CSCI-GA.3033-012

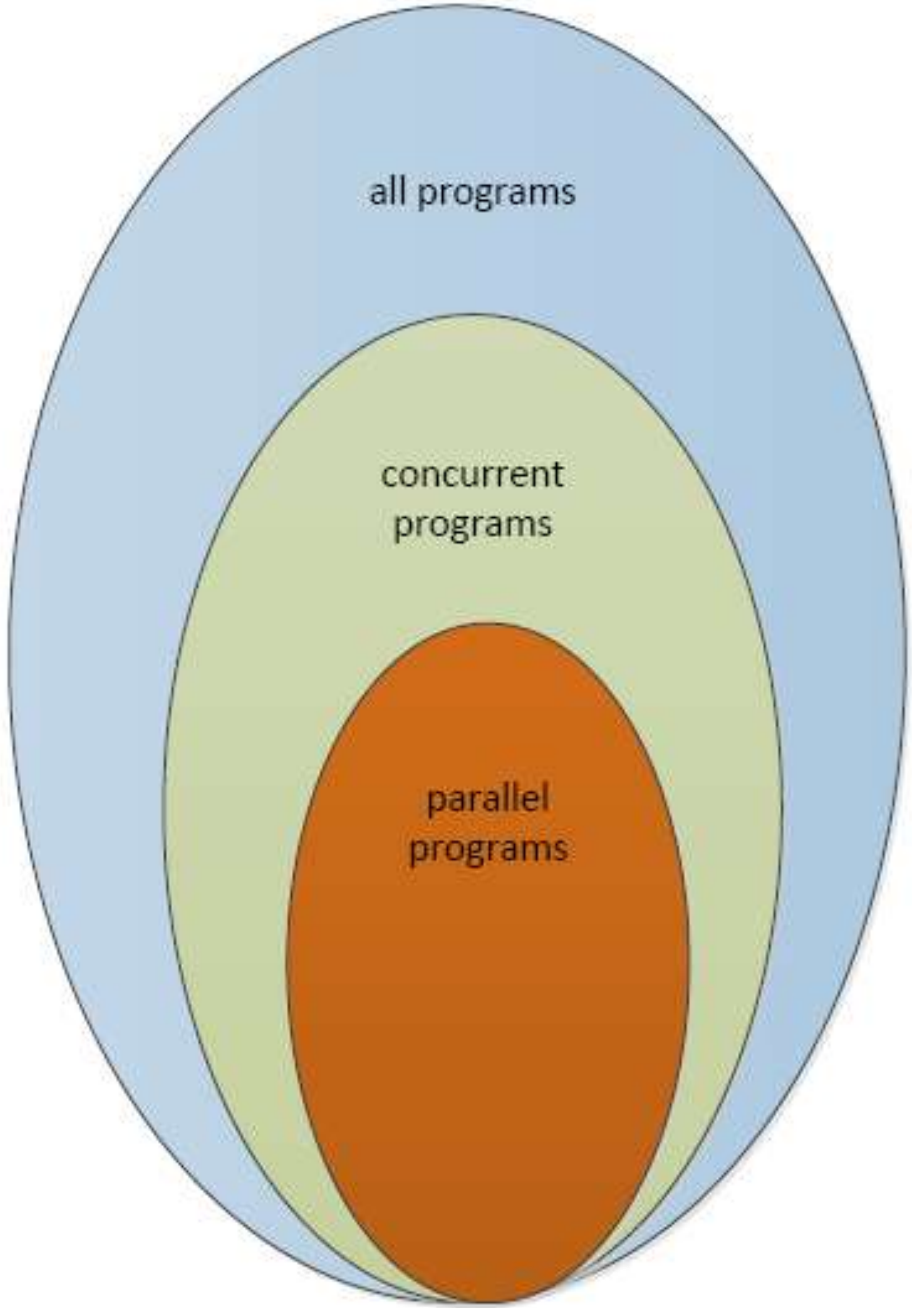
Multicore Processors: Architecture & Programming

