

Welcome



Overview of the week

29 April to 03 May, 2013

Week 18

	29 Monday	30 Tuesday	1 Wednesday	2 Thursday	3 Friday
			Worker's Day		
8 AM					
9 AM	Introduction to Course, Overview of Parallel Computing (M. Kuttel, UCT). Introduction to CUDA (J. Gain, UCT) CS LT303	CUDA applications I. John Stone (UIUC) CS 3.03	A brief OpenACC intro plus other general approaches to GPU computing: Libraries, tools, accessing CUDA from other languages, examples	Future Nvidia developments: Echelon project, Dragonfly interconnect, Maxwell and Volta	Supercomputers and GPUs: Presence in the top500, an overview of Titan supercom-
10 AM	Tea	Tea	Tea	Tea	Tea
11 AM	Programming in CUDA: the essentials : J. Stone	CUDA Applications II. John Stone (UIUC)	The Kepler architecture and six ways to enhance CUDA programs using its new capabilities. Manuel Ujaldon (U. Malaga)	Programming for hybrid architectures. J. Stone (UIUC)	Many core and the SKA. Simon Ratcliff (SKA)
Noon					Conclusions/wrap-up: Michelle Kuttel
1 PM	Lunch	Lunch	Lunch	Lunch	Lunch
2 PM	Prac 01 – Introduction to cluster computing – Hello World on the cluster – CUDA Runtime API – Vector Addition CS Honours Computer Lab	Prac 02 – Parallel Reduction CS Honours Computer Lab	Prac 03 – Numeric Integration CS Honours Computer Lab	Prac 04 – N-body Simulation CS Honours Computer Lab	
3 PM					
4 PM					



Overview of the week: Invited Lecturers

- **John Stone, UIUC**
 - **Monday, Tuesday, Thursday**
- **Manuel Ujaldón, University of Malaga**
 - **Wednesday, Thursday, Friday**



