**PREAMBLE**: COURSE BASICS
Recommended Texts (online/in Library)

- Course Notes: [www.computing.dcu.ie/~mcrane/CA4006.html](http://www.computing.dcu.ie/~mcrane/CA4006.html)

- Recommended Text:

- Additional Texts:
Assessment Details

- **25% Continuous Assessment:**
  - Java Concurrency Project (10%),
    - Set Week 5 of Semester,
    - Handed in Week 8
  - Web Services Project (15%)
    - Set Week 8 of Semester,
    - Handed in Week 12

- **75% May Exam**
  - Three hours long
  - 4 from 5 Questions
Course Outline

1. Intro to Concurrent & Distributed Processing
2. Support for Correctness in Concurrency
3. Concurrent & Distributed Architectures
4. Enhanced Concurrency Support in Java Language
5. Programming with MPI & OpenMP
6. Message-Oriented Communication in DS
7. Distributed Object- & Web-Based Systems
8. Safe Access to Distributed Shared Resources
LECTURE 1: INTRO TO CONCURRENT & DISTRIBUTED PROCESSING
Concurrent V Distributed Processing

• Both used to leverage *available* resources & boost performance.

• Hence much overlap between the two.

• All distributed systems must make use of some form of concurrent programming – otherwise no productivity.

• At simplest,
  
  – *Distributed computing* is mostly about infrastructure, physical or virtualized
  
  V
  
  – *Concurrent programming* (implemented in application layer) deals with computations executing simultaneously while synchronising/communicating with each other.
Concurrent Programming is facilitating the performance of multiple computing tasks at the same time.

Example:
- Using mouse, watching YouTube, updating a spreadsheet and scanning your PC only possible due to concurrent programming.
- In that scenario, it’s multi-tasking which allows several programs or processes to access the CPU without waiting turns.
- This permits intensive I/O processing and effective signal handling with resources being shared among multiple tasks.

Concurrency also occurs via implementation of multiple threads of computation (multi-threading) in a single process/program

Example:
- Print a document while continuing to type in another document.
- Without multi-threading, UIs would be very slow (system only handles one action at a time)
Concurrent V Distributed Processing (/3)

- **Distributed System** is a collection of *independent* computers that appears to its users as a *single coherent system*.

- Figure above shows a distributed system organized as *middleware*.

- Middleware:
  - Layer of s/w logically placed between users/apps and o/s, communications
  - Users/apps offered one single system view
  - Extends over multiple machines, offering each app same interfaces/program.

- Wherever/whenever users interact with dist’d system, interaction should be *uniform* (all same) & *consistent* (up to date)
Concurrent V Distributed Processing (/4)

- **Goals of a Distributed System:**
  - Making resources available:
    - make multiple resources available to multiple users/apps;
    - should share them in a controlled & efficient manner for economic reasons;
  - Distribution transparency:
    - should hide fact that processes/resources of dist’d system are physically distributed;
    - perhaps distribution is across multiple computers/locations;
    - i.e. should appear as a single system;
  - Openness:
    - should offer services according to standard rules;
    - should describe syntax & semantics of these services;
  - Scalability
    - e.g. size (number of users &/or processes)
    - geography (distance between nodes)
    - admin (number of administrative domains)
Concurrent V Distributed Processing (/5)

- **Result**: very many types of Distributed Systems
- Can have distribution:
  - Among different device types (Grid + desktop + sensors)
  - Among architectural tiers (fat-thin client spectrum)
  - Centralized/decentralized architectures (C/S → Bitcoin)
  - Between devices of differences in reliability
  - At different locations or even relocating at runtime
  - With different data representations
  - With possible replication of objects at different locations
SECTION 1: INTRO TO CONCURRENT PROCESSING
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:

1. Several computations can be in progress simultaneously, and
2. Potentially interacting with each other.

The computations may be:

– executing on multiple cores in the same chip,
– pre-emptively time-shared threads on the same processor, or
– executed on physically separated processors...
Concurrency, A Cure For All Woes?

• Multicore Systems;
  +
• Fast Networks;
  +
• Concurrency:
  =

Solution to today’s/tomorrow’s *Grand Challenge* problems in Climate Change, Bioinformatics, Astrophysics etc.? ...

...Sadly Not...
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

• 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.
• 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:
  – Improved performance by adding a concurrent procedure to 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!

