CSCI-GA.3033-017

Special Topics: Multicore Programming

Lecture 2
Parallelism, Concurrency, and Performance

Christopher Mitchell, Ph.D.

cmitchell@cs.nyu.edu || http://z80.me

Lecture 2 Outline

- GDB Overview/Review
- Parallelism and Concurrency
- Parallel Programming Models and Performance

GDB Overview/Review

- **`gdb --args ./binary argument argument`**

• `run`	Start the program
• `break`	Set breakpoint
• `continue`	Pick up where execution stopped
• `print`	Print variable contents
• `x`	Inspect memory
• `list`	Display lines of code
• `step`	Execute the next line of code
• `info`	Get list of breakpoints, sources, etc
• `help/apropos`	Get help with commands

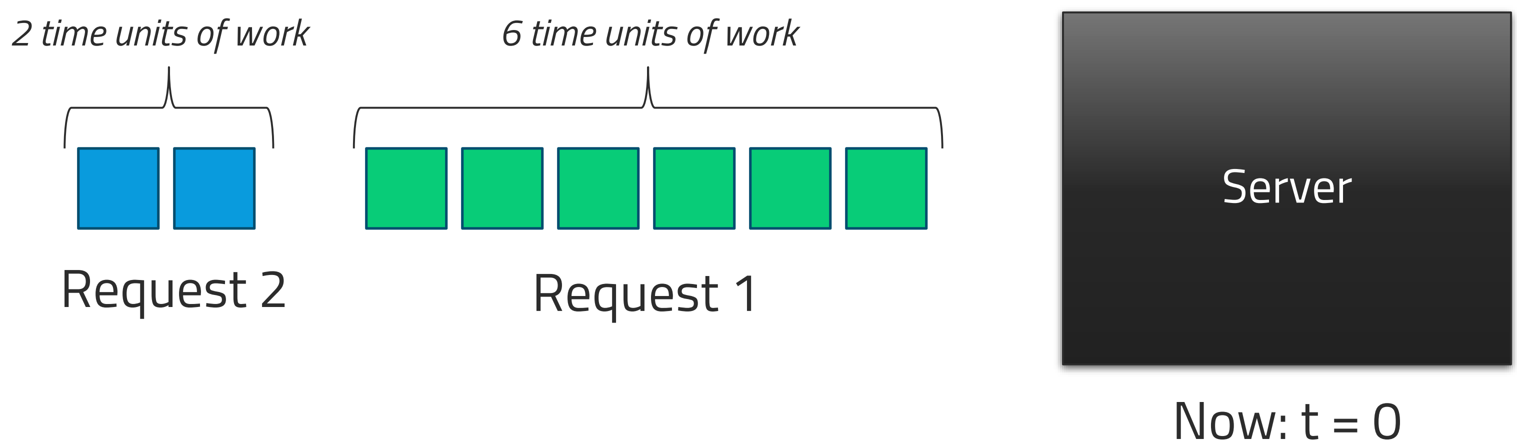
- Abridged reference: https://w3.cs.jmu.edu/bernstdh/Web/common/help/gdb.php

Parallelism and Concurrency

Same Meaning?

- Concurrency: At least two tasks are making progress in 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

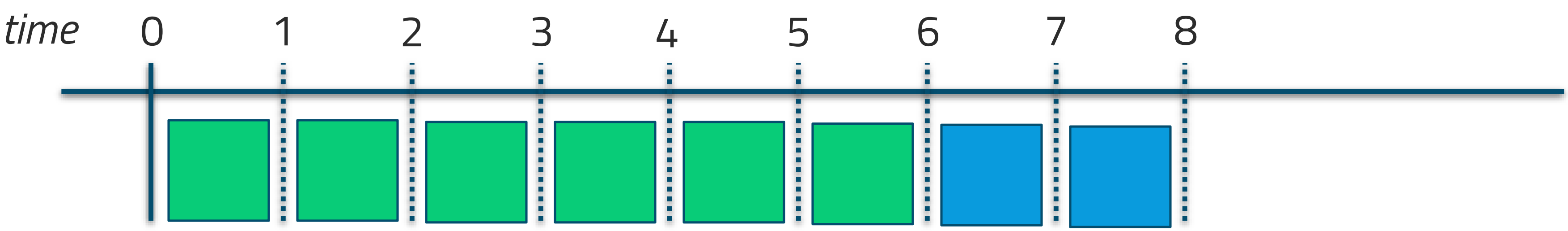
Example: Request Processing



How will this server react: (1) if it is serial, (2) if it is concurrent but not parallel, and (3) if it is parallel?

[Hint: look at total completion time and average completion time]

Example:
Request Processing

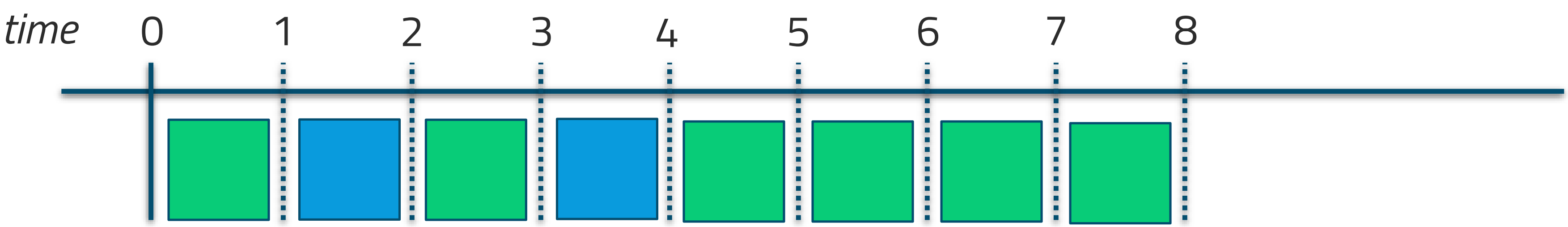


Serial:

- Average completion time (be careful)?
- Total completion time?
- Resource utilization?

Server

Example: Request Processing

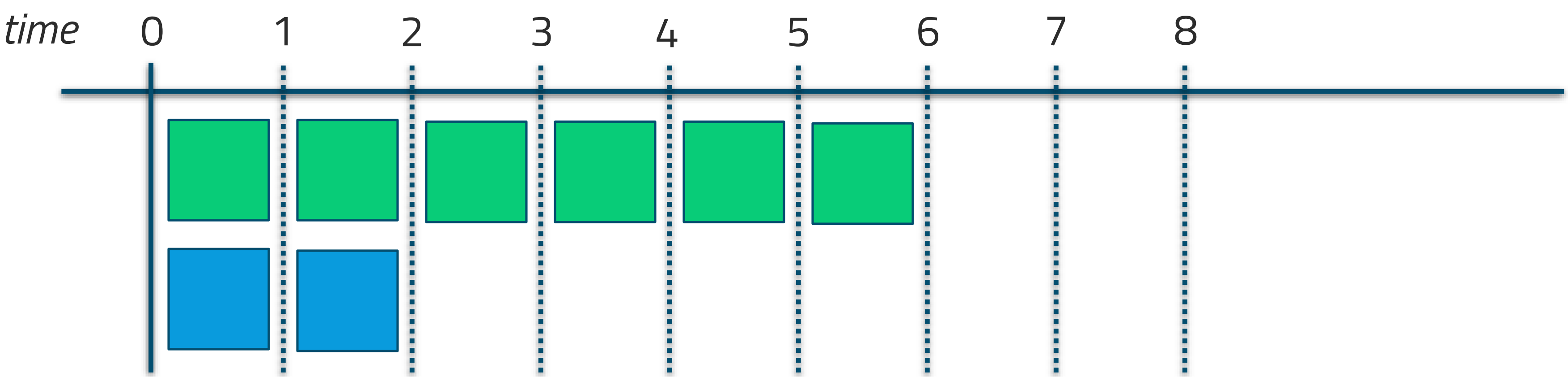


Concurrent (Without Parallelism):

- Average completion time (be careful)?
- Total completion time?
- Resource utilization?