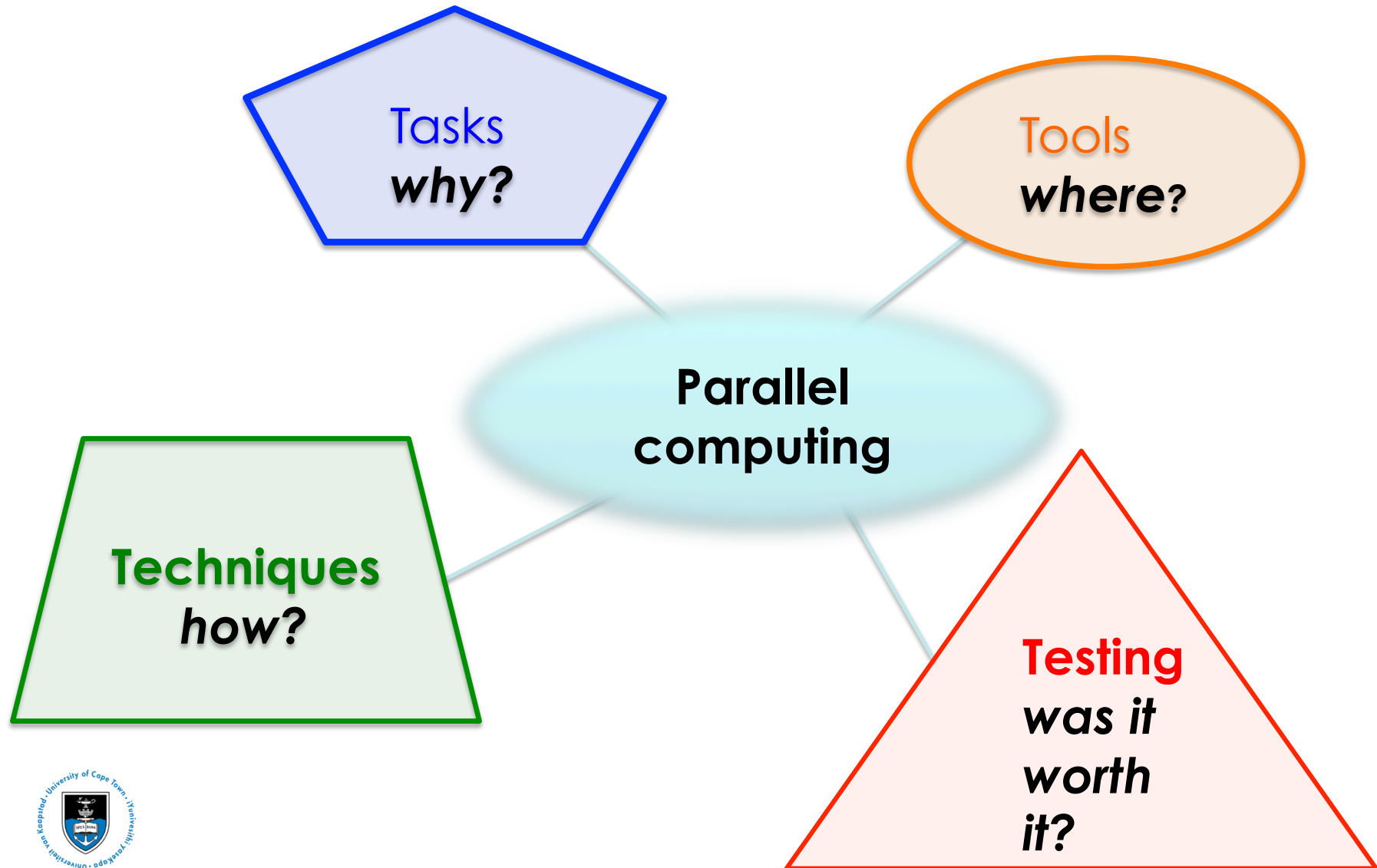
Overview of Parallel Computing

Michelle Kuttel mkuttel@cs.uct.ac.za

April/May 2013



Overview of parallel computing

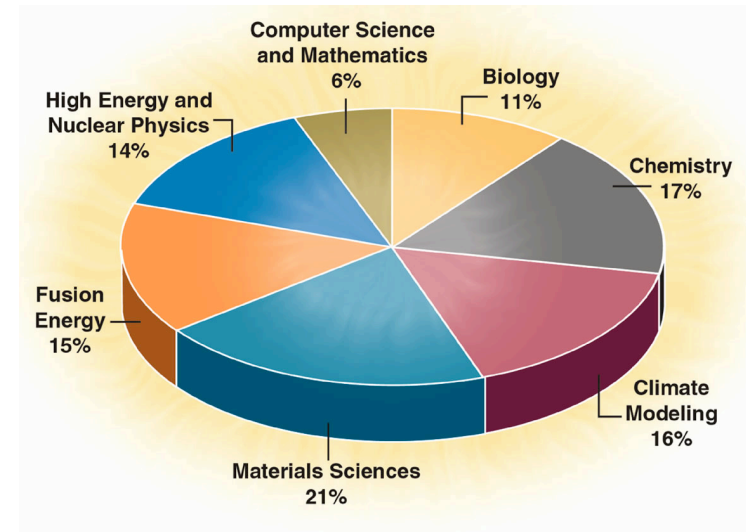




Why do we need parallel computing?

New **model for science**:

- theory+experiment+ simulation
- **Grand Challenge** problems
 - cannot be solved in a reasonable time by today's computers
 - Many are numerical simulations of complex physical systems:
 - weather/climate modelling
 - chemical reactions
 - Astronomical simulations
 - Computational fluid dynamics and turbulence
 - Particle physics
 - Finance - option pricing



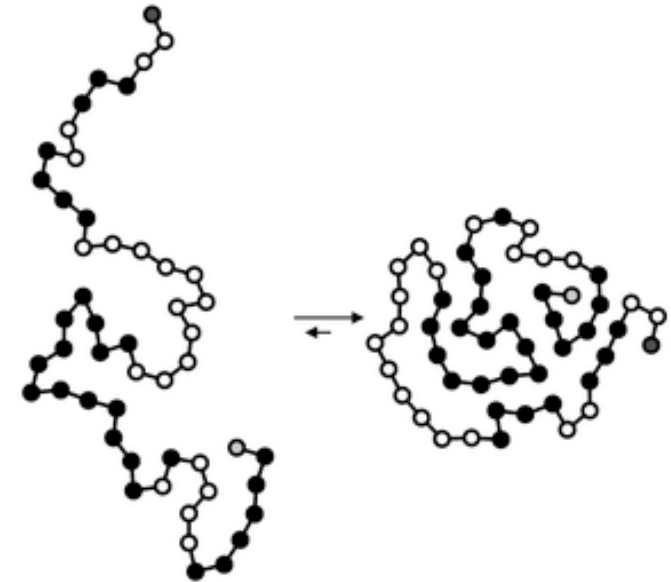
e.g. Usage of Oakridge National Laboratory (USA) CCS supercomputers in terms of processor hours by scientific discipline.

