CA463 Concurrent Programming

Lecturer Dr. Martin Crane

mcrane@computing.dcu.ie

Office: L2.51 Ph: x8974

Recommended Texts (online/in Library)

- **`www.computing.dcu.ie/~mcrane/CA463.html`**

- Recommended Text:

  Foundations of Multithreaded, Parallel and Distributed Programming, G.R. Andrews, Addison-Wesley, 2000. ISBN 0-201-35752-6

- Additional Texts:

  Principles of Concurrent and Distributed Programming, M. Ben-Ari, Addison-Wesley, 2006.

  Computer Organization and Architecture, W. Stallings, Pearson, 9th Ed, 2012

  The SR Programming Language, Concurrency in Practice, G.R. Andrews and R.A. Olsson, Benjamin/Cummings, 1993.

  Java Concurrency in Practice, Brian Goetz et al, Addison-Wesley, 2012

  Using MPI: Portable Parallel Programming with the Message Passing Interface , W. Gropp, E. Lusk, A. Skjellum, 2nd Edition, MIT Press 1999

Course Outline

- Introduction to Concurrent Processing
- Critical Sections and Mutual Exclusion
- Semaphores
- Monitors
- Message Passing & Other Communication Mechanisms (SR & Java)
- Enhanced Concurrency in Java
- Load Balancing and Resource Allocation
- Fault Tolerance

Assessment

- 25% Continuous Assessment:
  - Java Project,
  - Set Week 5/6
  - To be handed at end of Semester
- 75% January Exam (3 from 4 Questions)

Lecture 1: Intro to Concurrent Processing

- Basic Definitions;
- A Simple Analogy;
- More Advanced Definitions;
- Architectures for Concurrency;
- Concurrent Speedup;
- Applications to Multicore Computing.

A Preliminary Definition….

Concurrency is a property of systems in which several computations can be in progress simultaneously, and potentially interacting with each other.

The computations may be executing on multiple cores in the same chip, preemptively time-shared threads on the same processor, or executed on physically separated processors...

Concurrency, Somethin' for what ails ya?

- Multicore Systems;
- Fast Networks;
- Concurrency:

  the solution to today's (& tomorrow's) *Grand Challenge* problems in Climate Change, Bioinformatics, Astrophysics etc. etc.?

A Clear(er) Analogy of Concurrency

- *Concurrency* is about dealing with lots of things at once.
- *Parallelism* is about doing lots of things at once.

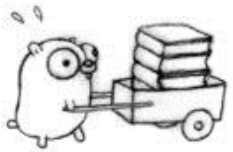
These are not the same, but they are related.

- *Concurrency* is about structure, *parallelism* is about execution.
- *Concurrency* provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.

- Example:
  - Concurrent: using MS Word, mediaplayer.
  - Parallel: calculating a Vector dot product, cells being updated in excel

A Simple Example Problem to Make Things More Concrete[1]

- Move a pile of obsolete language manuals to the incinerator.

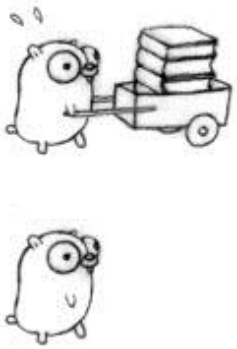


- With only one gopher this will take too long.

1. From R. Pike “Concurrency is not Parallelism”, Waza, Jan 11 2012

A Simple Example With Gophers (cont'd)

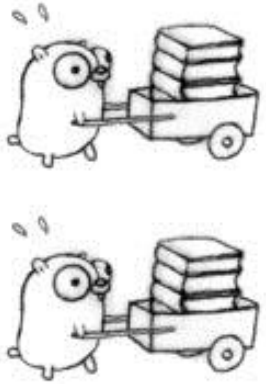
- Maybe more gophers…..



- More gophers are not enough; they need more carts.

More Gophers

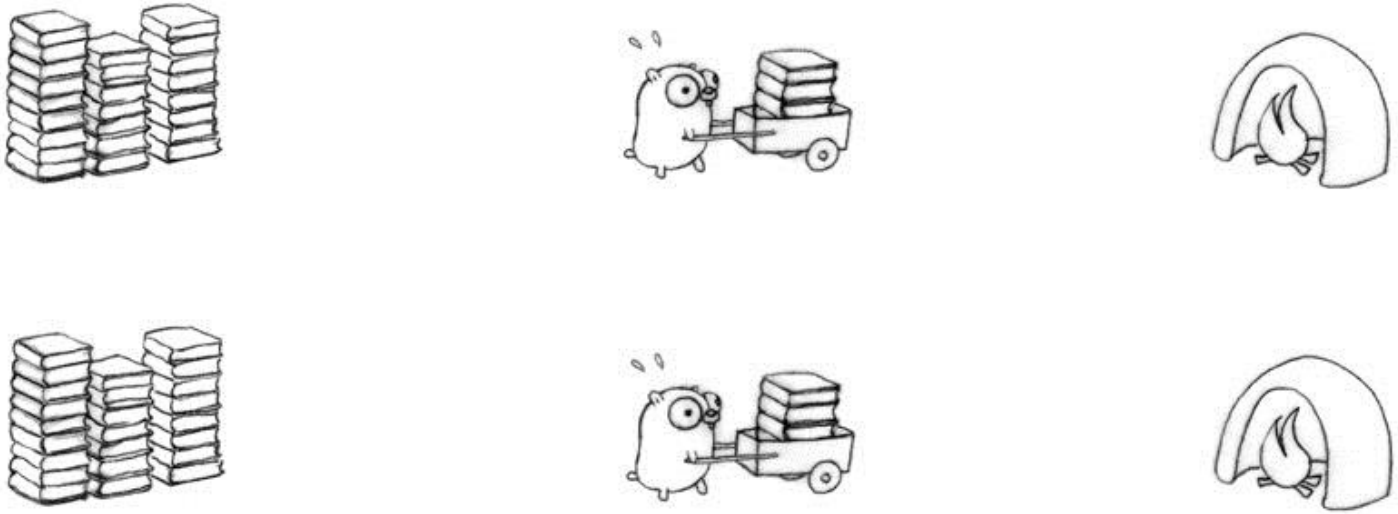
- More gophers and more carts



- Faster, but gives rise to bottlenecks at pile, incinerator.
- Also need to synchronize the gophers.
- A message (i.e. communication btw gophers) will do.

More Gophers

- Double everything
- Remove the bottleneck; make them really independent.

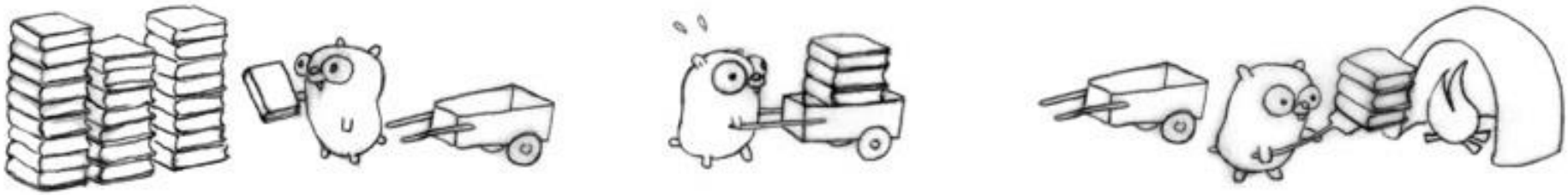


- This will consume input twice as fast.
- This is the *concurrent composition* of two gopher procedures.

More Gophers

- A Note on *Concurrent Composition*
- This design is not automatically parallel!
- What if only one gopher is moving at a time?
- Then it's still concurrent (that's in the design), just not parallel.
- However, it's automatically parallelizable!
- Moreover the concurrent composition suggests other models…

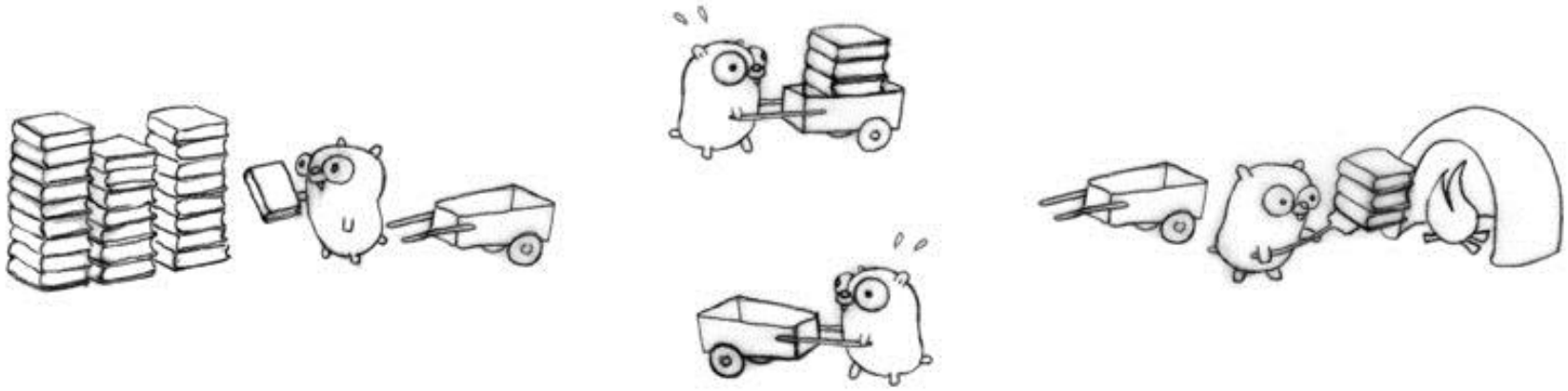
More Gophers: Another Design



- Three gophers in action, but with likely delays.
- Each gopher is an independently executing procedure, plus coordination (communication).

