Parallel programming with Java
Slides 1: Introduction

Michelle Kuttel
August/September 2012
mkuttel@cs.uct.ac.za
(lectures will be recorded)

Changing a major assumption

So far, most or all of your study of computer science has assumed that

one thing happens at a time

this is *sequential programming*

- everything is part of **one** sequence

This slide adapted from: Sophomoric Parallelism and Concurrency, Lecture 1, Dan Grossman

Changing a major assumption

Removing the sequential assumption creates both **opportunities** and **challenges**

- Programming:
  - **Divide work** among threads of execution and **coordinate** (synchronize) them
- Algorithms:
  - more **throughput**: work done per unit wall-clock time = speedup
  - more **organization required**
- Data structures:
  - May need to support **concurrent access** (multiple threads operating on data at the same time)

This slide adapted from: Sophomoric Parallelism and Concurrency, Lecture 1, Dan Grossman

Why is parallel programming important?

Parallel programming is a lot of work…

Writing correct and efficient multithreaded code is often much more difficult than for single-threaded (i.e., sequential) code

- Especially in common languages like Java and C
- So typically stay sequential if possible

- Why is it necessary?

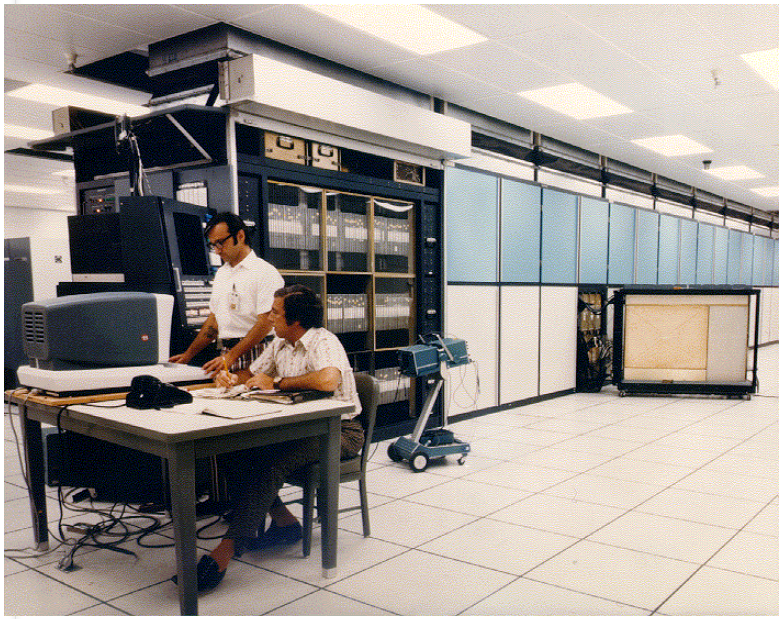
Origins of Parallel Computing

- None of the “future is parallel” argument is new :

“A Universal Computer Capable of Executing an Arbitrary Number of Sub-programs Simultaneously” J. Holland, **1959**, *Proc. East Joint Computer Conference,* Vol16, pp 108-113

Early Parallel Computing: ILLIAC IV

- ILLIAC IV project at University of Illinois 1965-76
- one of the most **infamous** supercomputers ever.



Designed to have fairly high parallelism:
- **256 processing elements driven by 4 CPUs** and a 13MHz clock.
- 64-bit registers

BUT, the final machine
- had only 16 processors, due to costs escalating from projected $8 million (1966) to $31 million (1972)
- Had to be moved due to student protests (Vietnam war)

In 1976 it ran its first successful application.
Used to perform computational fluid dynamics simulations, but
- actual performance of 15 MFLOPS compared to estimated performance of 1000 MFLOPS.

Photo Courtesy of Charles Babbage Institute, University of Minnesota, Minneapolis

Early Parallel Computing

Why not pursued more thoroughly before 1990's?