Example: Protein folding challenges

Problem: Given the composition of a protein, can you predict how it folds?

- Levinthal's paradox:
many proteins fold extremely quickly into a favourable conformation, despite the number of conformations possible
- NP-complete problem –

for a protein of 32 000 atoms, 1 petaflop system will still need **3 years** to fold one protein (100 microseconds of simulation time)



if you can fold 1, then you will want to fold more, assemble a whole cell, human body ... etc. etc.

Protein folding is an example of an N-Body Problem

- Many simulations involve computing the **interaction** of a large number of particles or objects. If
 - the force between the particles is completely described by adding the forces between all pairs of particles (*pairwise interactions*)
 - the force between each pair acts along the line between them
- this is called an **N-body** central force **problem**.
 - **e.g. astronomical bodies, molecular dynamics, fluid dynamics, simulations for visual effects industry, gaming simulations**
- It is straightforward to understand, relevant to science at large, and difficult to parallelize effectively.

Tasks
why?

Why do we need parallel computing?



Weta Digital data center (Wellington, NZ) used to render the animation for the movie "Avatar." (Photo: Foundry Networks Inc.)

more than **4,000 HP BL2x220c blades**



The (only) goal of parallel programming is - **SPEED!**

Aim to solve a given problem in **less wall-clock time**

e.g. run financial portfolio scenario risk analysis on all portfolios held by an investment firm within a time window.

- OR solve **bigger problems** within a certain time

e.g. more portfolios

- OR achieve **better solutions** in same time
- e.g. use a more accurate scenario model



Another goal: use the computing power you have!

- During last decade, parallel machines have become much more widely **available** and **affordable**
 - first Beowulf clusters, now multicore architectures and accelerators
- As parallelism becomes ubiquitous, parallel programming becomes essential
 - parallel programming is much harder than serial programming!

Tools
where?

2. Tools

Parallel processing is:

*the use of **multiple processors** to execute different parts of the same program simultaneously*

But this is a bit vague, isn't it?

What is a **parallel computer**?



Tools
where?

What is a parallel computer?

a set of processors that are able to work cooperatively to solve a computational problem

- How **big** a set?
- How **powerful** are the processing elements?
- How easy is it **scale up**? (increase number of processors)
- How do the elements **communicate** and **cooperate**?
- How is **data transmitted** between processors? What sort of **interconnection** is provided and what operations are available to sequence the actions carried out on different processors?
- What are the **primitive abstractions** that hardware and software provide to the programmer?
- How does it all translate into **performance**?

Tools
where?