Server

Example:
Request Processing

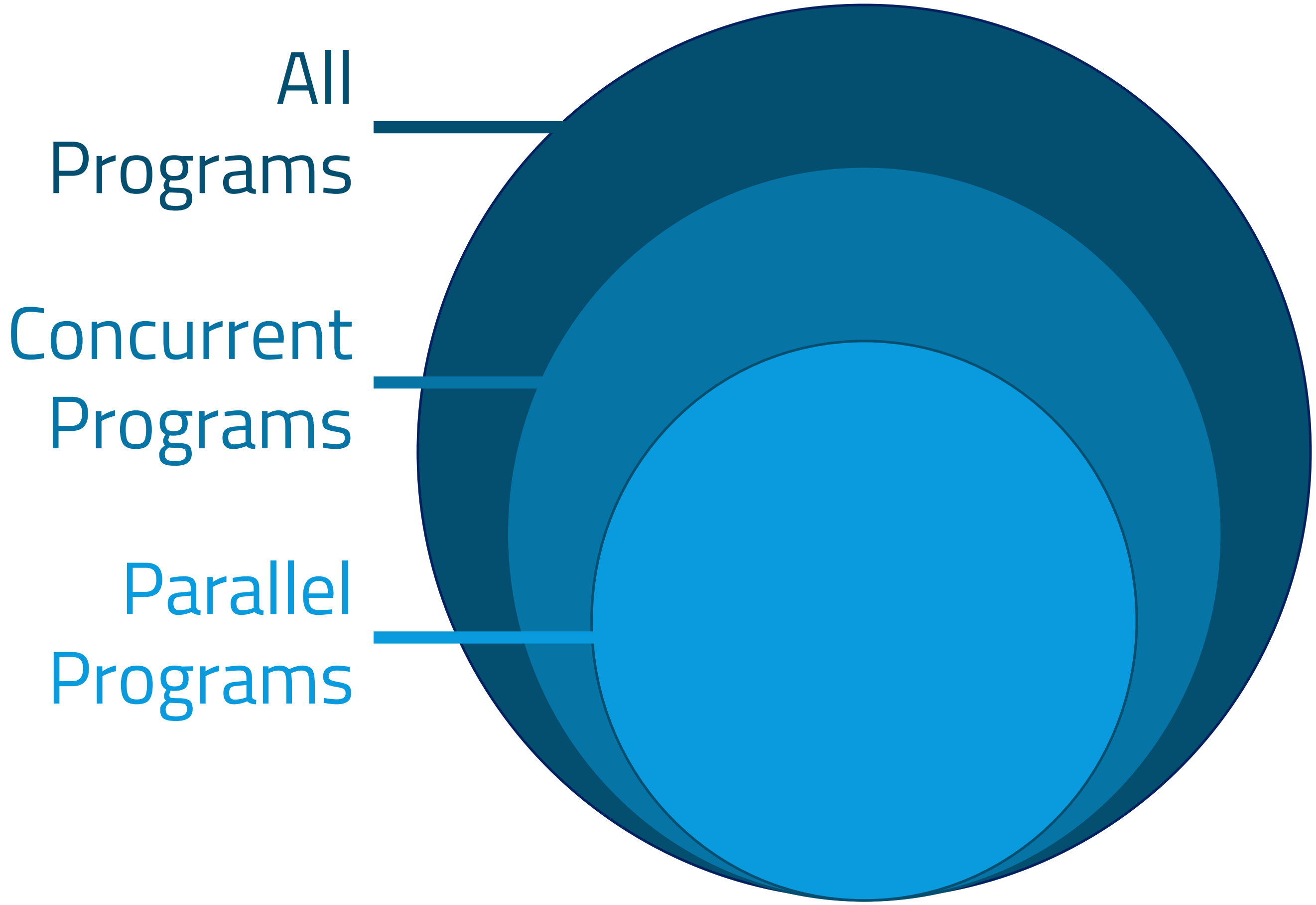


Parallel:

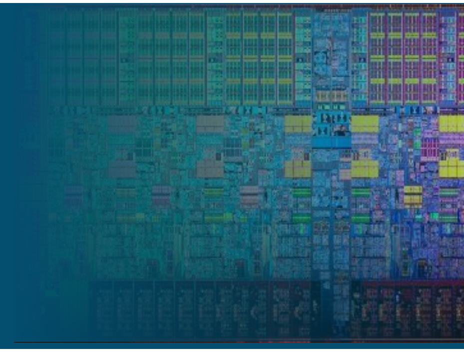
- Average completion time?
- Total completion time?
- Resource utilization?

Server

Parallel and Concurrent Programs



Simply Speaking...



Concurrency + Parallelism

=

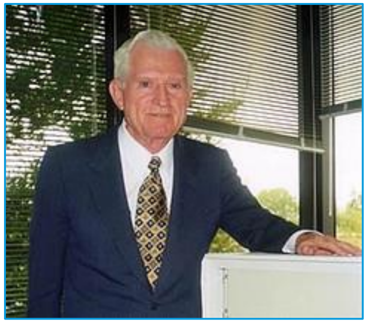
Performance

(but how much?)

Questions!

- If we have as much hardware as we want, do we get as much parallelism as we wish?
- If we have 2 cores, do we get 2x speedup?
  - Think back to the “resource utilization” question.

Amdahl’s Law



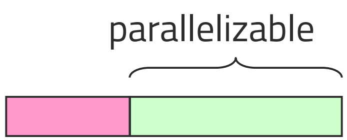
Gene M. Amdahl

- How much of a speedup one could get for a given parallelized task?
- Amdahl’s Law (1967):

If F is the fraction of a calculation that is sequential then the maximum speed-up that can be achieved by using P processors is **$1/(F+(1-F)/P)$**

Amdahl's Law

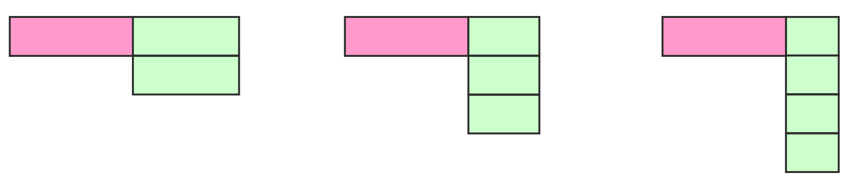
If F is the fraction of a calculation that is sequential then the maximum speed-up that can be achieved by using P processors is **1/(F+(1-F)/P)**.



Sequential portion = 40% -> F = 0.4
Execution time = 1.0 units

1CPU

1/ (0.4+ 0.6/2)
= 1/0.7 = 10/7



2CPUs 3CPUs 4CPUs

What Was Amdahl Saying?

1. Don't invest blindly on large number of processors.
2. Having faster cores (or processor at his time) makes more sense than having many cores.

Was he right?

- In 1967, many programs had long sequential parts.
- This is not necessarily the case nowadays.
- It is not very easy to find F (the sequential portion)

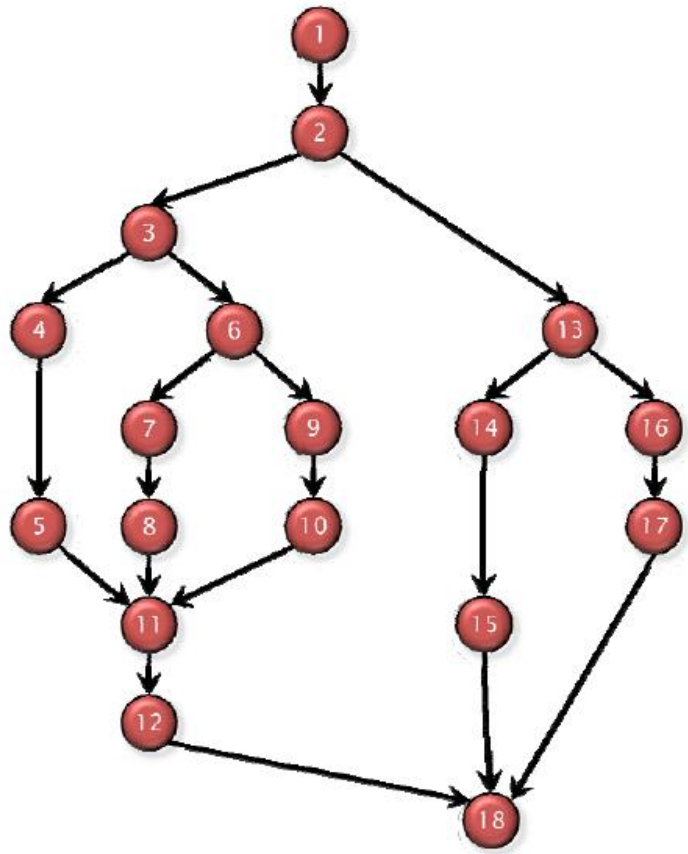
So …

- Decreasing the serialized portion is of greater importance than adding more cores
- Only when a program is mostly parallelized, does adding more processors help more than parallelizing the remainder.
- Gustafson’s law: computations involving arbitrarily large data sets can be efficiently parallelized

So …

- 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.

