

ID1217 Concurrent Programming

Lecture 1



ROYAL INSTITUTE
OF TECHNOLOGY

Introduction.

Parallel Programming Concepts, Models and Paradigms

Vladimir Vlassov
KTH/EECS

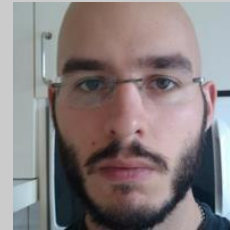
Course Staff

Course leader and lecturer

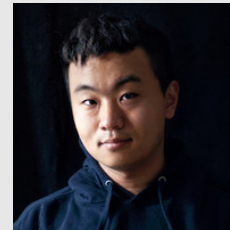


Vladimir Vlassov
Associate Professor, PhD
vladv@kth.se

Teaching assistant



Edward Tjörnhammar
PhD student
edwardt@kth.se



Tianze Wang
PhD student
tianzew@kth.se

Course Material and Literature

- Lecture slides are available in Canvas
- Literature
 - ***An Introduction to Parallel Programming***, by Peter Pacheco, Morgan Kaufmann, 2011, ISBN 978-0-12-374260-5
- Additional reading
 - ***The Art of Multiprocessor Programming***, by Maurice Herlihy, and Nir Shavit, Elsevier Science & Technology Books (Morgan Kaufmann Publishers), 2008; ISBN 978-0-12-370591-4
- Advanced reading
 - ***Multicore Application Programming: for Windows, Linux, and Oracle Solaris (Developer's Library)*** by Darryl Gove, Addison-Wesley Professional, 2010, ISBN 978-0-321-71137-3

Course Layout

- Lectures (18)
- Programming assignments:
 - Programming project (1) – can be done in group of 2 students
 - Homework (5) – 4 homework out of 5 are required to pass the course
- Examination requirements:
 - Approved written exam (5 hours) – 4,5 hp
 - Approved programming assignments – 3 hp

Bonus Policy

- Each assignment (project, homework) is awarded bonus points on your first ID1217 exam whenever you take it, if all of the following three requirements are met:
 1. you submit your assignment on time, i.e., before/on the corresponding submission deadline;
 2. you present and demonstrate your assignment in person to a course teaching assistant within a week after the deadline;
 3. your assignment is accepted.

Assignment	Max bonus points
Programming project	10 points
Each homework	3 points

- You can collect up to $10 + 3 \times 5 = 25$ bonus points.
- The full bonus can be reduced for errors or poor design and implementation, i.e. partial bonus can be given.
- The bonus is valid only once on your first ID1217 exam whenever you take it.

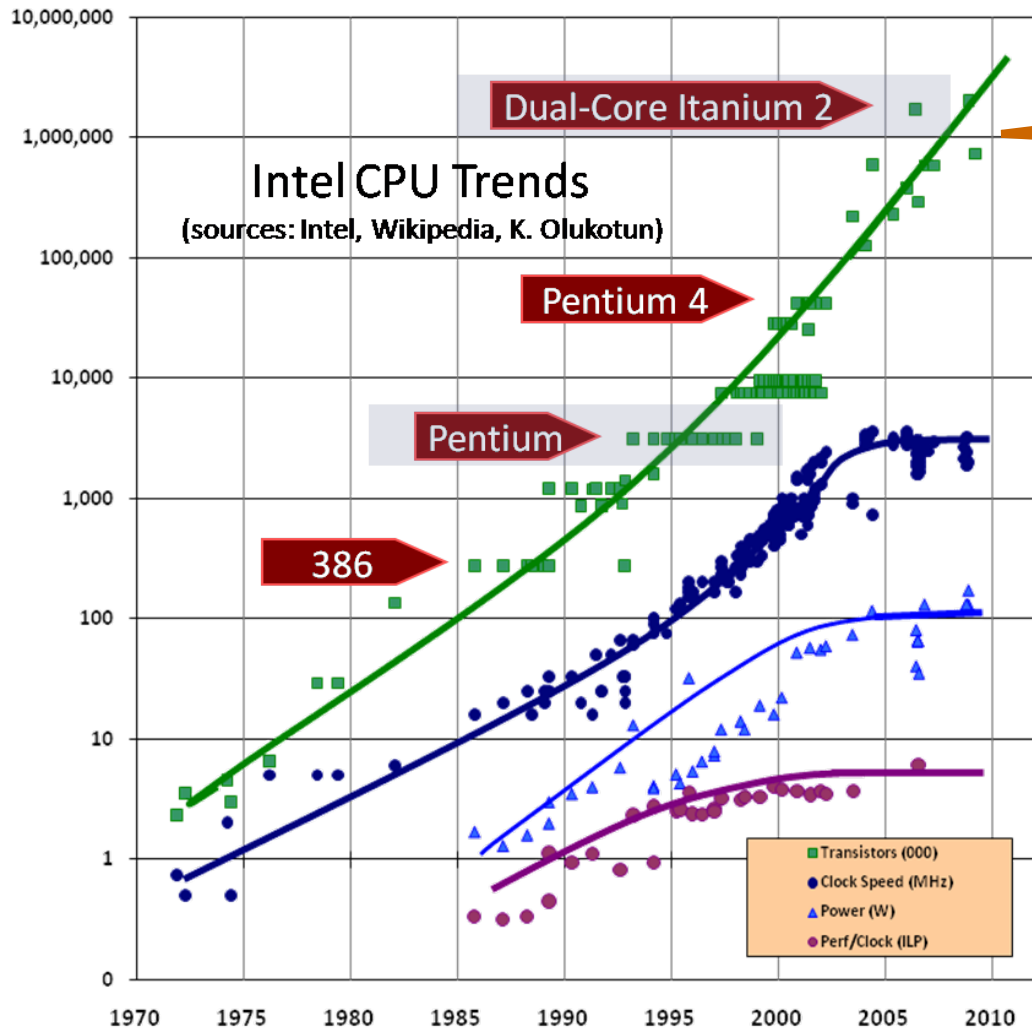
Contents of the Course

- Introduction. Parallel programming concepts, models, and paradigms
- Shared memory programming
 - Processes and synchronization;
 - Introduction to the formal semantics of concurrent programs with shared variables;
 - Synchronization mechanisms: locks, flags, barriers, condition variables, semaphores;
 - Introduction to multicore architectures and systems;
 - OpenMP. Task-centric programming models;
 - Performance models;
 - Monitors and concurrent objects (locking, lock-free, wait-free);
 - Implementation of processes and synchronization.
- Distributed memory programming
 - Message passing, RPC and rendezvous, RMI, MPI; Paradigms for process interaction.
- Introduction to parallelism in scientific computing
- Overview of programming environments: Pthreads, OpenMP, MPI, multithreading in Java

Outline (today)

- Introduction
- Parallel programming concepts: task, process, state, etc.
- Parallel programming models
 - Shared address space (a.k.a. shared memory) programming model
 - Message passing (a.k.a. dist. memory) programming model
- Parallel programming basic paradigms
 - Iterative parallelism
 - Recursive parallelism
 - Producers and consumers (pipelines)
 - Clients and servers
 - Interacting peers

The Power Wall: why frequency does not increase anymore (2005)