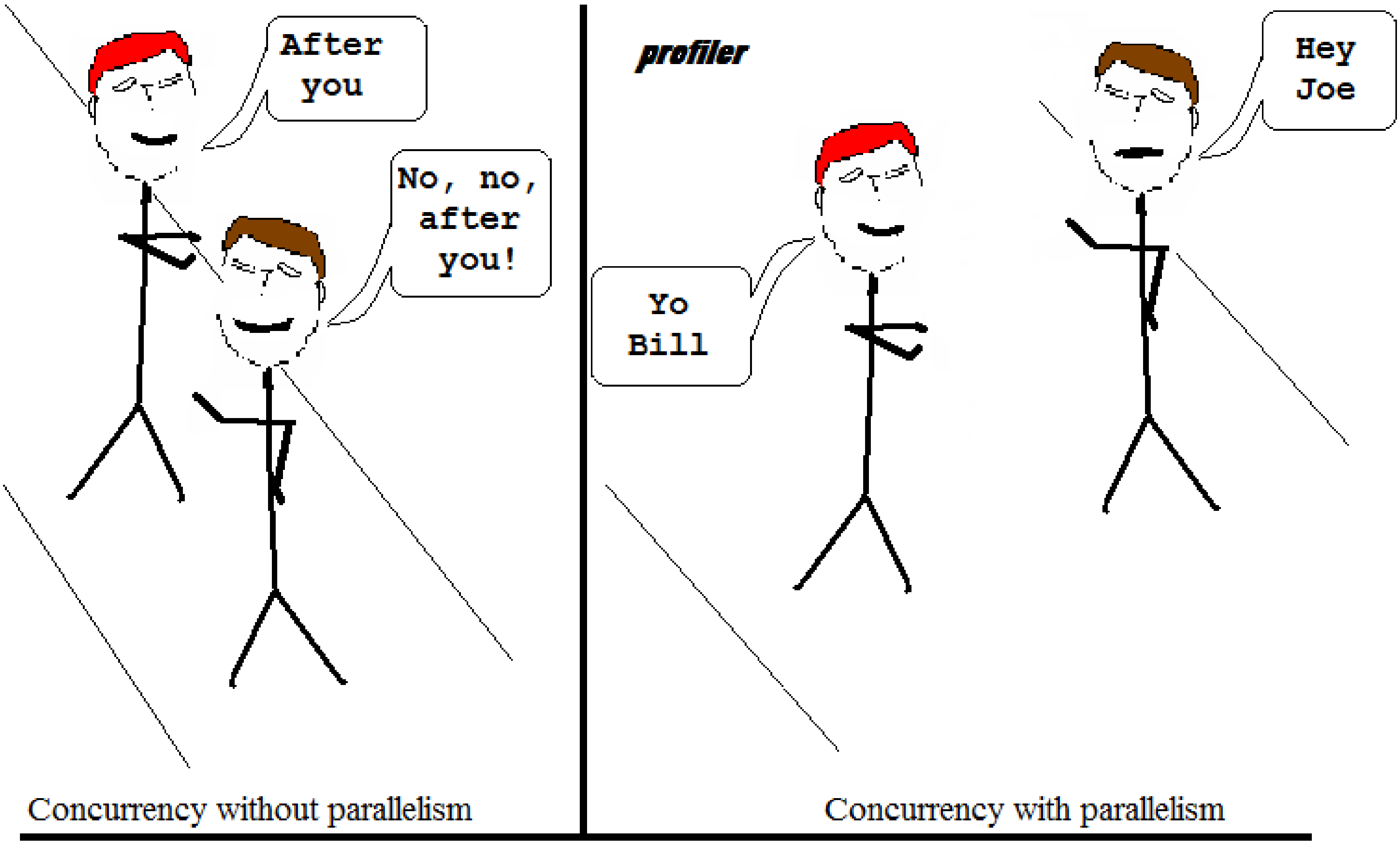
Lecture 2: Concurrency and Parallelism

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


all programs
concurrent programs
parallel programs



Performance tuning technique number 106: Concurrency vs. Parallelism

Questions!

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

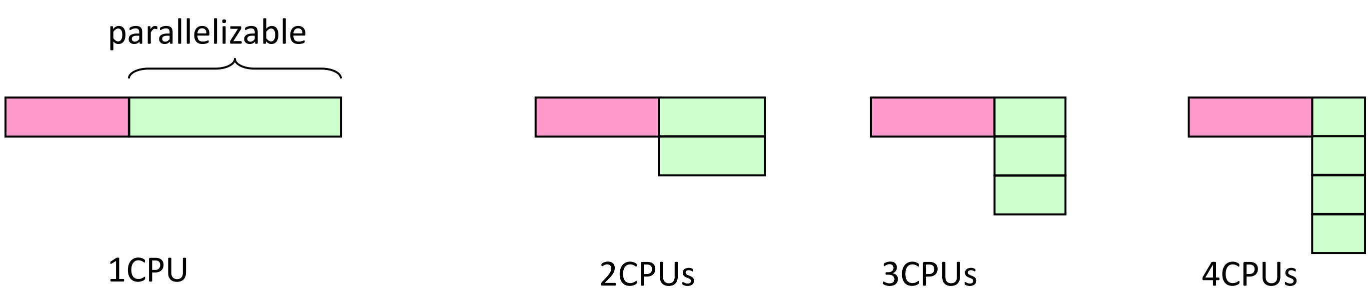
If we have 2 cores, do we get 2x speedup?

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



What Was Amdahl Trying to Say?