- Because of dramatic increase in uniprocessor speed, the need for parallelism turned out to be less than expected

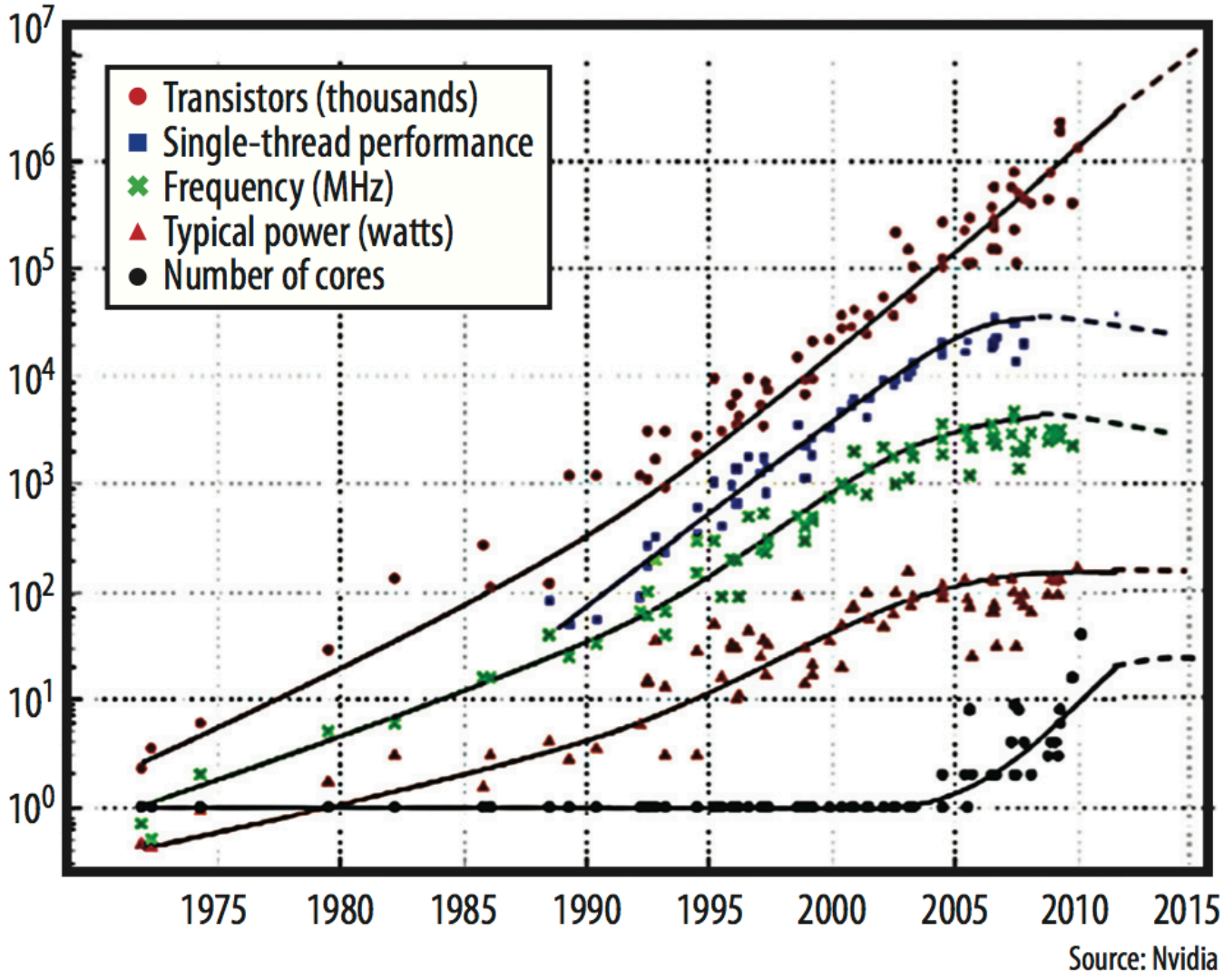
From roughly 1980-2005, desktop computers became exponentially faster at running sequential programs

  - traditional doubling of clock speeds every 18–24 months

Now, the “Power Wall”

Nobody knows how to continue the speed increase

- Increasing clock rate generates too much heat: power and cooling constraints limit increases in microprocessor clock speeds
- Relative cost of memory access is too high



Will Power Problems Curtail Processor Progress? Neal Leavitt, Computer, Vol 45, number 5, pages 15-17

Era of multicore

We can keep making "wires exponentially smaller" (Moore's "Law"), so put multiple processors on the same chip ("multicore")

- During the next decade, the level of parallelism on a single microprocessor will rival the number of nodes in the most massively parallel supercomputers of the 1980s*
- By 2020, extreme scale HPC systems are anticipated to have on the order of 100,000–1,000,000 sockets, with each socket containing between 100 and 1000 potentially heterogeneous cores*

- These enormous levels of concurrency must be exploited efficiently to reap the benefits of such exascale systems.