• Remember 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.
- Now it's broken down, parallelization falls out & correctness is easy.
- In our book transport problem, substitute:
  - *book pile* => *web content*
  - *gopher* => *CPU*
  - *cart* => *marshalling, rendering, or networking*
  - *incinerator* => *proxy, browser, or other consumer*
- 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.
    • Cannot directly access shared data in other processes.
    • Process resources allocated to it via 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, stores it in its own memory cache.
    • A thread can re-read the shared data.
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 many tasks in parallel.
- Multiple compute 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 thread within an application.

*Image* 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
    — They are defined statically by compiler or dynamically by hardware
SECTION 2: ARCHITECTURAL CLASSIFICATION SYSTEMS
Computer Architecture Taxonomies for Concurrency

Processor Organizations

- Single Instruction, Single Data Stream (SISD)
- Single Instruction, Multiple Data Stream (SIMD)
- Multiple Instruction, Single Data Stream (MISD)
- Multiple Instruction, Multiple Data Stream (MIMD)

Uniprocessor

- Vector Processor
- Array Processor

- Shared Memory (tightly coupled)
- Distributed Memory (loosely coupled)

- Symmetric Multiprocessor (SMP)
- Nonuniform Memory Access (NUMA)

Clusters
**Flynn’s Classification**

- **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 (/3)

- **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 & NUMA systems (more later)
  - Most modern Supercomputers use MIMD with SMPs for specific tasks.
  - Suited more to *functional decomposition* than *domain decomposition* (more shortly)
More on MIMD

- General purpose processor; 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 ~ 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. *Non-uniform 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 (/2)

• **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* are real-time or *batch*
  – Communication via fixed path or network connections
  – Alternative to SMP giving high performance & high availability
Program Decomposition

- Three methods to decompose problem into smaller processes for parallel execution: *Functional Decomposition*, *Domain Decomposition*, or combination

**Functional Decomposition**
- Decompose *problem into different processes* for distribution to processors for simultaneous execution
- Good when no static structure or fixed number of calculations

**Domain/Data Decomposition**
- Partition *problem data* & distribute to processors for simultaneous execution
- Good where:
  - Static data (solve large matrix)
  - Fixed domain but dynamic computation in various regions (fluid vortices models)
### Lecture 1: Introduction

#### TOP 500

**The List.**

[Image of TOP 500 logo]

**PRESENTED BY**

- Innovative Computing Laboratory
- Lawrence Berkeley National Laboratory
- ISC Group

**FIND OUT MORE AT**

top500.org

<table>
<thead>
<tr>
<th>NAME</th>
<th>SPECS</th>
<th>SITE</th>
<th>COUNTRY</th>
<th>CORES</th>
<th>TFLOPS</th>
<th>POWER (MW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tianhe-2 (Milkyway-2)</td>
<td>Intel Ivy Bridge (12C 2.2 GHz) &amp; Xeon Phi (57C 1.1 GHz). Custom interconnect</td>
<td>NUDT</td>
<td>China</td>
<td>3.129,000</td>
<td>33.9</td>
<td>17.6</td>
</tr>
<tr>
<td>Titan</td>
<td>Cray XK7, Opteron 6274 (16C 2.2 GHz) + Nvidia Kepler GPU. Custom interconnect</td>
<td>DOE/SC/ORNJ</td>
<td>USA</td>
<td>560,640</td>
<td>17.6</td>
<td>8.2</td>
</tr>
<tr>
<td>Sequoia</td>
<td>IBM BlueGene/Q. Power BQC (16C 1.60 GHz). Custom interconnect</td>
<td>DOE/NNSA/LLNL</td>
<td>USA</td>
<td>1,572,684</td>
<td>17.2</td>
<td>7.9</td>
</tr>
<tr>
<td>K computer</td>
<td>Fujitsu SPARC64 VIIIFx (8C 2.0 GHz). Custom interconnect</td>
<td>RIKEN AICS</td>
<td>Japan</td>
<td>705,024</td>
<td>10.5</td>
<td>12.7</td>
</tr>
<tr>
<td>Mira</td>
<td>IBM BlueGene/Q. Power BQC (16C 1.60 GHz). Custom interconnect</td>
<td>DOE/SC/ANL</td>
<td>USA</td>
<td>786,422</td>
<td>8.59</td>
<td>3.95</td>
</tr>
</tbody>
</table>

**PERFORMANCE DEVELOPMENT**

[Graph showing performance development over time]

- **HPLINPACK**
  - A Portable Implementation of the High Performance Linpack Benchmark for Distributed Memory Computers
  - More information: [http://icl.uky.edu/hpl/](http://icl.uky.edu/hpl/)

---

*CA4006 Lecture Notes (Martin Crane 2015)*

Page 43
Lecture 1: Introduction
SECTION 3: SCALABILITY: METRICS & BARRIERS TO IT
The Ideal V The Real World...