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

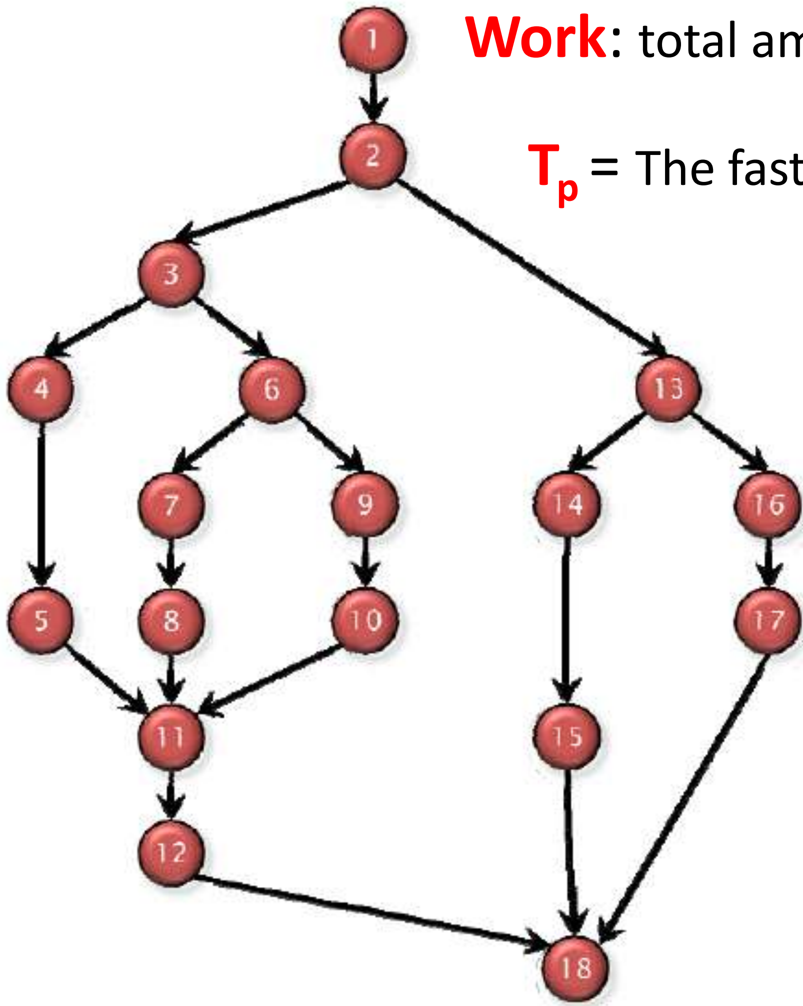
Was he right?

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

DAG Model for Multithreading

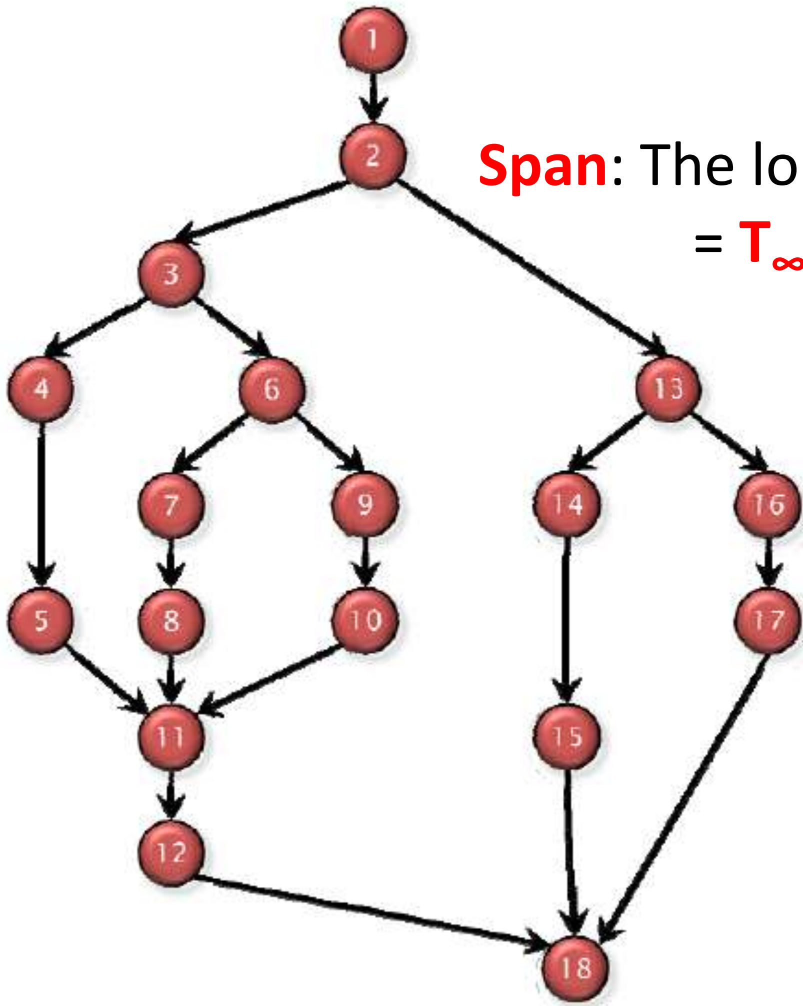


Work: total amount of time spent on all instructions

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:

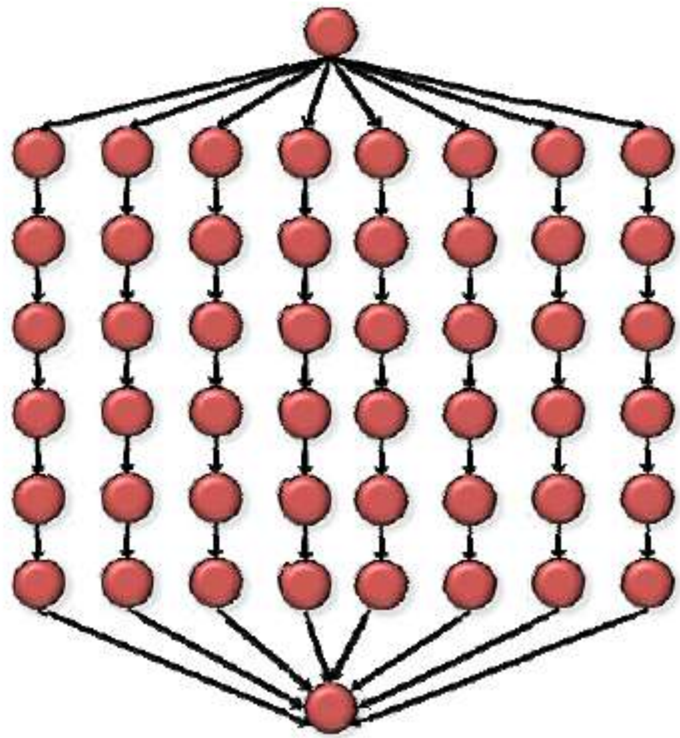
$$T_P \geq T_\infty$$

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: $T_\infty = 8$

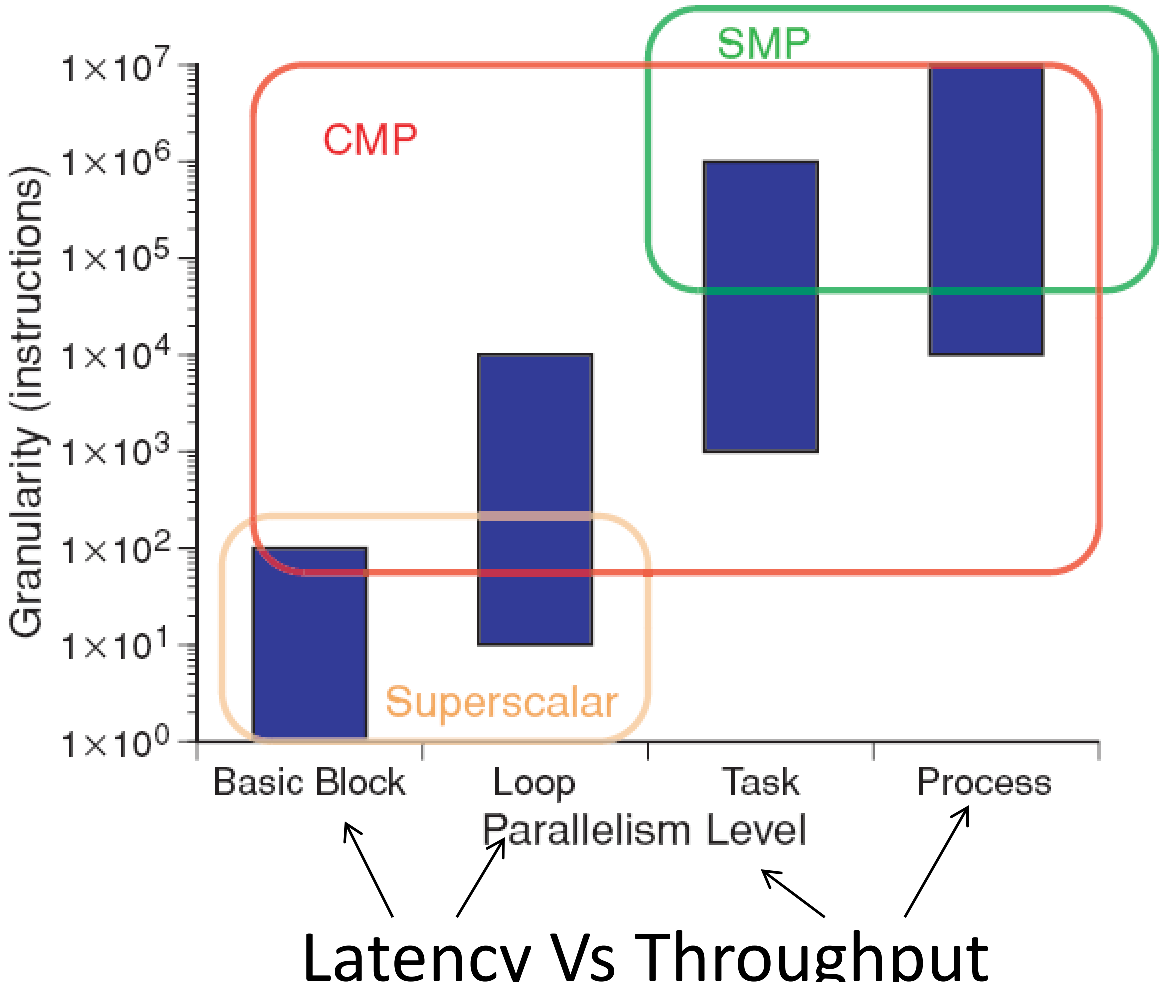
Parallelism: $T_1/T_\infty = 6.25$

At What Level Can We Reason About Parallelism (algorithm, high-level language, assembly)?