Transistor
count still
rising

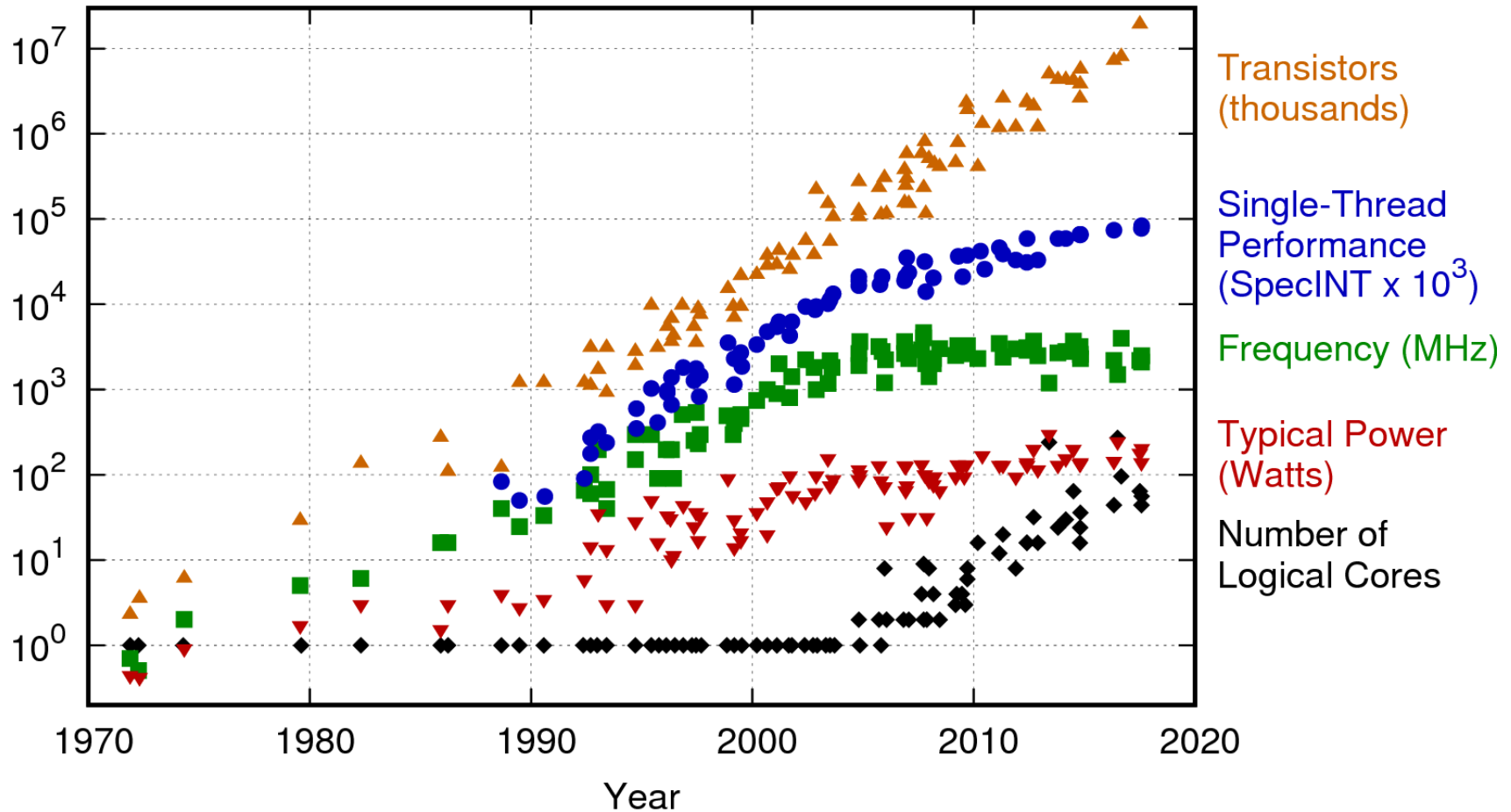
Clock speed
flattening
sharply

Moore's law: The number of transistors on a chip tends to double about every two years. [Gordon Earle Moore, co-founder of Intel]

Chart from "The free lunch is over" by Herb Sutter, 2005 (graph updated August 2009)

Moore's Law (cont'd) (2017)

42 Years of Microprocessor Trend Data

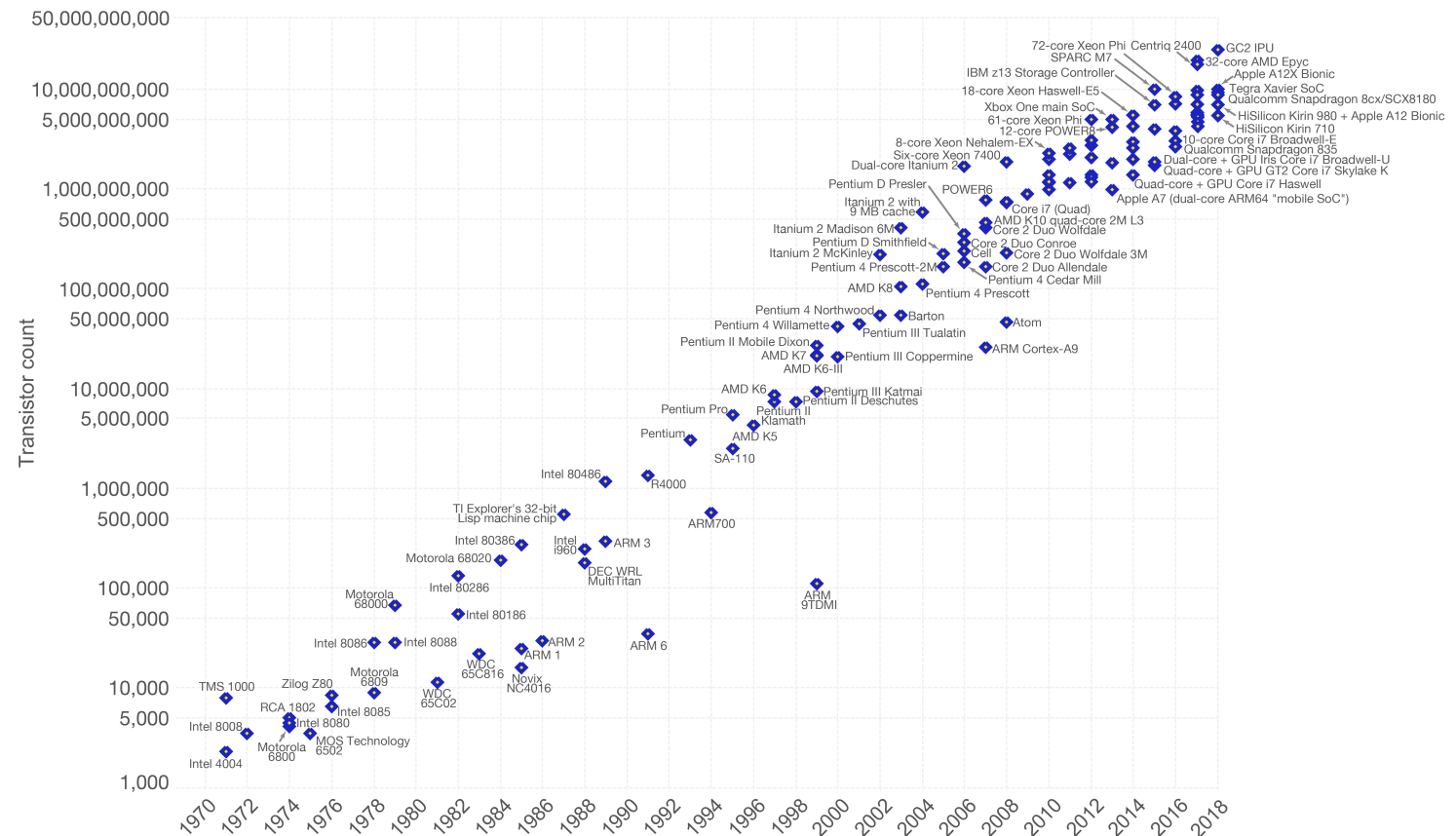


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Moore's Law (cont'd) (2019)

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

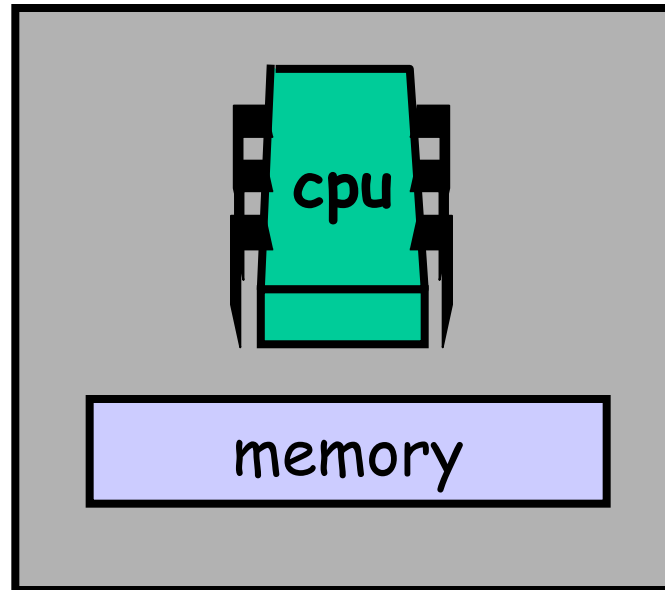
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