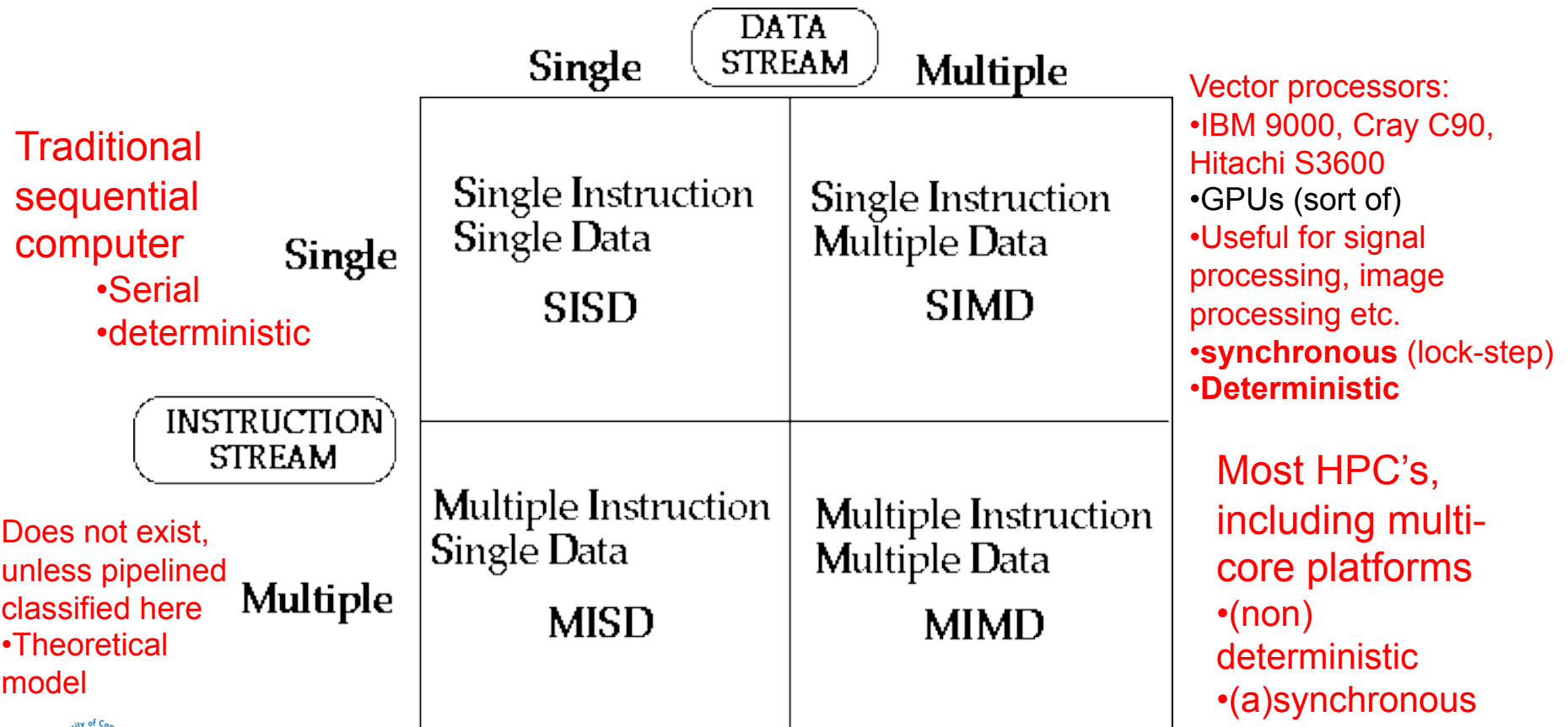
A parallel computer is

- Multiple processors on multiple separate computers working together on a problem (cluster)
- or a computer with multiple internal processors (multicore and/or multiCPUs) ,
- or a cpu with an accelerator (e.g. GPU)
- Or multicore with accelerators
- Or multicore with accelerators in a cluster
- Or ...a cloud?
- Or.....

Tools
where?

Flynn's Taxonomy

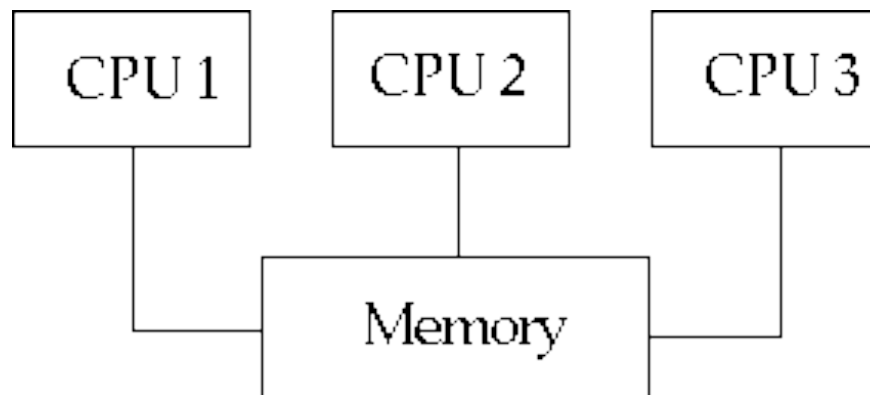
- One of the oldest classifications, proposed by Flynn in 1972
- Classified by instruction delivery (2 chars) and data stream (2 chars)



Tools
where?

Traditional parallel architectures: Shared Memory

- All memory is placed into a single (physical) address space. Processors connected by some form of interconnection network
- Single virtual address space across all of memory. Each processor can access all locations in memory.
- Shared memory designs are broken down into two major categories – **SMP** and **NUMA** - depending on whether or not the access time to shared memory is uniform or non-uniform.