DAG Model for Multithreading

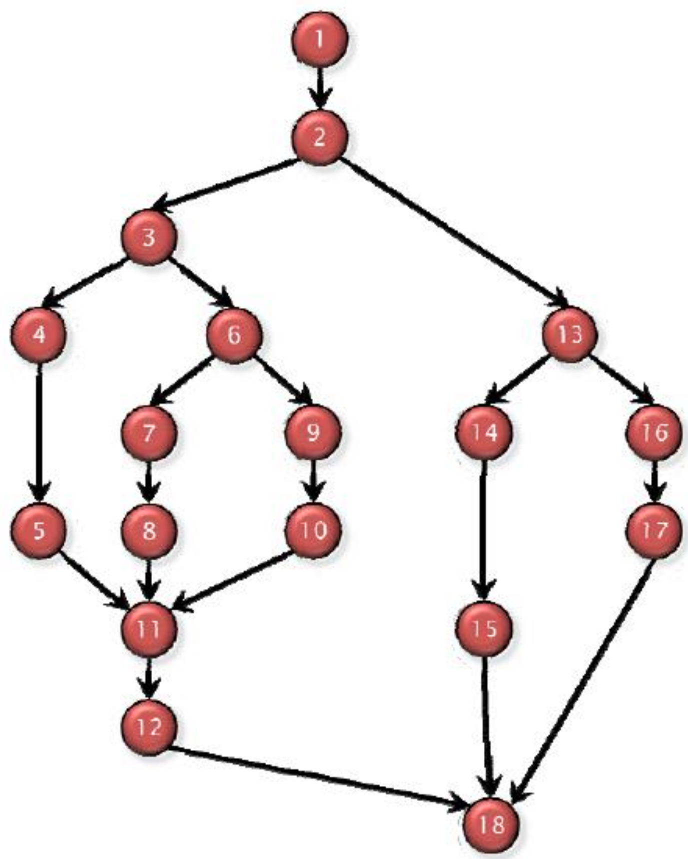


Work: total amount of time spent on all instructions

T_p = The fastest possible execution time on P processors

Work Law: $T_P \geq T_1/P$

DAG Model for Multithreading



Span: The longest path of dependence in the DAG = $\mathbf{T_\infty}$

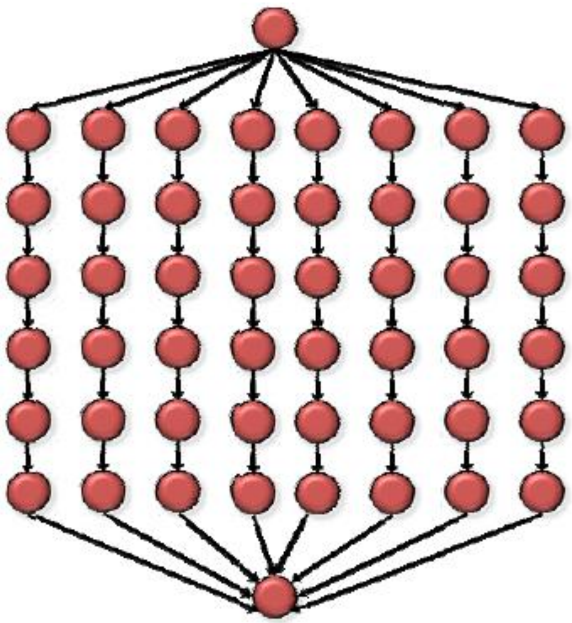
Span Law: $\boxed{T_P \geq T_\infty}$

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: $T_1 / T_\infty = 6.25$

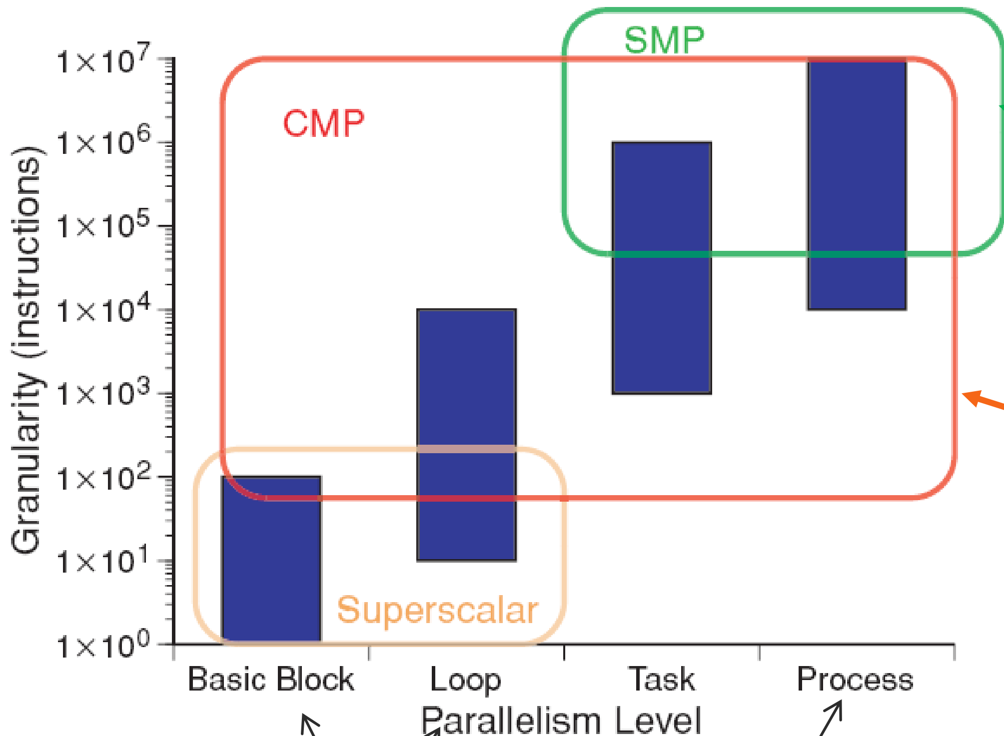
Reasoning about Parallelism

- At what level can we reason about parallelism?
  - Algorithm?
  - High-level language (eg, C++)?
  - Assembly?
  - Individual instructions?

Is Thread The Only Parallelism Granularity?

- Instruction level parallelism (ILP)
  - Superscalar
  - Out-of-order execution
  - Speculative execution
- Thread level parallelism
  - Hyperthreading technology (aka SMT)
  - Multicore
- Process level parallelism
  - Multiprocessor system
  - Hyperthereading technology (aka SMT)
  - Multicore

That Was The Software
How about the Hardware?