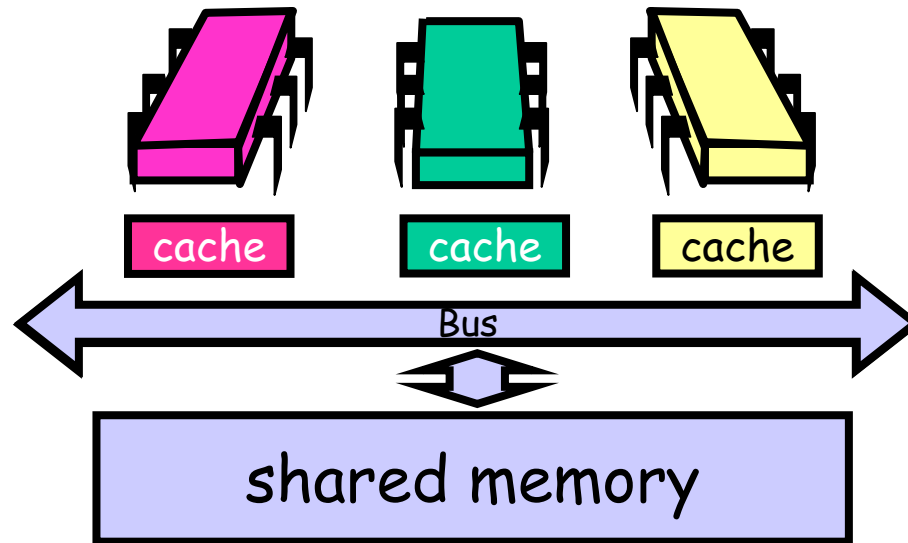
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Licensed under [CC-BY-SA](#) by the author Max Roser.

Earlier on your desktops and laptops: The Uniprocessor

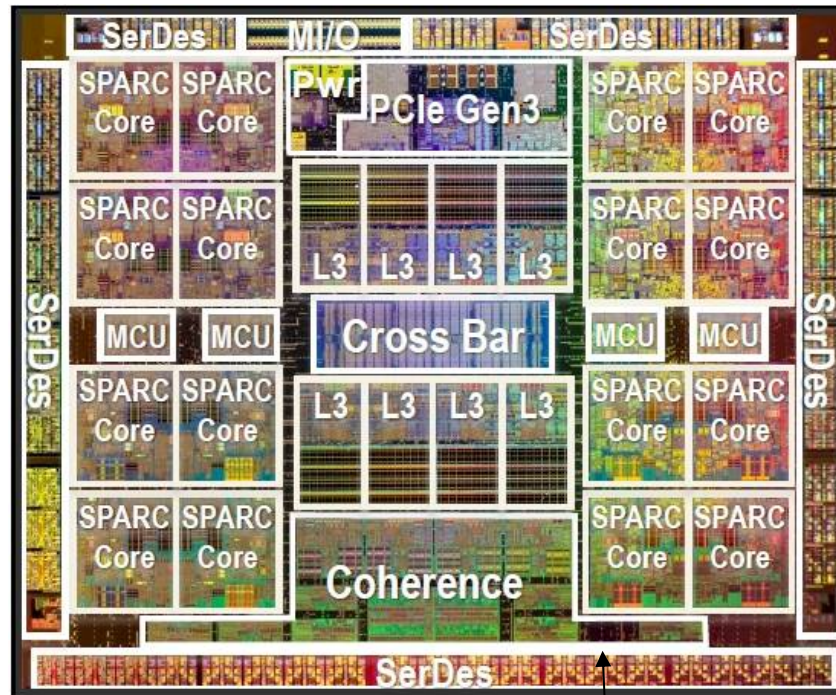


Earlier in the Enterprise: The Shared Memory Multiprocessor (SMP: Symmetric Multiprocessor)



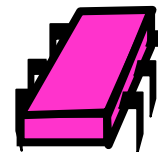
Recent Laptops, Desktops and Servers: The Multicore Processor (CMP: Chip Multiprocessor)

All on the
same chip



Oracle SPARC T5 (2013)

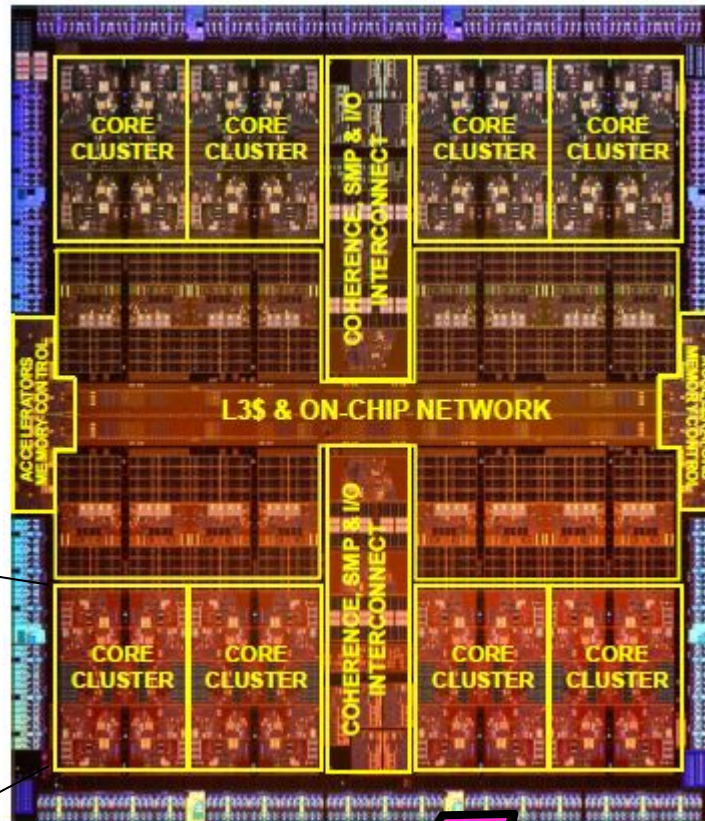
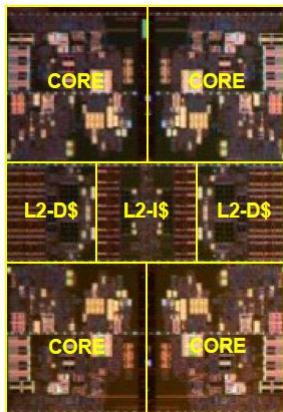
- 16 cores,
128 threads
- Private L1/L2, shared
L3 cache
- Cross-bar interconnect
- Cache coherent



"Art of Multiprocessor Programming" by
Maurice Herlihy, and Nir Shavit

Recent Laptops, Desktops and Servers: The Multicore Processor (CMP: Chip Multiprocessor)

All on the
same chip



Oracle SPARC M7 (2015)

- 32 cores,
256 threads
- Private L1 cache
- Cluster shared L2 cache
- Chip shared L3 cache
- On-chip network
- Cache coherent