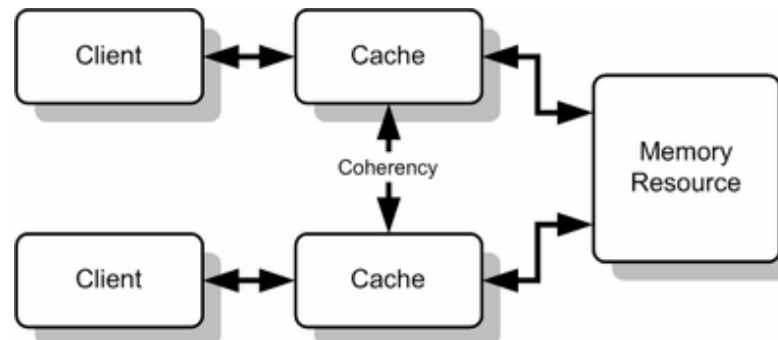
Tools
where?

Shared Memory: Advantages

- Shared memory is attractive because of the convenience of sharing data
 - Communication occurs implicitly as a result of conventional memory access instructions (write and read variables)
 - easiest to program:
 - provides a familiar programming model
 - allows parallel applications to be developed incrementally
 - supports fine-grained communication in a cost-effective manner
 - no real data distribution or communication issues.

Shared Memory: Disadvantages

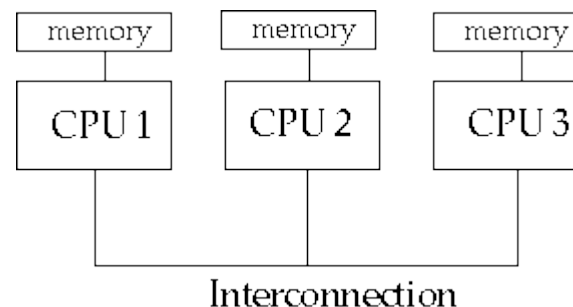
- Why doesn't every one use shared memory ?
 - Limited numbers of processors (tens) –
 - Only so many processors can share the same bus before conflicts dominate.
 - Limited memory size – Memory shares bus as well. Accessing one part of memory will interfere with access to other parts.
 - Cache coherence requirements
 - data stored in local caches must be consistent



Tools
where?

Traditional parallel architectures: Distributed Memory

- “share-nothing” model - separate computers connected by a network
- Memory is physically distributed among processors; each local memory is directly accessible only by its processor.
- Each node runs its own operating system
- Communication via explicit IO operations



Tools
where?

Architectural Considerations: Distributed memory

- A distributed memory multicomputer will physically *scale* easier than a shared memory multicomputer.
 - potentially infinite memory and number of processors
- Big gap between programming method and actual hardware primitives
 - Communication is over an interconnection network using operating system or library calls
- Access to local data fast, remote slow
 - data distribution is very important.
 - We must minimize communication.

Tools
where?

Current parallel architectures: **Supercomputers**

Fastest and most powerful computers in terms of processing power and I/O capabilities.

www.top500.org

- semi-annual listing put together by University of Mannheim in Germany (**Linpack benchmark**)
- No. 1 Position on Latest TOP500 List (Nov, 2012): Titan from Oak Ridge National Laboratory
 - 17.59 **Petaflop/s** (quadrillions of calculations per second) on the Linpack benchmark.
 - Titan has 560,640 processors, including **261,632 NVIDIA K20x** accelerator cores.

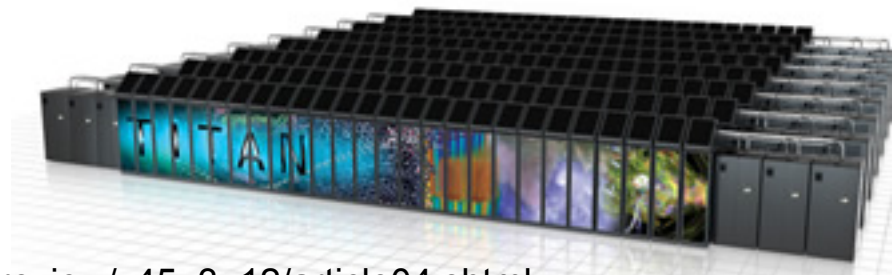


image from http://www.ornl.gov/info/ornlreview/v45_3_12/article04.shtml

Tools
where?

Current parallel architectures:

Supercomputers

Current supercomputers **combine distributed and shared memory and accelerators:**

- A total of **62 systems** on the www.top500.org list are using **Accelerator/Co-Processor technology**:
 - Titan and the Chinese Tianhe-1A system (No. 8) use **NVIDIA GPUs** to accelerate computation
 - Stampede and six others are accelerated by the new **Intel Xeon Phi** processors.
 - Six months ago, 58 systems used accelerators or co-processors.