Symmetric Multiprocessing: Multiple CPUS

Chip-Level Multiprocessing: Multiple Cores

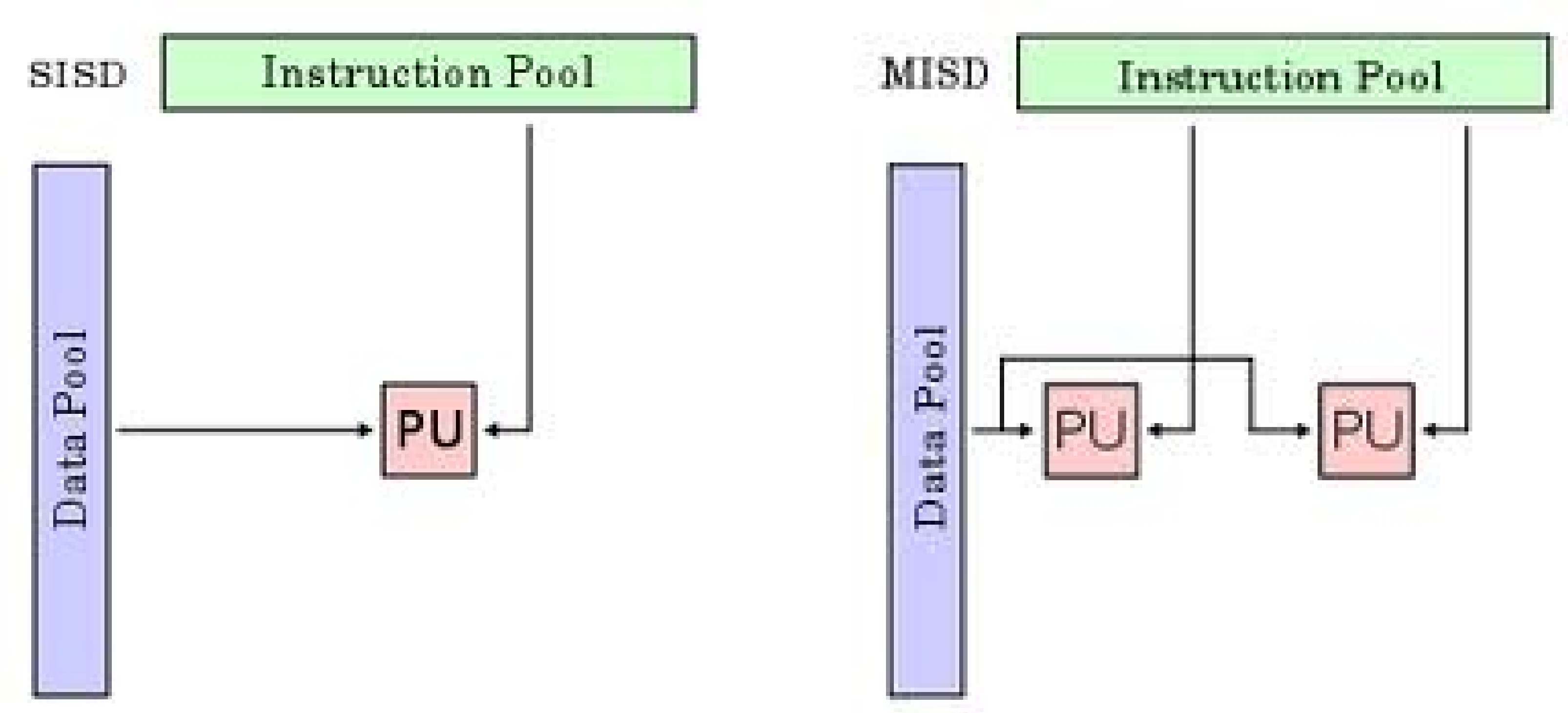
Latency Vs Throughput

Flynn Classification

- A taxonomy of computer architecture
- Proposed by Michael Flynn in 1966
- Classifies by:
  - Instruction parallelism
  - Data parallelism

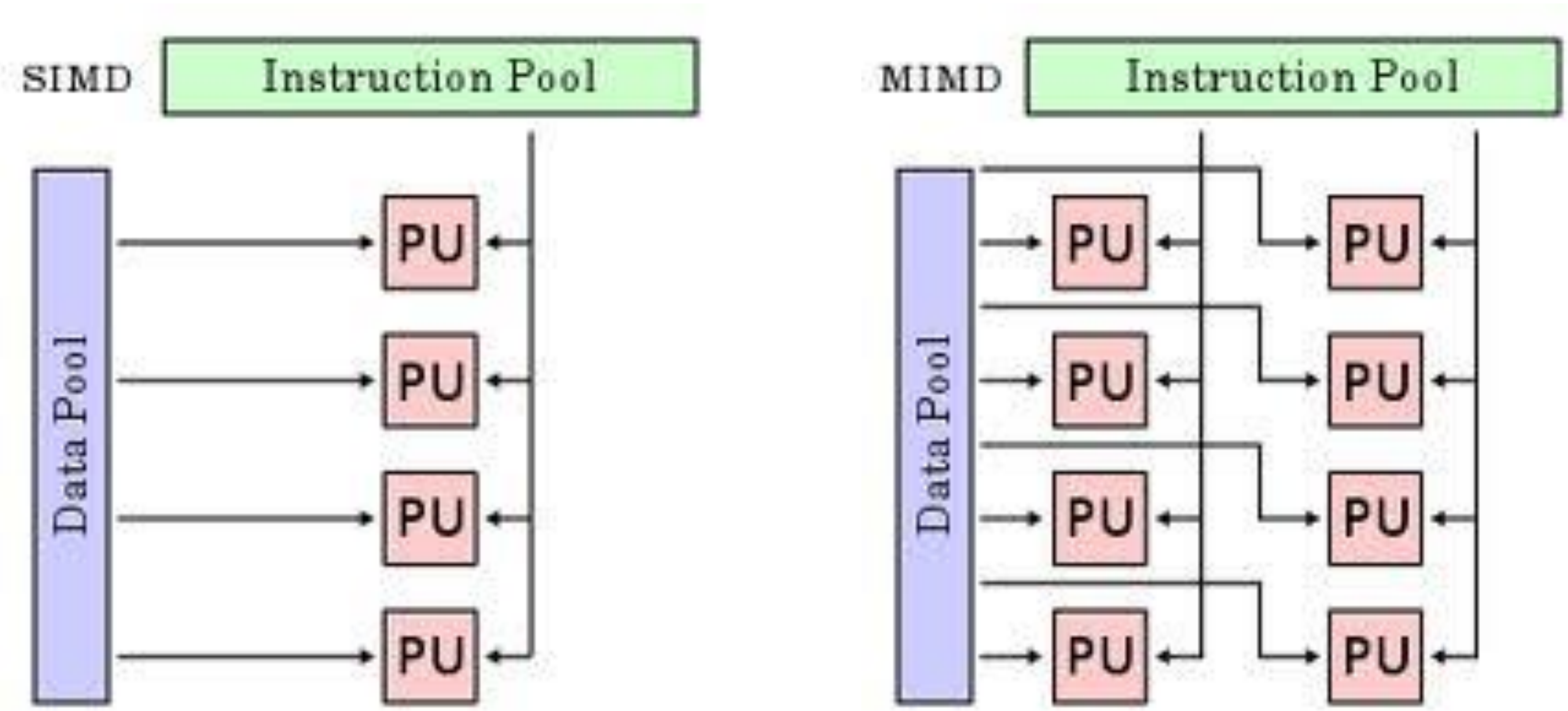
	Single instruction	Multiple instruction
Single data	**SISD**	**MISD**
Multiple data	**SIMD**	**MIMD**