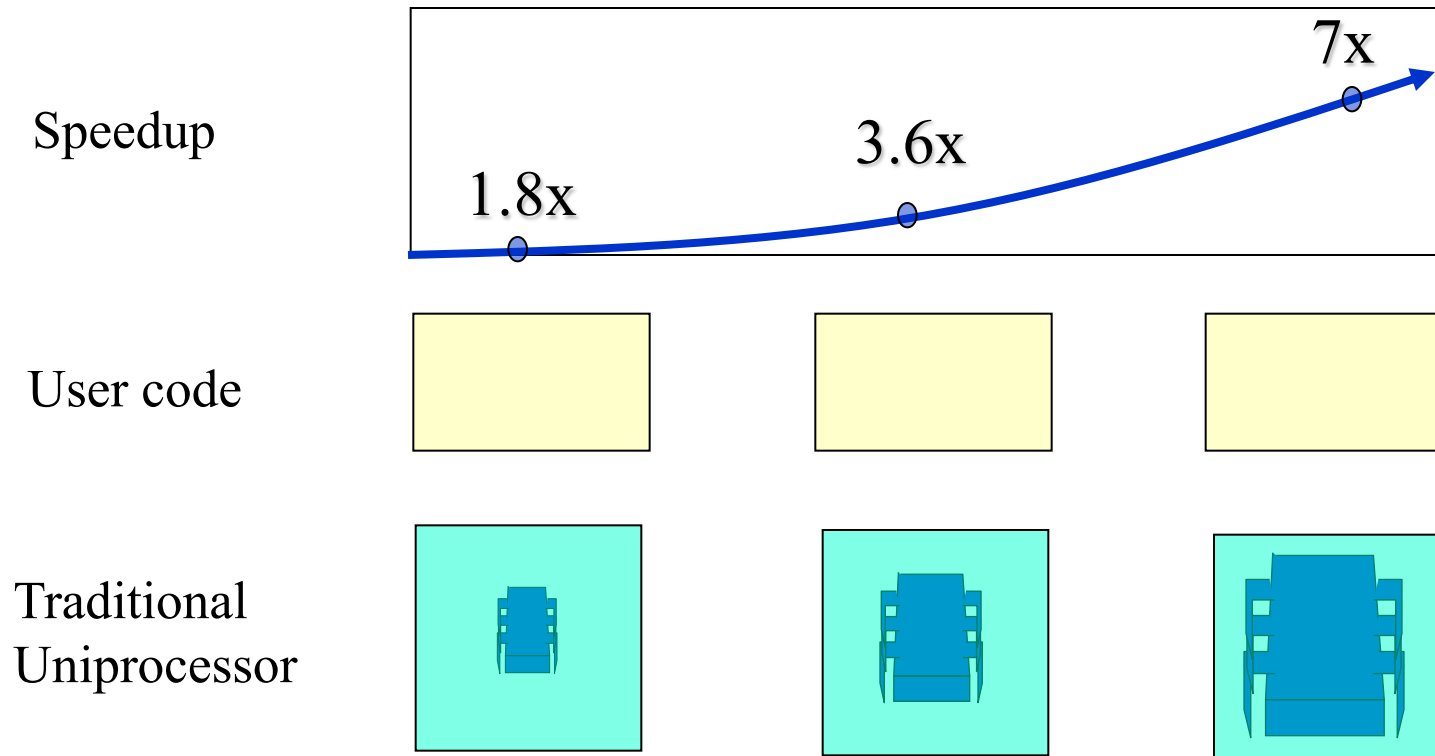
"Art of Multiprocessor Programming" by
Maurice Herlihy, and Nir Shavit

Why Do We Care?

- Time no longer cures software bloat
 - The “free ride” is over
 - The free ride where you write software once and trust Intel, Oracle, IBM, and AMD to make it faster is no longer valid.
- When you double your program’s path length
 - You can’t just wait 6 months
 - Your software must somehow exploit twice as much concurrency

Traditional Scaling Process for Software

- Write software once, trust Intel to make the CPU faster to improve performance.