*Parallel Computing, Volume 37, Issue 9, September 2011, Pages 499-500, **Emerging Programming Paradigms for Large-Scale Scientific Computing**

What to do with multiple processors?

Run multiple totally different programs at the same time

- Already do that? Yes, but with time-slicing

Do multiple things at once in one program

- Our focus – more difficult
- Requires rethinking everything, from asymptotic complexity to how to implement data-structure operations

Why Parallel Programming?

- **Multicore** architectures are now ubiquitous
- **Performance gain** from multiprocessing hardware
- Increased application **throughput**
  - an I/O call need only block one thread.
- Increased application **responsiveness**
  - high priority thread for user requests.
- More appropriate structure
  - for programs which interact with the environment, control multiple activities and handle multiple events.

Why NOT parallel programming?

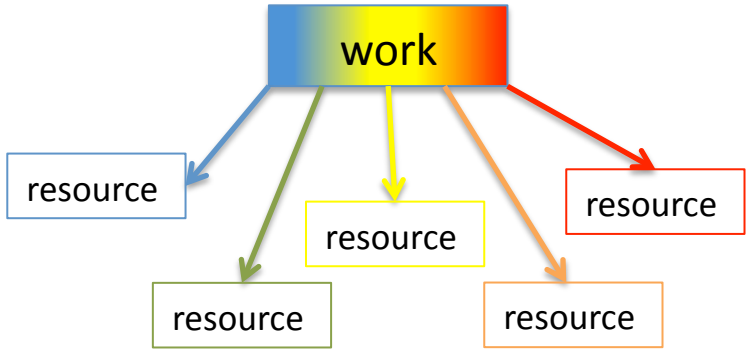
Increased **pain (complexity)** of the programs:

Managing this complexity and the **principles and techniques** necessary for the construction of well-behaved parallel/concurrent programs is the main subject matter of this module

Concurrency versus parallelism

Parallel:

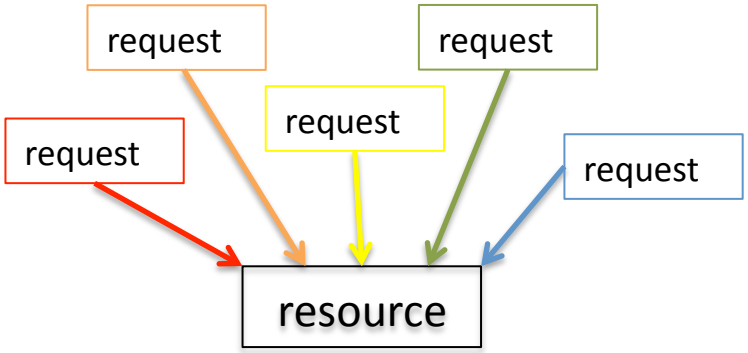
Use extra computational resources to solve a problem faster



These terms are NOT standard, but perspective is important
Many programmers confuse these terms

Concurrent:

Correctly and efficiently manage access to shared resources



*A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency, Dan Grossman, online notes, version Feb 2012

Why is Parallel/Concurrent Programming so Hard?

- Try preparing a seven-course banquet
  - By yourself
  - With one friend
  - With twenty-seven friends …

Cooking analogy

A serial program is a recipe for one cook

- the cook does one thing at a time (sequential)

Parallel options

- use lots of helpers to slice lots of potatoes (**data decomposition**)
- use helpers to dedicate to different tasks, such as beating eggs, slicing onions, washing up (**task decomposition**)
- have a list of tasks and assign them to workers as they are free (**work pool**)
- But too many chefs = too much time coordinating!