Even More Gophers: Finer-grained concurrency

- Add another gopher procedure to return empty carts.



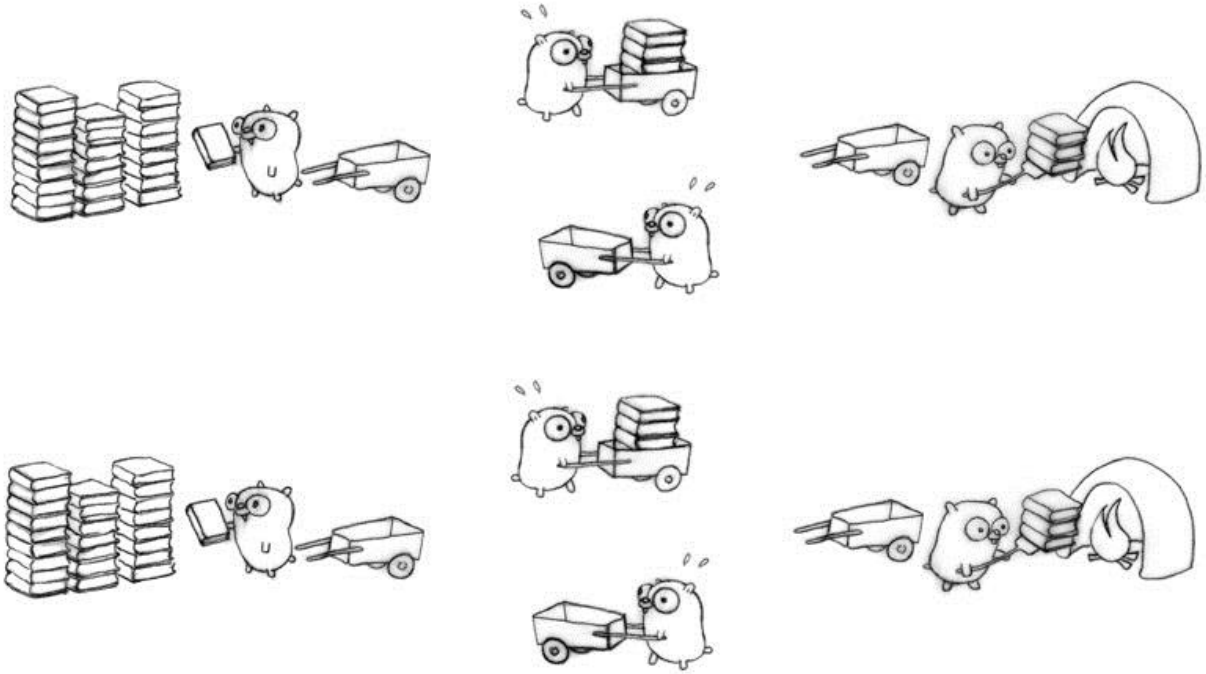
- 4 gophers in action for better flow, each doing a simple task.
- If we arrange everything right (implausible but not impossible), that's 4 times faster than our original 1-gopher design.

Even More Gophers (cont'd): Finer-grained concurrency

- Observation:
  - We improved performance by adding a concurrent procedure to the existing design.
  - More gophers doing more work; it runs better.
  - This is a deeper insight than mere parallelism.
- Four distinct gopher procedures:
  - load books onto cart
  - move cart to incinerator
  - unload cart into incinerator
  - return empty cart
- Different concurrent designs enable different ways to parallelize.

A Simple Example With Gophers (cont'd): More parallelization!

- Can now parallelize on the other axis; the concurrent design makes it easy. 8 gophers, all busy!



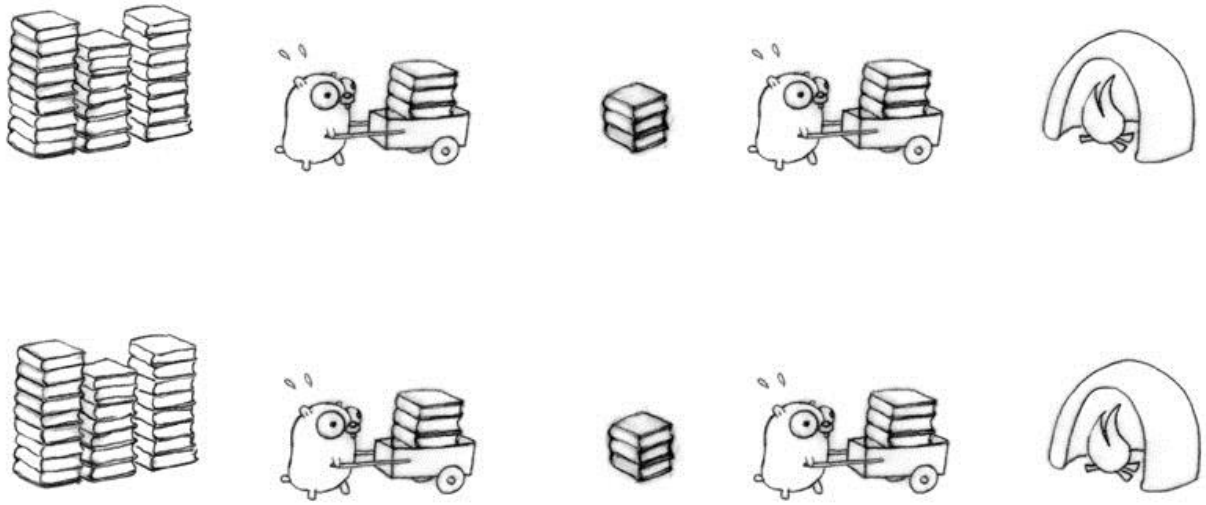
- Or maybe no parallelization at all! Keep in mind, even if only 1 gopher is active at a time (zero parallelism), it's still a correct & concurrent solution.

Even More Gophers (cont'd): Another design

- Here's another way to structure the problem as the concurrent composition of gopher procedures.
- Two gopher procedures, plus a staging pile.

Even more Gophers (cont'd): Another Design

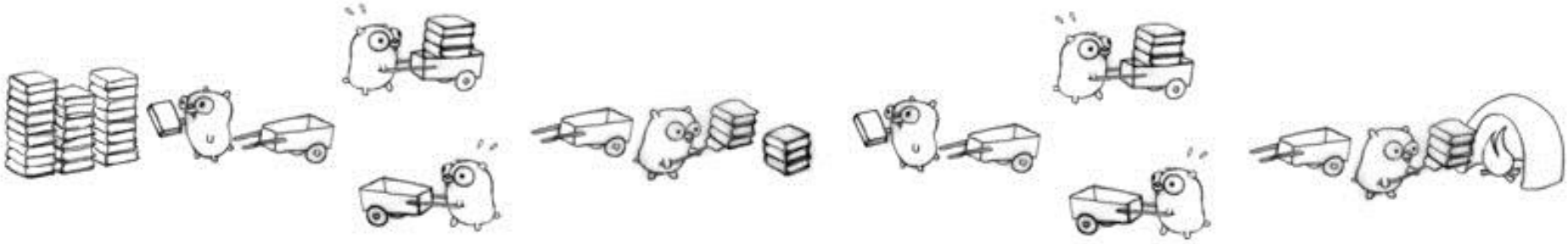
- Parallelize this in the usual way:



- i.e. run more concurrent procedures to get more throughput.