Single Data Architectures



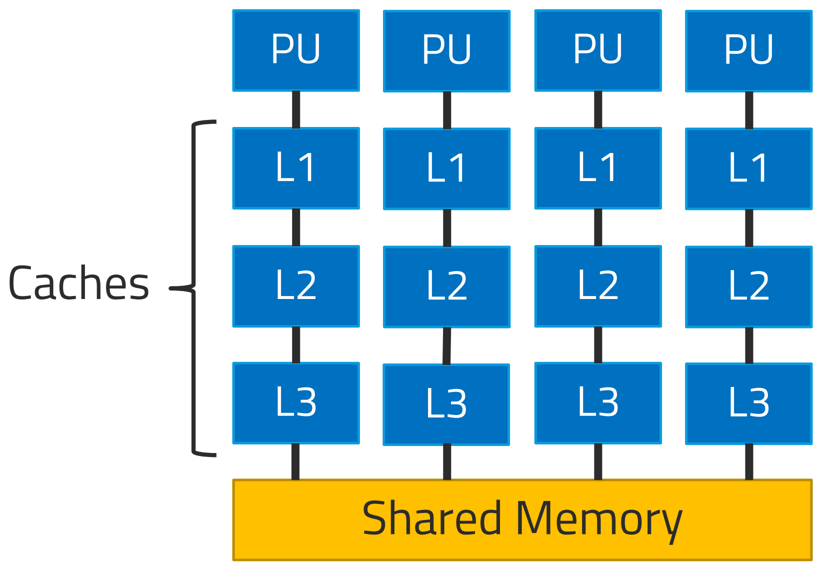
PU = Processing Unit

Multiple Data Architectures

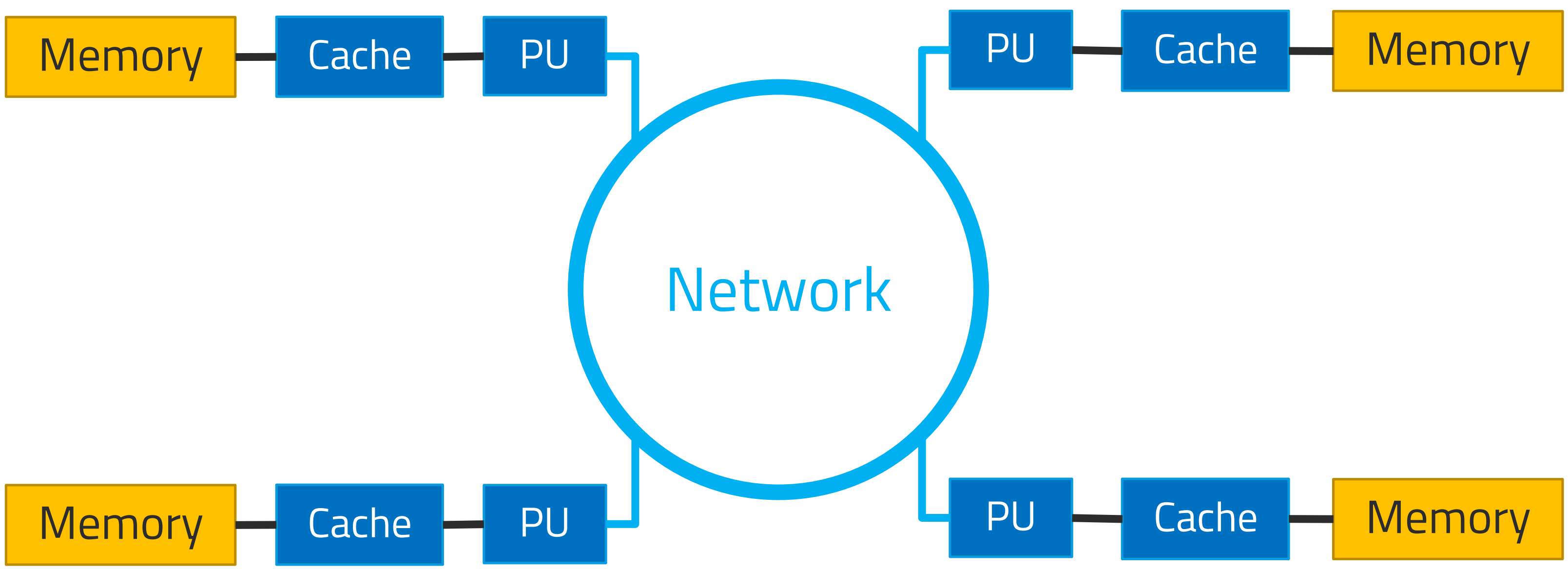


PU = Processing Unit

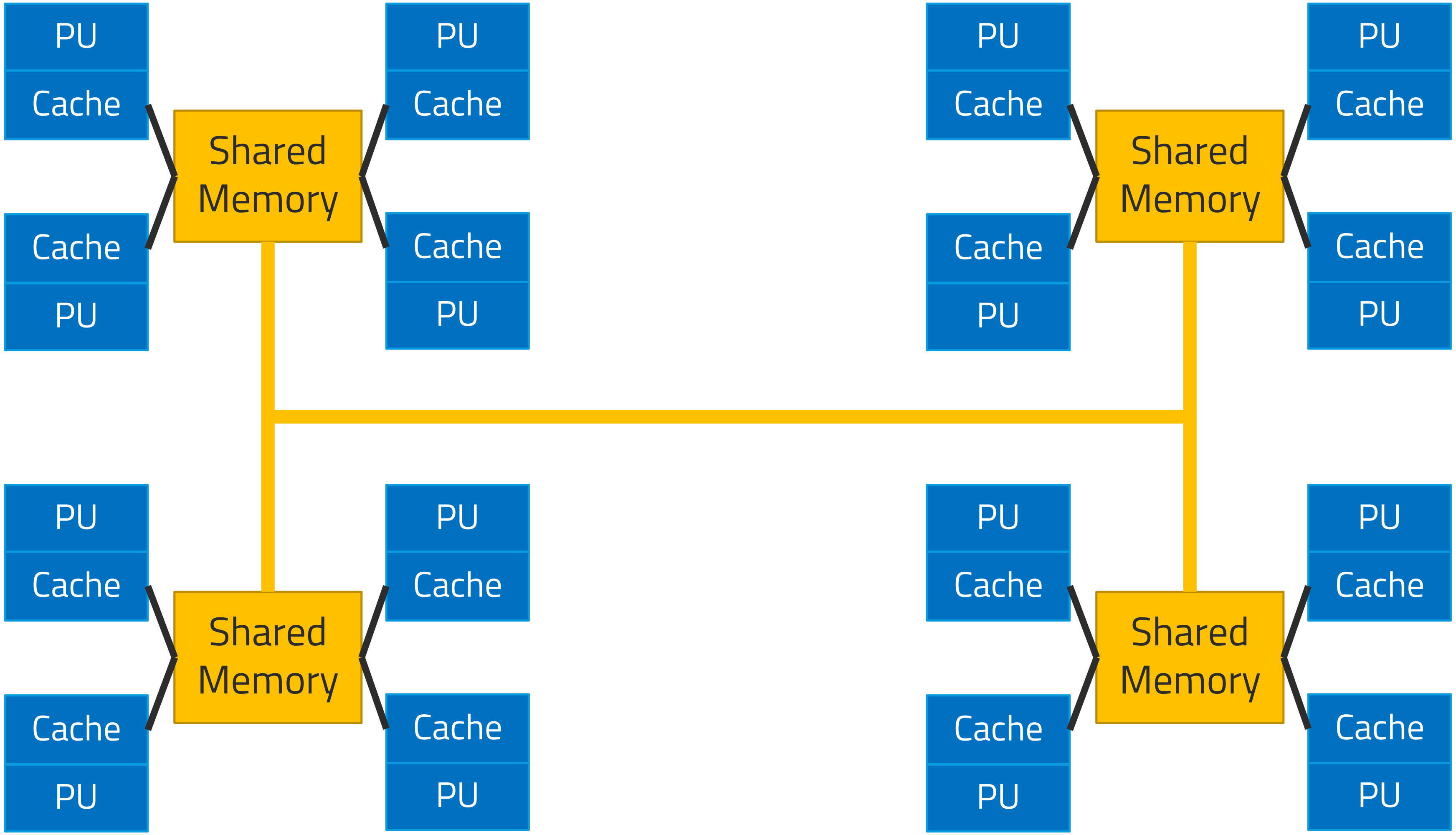
MIMD Memory Models: Shared Memory



MIMD Memory Models: Distributed Memory



MIMD Memory Models: 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
Parallel Computing @ Intel

- 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

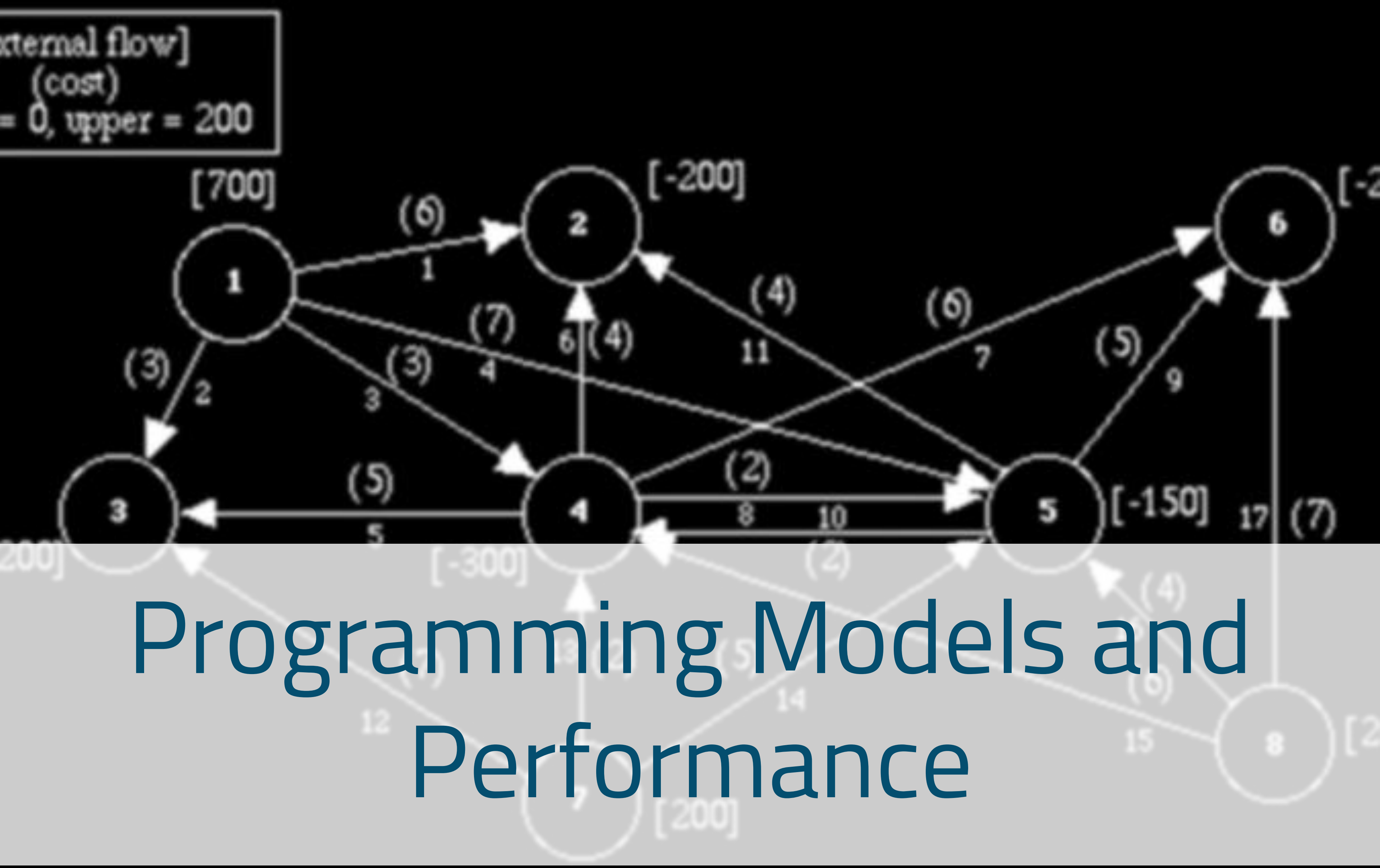
ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER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