This slide adapted from: Sophomoric Parallelism and Concurrency, Lecture 1, Dan Grossman

Cooking analogy

Concurrency example

- Lots of cooks making different things, but only 4 stove burners
- want to allow access to burners, but not cause spills or incorrect burner settings

Parallelism Example

Parallelism: Use extra computational resources to solve a problem faster (increasing throughput via simultaneous execution)

Pseudocode for array sum

- Bad style for reasons we'll see, but may get roughly 4x speedup

```
int sum(int[] arr){
  res = new int[4];
  len = arr.length;
  FORALL(i=0; i < 4; i++) { //parallel iterations
    res[i] = sumRange(arr,i*len/4,(i+1)*len/4);
  }
  return res[0]+res[1]+res[2]+res[3];
}
int sumRange(int[] arr, int lo, int hi) {
   result = 0;
   for(j=lo; j < hi; j++)
      result += arr[j];
   return result;
}
```

This slide from: Sophomoric Parallelism and Concurrency, Dan Grossman

Concurrency Example

Concurrency: Correctly and efficiently manage access to shared resources (from multiple possibly-simultaneous clients)

Pseudocode for a shared chaining hashtable

- Prevent *bad interleaving* (correctness)
- But allow some concurrent access (performance)

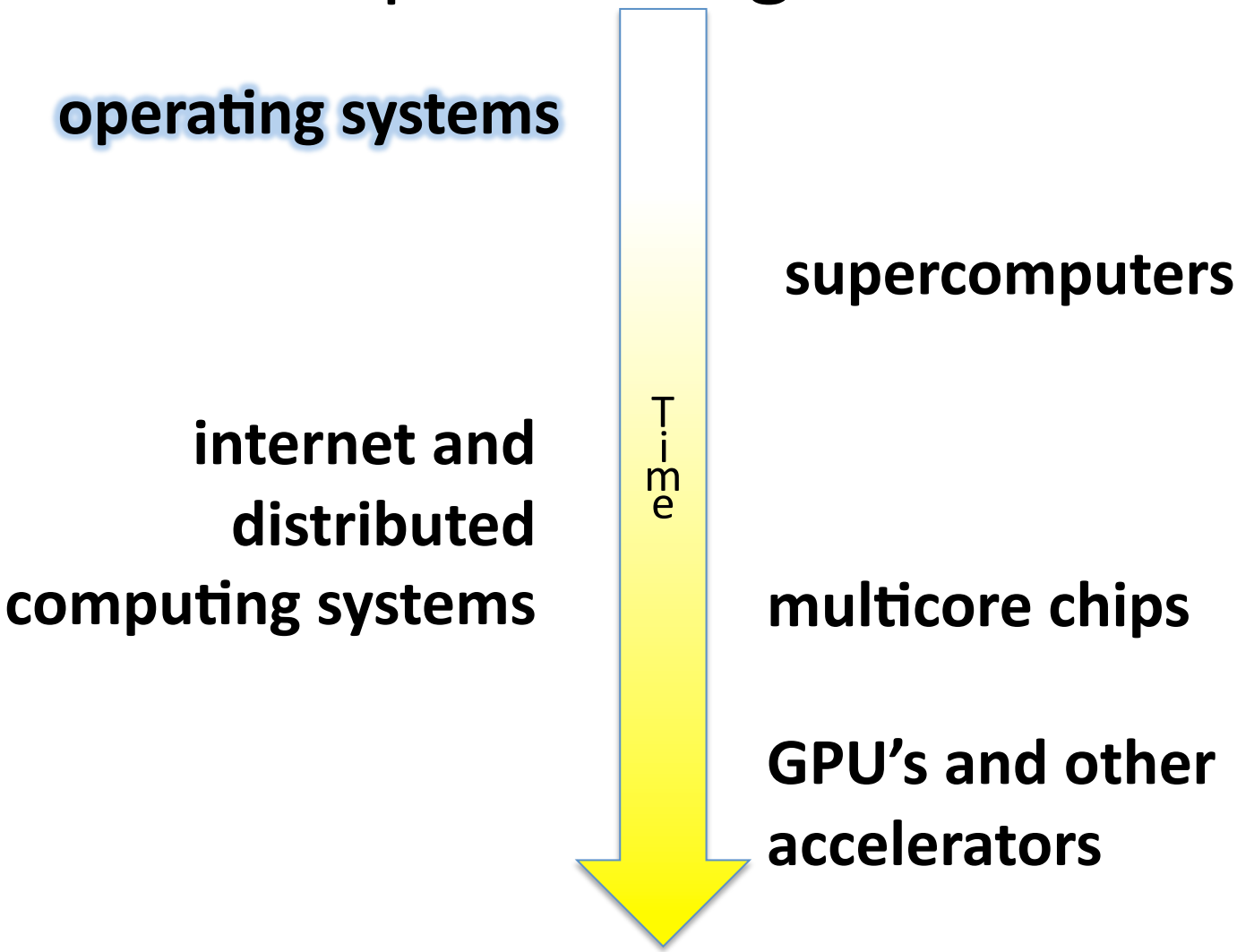
```
class Hashtable<K,V> {
   …
   void insert(K key, V value) {
      int bucket = …;
      prevent-other-inserts/lookups in table[bucket]
      do the insertion
      re-enable access to arr[bucket]
   }
   V lookup(K key) {
   (like insert, but can allow concurrent
    lookups to same bucket)
   }
}
```