Is Thread The Only Parallelism Granularity?

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

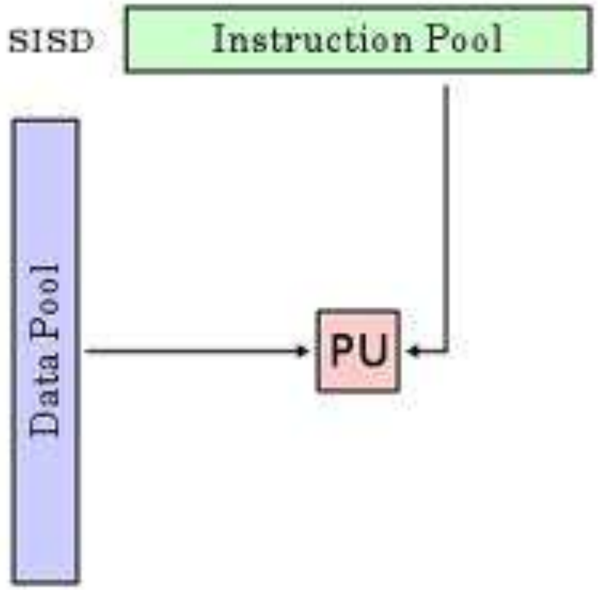
That Was The Software
How about the Hardware?

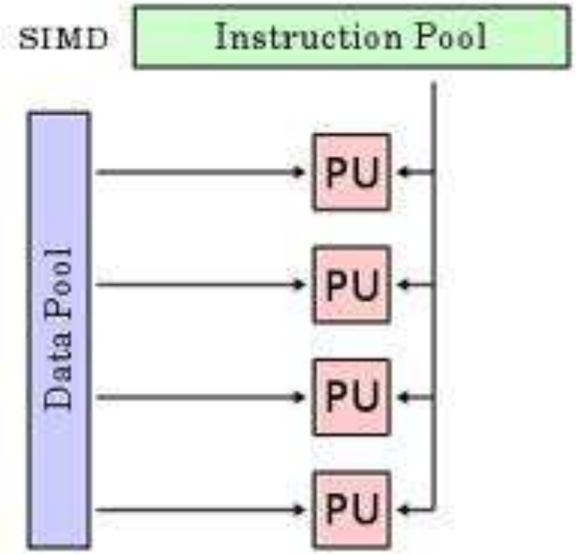


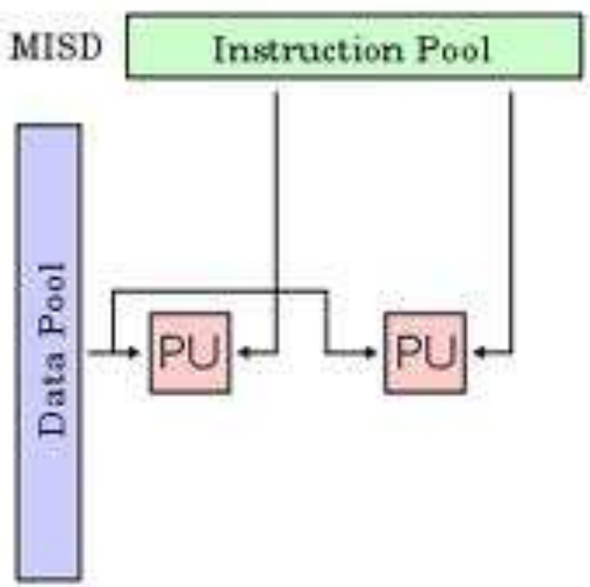
A Quick Glimpse on: Flynn Classification