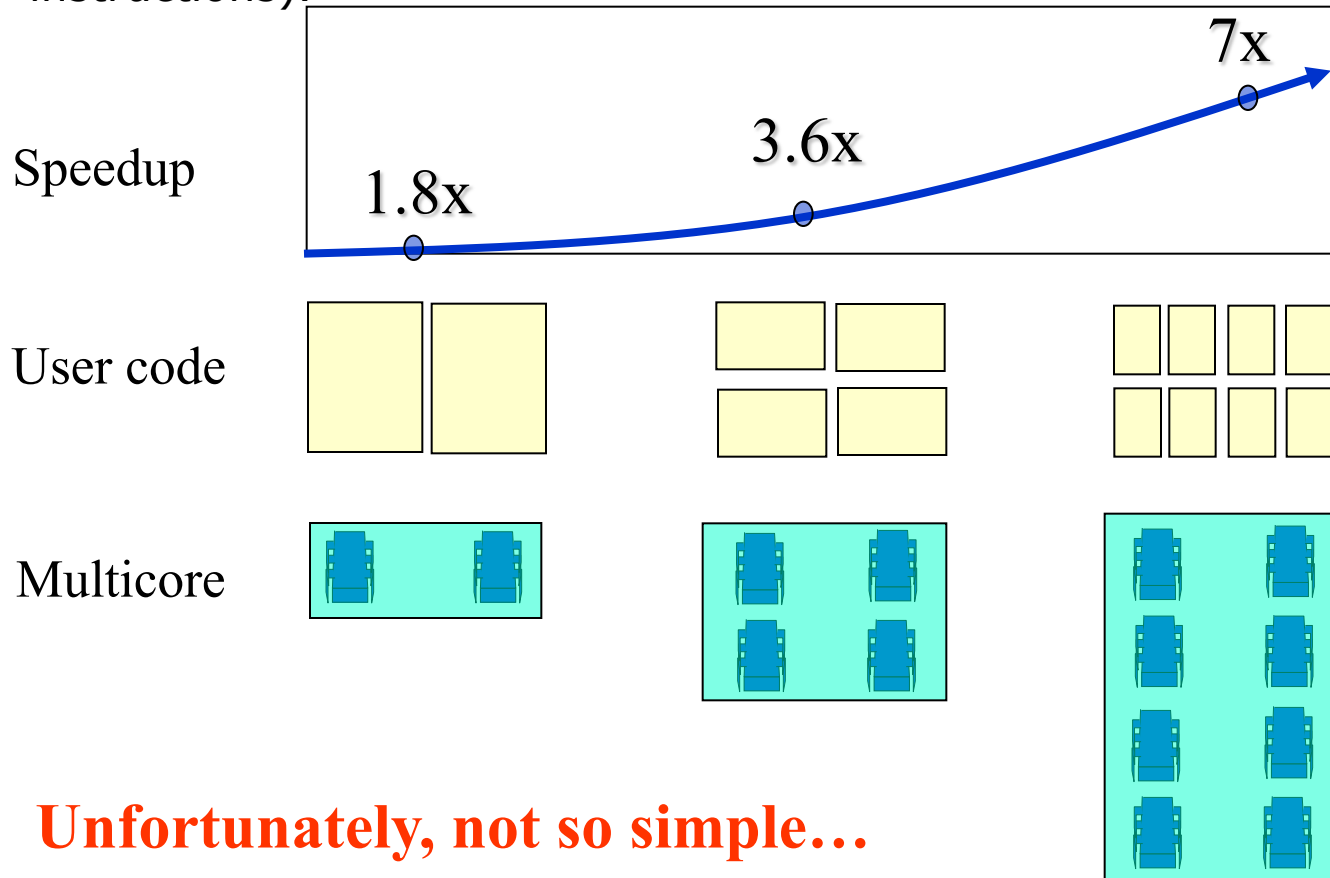


Time: Moore's law

→
"Art of Multiprocessor Programming" by Maurice
Herlihy, and Nir Shavit

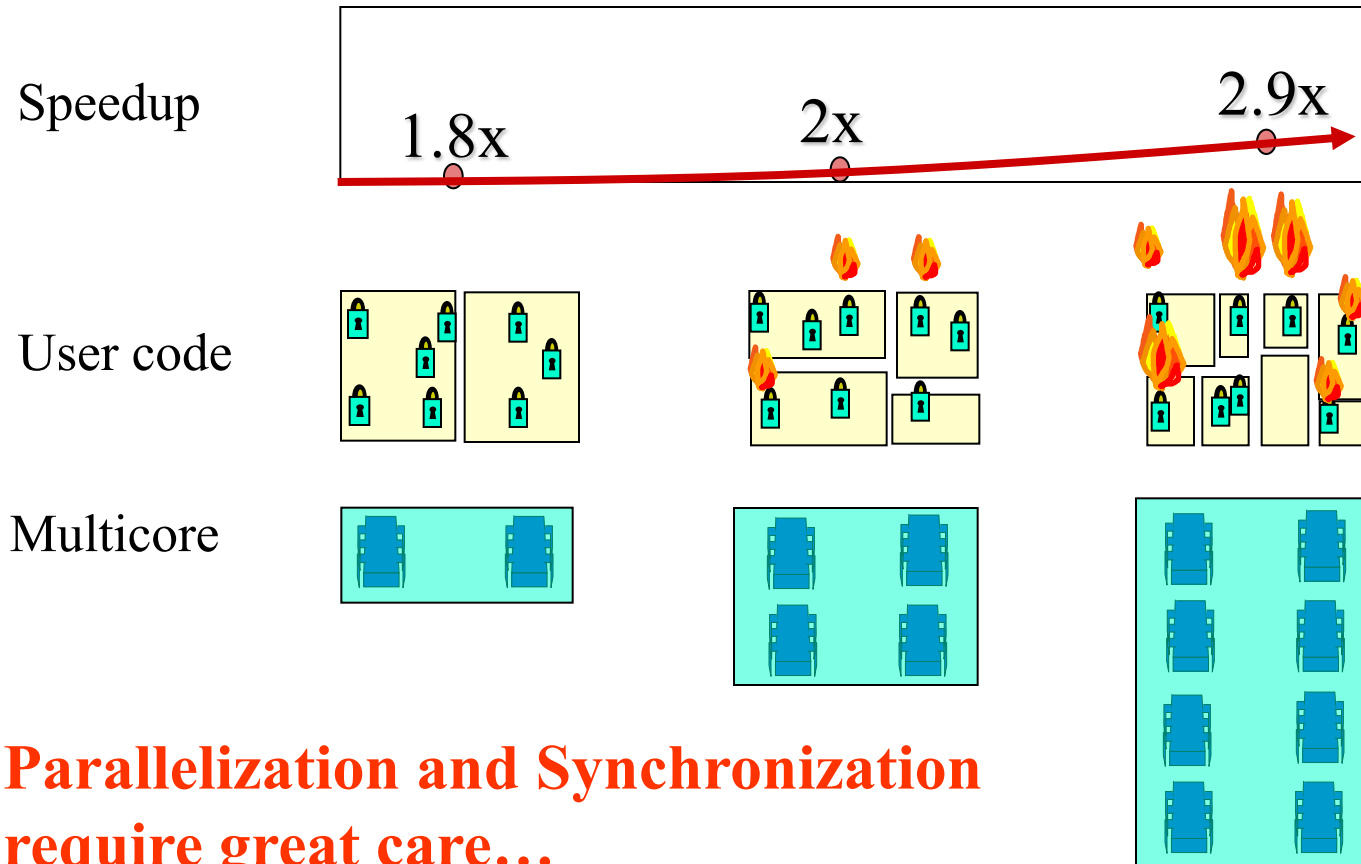
Multicore Scaling Process

- We have to parallelize the code to make software faster, and we cannot do this automatically (except in a limited way on the level of individual instructions).



Unfortunately, not so simple...

Real-World Scaling Process



**Parallelization and Synchronization
require great care...**

Sequential Program Versus Parallel Program

- **Sequential program**
 - Represents a sequence of actions that produces a result
 - Contains a single thread of control
 - Usually, it is called a **task** (to be executed by/in a **process** or **thread**)
- **Parallel (concurrent, distributed) program**
 - Represents several tasks that can be done sequentially or in parallel
 - Contains two or more threads of control
 - Executed by two or more processes (threads) that cooperate in solving a problem
 - Each process executes a sequential (single-threaded) program and has its own state
 - Processes communicate via shared memory or/and by message passing
 - Can be executed
 - Concurrently on a single-processor machine or
 - In parallel on a multiprocessor or on several computers.

Programming with Processes

- **Concurrent Programming**
 - Several concurrent processes (threads) share a single CPU or a multiprocessor
 - Shared memory programming model
 - Motivations:
 - Modeling of concurrency in the (real) world
 - Improving performance, utilization and scalability
- **Distributed Programming**
 - Processes distributed among computers communicate over network
 - Message passing programming model
 - For distributed computing: Users, data, machines are geographically distributed
 - For high performance (scalable) computing
- **Parallel Programming**
 - Parallel (concurrent) processes execute on their own processors
 - Both, shared memory and message passing programming models
 - For high performance computing : solve a problem faster or/and solve a larger problem

Speedup

- Major goal of applications in using a parallel machine is ***speedup which is offered*** by the parallel machine
 - Estimated using benchmark programs

$$\text{Speedup } (n \text{ processors}) = \frac{\text{Performance } (n \text{ processors})}{\text{Performance } (1 \text{ processor})}$$

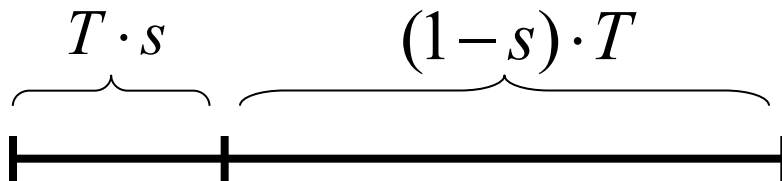
- Major goal of applications in being executed in parallel is ***speedup which can be achieved*** due to parallel execution
 - for a fixed problem size (input dataset),
performance = 1/time:

$$\text{Speedup}_{\text{fixed problem size}} (n \text{ processors}) = \frac{\text{Execution time } (1 \text{ processor})}{\text{Execution time } (n \text{ processors})}$$

Limited Parallelism (Concurrency). Amdahl's Law

- **Amdahl's Law**: If fraction s of sequential execution is inherently serial (cannot be parallelized), then the speedup that can be achieved by parallel execution is limited by $1/s$, i.e. the speedup is limited by the time needed for the sequential fraction of the program.

$$\text{Speedup}(n \text{ proc}) = \frac{\text{Sequential execution time}}{\text{Parallel execution time } (n \text{ proc})} = \frac{T}{T \cdot s + \frac{T \cdot (1-s)}{n}} = \frac{1}{s + \frac{1-s}{n}}$$

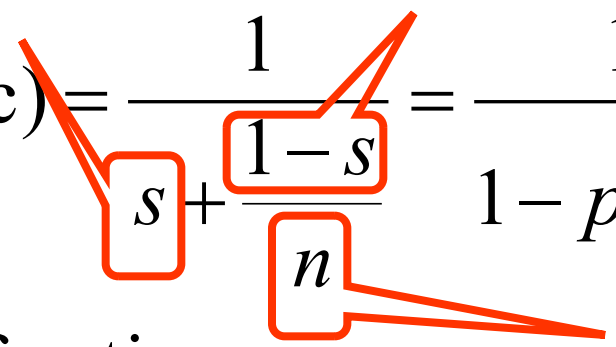


$$\lim_{n \rightarrow \infty} \frac{1}{s + \frac{1-s}{n}} = \frac{1}{s}$$

Amdahl's Law

**Sequential
fraction**

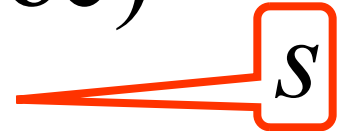
**Parallel
fraction**

$$\text{Speedup}(n \text{ proc}) = \frac{1}{s + \frac{1-s}{n}} = \frac{1}{1-p + \frac{p}{n}}$$


s is sequential fraction;