Tools
where?

Supercomputers

Supercomputers are not getting faster, they are getting "wider":

processors handle hundreds of parallel threads of data

changes the way programmers must work –
disruptive technology

Techniques
how?

3. Techniques

- How do you write and run a parallel program?

Techniques
how?

Parallel Programming

- The goal of parallel programming technologies is to improve the “**gain-to-pain**” ratio
- Parallel language must support 3 aspects of parallel programming:
 - specifying parallel execution
 - communicating between parallel threads
 - expressing synchronization between threads

Techniques
how?

Programming a Parallel Computer

- can be achieved by:
 - an entirely **new language** – e.g. Erlang
 - a **directives-based data-parallel language** e.g. HPF (data parallelism), OpenMP (shared memory + data parallelism)
 - an existing high-level language in combination with **a library** of external procedures (**e.g. message passing in MPI, threads in CUDA**)
 - **threads (shared memory – Pthreads, Java threads)**
 - a **parallelizing compiler**
 - other approaches – e.g. object-oriented parallelism



Techniques
how?

Parallel programming for supercomputers:

- For HPC services, most users expected to use standard **MPI** or **OpenMP**, using either Fortran or C

Techniques
how?

MPI

- MPI addresses the message-passing model
 - A computation is a collection of *processes* communicating via *messages*
- A library, not a language
 - Specifies the *names*, *call sequences* and *results* of subroutines to be called from Fortran, C and C++ programs
- A specification, not a particular implementation
 - All parallel computer vendors offer an implementation for their machines and free implementations can be downloaded off the internet (e.g openmpi, lam-mpi, mpich)

Techniques
how?

"hello world" program in C++

```
#include <iostream>
#include <mpicxx.h> // MPI header file for C++
using namespace std;

int main(int argc, char *argv[]) {
    MPI::Init(argc, argv);
    int myid = MPI::COMM_WORLD.Get_rank();
    cout << "Node " << myid << " : Hello world!"<< endl;
    MPI::Finalize();

    return EXIT_SUCCESS;
}
```

Techniques
how?

Message-Passing MPI

- ubiquity means that no other technology can beat it for portability
- availability of MPI-based libraries that provide high-performance implementations of commonly-used algorithms
- however, explicit communication requirements can place an additional burden on programmer

Techniques
how?

Parallel languages: OpenMP

- OpenMP : Open specifications for Multi Processing
 - The OpenMP interface is an alternative multithreading interface specifically designed to support parallel programs
 - An OpenMP program is not appropriate for a distributed memory environment such as a cluster of workstations: OpenMP has no message passing capability.
 - OpenMP recommended when goal is to achieve **modest** parallelism on a **shared memory** computer

Techniques
how?

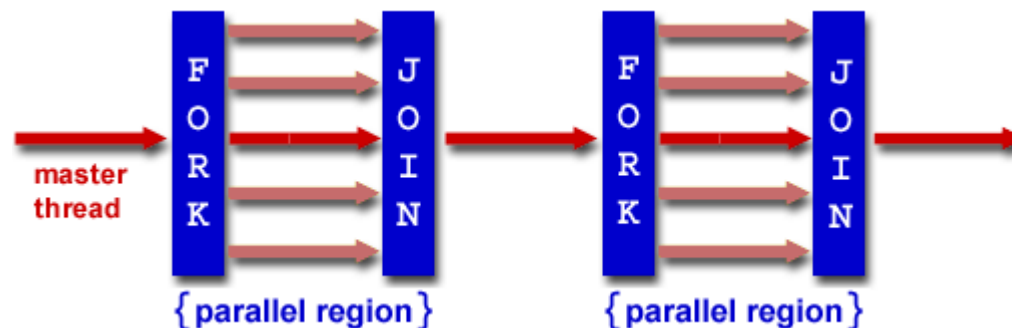
Parallel languages: OpenMP

- OpenMP is the software standard for shared memory multiprocessors
 - parallel programming model for shared memory and distributed shared memory multiprocessors
- recent rise of multicore architectures makes OpenMP much more relevant
 - though MPI can run on shared memory machines (passing “messages” through memory), it is much harder to program.
 - multiprocessor architectures increasingly providing hardware support for cache coherency

Techniques
how?

Runtime Execution Model

- OpenMP uses the highly structured **Fork - Join Model** of parallel execution :
- All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered.