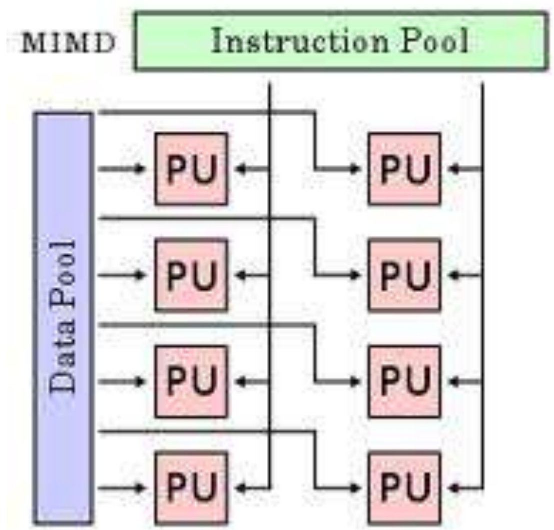
- A taxonomy of computer architecture
- Proposed by Michael Flynn in 1966
- It is based on two things:
  - Instructions
  - Data

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



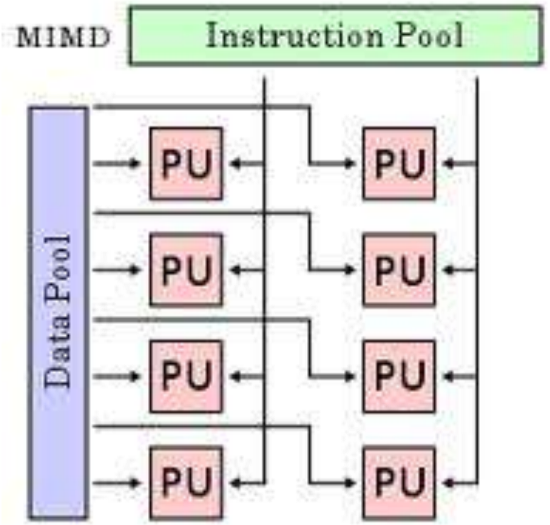






PU = Processing Unit

More About MIMD

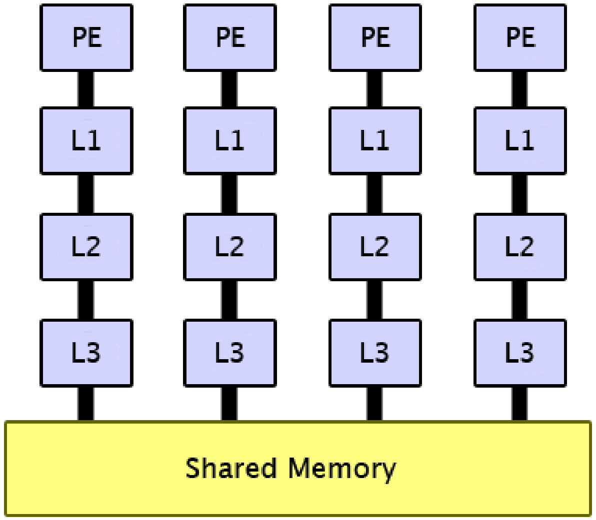