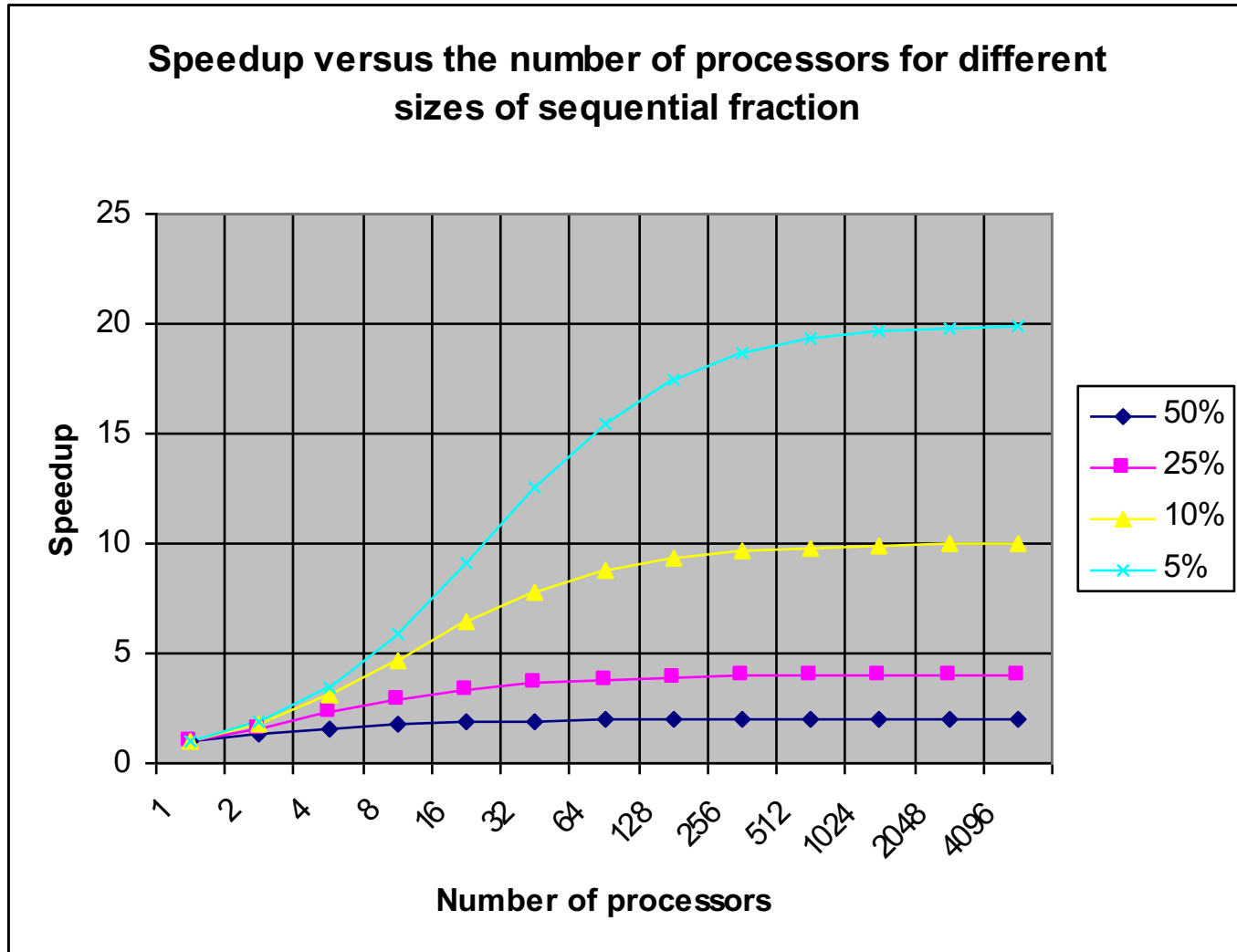
$p = (1 - s)$ is parallel fraction

**Number of
processors**

$$\text{Theoretical max speedup}(\infty \text{ proc}) = \frac{1}{s}$$


**Sequential
fraction**

Amdahl's Law



Example

Only 2,17x speedup for the money paid
for 10 CPUs and programming efforts
(60% of parallelized execution)

Assume ten processors ($n = 10$). How close to 10-fold speedup?

Concurrent	Sequential	Speedup
60%	40%	$1/(0.4 + 0.6/10) = 2.17$
80%	20%	$1/(0.2 + 0.8/10) = 3.57$
90%	10%	$1/(0.1 + 0.9/10) = 5.26$
99%	1%	$1/(0.01 + 0.99/10) = 9.17$

With 99% parallelized we are now
utilizing 9 out of 10.

The Moral

- In many applications, 90% of the execution can be parallelized rather easily.
- From Amdahl's law we see that its worth our effort to try and parallelize even the last 10% which are responsible for 50% of the utilization of our multiple processors.
- Making good use of our multiple processors (cores) means
- Finding ways to effectively parallelize our code
 - Minimize sequential parts
 - Reduce idle time in which threads **wait** without work

Impediments To Speedup

- Inherently sequential parts (Amdahl's law) – unavoidable (unless change algorithms)
- Load imbalance
- Synchronization and context switch overheads: critical sections, condition synchronization, delays, process creation (fork/join)

Gustafsson's Law

- The Amdahl's law is formulated for a fixed problem size having a fixed serial fraction ***s***.
- By allowing ***s*** to vary, the Amdahl's methodology can accommodate a varying problem size.
- For many applications, increasing the number of processors ***n*** allows solving the problem of a larger size, i.e., the problem size grows with ***n***, and the serial fraction ***s*** varies with ***n***.
- **The Gustafsson's law** (for a varying problem size, hence the varying serial fraction)

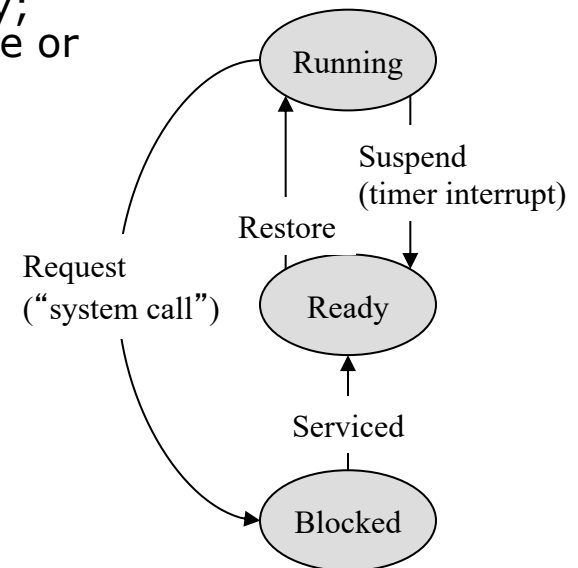
$$\text{Speedup}(n) = n - (n - 1) \cdot s(n)$$

Hardware

- **A single-CPU processor**
 - For multiprogramming and concurrent programming with shared memory
- **Shared-memory multiprocessor**
 - For concurrent and parallel programming with shared memory
 - Centralized shared memory (UMA, SMP)
 - Distributed shared memory (NUMA)
 - Multi-core processors (chip-multiprocessors)
- **Distributed memory multiprocessor, Computer clusters and networks**
 - For distributed and parallel programming with message passing
- **Hierarchical multiprocessor**
 - Distributed memory MP with SMP (or multicore) nodes
 - For distributed and parallel programming with message passing and shared memory

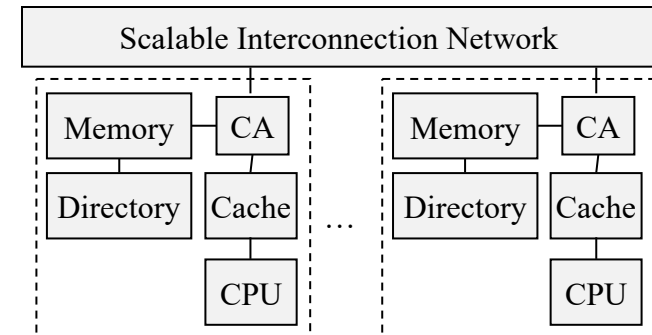
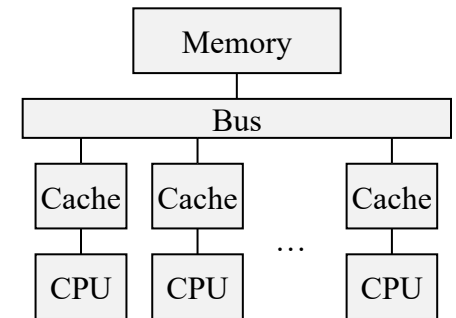
Multiprogramming on a Single Processor

- **Time-sharing:** The CPU time is shared between several programs by time slicing
 - OS controls and schedules processes
 - When a time slice expires or the process blocks for some reason (e.g. IO operation, blocking synchronization operation), OS makes a process (context) switch
- **Context switch:** Suspend the current process and restore a new process:
 1. Save the current process state to the memory; Add the process to the tail of the ready queue or to a wait queue;
 2. Take a process from the head of the ready queue; Restore the proc state and make it running.
- Resume a blocked process:
 - Move a process from the wait queue to the ready queue.