Techniques
how?

OpenMP

- Programming with OpenMP:
 - begin with parallelizable algorithm, SPMD model
- Annotate the code with parallelization and synchronization directives (pragmas)
 - Assumes you know what you are doing
 - Code regions marked parallel are considered independent
 - Programmer is responsibility for protection against races
- Test and Debug

Techniques
how?

OpenMP Hello World

```
int main(int argc, char *argv[]) {  
    #pragma omp parallel  
    printf("Hello, world.\n");  
    return 0;  
}
```

The *omp* keyword distinguishes the pragma as an OpenMP pragma, so that it is processed by OpenMP compilers and ignored by non-OpenMP compilers.


OpenMP **preserves sequential semantics:**

- A serial compiler will ignore the #pragma statements and produce the usual serial executable.
- An OpenMP-enabled compiler will recognize the pragmas and produce a parallelized executable suitable for running on a shared-memory machine.
- simplifies development, debugging and maintenance

Testing
was it
worth it?

4. Testing

- How do demonstrate that parallel computing is worth the effort?
- Identification of the causes of **inefficiency** of parallel algorithms and **quantification** of their importance are the basic steps to optimizing the performance of an application
- This is where the science comes in ...



Testing
was it
worth it?

Performance analysis

- requires a good understanding of how all levels of a system behave and interact
 - from processor architecture to algorithm
- enormous amount of well-thought experimentation and benchmarking is needed in order to optimize performance

Testing
was it
worth it?

Speedup

Speedup is the factor by which the time is reduced compared to a single processor

Speedup for P processes =
$$\frac{\text{time for 1 process}}{\text{time for } P \text{ processes}}$$

$$= T_1 / T_P.$$

In the ideal situation, as P increases, so T_P should decrease by a factor of P .

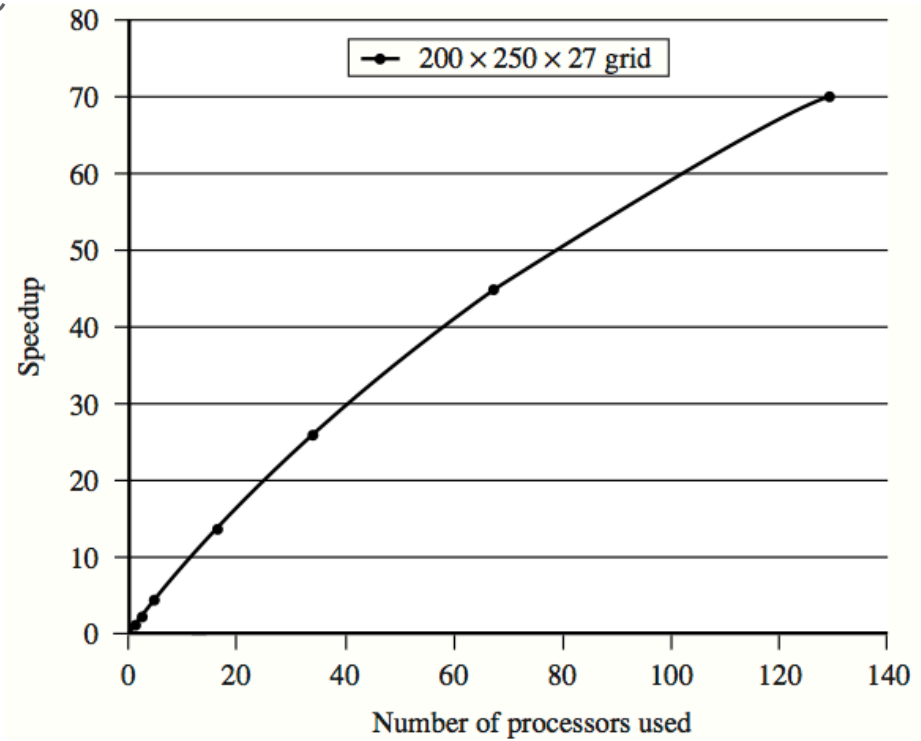


Figure 1.1

Performance of the MM5 weather code.



Figure from "Parallel Programming in OpenMP, by Chandra et al.

Testing
was it
worth it?

Amdahl's Law: Recap


Sequential
fraction

Parallel
fraction

Speedup
=

$$\frac{1}{1 - p + \frac{p}{n}}$$

Number of
processors



Testing
was it
worth it?