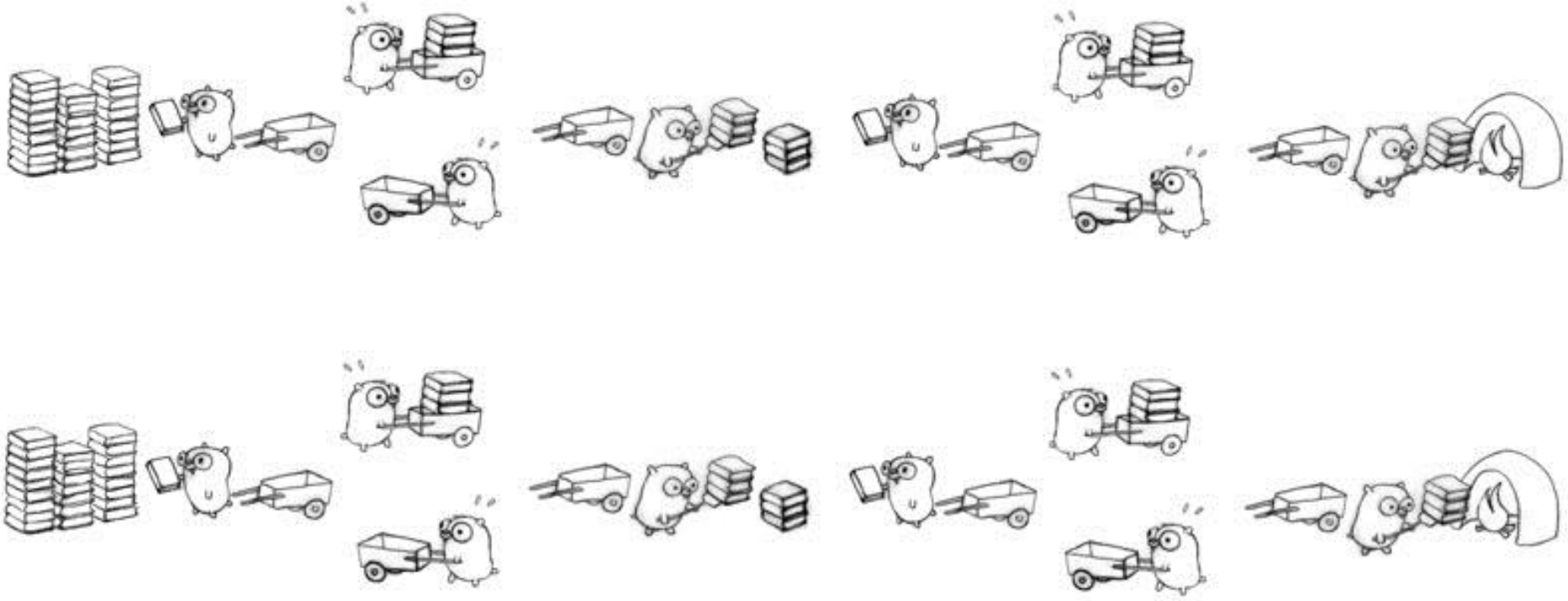
Even More Gophers (cont'd): A Different Way…

- Bring a staging pile to the multi-gopher concurrent model:



Even More Gophers (cont'd): A Different Way…

- Full on optimization:



The Lesson from all This…

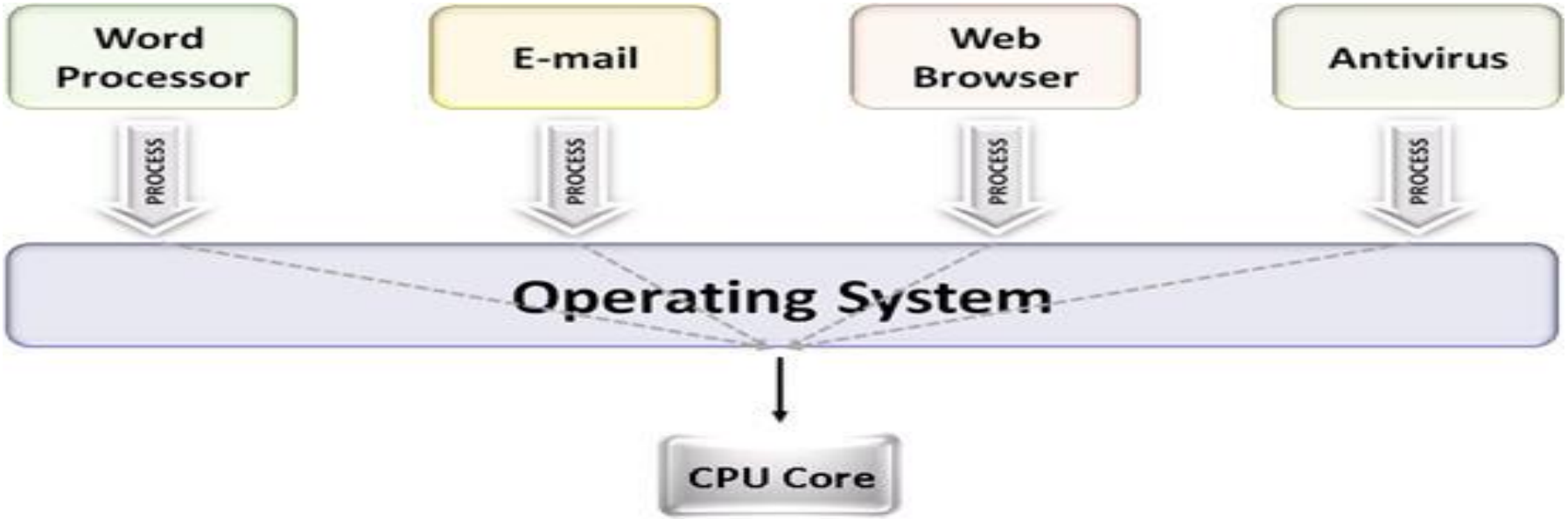
- There are many ways to break the processing down.
- That's concurrent design.
- Once we have the breakdown, parallelization can fall out & correctness is easy.
- In our book transport problem, substitute:
  - book pile => web content
  - gopher => CPU
  - cart => marshaling, rendering, or networking
  - incinerator => proxy, browser, or other consumer
- It becomes a concurrent design for a scalable web service with Gophers serving web content.

What have we learned thus far?

- Concurrency is the *ability* to run several parts of a program or several programs in parallel.
- A modern multi-core computer has several CPU's or several cores within one CPU.
- Here we distinguish between processes and threads:
  - *Process:* runs independently and isolated of other processes. It cannot directly access shared data in other processes. The process resources are allocated to it via the OS, e.g. memory & CPU time.
  - *Threads:* (or lightweight processes) have their own call stack but can access shared data. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data.
- Don’t confuse a Process with a Processor (i.e. s/w & h/w)

Concurrency: Some More Definitions

- Multi-tasking: A single CPU core can only run 1 task at once,
  - means CPU actively executes instructions for that one task
- Problem solved by scheduling which task may run at any given time and when another waiting task will get a turn.
- Amounts to time-slicing between the tasks



Single-core systems schedule tasks on 1 CPU to multitask

Concurrency: Some More Definitions (cont'd)

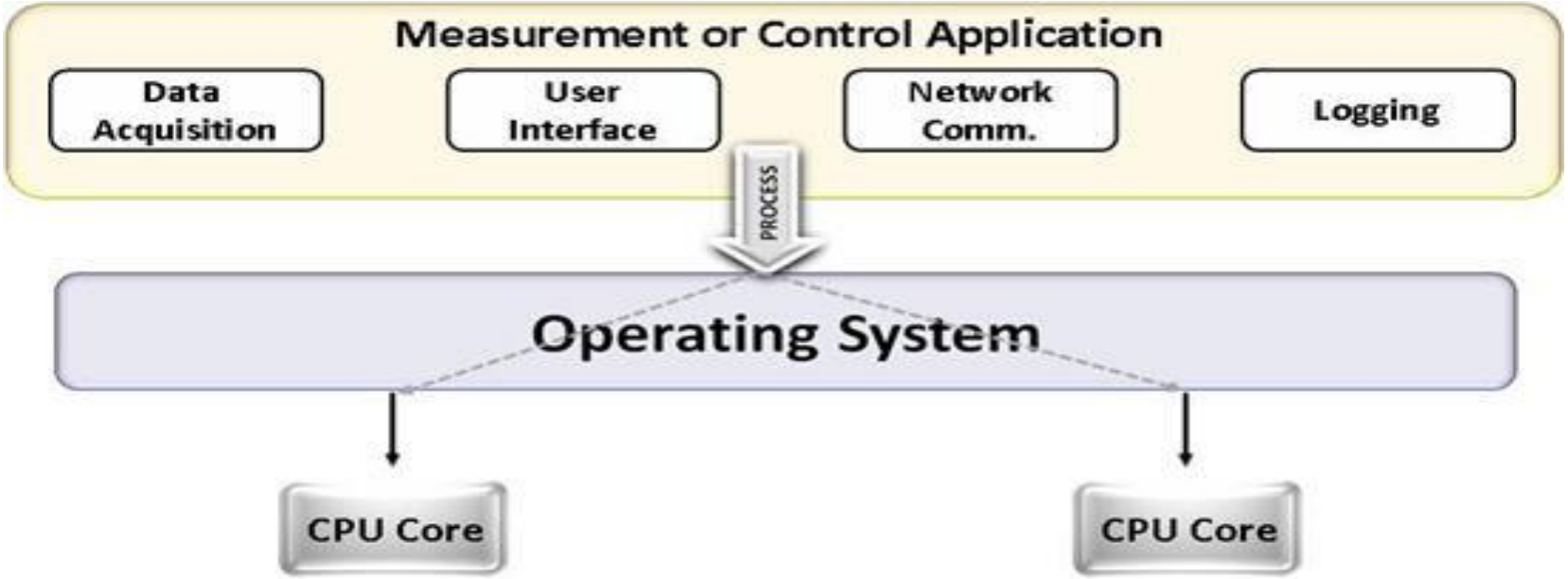
- Multi-Core: multitasking OSs can truly run multiple tasks in parallel.
- Multiple computing engines work independently on different tasks.
- OS Scheduling dictates which task runs on the CPU Cores.