• Ultimately would like 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;
  – Race Conditions;
  – Etc., etc.,
The Ideal V The Real World... (/2)

• **Metrics for Concurrency:**
  – Time spent on a calculation (i.e. *latency*, units [T])
  – Rate to produce series of results (*throughput*, units [T^{-1}])
  – *Power* consumed on a calculation
  – *Platform cost* required for the computation
  – How effectively computational power used in parallel program (*Efficiency*)

• Some of these we will return to later in the course
Scalability

• How much faster can a given problem be solved with N workers instead of one?
• How much more work can be done with \(N\) workers instead of one?

• **Strong Scaling**: same problem size, add more workers/processors
  – **Goal**: Minimize time to solution for a given problem

• **Weak Scaling**: same work per worker, add more workers/processors (overall problem size increases)
  – **Goal**: solve larger problems.
Barriers to Scalability

• **Fine- & Coarse-Grained, Embarrassing Parallelism:**

• Classify applications on how often their subtasks must synchronize/ inter-communicate:
  – *Fine-grained parallelism* exhibited if application subtasks must communicate multiple times per second;
  – *Coarse-grained parallelism*, if they don’t communicate as often,
  – *Embarrassing parallelism*, if they rarely/never have to communicate.

• Embarrassingly parallel applications are considered the easiest to parallelize.
Barriers to Scalability (/2)

- **Fine-grain parallelism** (typically loop level)
  - Can be done incrementally, one loop at a time
  - Does not require deep knowledge of the code
  - Many loops must be parallel for decent speedup
  - Potentially many synchronization points (at end of each parallel loop)

- **Coarse-grain parallelism**
  - Make larger loops parallel at higher call-tree level potentially in-closing many small loops
  - More code is parallel at once
  - Fewer synchronization points, reducing overhead
  - Needs deeper knowledge of code
Barriers to Scalability (/3)

• **Load imbalance:**
  – Longest-running thread determines time to execute parallel code segment
  – Unequal work load distribution leads to idle processors, others work too much
  – Coarse grain parallelization, gives more opportunities for load imbalance

• **Too many synchronization points:**
  – Compiler puts synchronization points at start/exit of each parallel region
  – If too many small loops have been made parallel, synchronization overhead will compromise scalability.
SECTION 4: AMDAHL’S LAW FOR SINGLE & MULTI-CORE SYSTEMS
Amdahl’s Law

• Gene Amdahl divided a program into two sections,
  – an inherently serial section (accounts for many of above issues)
  – and other which can be parallel.

• Let $\alpha$ be inherently serial fraction of program.

• Then the Speedup is given by:

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

• So, if the serial fraction $\alpha = 5\%$, then $S \leq 20$. 
Amdahl’s Law (/2)

• This can be better seen in the following schematic:
Amdahl’s Law (/3)

• How does speedup change with different $\alpha$?
Amdahl’s Law (/4)

• Graph of $S$ against $1 - \alpha$ for different $P$
Amdahl’s Law (/5)

• The Sandia Experiments
  – Karp prize for first program to achieve a speed-up of 200 or better.
  – In 1988, Sandia reported speed-up $> 1,000$
  – This was on 1,024 processor system on three different problems.

• How is this possible?

• Moler’s Law
  – Amdahl assumed serial fraction $\alpha$, independent of problem size.
  – Sandia experiments showed this to be false.
  – As problem size increases, inherently serial parts of program stay same or increase more slowly than problem size.
  – So Amdahl’s law should be

\[ S \leq \frac{1}{\alpha(n)} \]
Amdahl’s Law (/6)

• So Amdahl’s law should be

\[ S \leq \frac{1}{\alpha(n)} \]

• If problem size, \( n \uparrow, \alpha(n) \downarrow \) & potential Speedup \( \uparrow \).

• 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 (/7)

• So the speedup for a parallel program is not fixed, it’s influenced by a number of factors.