ABCPL	CORRELATE	GLU	Mentat	Parafrase2	pC++
ACE	CPS	GUARD	Legion	Paralation	SCHEDULE
ACT++	CRL	HAsL.	Meta Chaos	Parallel-C++	SciTL
Active messages	CSP	Haskell	Midway	Parallaxis	SDDA.
Adl	Cthreads	HPC++	Millipede	ParC	SHMEM
Adsmith	CUMULVS	JAVAR.	CparPar	ParLib++	SIMPLE
ADDAP	DAGGER	HORUS	Mirage	ParLin	Sina
AFAPI	DAPPLE	HPC	MpC	Parmacs	

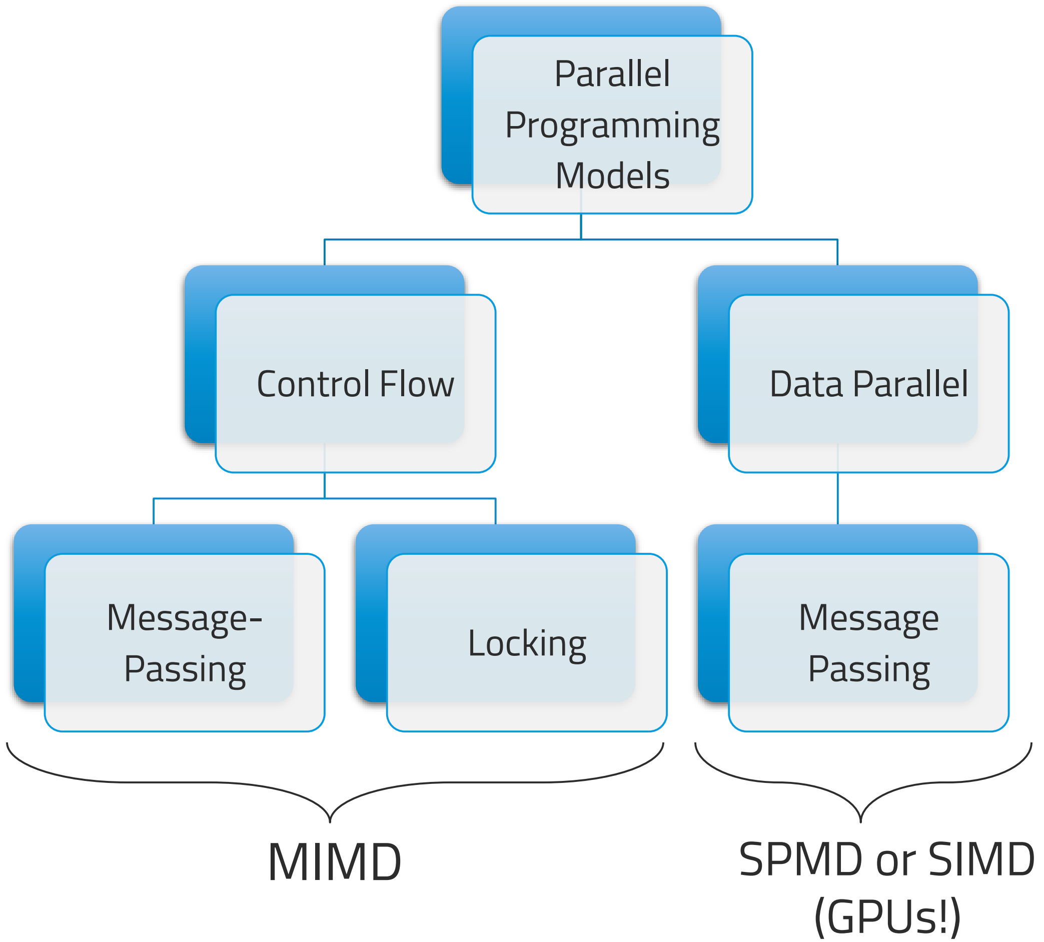
We don't want to scare away the programmers … Only add a new API/language if we can't get the job done by fixing an existing approach.

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



Programming Models and Performance

Parallel Programming Models



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?

Any Paradigm on Any Hardware

- 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

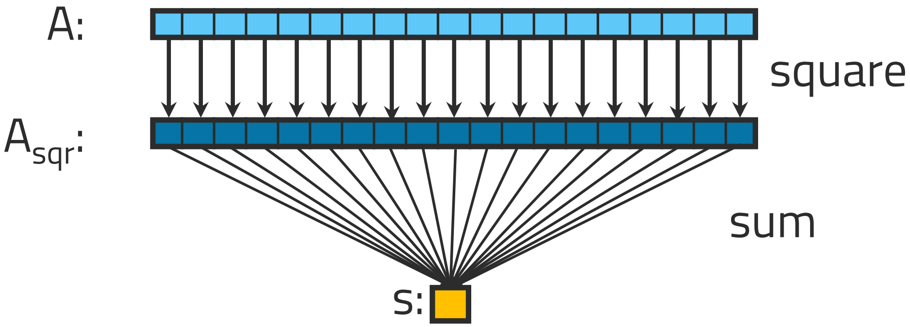
Example

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

Thanks to Katherine Yelick.

Example

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

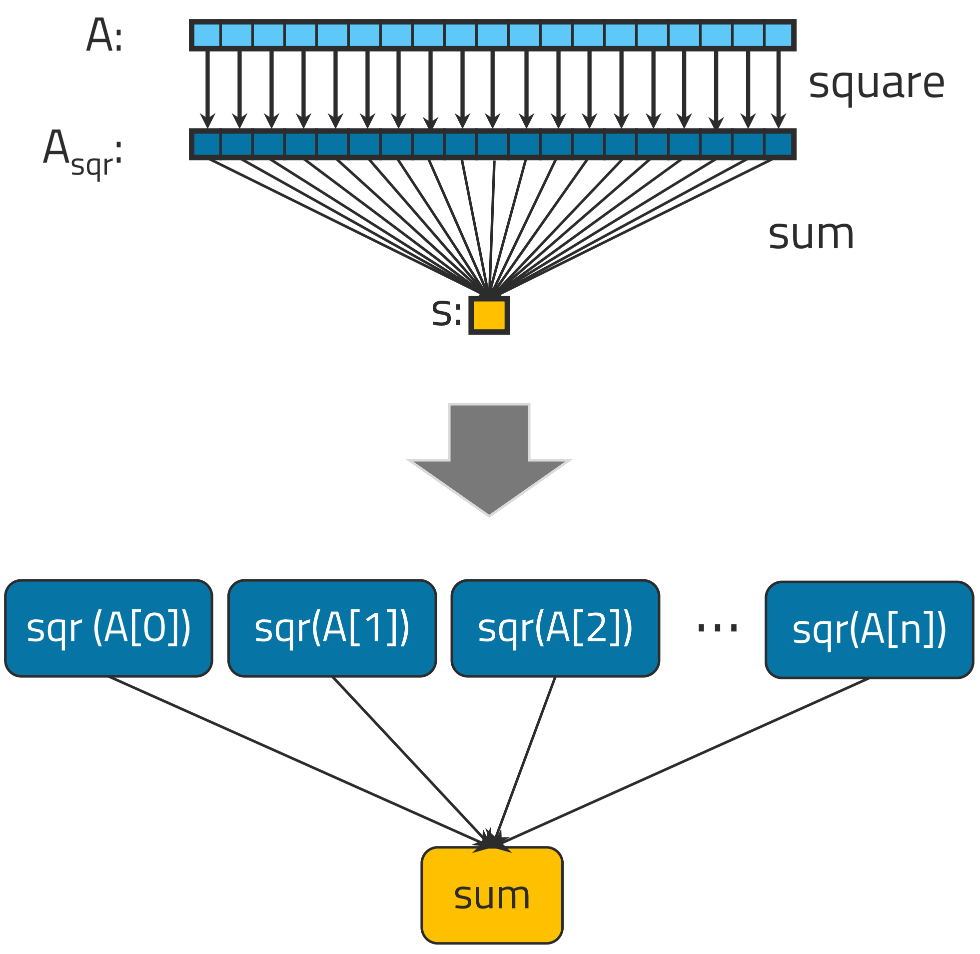


- How can we parallelize this?
- How long will it take if we have unlimited number of processors?