Dual-core systems enable multitasking operating systems to execute 2 tasks simultaneously

Concurrency: Some More Definitions (cont'd)

- Multi-threading: extends multitasking to application-level,
  - subdivides operations in one application into individual threads.
- Each thread can (conceptually) run in parallel.

- OS divides processing time not only among different applications, but also among each of the threads within an application.

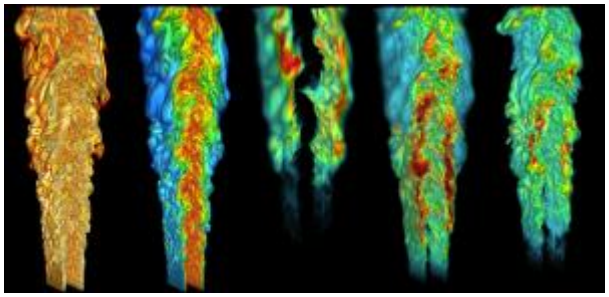


Dual-core system enables multithreading

Concurrency: Side note on Multi-threading

- *Implicit and Explicit Multi-Threading*
- All commercial processors and most experimental ones use *explicit multithreading*
  - Concurrently execute instructions from different explicit threads
  - Interleave instructions from different threads on shared pipelines or parallel execution on parallel pipelines
- *Implicit multithreading* is concurrent execution of multiple threads extracted from single sequential program
  - Implicit threads defined statically by compiler or dynamically by hardware

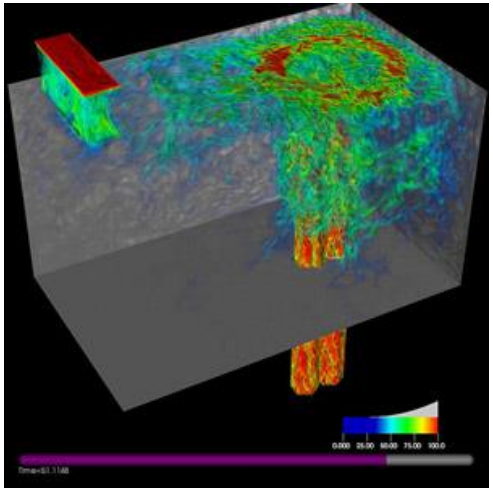
Why Concurrency?



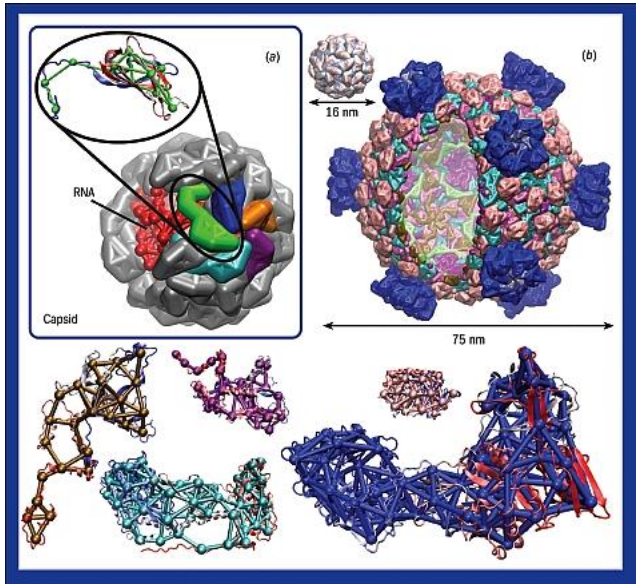
© ExACT

Combustion/Turbulence Applications

Nuclear Safety Applications



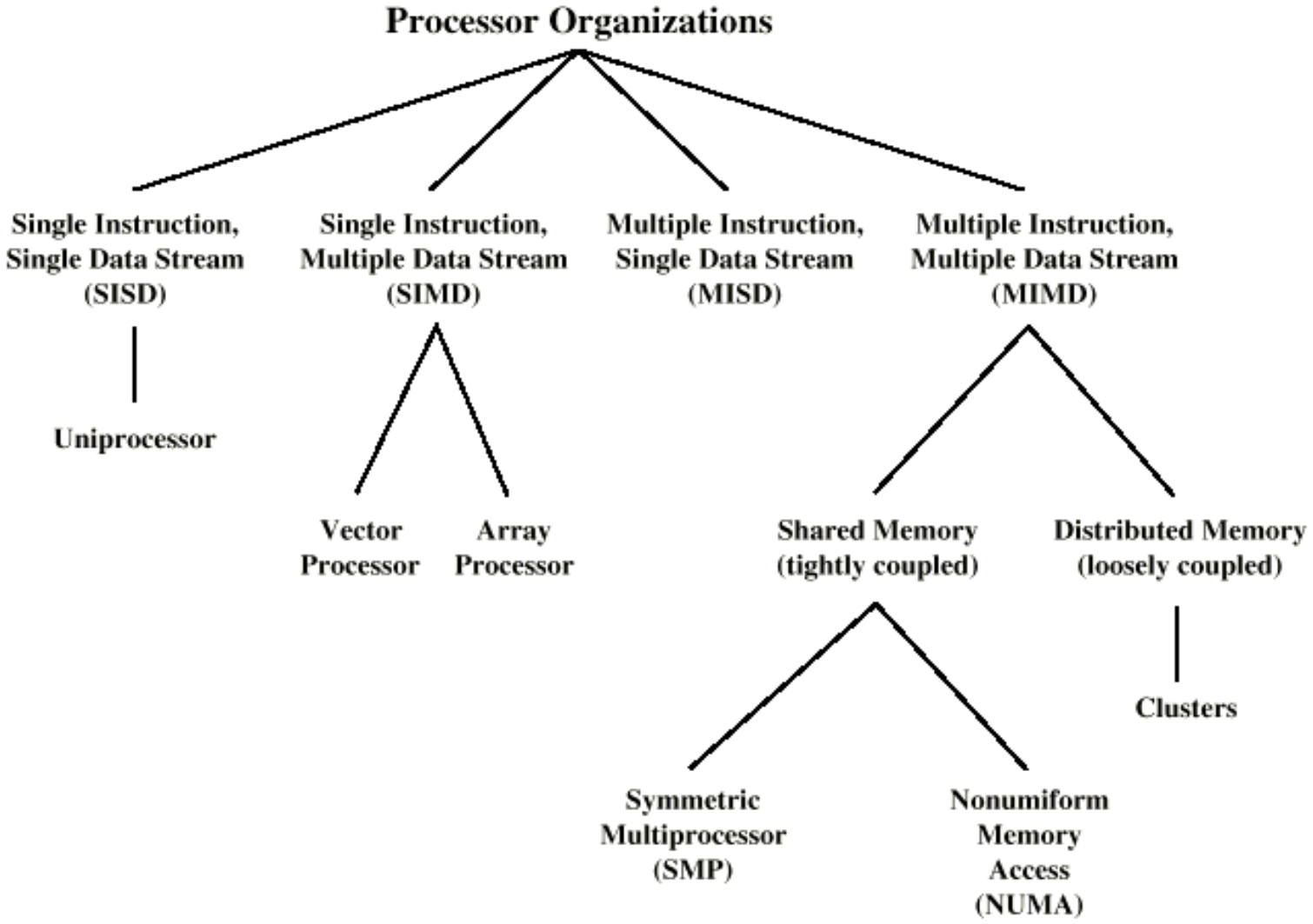
© Argonne National Labs



© UIUC

Life Science/Bioinformatics/Biomedical Applications

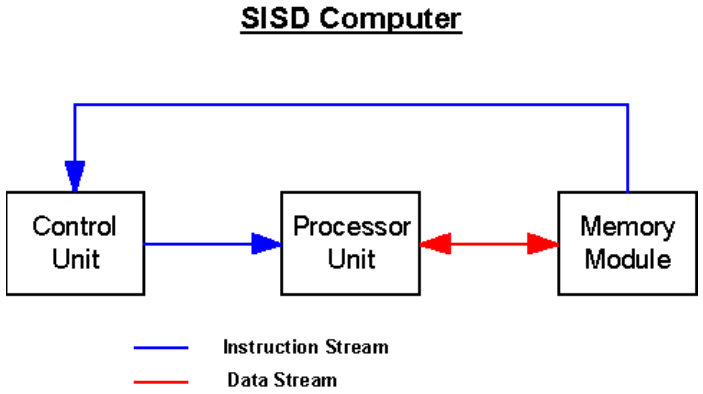
Computer Architecture Taxonomies for Concurrency



Computer Architecture Taxonomies (cont'd)