Shared-Memory

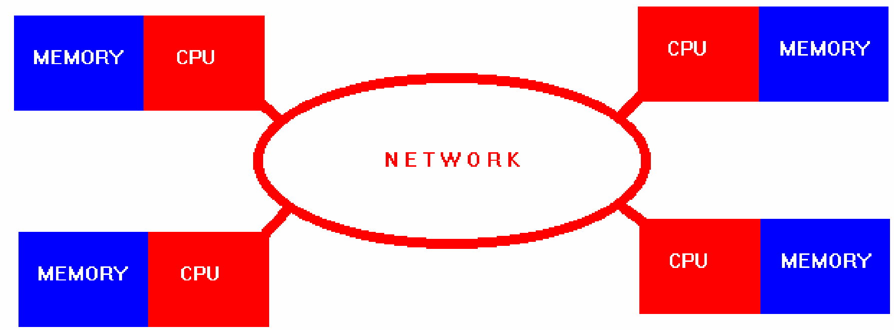
Distributed-Memory

Or hybrid

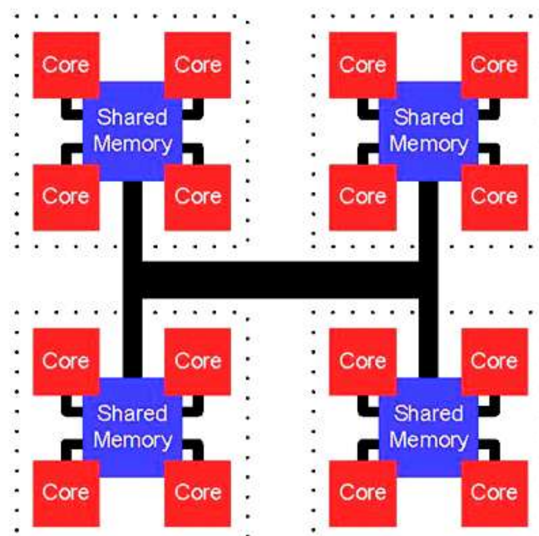
Shared Memory



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**

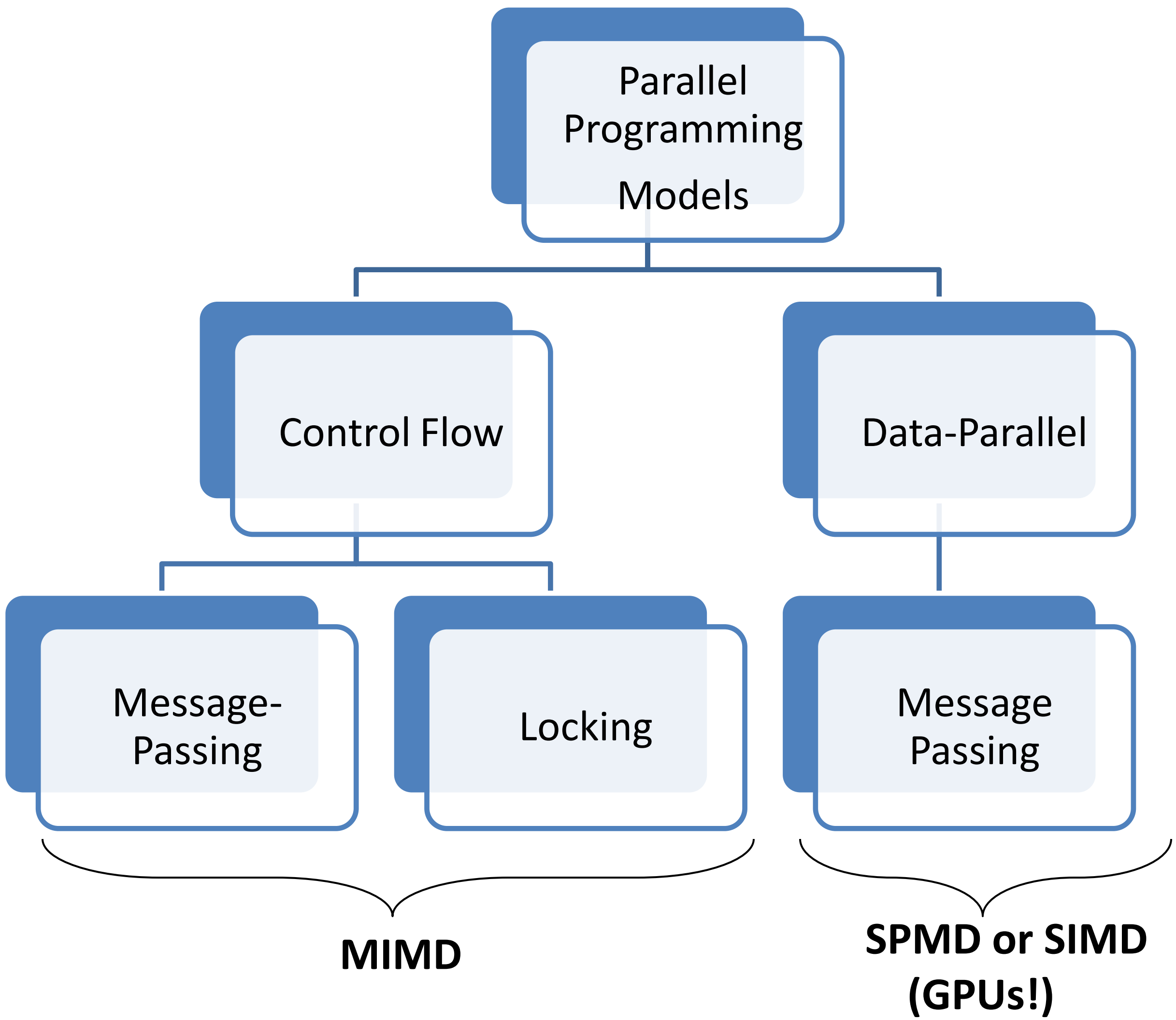
Dilemma:
- Parallel hardware is ubiquitous
- Parallel software is not!