• By Sullivan’s Theorem:
  – \( \text{Speedup} = \min(P, C) \)
  – where \( P \) is number of processors &
  – \( C \) is the concurrency of the program.

  If \( N \) is number of operations in execution graph, & \( D \) is longest path through graph then concurrency \( C = \frac{N}{D} \).

• Max speed-up is property of parallel program structure.
Gustafson-Barsis’ Law

• **Gustafson-Barsis’ Law:**
  - Recall *Strong scalability*\(^1\) assumes that problem size remains fixed in size as \(P\) increases – this is rarely the case.
    - Games consoles don’t run old 8-bit games fast but more complex games
    - Expectations of problem size alter with hardware developments
  - As a result, Gustafson (1988) suggested reformulating Amdahl’s law:
    - ...speedup should be measured by scaling the problem to number of processors, not by fixing problem size.

• **Weak scalability** measures speedup by increasing problem size.

• Gustafson-Barsis’ Law shows the limits of *weak scalability*:
  \[
  S' = S(\alpha + (1 - \alpha)P)
  \]
  where \(S'\) is the *Scaled Speedup*

\(^1\)reflected in Amdahl’s law
Gustafson-Barsis’ Law (/2)

- **Illustrations of Gustafson’s Law**

- Comparing Amdahl’s Law with Gustafson’s Law:

![Illustration of Amdahl and Gustafson's Laws](image)

*Figure: Comparison of Amdahl's Law (linear speedup with problem size fixed) and Gustafson's Law (non-linear speedup with execution time fixed)*
Amdahl’s Law for Multi-Core Computing

- Parallel h/w increased in complexity with arrival of multicore chips.
- More Degrees of Freedom for designers to contend in MC chips than with single-core designs e.g. no. cores, simple/complex pipelines etc.
- Such problems will 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: 

\[ S = \frac{1}{1-f+n\frac{f}{n}} \]
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
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 \( \text{perf}(r) \).
- They assumed functional relationship to be \( \text{perf}(r) = \sqrt{r} \).
- The resulting speedup (assuming a SMP where all \( n \) cores are identical) is given by:
  \[
  S_{\text{smp}}(f, n, r) = \frac{1}{1-f} + \frac{f}{\text{perf}(r)n/r}
  \]
  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**:
  \[ S_{\text{smp}}(f, n, r) = \frac{1}{1-f \frac{1}{\text{perf}(r)} + f \frac{n}{\text{perf}(r) \cdot r}} \text{ for } n=256 \text{ cores} \]

- 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 more powerful cores.
- Here assume that only one core is more powerful.
- With resource budget of 16 BCEs, ASMP can have $1 \times 4$ BCE core & 12 single BCE cores (see diagram).
- In general, chip has $1 + n - r$ cores since one larger uses $r$ resources & rest have $n - r$ resources

- Resulting speedup: $S_{asmp}(f, n, r) = \frac{1}{1-f \cdot perf(r) + f \cdot perf(r) + n-r}$

i.e., overall performance made up of:
- a single (more powerful) $r$-BCE core on serial code part $(1 - f)$ & all cores on parallelisable part, $(f)$, delivering $perf(r) + (n - r)$.

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

Lecture 1: Introduction

CA4006 Lecture Notes (Martin Crane 2015)
Hill & Marty on Amdahl’s Law (cont’d): ASMP

• Graphing: \( S_{\text{asmp}}(\alpha, n, r) = \frac{1}{1-f}\frac{1}{\text{perf}(r)}+\frac{f}{\text{perf}(r)+n-r} \) 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 than 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.
  3. So local inefficiency is ok if global efficiency is increased (e.g. \( T_{\text{sequential}} \) reduced)
Summary

• Concurrency is about *dealing with* lots of things at once, Parallelism is about *doing* lots of things at once.
• Distributed Systems involve more issues than Concurrent ones.
• Terms such as *Multi-tasking, Multi-core, Multi-threading* (both *implicit* & *explicit*) important in concurrent systems.
• Flynn’s classification is established but essential architectural classification system in concurrency.
• When coding in parallel *Functional* & *Domain Decomposition* should be considered.
• Ability of programs to scale is important but many barriers exist.
  – E.g. Fine-/Coarse-grain Parallelism, load imbalance & synchronization.
• Amdahl’s law is a simple way to account for some of these barriers.
• Has been extended by Hill & Marty to *Multi-core* processors.