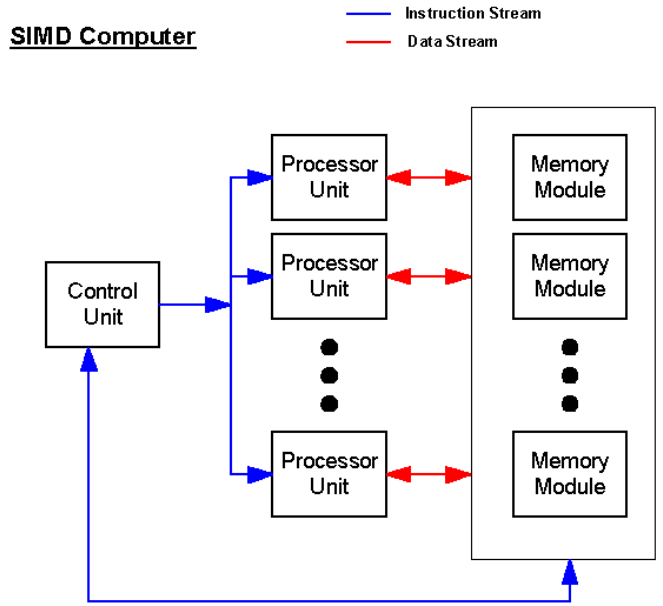
Classification According to Flynn

- *SISD Single Instruction Single Data*



  - Single processor
  - Single instruction stream
  - Data stored in single memory
  - Uni-processor
  - Old but still common (RISC)

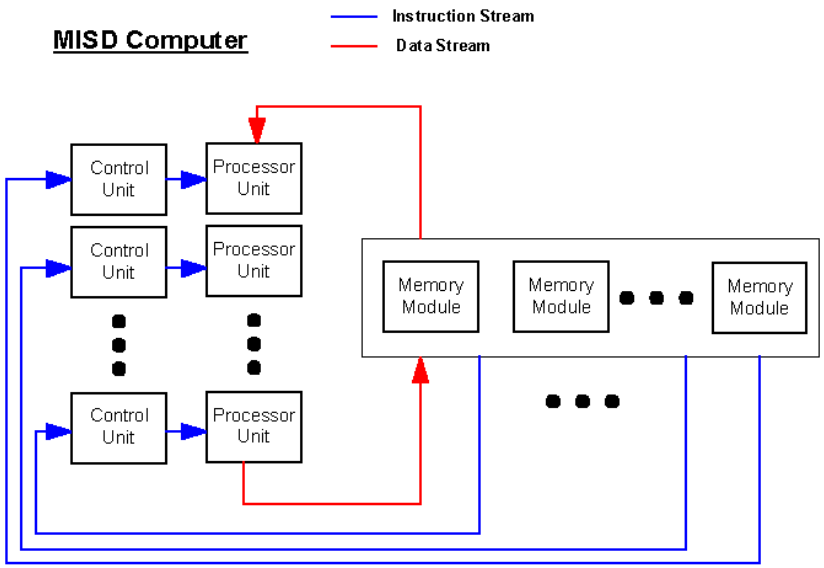
- *SIMD Single Instruction Multiple Data*



  - Single machine instruction controls simultaneous execution
  - Number of processing elements each with associated data memory
  - Each instruction executed on different set of data by different processors
  - Vector & array processors (for graphics)

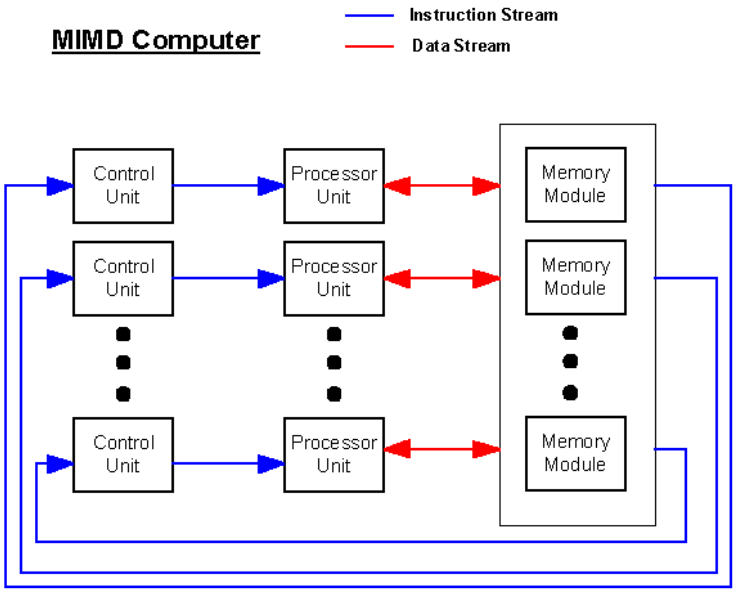
Computer Architecture Taxonomies (cont'd)

- *MISD Multiple Instruction Single Data*



  - Sequence of data
  - Transmitted to set of processors
  - Each processor executes different instruction sequence
  - No prototype so far (Cryptographic Algorithms?)

- *MIMD Multiple Instruction Multiple Data*



  - Set of processors
  - Simultaneously execute different instruction sequences on different data
  - SMPs, clusters and NUMA systems (more later)
  - Most modern Supercomputers use MIMD with SMPs for specific tasks.

More on MIMD

- General purpose proc'r; each can process all instructions necessary
- Further classified by method of processor communication
- Tight Coupling

1. *Symmetric Multi-Processing (SMP)*

– Processors share memory & communicate via that shared memory
– Memory access time to given area of memory is approximately the same for each processor

2. *Asymmetric Multi-Processing (ASMP)*

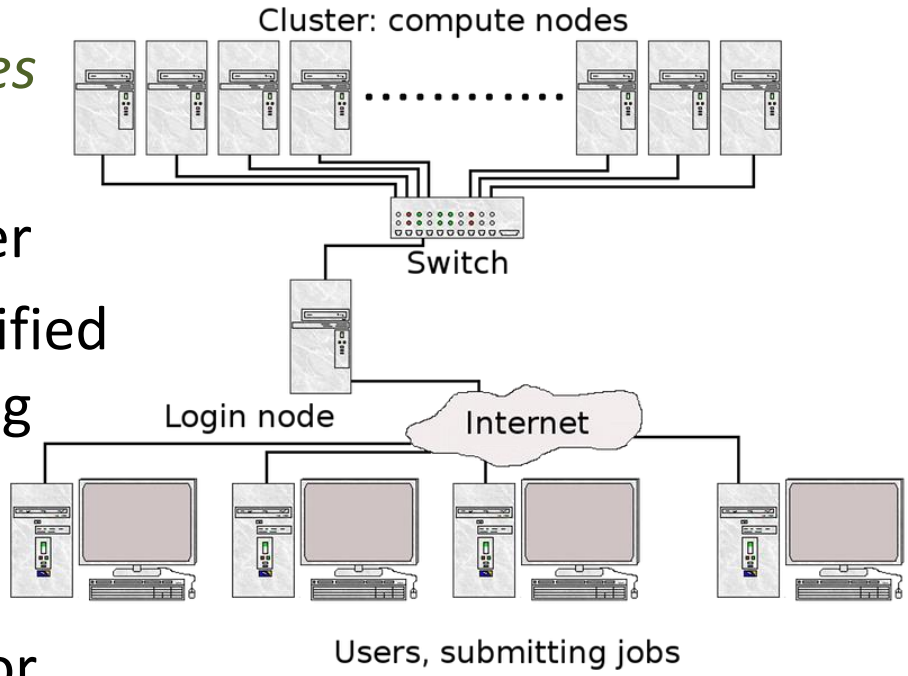
– For SMP some cores used more than others (& some mostly unused)
– With ASMP consume power & increase compute power only on demand

3. *Nonuniform Memory Access (NUMA)*

– Access times to different regions of memory may differ depending on memory location relative to a processor
– Benefits limited to particular workloads, e.g. servers where data are often associated strongly with certain tasks or users

More on MIMD (cont'd)

- Loose Coupling: *Clusters*
  - Collection of independent *nodes* (uniprocessors or SMPs)
  - Interconnected to form a cluster
  - Working together (often) as unified resource or different users using *partitions*
  - *Jobs* can be real-time or *batch*
  - Communication via fixed path or network connections
  - Alternative to SMP giving high performance & high availability



TOP500 JUNE 2014

PRESENTED BY

ICL UT INNOVATIVE COMPUTING LABORATORY at UNIVERSITY of TENNESSEE

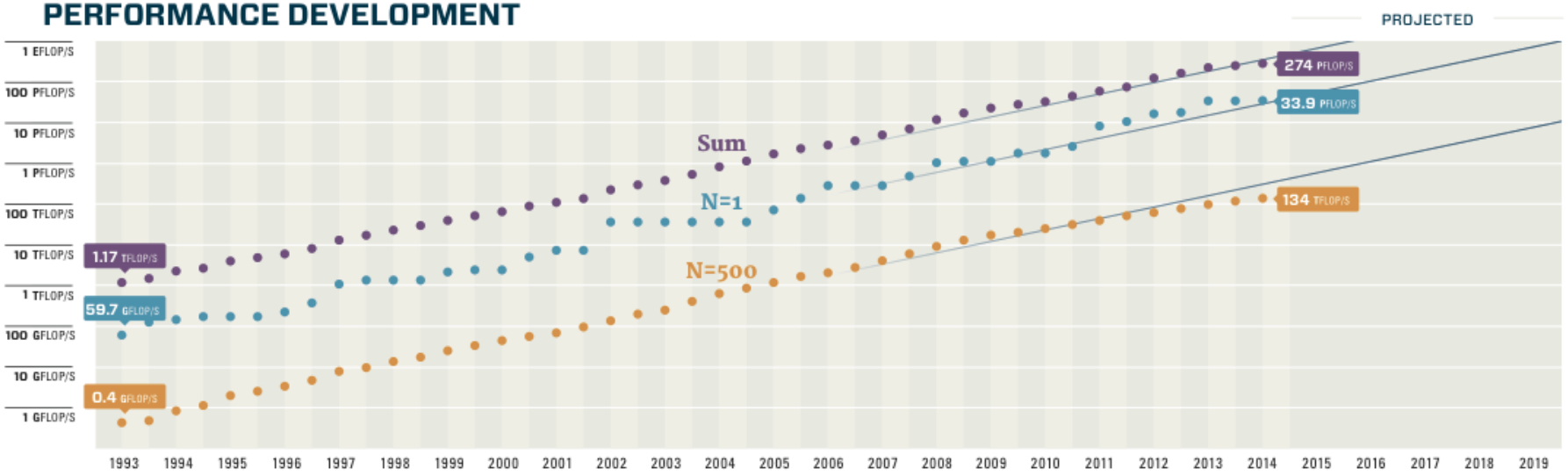
BERKELEY LAB Lawrence Berkeley National Laboratory