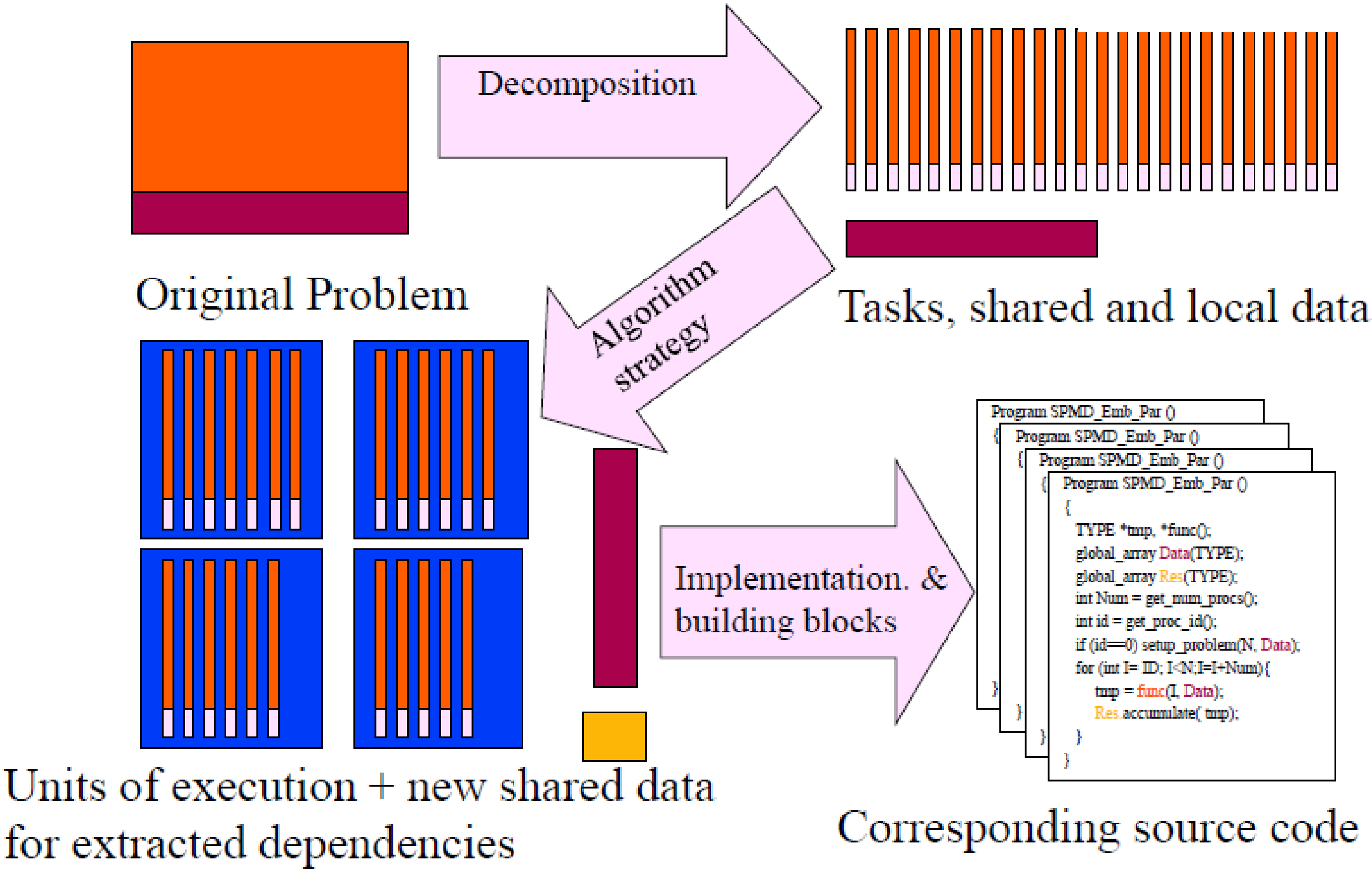
Example

- First, decompose your problem into a set of tasks
  - There are many ways of doing it.
  - Tasks can be of the same, different, or undetermined 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.

Example



Writing a Parallel Program



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

1. Good performance
2. On a wide range of parallel machines
3. Minimal tuning to hardware in early stage

We need an analytical model that can predict the performance of our algorithm on a wide range of machines and must strike a balance between detail and simplicity.

Three Main Computational Models

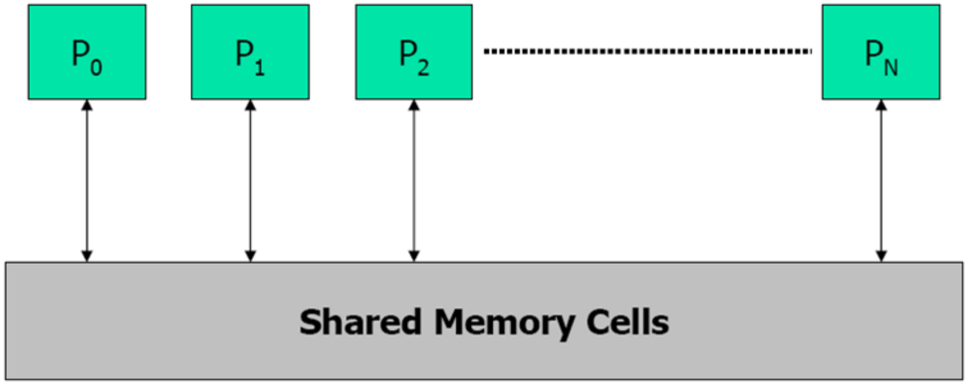
PRAM

LogP

BSP

PRAM Model

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



PRAM Model

- Can emulate a message-passing machine by partitioning memory into private memories.
- No communication cost (i.e. infinite bandwidth and zero latency).
- Infinite memory
- 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"