Shared-Memory Multiprocessors

- A shared-memory MP offers a shared address space
 - Processes executing on different processors have access to a common (shared) memory
- **Symmetric multiprocessors** (SMP, UMA)
 - Centralized shared memory
 - Bus (with “broadcast” capability)
 - Snoopy cache coherence protocols
- **Distributed shared memory MP** (NUMA)
 - Shared memory is distributed among nodes
 - Scalable interconnection network
 - Directory-based cache coherence protocols



Multi-Core Processors

- Combines several independent cores into a single package on the same die.
 - Tightly-coupled multiprocessor on a chip
 - May share L2 and L3 cache or have separate caches
 - Shared the same interconnect to the rest of the system (memory)
- Allows **TLP** – **thread-level-parallelism** on the chip
- **Simultaneous multithreading**: One core may support multiple hw threads (a.k.a. also multiple virtual CPUs)
 - **Hyper-threading** – in Intel's CPUs, e.g. Intel Xeon CPUs
 - The total number of virtual threads (CPUs) = the number of cores times the number of hw threads per core.

Some Recent High-End Multicore Processors

Manufacturer/Brand	Example	Technology	#cores	#threads	Max Frequency	TDP
Intel Core	i9-9980XE	14 nm	18	36	4.5 GHz	165 W
Intel Xeon	W-3175X	14 nm	28	56	3.8 GHz	255 W
Intel Xeon Phi	7295	14 nm	72	288	1.6 GHz	320 W
AMD Ryzen	2990WX	14 nm	32	64	4.2 GHz	250 W
Oracle SPARC	M8	20 nm	32	256	5.0 GHz	n/a
IBM POWER9	SMT8	14 nm	24	96	4.0 GHz	n/a

Distributed Memory Multiprocessors and Multi-Computers

- There is no HW-supported shared memory
 - Each node has its own local memory
 - Processes on different nodes communicates by message passing
 - Major problem is the communication latency
- **Massively parallel processors (MPP) and computer clusters**
 - Tightly coupled computers connected with a high-speed interconnection network
- **Computer networks**
 - Loosely coupled computers (LAN or WAN)

Parallel Programming Concepts

- **Task** – an arbitrary piece of un-decomposed work in parallel computation
 - **Executed sequentially**; concurrency is only across tasks
 - Fine-grained versus coarse-grained tasks
- **Process (thread)** – an abstract entity that performs tasks assigned to it
 - Each process has its state and a unique ID
 - Processes communicate and synchronize to perform their tasks via shared memory or/and by message passing
 - Two types of processes:
 - Heavy-weight processes
 - Threads (light-weight processes)

Parallel Programming Concepts (cont'd)

- A **process state (context)** includes all information needed to execute the process:
 - In CPU: contents of PC, NPC, PSW, SP, registers
 - In memory: contents of text segment, data segment, heap, stack
 - Can be cached in data and instruction caches
 - A portion (but not stack) can be shared with other processes
- A **context switch** – terminate or suspend the current process and start or resume another process
 - Performed by an OS kernel
 - The CPU state of the current process must be saved to the memory
 - “Dirty” cache lines can be written back to memory on replacement

Parallel Programming Concepts (cont'd)

- **Processor (core, virtual CPU)** – a physical engine on which processes execute
 - Processes virtualize the computer to the programmer
 - First write program in term of processes, then map to processors
- **Concurrent program (execution)** is formed of several processes (threads) that share a (multi)processor
 - Usually more processes than processors
- **Parallel program (execution)** is formed of several processes each executes on its own processor
 - Usually more processors than processes
- **Distributed program (execution)** is formed of processes that are distributed among processors and communicate over network
 - No shared memory

Heavy-Weight Processes (HWP)

- A **heavy-weight process** has its own virtual address space not shared with other HWPs
 - Entire process state is private
 - An HWP can be multithreaded
 - An HWP can create another HWP (a copy of itself or a different process)
- Example: UNIX processes

Example: UNIX Processes

- **fork** creates a new process (child) which is an exact copy of the parent
 - The child starts execution at **fork**, gets own PID, and has its own state.
 - COW – Copy On Write policy
 - Processes are “joined” by **wait** in parent and **exit** in child.
 - On success, **fork** returns 0 to child and the child’s PID to parent

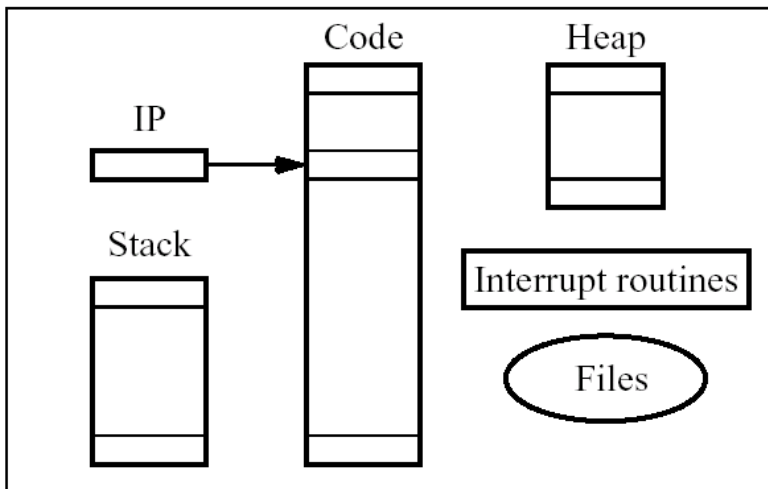
```
pid = fork();
if (pid == 0) {
    Code to be executed by child;
    exit(0);
} else {
    Code to be executed by parent;
    wait(pid); // join
}
```

- **fork** followed by **exec** (in child) is used to execute a completely different program in the child process.

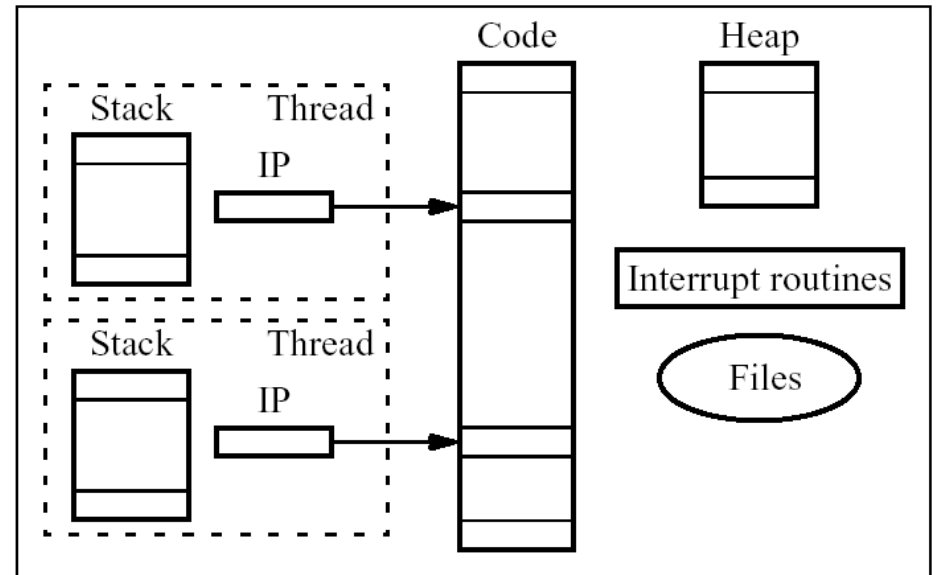
Threads

- A **thread** is essentially a program counter, an execution stack, and a set of registers – thread context.
 - All the other data structures and the code belong to a HWP where threads are created, and are shared by the threads.
 - Each thread is assigned a unique thread ID.
- Example: **Pthreads** – POSIX (IEEE Portable OS Interface) threads
 - **pthread_create** creates a new thread
 - The thread executes a given function, has its own stack and registers, but share global variables with other threads.
 - Threads can be “joined” by **pthread_join** in parent and **return** or **pthread_exit()** in child