PROMETEUS PROFESSOR MEUER TECHNOLOGIEBERATUNG

FIND OUT MORE AT www.top500.org

	NAME	SPECS	SITE	COUNTRY	CORES	RMAX PFLOP/S	POWER MW
1	Tianhe-2 (Milkyway-2)	NUDT, Intel Ivy Bridge (12C, 2.2 GHz) & Xeon Phi (57C, 1.1 GHz), Custom interconnect	NSCC Guangzhou	China	3,120,000	33.9	17.8
2	Titan	Cray XK7, Operon 6274 (16C 2.2 GHz) + Nvidia Kepler GPU, Custom interconnect	DOE/SC/ORNL	USA	560,640	17.6	8.2
3	Sequoia	IBM BlueGene/Q, Power BQC (16C 1.60 GHz), Custom interconnect	DOE/NNSA/LLNL	USA	1,572,864	17.2	7.9
4	K computer	Fujitsu SPARC64 VIIIfx (8C, 2.0GHz), Custom interconnect	RIKEN AICS	Japan	705,024	10.5	12.7
5	Mira	IBM BlueGene/Q, Power BQC (16C, 1.60 GHz), Custom interconnect	DOE/SC/ANL	USA	786,432	8.59	3.95

PERFORMANCE DEVELOPMENT



TOP500®
JUNE 2014

PRESENTED BY

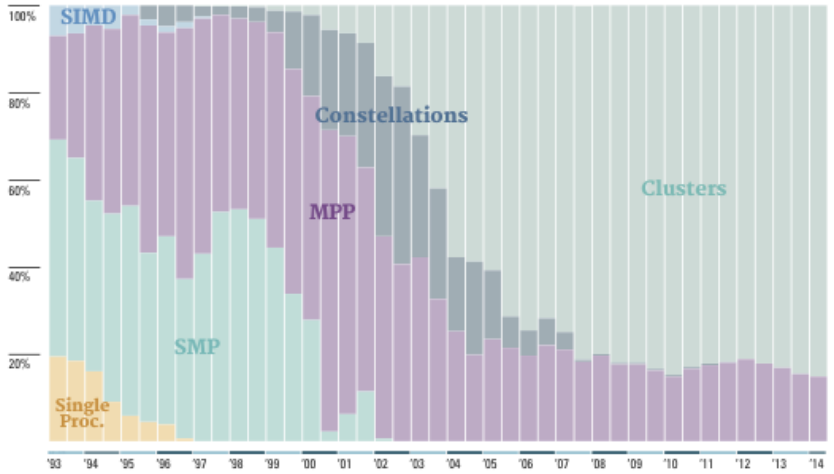
ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE

BERKELEY LAB Lawrence Berkeley National Laboratory

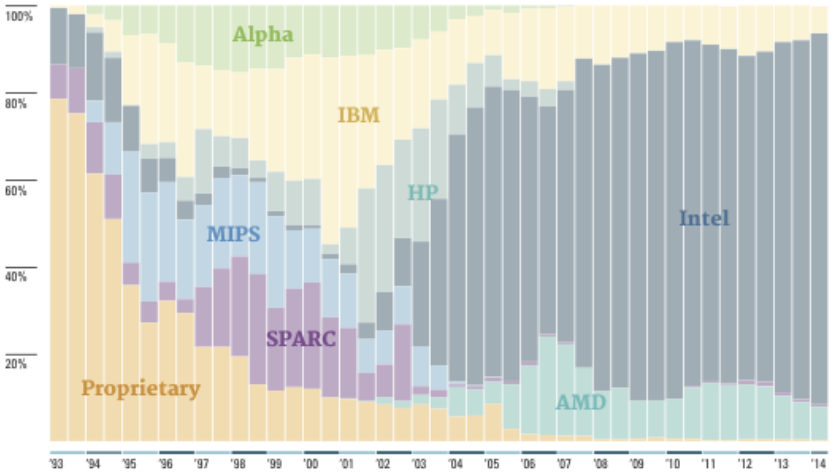
PROMETEUS PROFESSOR MEUER TECHNOLOGIEBERATUNG

FIND OUT MORE AT
www.top500.org

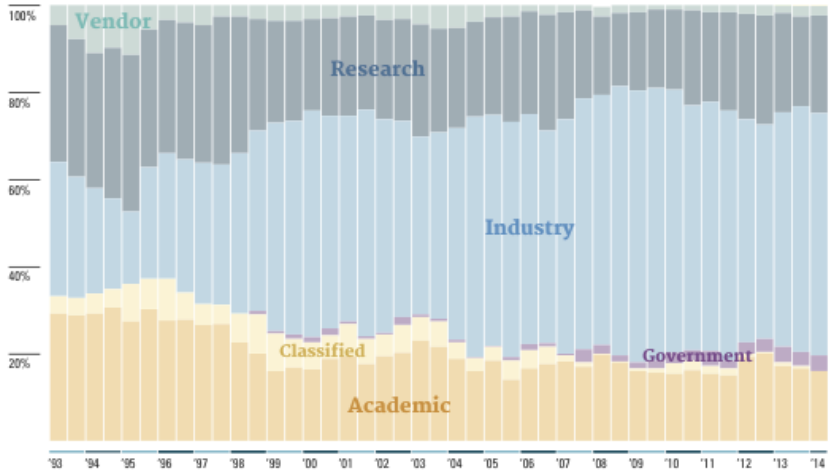
ARCHITECTURES



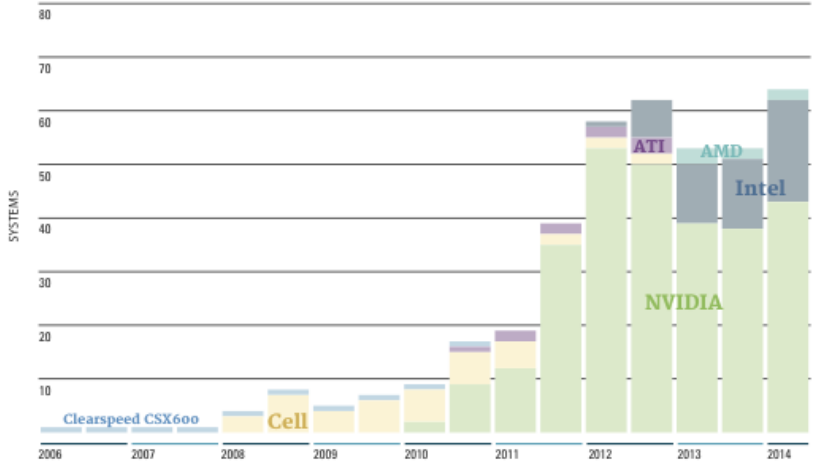
CHIP TECHNOLOGY



INSTALLATION TYPE



ACCELERATORS/CO-PROCESSORS



Good News!

- This is Great!

- All we need to solve really difficult problems is to throw multithreading on multicore machines at them.
- No more problems…….

- There is always a but and this time it's a BIG one!