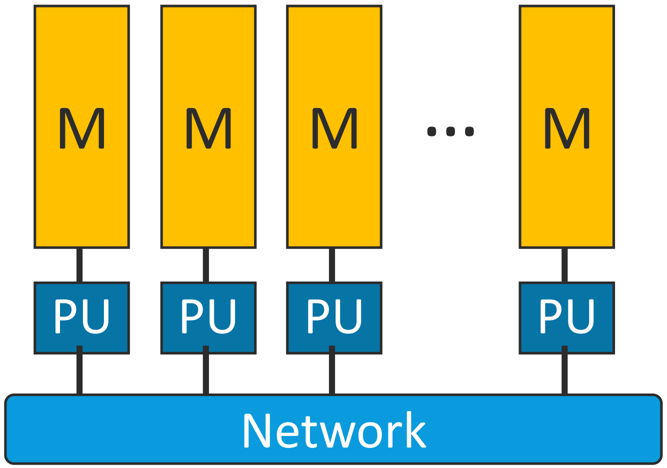
Pros/Cons of PRAM

+ Simple to use

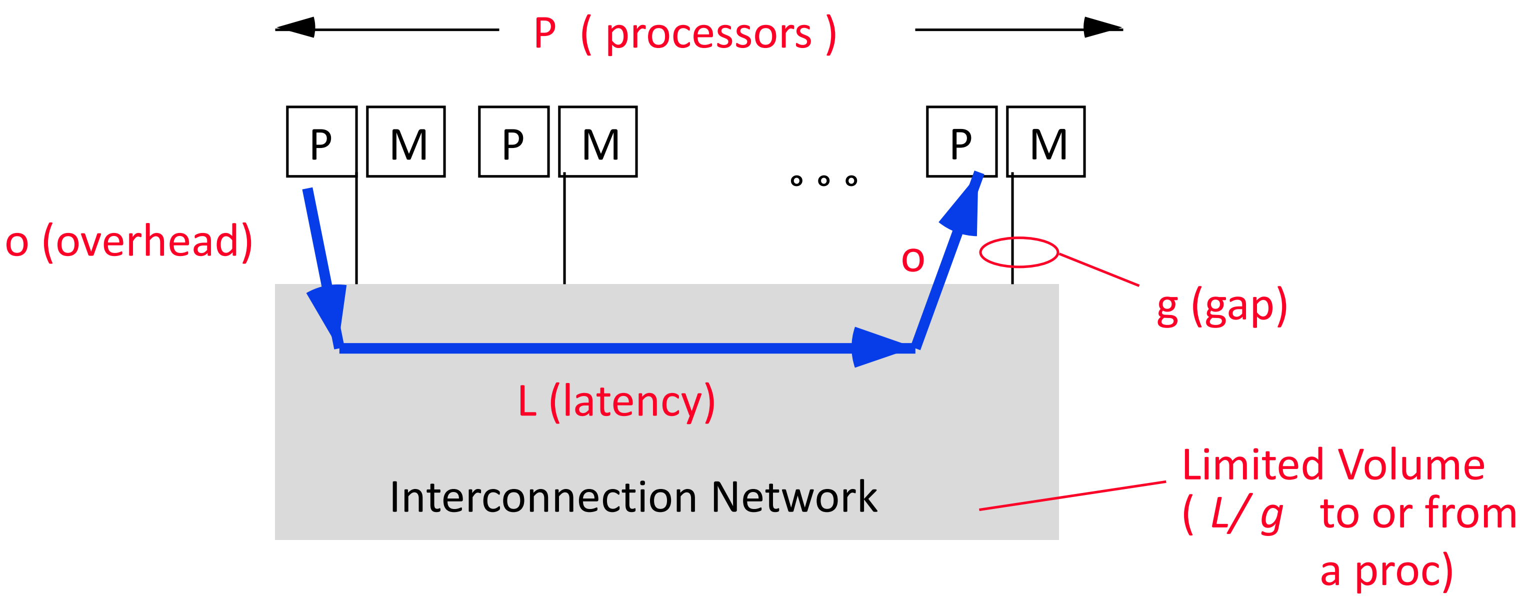
- Unrealistic → performance prediction is inaccurate

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

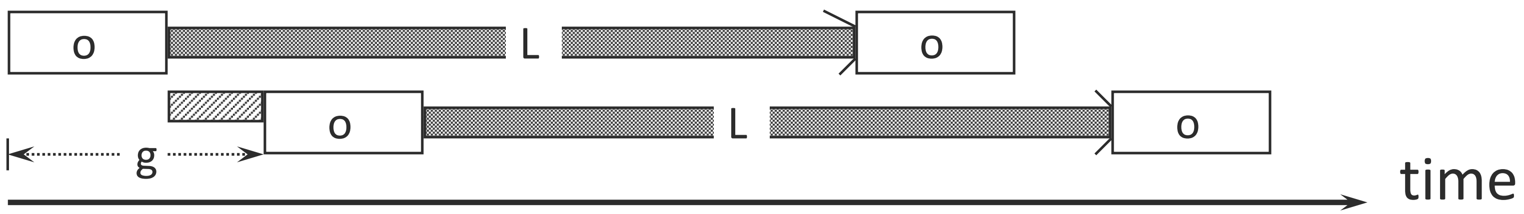


P (processors)
P
M
P
M
P
M
o (overhead)
o
g (gap)
L (latency)
Interconnection Network
Limited Volume
(L/ g to or from
a proc)

Using the LogP Model

- One processor sends *n* words to another processor

  $2o + L + g(n-1)$



Pros/Cons of the LogP Model

+ Simple, 4 parameters

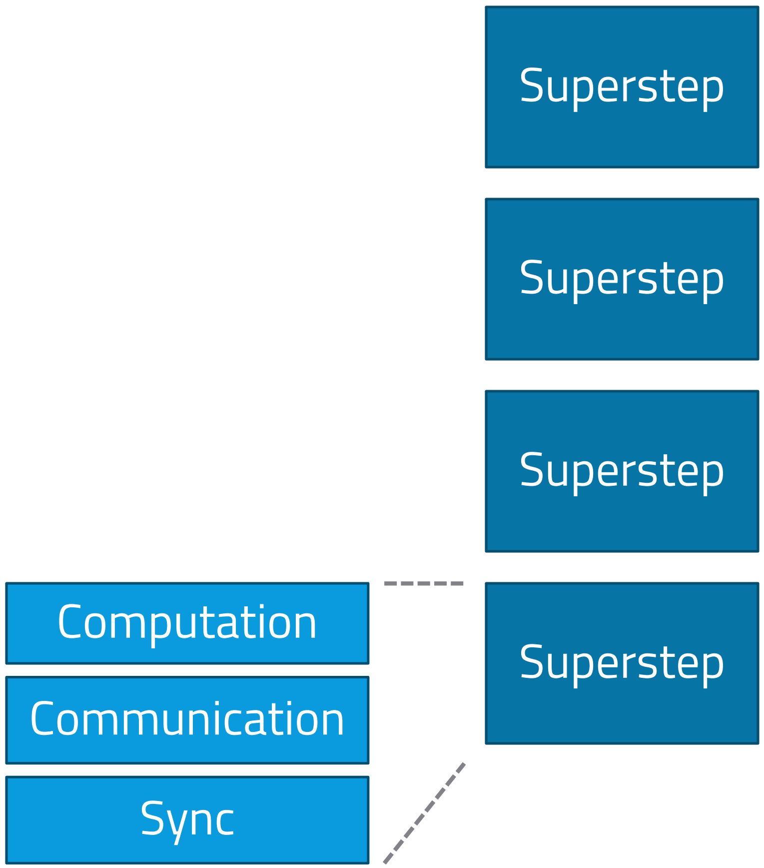
+ Can easily be used to guide the algorithm development

- Does not take contention into account → can sometimes underestimate communication time.

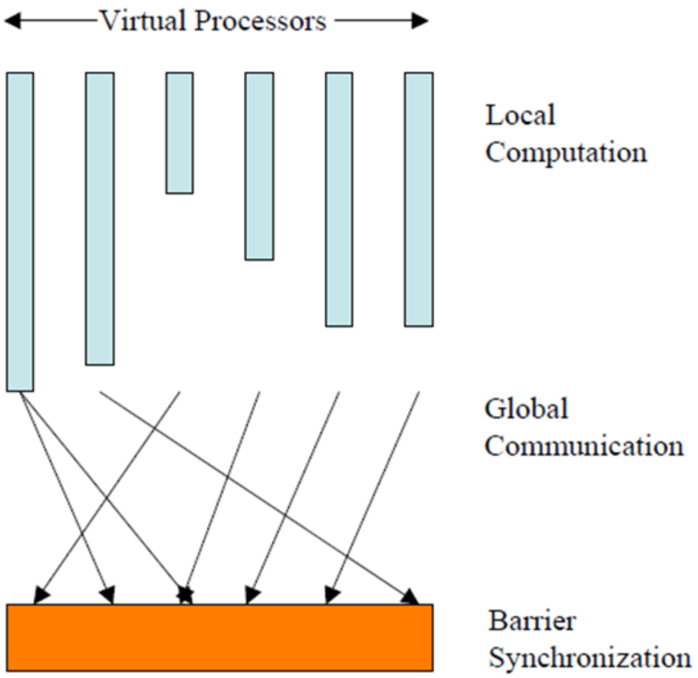
There are many variations to the LogP model, making it more accurate but more complex (e.g. LogGP, logGPC, pLogP, ...)

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:



BSP Model



Virtual Processors
Local Computation
Global Communication
Barrier Synchronization

BSP Model

- **Variables**
  - **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
- Assume w_i is the computation time for work on processor p during a superstep.
- Cost of a superstep: **?**

BSP Model

- **Variables**
  - **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
- Assume w_i is the computation time for work on processor p 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.
- You need to know the difference between: threads/processors/tasks.
- Knowing the hardware will help you generating a better task dependency graph.

- **Homework 1**