Focus on parallelism

- In reality, parallelism and concurrency are mixed
  - Common to use threads for both
  - If parallel computations need access to shared resources, then the concurrency needs to be managed
- However, in the first half of this course we focus on **parallelism**
  - Structured, shared-nothing parallelism
- once you are comfortable with threads, we will talk about mutual exclusion, interleaving etc.

This slide adapted from: Sophomoric Parallelism and Concurrency, Lecture 1, Dan Grossman

Where does parallel/concurrent processing occur?



Everywhere

Many programs today are inherently concurrent or parallel:

- operating and real-time systems,
- event-based implementations of graphical user interfaces,
- high-performance parallel simulations like climate prediction using weather models,
- Internet applications like multiuser games, chats and ecommerce.

This explosion of parallelism has **two significant challenges** from a programming perspective:

- how to best manage all the available resources to ensure the most efficient use of the peak performance provided by the hardware designs.
- how the enormous potential of these systems can be effectively utilized by a wide spectrum of scientists and engineers who are not necessarily parallel programming experts.

from: Parallel Computing, Volume 37, Issue 9, September 2011, Pages 499-500, **Emerging Programming Paradigms for Large-Scale Scientific Computing**

What is a parallel computer?

Parallel processing is:

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

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

What is a parallel computer?

TOP500

Created in 1993

- a list of the top 500 super computers
- updated twice yearly

- Each computer is ranked according to performance on High Performance Linpack (HPL) software package from the Innovative Computing Laboratory at the University of Tennessee:
  - **LINPACK** provides 3 separate benchmarks to evaluate a systems performance on a dense system of linear equations

Class volunteer?

- Outline of machine currently at top of top500

The problem for application programmers is further compounded by the **diversity of multicore architectures** that are now emerging, ranging from:

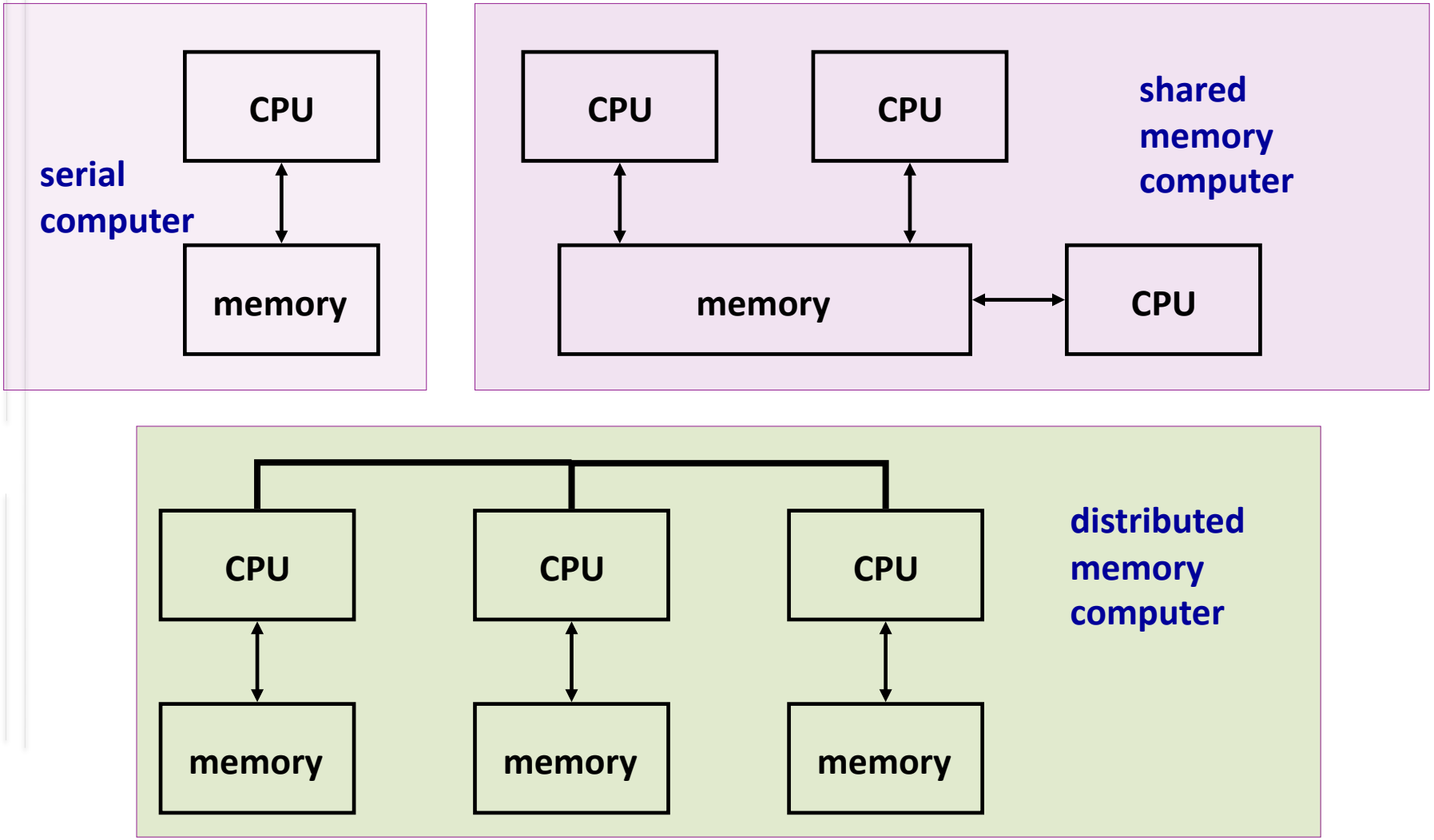
- complex out-of-order CPUs with deep cache hierarchies,
- to relatively simple cores that support hardware multithreading,
- to chips that require explicit use of software-controlled memory.

from: Parallel Computing, Volume 37, Issue 9, September 2011, Pages 499-500, **Emerging Programming Paradigms for Large-Scale Scientific Computing**

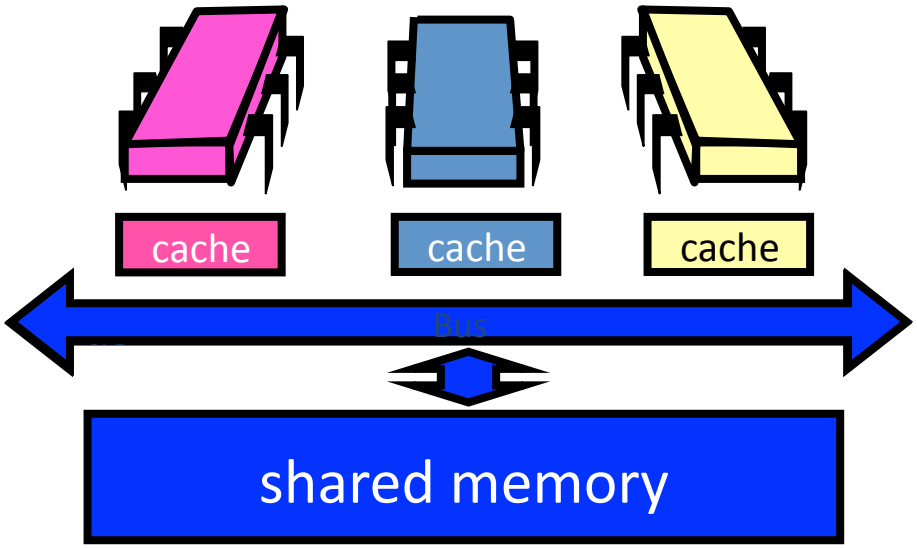
A parallel computer is

- Multiple processors on multiple separate computers working together on a problem (cluster)
- Or a computer with multiple internal processors (multicore and/or multiCPUs) ,
- Or accelerators (GPU's)
- Or multicore with GPU's
- Or multicore with GPU's in a cluster
- Or …a cloud?
- Or…. the internet?