Amdahl's Law

- Ultimately we would like the system throughput to be directly proportional to the number of CPUs.
- Unfortunately this 'perfect' scenario is impossible to realise for various reasons:
  - Poor Design (how problem is broken down & solved);
  - Code Implementation (I/O, inefficient use of memory…);
  - Operating System Overhead;
  - Etc., etc.,

Amdahl's Law (cont'd)

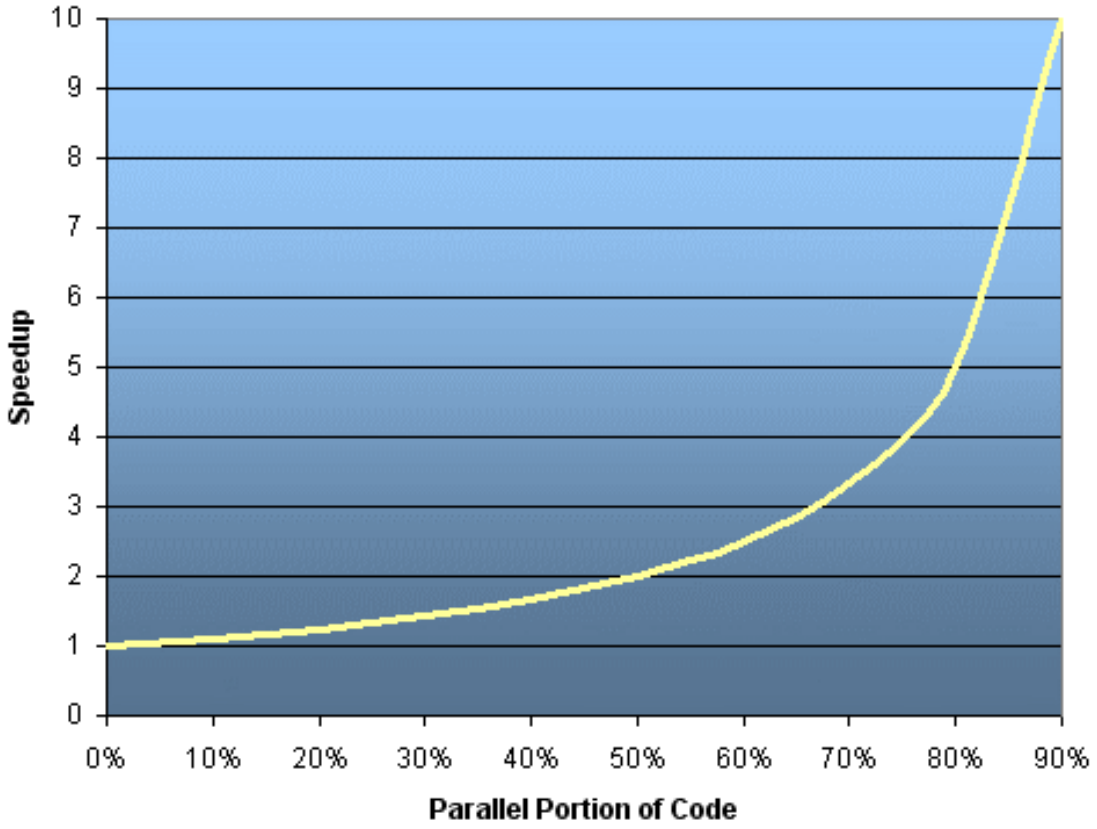
- Gene Amdahl divided a program into 2 sections,
  - one that is inherently serial
  - and the other which can be parallel.
- Let α be the fraction of program which is inherently serial.
- Then the Speedup

$$S = \frac{T(\alpha+(1-\alpha))}{T(\alpha+\frac{1-\alpha}{P})} = \frac{P}{1+(P-1)\alpha}$$

- So, if the serial fraction α = 5%, then S $\leq$ 20.

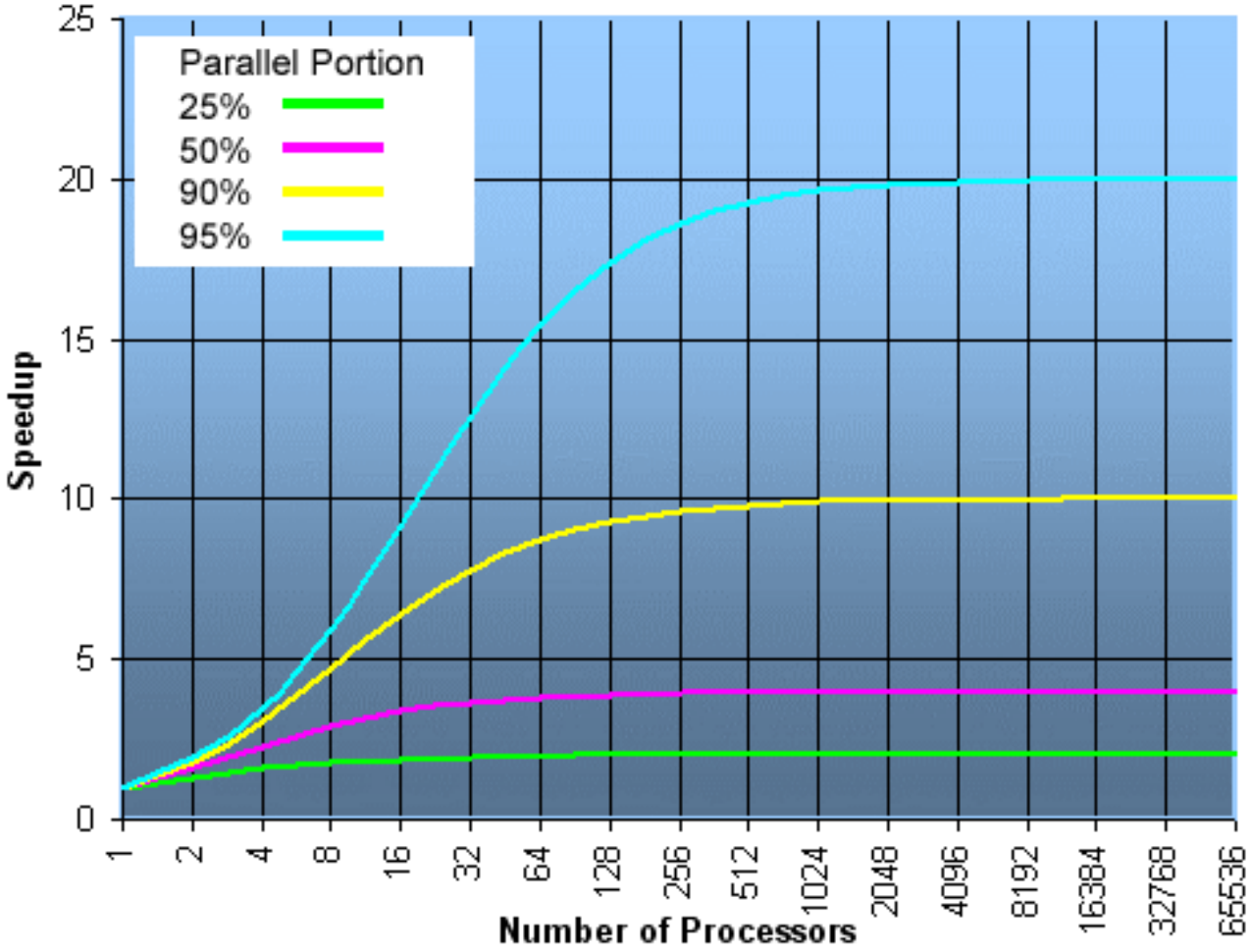
Amdahl's Law (cont'd)

- How does speedup change with different α?



Amdahl's Law (cont'd)

- Graph of S against 1 - α for different P



Amdahl's Law (cont'd)

- The Sandia Experiments
  - The Karp prize was established for the first program to achieve a speed-up of 200 or better.
  - In 1988, a Sandia laboratories team reported a speed-up of over 1,000 on a 1,024 processor system on three different problems.
- How is this possible?

- Moler's Law
  - Amdahl's Law assumes that serial to parallel fraction, α, is independent of the size of the program.
  - The Sandia experiments showed this to be false.
  - As the problem size increased the inherently serial parts of a program remained the same or increased at a slower rate than the problem size.
  - So Amdahl's law should be

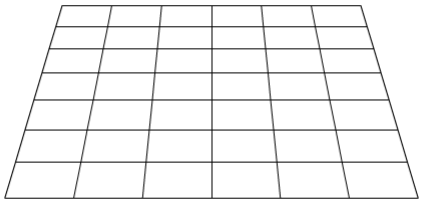
$$S \leq \frac{1}{\alpha(n)}$$

Amdahl's Law (cont'd)

- So Amdahl's law should be

$$S \leq \frac{1}{\alpha(n)}$$

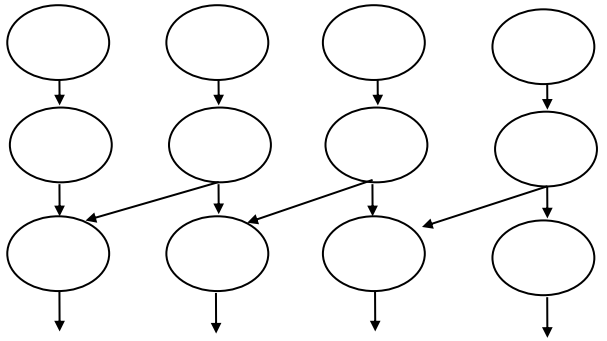
- So as the problem size, n, increases, $\alpha(n)$ decreases and the potential Speedup increases.
- Certain problems demonstrate increased performance by increasing the problem size.
- For example: Calculations on a 2D Grid
- *Regular Problem Size Timings:*
  - Grid Calculations: 85 seconds 85%
  - Serial fraction: 15 seconds 15%
- *Double Problem Size Timings:*
  - 2D Grid Calculations: 680 seconds 97.84%
  - Serial fraction: 15 seconds 2.16%



Amdahl's Law (cont'd)

- So the speedup for a parallel program is not fixed, it's influenced by a number of factors.
- By Sullivan's Theorem:
  - Speedup = $\texttt{min}(P, C)$
  - where P is the number of processors &
  - C is the concurrency of the program.