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

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?

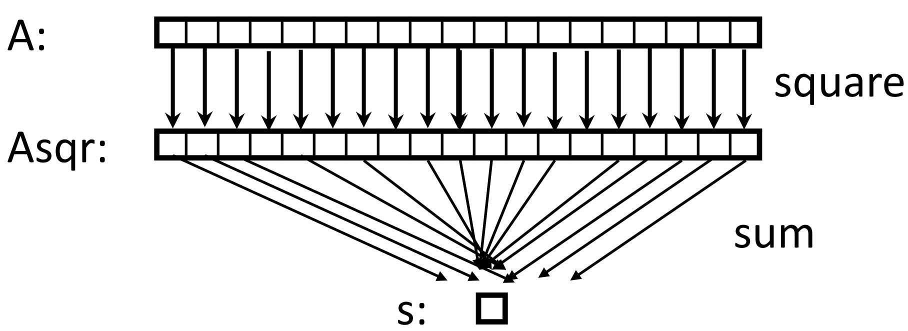
It Is Important to Note

- You can run any paradigm on any hardware (e.g. an MPI on shared-memory)
- The same program can have different type of parallel paradigms
- 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 type of parallelisms

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?**

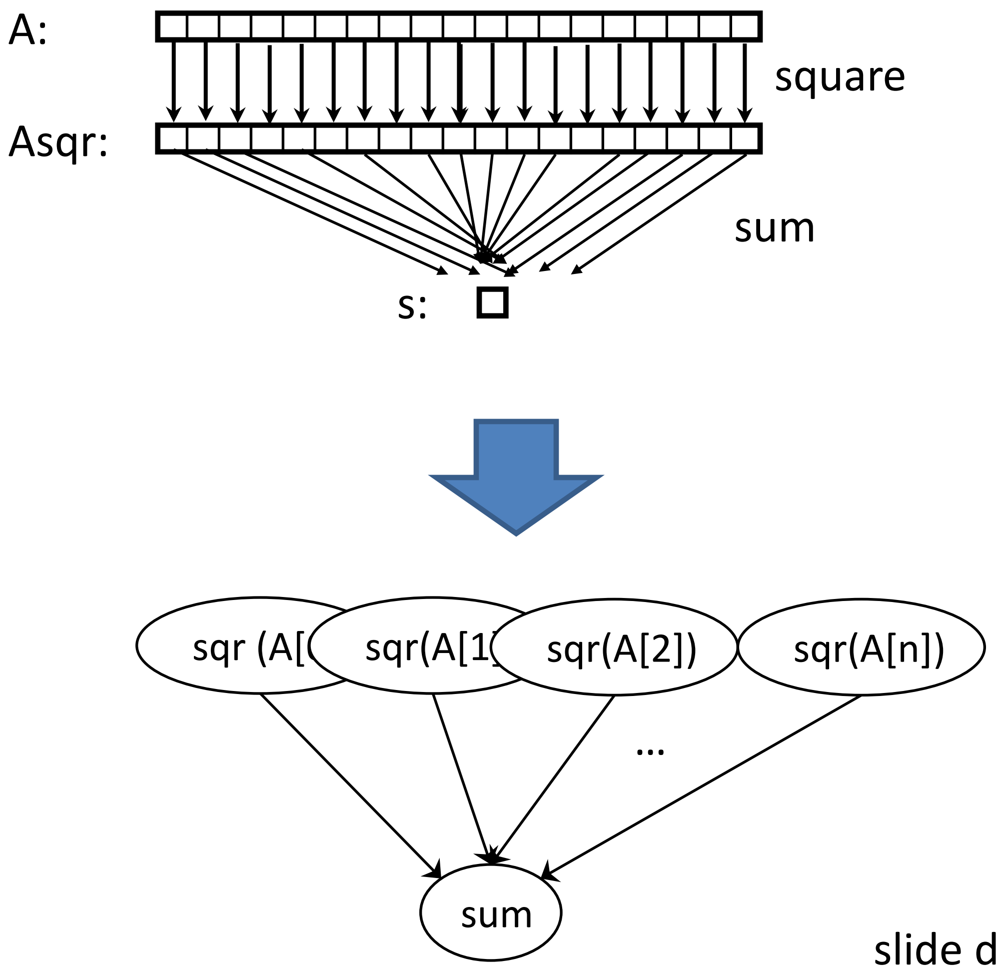
slide derived from Katherine Yelick

Example

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

slide derived from Katherine Yelick

Example



slide derived from Katherine Yelick

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

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 among: threads/processors/tasks.
- Knowing the hardware will help you generating a better task dependency graph.