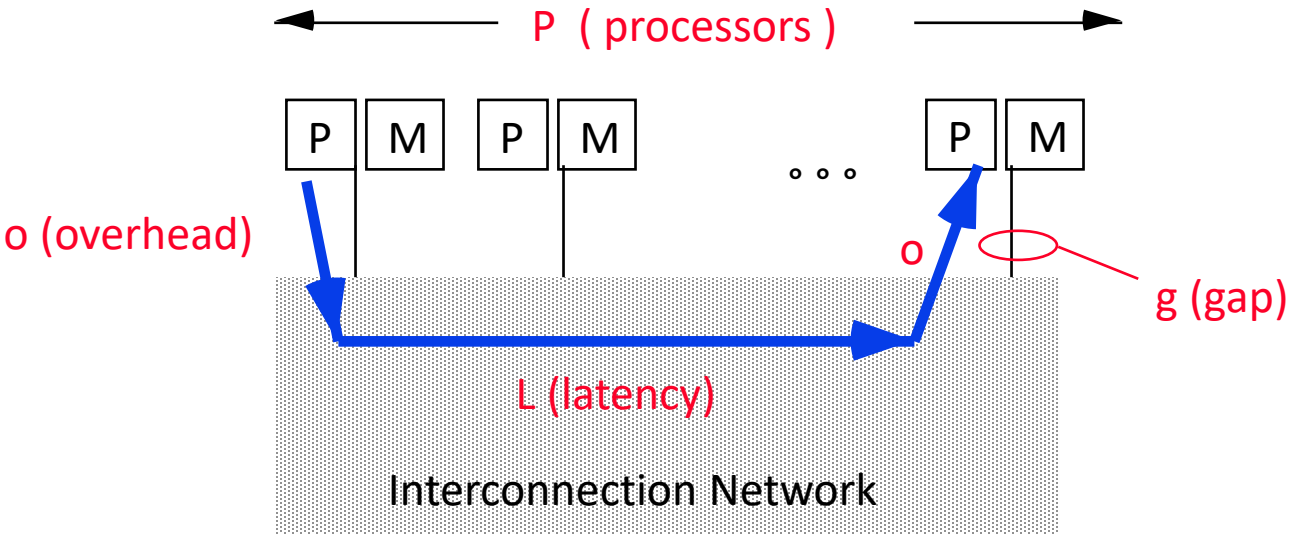
+ Simple to use
- Unrealistic → performance prediction is inaccurate

LogP Model



- Distributed memory
- No specification of interconnection network
- Based on:
  - Latency of communication
  - Overhead in processing transmitted/received messages
  - Gap between consecutive transmissions (i.e. bandwidth limitation)
  - Processing power

LogP Model


P (processors)
P
M
P
M
. . .
P
M
o (overhead)
o
g (gap)
L (latency)
Interconnection Network

Pros/Cons of the LogP model

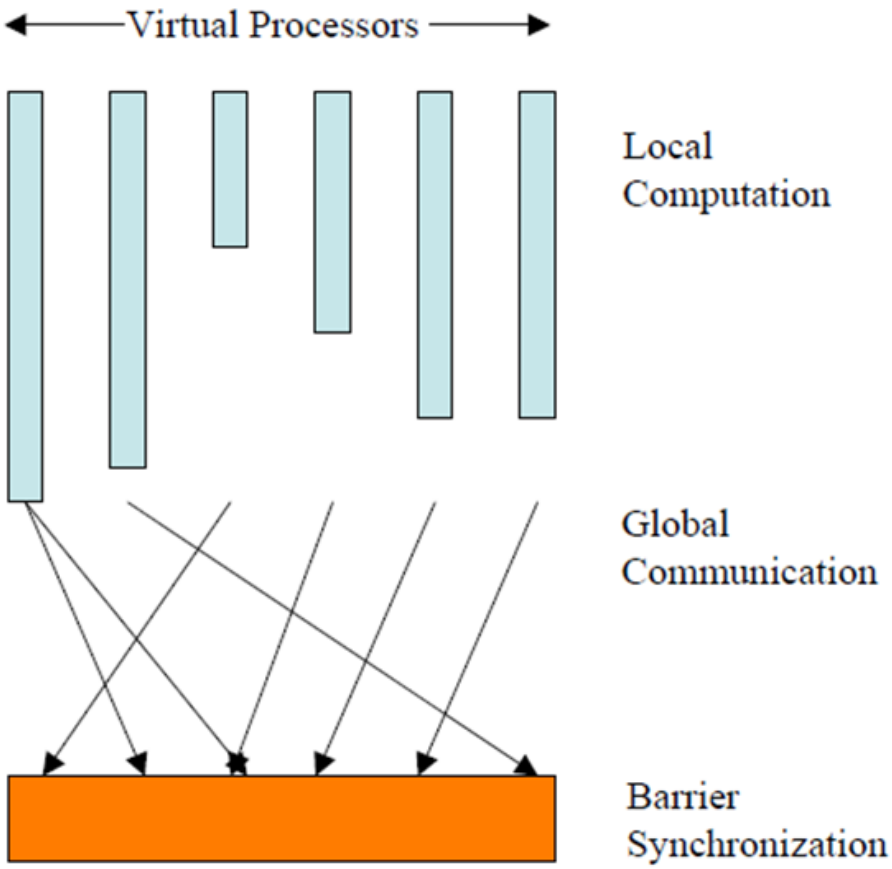
+ Simple, 4 parameters
+ Can easily be used to guide the algorithm development
- Does not take contention into account → can sometimes underestimate communication time.

There are many variations to the LogP model, making it more accurate but more complex (e.g. LogGP, logGPC, pLogP, …)

BSP Model

- Bulk Synchronous Parallel
- A BSP computer consists of
  - A set of processor-memory pairs
  - A communication network that delivers messages in a point-to-point manner
  - Mechanism for barrier synchronization for all or a subset of the processes
- BSP programs composed of supersteps
  - Each superstep consists of three ordered stages:
    - Computation
    - Communication
    - Barrier synchronization

BSP Model


Virtual Processors
Local
Computation
Global
Communication
Barrier
Synchronization

BSP Model

- p: number of processors
- s: processor computation speed (flops/s)
- h: the maximum number of incoming or outgoing messages per processor
- g: the cost of sending a message.
- l: time to do a barrier synchronization

Using BSP Model

- Assume w_i is the computation time for work on processor i during a superstep.
- Cost of a superstep:

$$\max_{i=1}^{p}(w_i) + \max_{i=1}^{p}(h_i g) + l$$

Pros/Cons of BSP Model

+ Simple
+ predictable performance
- Not very good if locality is important
- BSP does not distinguish between sending 1 message of length m, or m messages of length 1.

Be Careful!

- All these models are just approximations.
- They do not model memory which can greatly affect their predictions.
  - There are memory models though.
- An implementation of a good parallel algorithm on a specific machine will surely require tuning. But first, pick/design an algorithm based on one of the models discussed.

Conclusions

- Concurrency and parallelism are not exactly the same thing.
- There is parallelism at different granularities, with methods to exploit each parallelism granularity.
- Knowing the hardware will help you generating a better task dependency graph.
- With the end of Moore's law, parallel programming becomes even more important.