An Execution Graph



If N is the number of operations in the execution graph, and D is the longest path through the graph then the concurrency $C = \frac{N}{D}$.

- The maximum speed-up is a property of the structure of the parallel program.

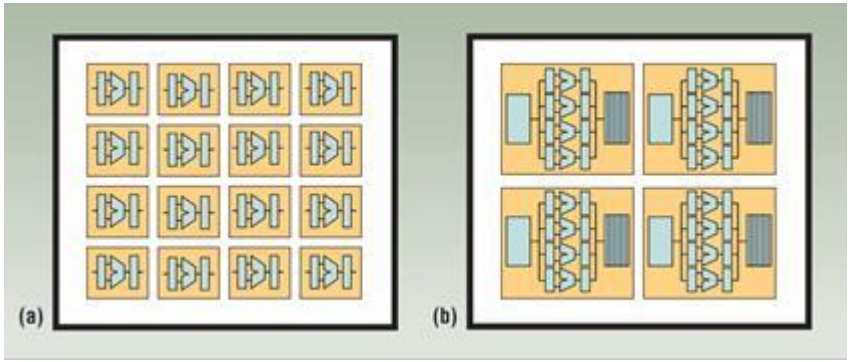
Amdahl's Law for Multi-Core Computing

- Parallel hardware has become more complex in recent years with the development of multicore chips.
- Designers have more DoF to contend with MC chips than single-core designs (uniprocessors) e.g. no. cores, simple/complex pipelines etc.
- Such problems are set to become even more complex as move to thousands of cores per chip.
- Can also move towards more complex chip configurations with either an SMP or ASMP allocating cores to specific functions.
- Recall Amdahl's law for Speedup: $S = \frac{T(\alpha+(1-\alpha))}{T(\alpha+\frac{1-\alpha}{P})}$
- Let $f = 1 - \alpha$, be the *parallelisable* fraction, n the number of cores then: $S = \frac{1}{1-f+\frac{f}{n}}$

Hill & Marty's Extension To Amdahl's Law

- So, taking Amdahl's law for Speedup: $S = \frac{1}{1-f+\frac{f}{n}}$
- Hill and Marty[1] extend this to account for multicore costs
- They use *base core equivalent* or *BCE,* a generic unit of *cost*, accounting for area, power, dollars, or a combination.
- For 1 unit of BCE, a single processor delivering a single unit of baseline performance can be built.
- A budget of n BCE's, can be used for a single n -BCE core, n single-BCE cores, or in general $\frac{n}{r}$ cores each consuming *r* BCEs

SMP with
(a) 16 Cores &
(b) 4 Cores,
Respectively



1. M.D. Hill & M. R. Marty, 'Amdahl's law in the Multicore Era', IEEE Computer Society, July 08

Hill & Marty on Amdahl's Law (cont'd): SMP

- Using a generic single-core performance model, authors assume an r-BCE core can give performance of *perf(r)*.
- They assumed the functional relationship to be *perf(r)=*$\sqrt{r}$
- The resulting speedup (assuming a SMP where all n cores are identical) is given by: $S_{\text{smp}}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r).\frac{n}{r}}}$

  i.e., overall performance made up of a single r-BCE core on serial code part $(1-f)$ & all $\frac{n}{r}$ cores on parallelisable part, (f)

Hill & Marty on Amdahl's Law (cont'd): SMP

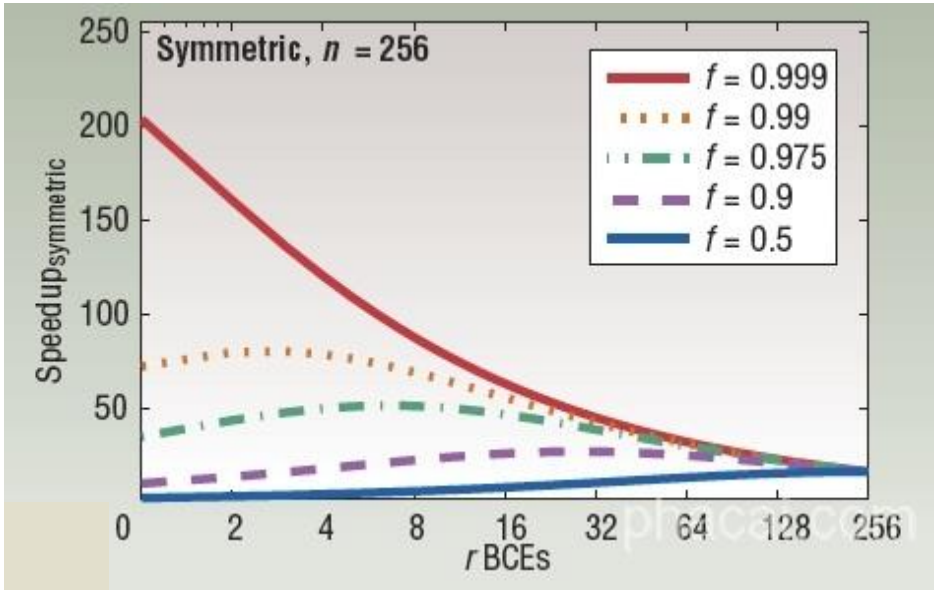
- Graphing : $S_{\text{smp}}(f, n, r) = \dfrac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r).\frac{n}{r}}}$ for n=256 cores

- We see the following:
  - For r=1 base cores, $\frac{n}{r}$=256 cores,

    get a relatively high speedup.
  - For r=256 base cores, $\frac{n}{r}$=1 cores

    get a pretty poor speedup.
  - For $f = 0.975$, max speedup= 51.2

    occurs for r=7.1 base cores, $\frac{n}{r}$=36 cores.



- *Implications:*

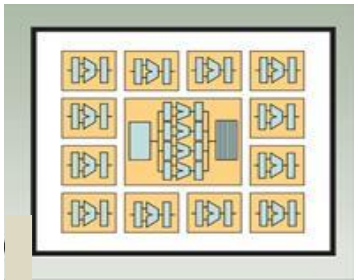
1. For SMP chips, need $f \approx 1$ so have to parallelise code to the max!
2. Use more BCEs per core, r>1 (see example above for max speedup).

Hill & Marty on Amdahl's Law (cont'd): ASMP

- Alternative to SMP is Asymmetric MP where some of the cores are more powerful than the others.
- Here assume that only one core is more powerful.
- With a resource budget of 16 BCEs for example, ASMP can have 1 X 4 BCE core & 12 single BCE cores (see diagram).
- In general chip will have $1 + n - r$ cores since one larger uses r resources & rest have $n - r$ resources
- The resulting speedup is given by: $S_{\text{asmp}}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}}$

i.e., overall performance made up of:

- a single (more powerful) r-BCEcore on serial code part $(1 - f)$ &
- all cores on parallelisable part, (f), delivering $perf(r) + (n - r).1$

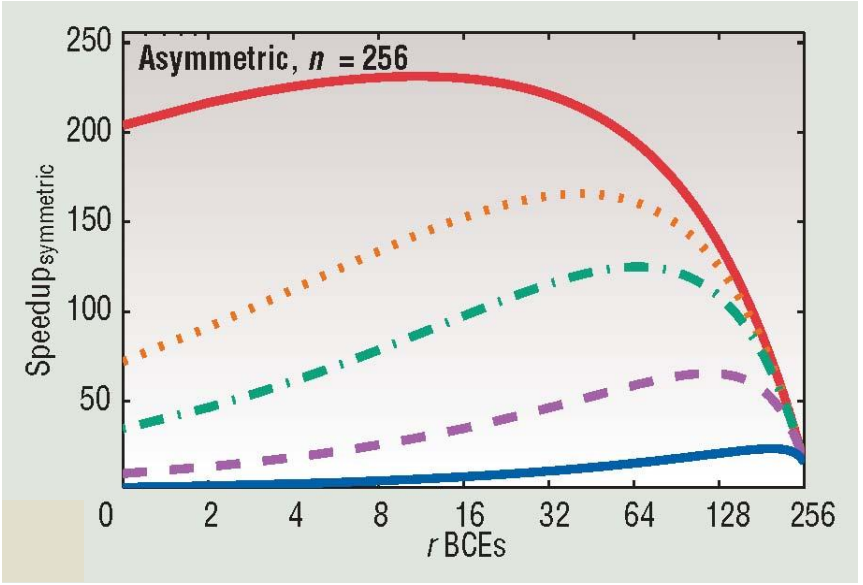


ASMP with 1 chip of
4 times the power
Of the 12 others

Hill & Marty on Amdahl's Law (cont'd): ASMP

- Graphing : $S_{\text{asmp}}(\alpha, n, r) = \dfrac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}}$ for n=256 cores

- Something very different to SMP:
  - For ASMP, max speedup often reached between $1 \leq r \leq 256$
  - For ASMP, often larger speedup that SMPs (and never worse) e.g. $f = 0.975$, N=256, max speedup= 125 (v. SMP 51.2)



- *Implications:*

1. ASMP has great potential for those codes with high serial fraction (small f)
2. Denser multicore chips increase both speedup benefits of going asymmetric and optimal performance of single large core. Hence local inefficiency is tolerable if global efficiency is increase (e.g. by reducing time on sequential phase).