Memory Parallelism



We focus on:
The Shared Memory Multiprocessor (SMP)



- 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.

Communication between processes

Processes must communicate in order to synchronize or exchange data

- if they don't need to, then nothing to worry about!

The challenge in concurrent programming comes from the need to *synchronize the execution of different processes and to enable them to communicate.*

Broadly speaking, researchers have taken **two paths** for leveraging the parallelism provide by modern platforms:

- The first focuses on **optimizing parallel programs** as aggressively as possible by **leveraging** the knowledge of the **underlying architecture**
- The second path **provides the tools**, libraries, and runtime systems to **simplify** the complexities of parallel programming, without sacrificing performance, thereby allowing domain experts to leverage the potential of high-end systems.

from: Parallel Computing, Volume 37, Issue 9, September 2011, Pages 499-500, **Emerging Programming Paradigms for Large-Scale Scientific Computing**

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 for message passing (MPI)
  - threads (shared memory – Pthreads, Java threads)
  - a parallelizing compiler
  - object-oriented parallelism (?)

Parallel Programming Models

- Different means of communication result in different models for parallel programming:
- **Shared-memory model** (this course)

- Message-passing
  - each thread has its own collection of objects.
  - communication is via explicitly sending and receiving messages
- Dataflow
  - program written as a DAS
  - a node executes after all its predecessors in the graph
- Data parallelism
  - primitives for things like: “apply function to every element of an array in parallel”

Parallel programming technologies

Technology converged around 3 programming environments:

OpenMP

simple language extension to C, C++ and Fortran to write parallel programs for shared memory computers **(shared memory model)**

MPI

A message-passing library used on clusters and other distributed memory computers **(message passing model)**

Java

language features to support parallel programming on shared-memory computers and standard class libraries supporting distributed computing **(shared memory model and message passing model)**

Our Needs

To write a shared-memory parallel program, need new primitives from a programming language or library

- Ways to create and *run multiple things at once*
  - Let's call these things **threads**
- Ways for threads to *share memory*
  - Often just have threads with references to the same objects
- Ways for threads to *coordinate (a.k.a. synchronize)*
  - For now, a way for one thread to wait for another to finish
  - Other primitives when we study concurrency

This slide from: Sophomoric Parallelism and Concurrency, Lecture 1, Dan Grossman

Theory not in this course: Modelling processes

We can model processes as **finite state machines**

A simplification that considers a process as having a **state** modified by **atomic actions**.

Each action causes a **transition** from the current state to the next state.

The order in which actions are allowed to occur is determined by a **transition graph**

abstract representation of the program

The use of state machines as an abstract model for processes is widely used in the study of concurrent and distributed algorithms.

Some terminology you will encounter

parallel concurrent

threads processors processes

data race synchronization thread safety

correctness isolation

mutual exclusion

locks monitors

liveness deadlock starvation

Required and Recommended Reading

A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency

Dan Grossman, online notes (and up on Vula)

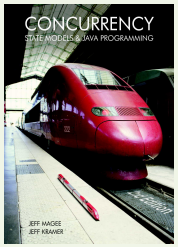
Java Concurrency in Practice

Joshua Bloch, Brian Goetz, Tim Peierls, Joseph Bowbeer, David Holmes, Doug Lea

For more "theoretical" approaches:

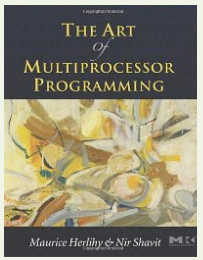
Concurrency: State Models and Java Programs, 2nd Edition

Jeff Magee and Jeff Kramer



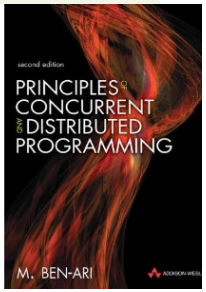
The Art of Multiprocessor Programming

Maurice Herlihy



Principles of Concurrent and Distributed Programming (2nd Edition)

Mordechai Ben-Ari



Tutorial

- Released 27th August

- Hand in: 9am on Friday 21st of September
  - with a 5% penalty if they take the long weekend and handin by Tuesday 24th at 9am
  - with 10% per day (OR PART THEREOF) thereafter