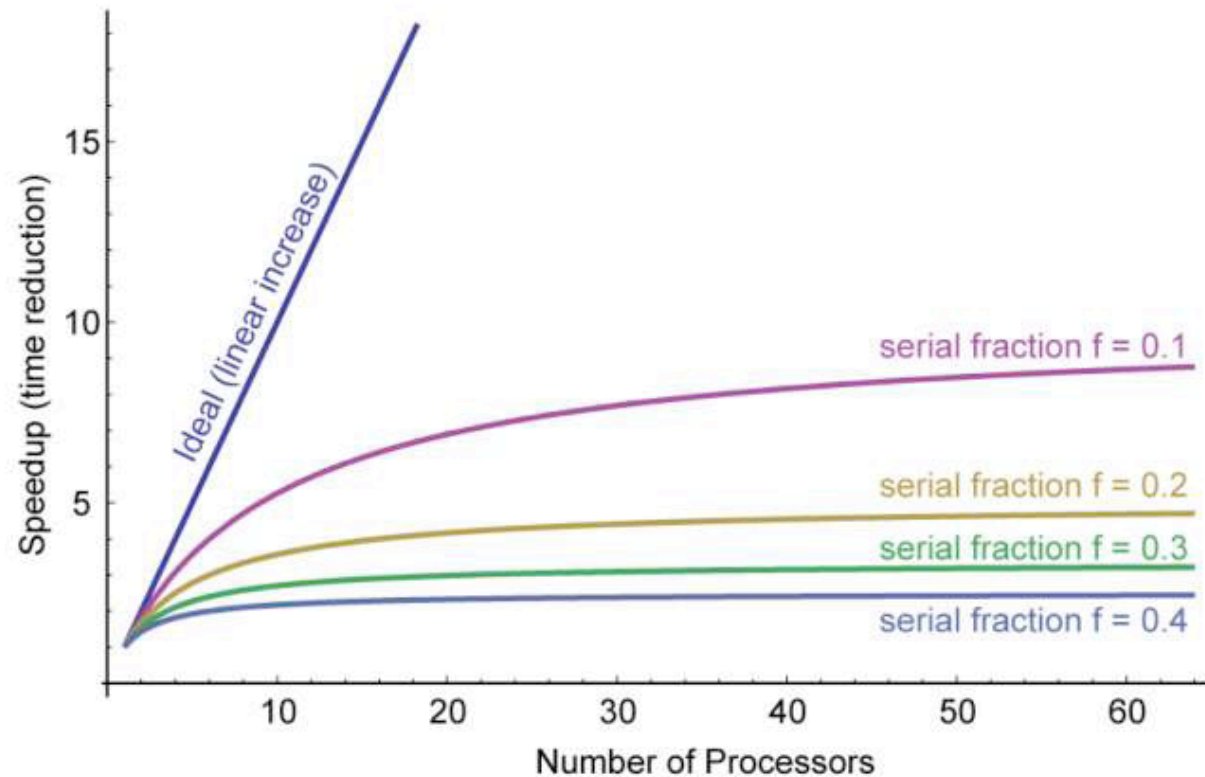
Amdahl's law^{*}

parallelism (infinite processors) = $1/(1-p)$

^{*} G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", *AFIPS Proc. Of the SJCC*, **30**,438-485,1967

Testing
was it
worth it?

Graphing Amdahl's Law



graphic from lecture slides: Defining Computer "Speed": An Unsolved Challenge, Dr. John L. Gustafson, Director Intel Labs, 30 Jan 2011

Testing
was it
worth it?

Why such bad news

$$T_1 / T_P = 1 / (S + (1-S)/P)$$

$$T_1 / T_\infty = 1 / S$$

- Suppose 33% of a program is sequential
 - Then a billion processors won't give a speedup over 3
- Suppose you miss the good old days (1980-2005) where 12ish years was long enough to get 100x speedup
 - Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1
 - For 256 processors to get at least 100x speedup, we need
$$100 \leq 1 / (S + (1-S)/256)$$
Which means $S \leq .0061$ (i.e., 99.4% perfectly parallelizable)

Testing
was it
worth it?

Scalability

- **strong scaling:**

- defined as how the solution time varies with the number of processors for a fixed *total* problem size.

- **weak scaling:**

- defined as how the solution time varies with the number of processors for a fixed problem size *per processor*.

Testing
was it
worth it?

- What if $t_1 = t_2$, and **work** is what changes?