Process versus Thread



(a) Process



(b) Threads

Parallel Programming Models

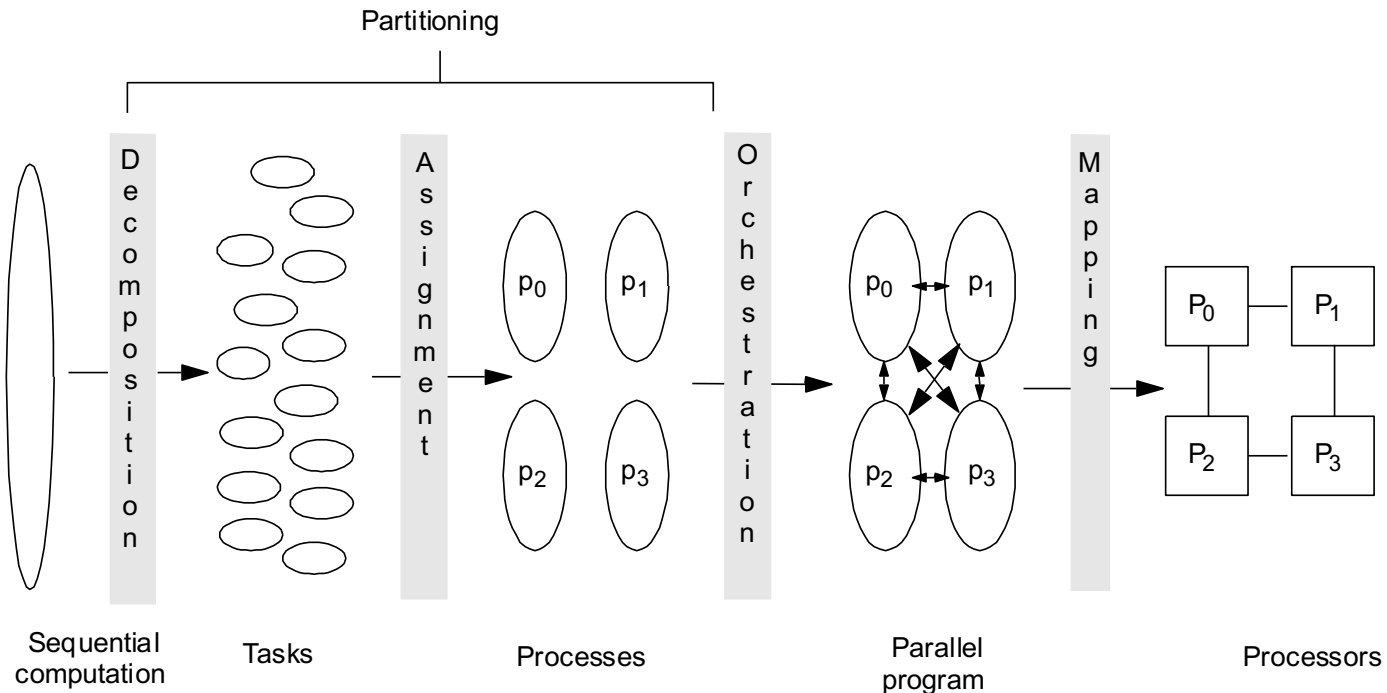
- A programming model defines how processes communicate and synchronize
- **Shared memory programming model**,
a.k.a. Shared address space (SAS) model.
 - Portions of virtual address spaces of processes are shared (common)
 - Processes communicate via shared memory (common variables) by conventional reads and writes
 - Explicit synchronization mechanisms (libraries) such as locks, barriers, semaphores, monitors , in an imperative (sequential) language, e.g. C or Java.
- **Distributed memory programming model**,
a.k.a. Message passing programming model
 - No shared memory, only communication channels are shared
 - Processes communicate by sending/receiving messages over the channels.
 - Synchronization is implicit: a message to be received must be sent

Some Existing Synchronization Mechanisms

- To synchronize thread, to coordinate accesses to shared memory
 - Locks
 - Condition variables
 - Barriers
 - Semaphores
 - Monitors (synchronized objects)
 - Concurrent objects (lock-based, lock-free, wait-free)
 - Lock-free using Compare-and-Swap (CAS)
 - Transactional memory

Four Steps in Creating a Parallel Program

- (1) Decomposition; (2) Assignment; (3) Orchestration; (4) Mapping
- Done by programmer or system software (compiler, runtime, ...).
 - The programmer does it explicitly



1. Decomposition

- Break up computation into **tasks** to be divided among processes
 - Tasks may become available dynamically
 - Number of available tasks may vary with time
 - Number of tasks available at a time is upper bound on achievable speedup
 - i.e. identify concurrency and decide level at which to exploit it
- Goal: Enough tasks to keep processes busy, but not too many
- Simple way to identify concurrency in a sequential program is to look at loop iterations
 - dependence analysis; if not enough concurrency (loops are sequential), then look further: examine fundamental dependences, ignoring loop structure

2. Assignment

- Specifying mechanism to **divide work up among threads**
 - E.g. which process computes forces on which stars, or which rays
 - Together with decomposition, also called **partitioning**
 - Balance workload, reduce communication and management cost
- Structured approaches usually work well
 - Code inspection (parallel loops) or understanding of application
 - Well-known heuristics
 - Static assignment (when the number of tasks is known)
 - Dynamic assignment
 - “**Bag of tasks**” for the fixed number of processes (“work farm”)
 - Create a new process for each new task
- **Work-dealing** (offload work to others) versus **work-stealing** (steal work from others)

3. Orchestration

- Naming data
- Structuring communication
 - Shared versus private memory
 - Messaging
- Synchronization
- Organizing data structures and scheduling tasks temporally
- Goals
 - Reduce cost of communication and synch. as seen by processors
 - Hide communication and/or synchronization latency by multithreading and/or data pre-fetching

4. Mapping

- After orchestration, already have parallel program
- Two aspects of mapping:
 - Which processes will run on same processor, if necessary
 - Which process runs on which particular processor
 - mapping to a network topology
- Two extremes:
 - (1) **Space-sharing**: machine divided into subsets, one appl in a subset
 - (2) **Complete resource management control to OS**
 - Real world is in between: user specifies desires, system may ignore
- Usually adopt the view: process \leftrightarrow processor

Programming Notation Used in Lectures

- A language similar to C
- Declarations – much like in C and Matlab
 - Primitive types (**int**, **double**, **char**, etc.)
 - Arrays: **int c[1:n]**, **double c[n, n]**
 - Process declaration **process** – start one or several processes in background (detached thread)
- Statements:
 - Assignment statement
 - Control flow statements: **if**, **while**, **for** – much like in C
 - **for** with quantifiers:
for [i=0 to n-1, j = 0 to n-1] ...;
for [i=1 to n by 2] ...; # odd values from 1 to n
for [i=1 to n st i!=x] ...; # every value except i=x, “st” stands for “such that”
 - **Concurrent statement**: **co...oc**
 - starts two or more threads in parallel and then waits until all the threads complete
 - similar to the openMP “**parallel**” pragma
 - **Await statement**: **await**
 - used for synchronization – will be introduced later

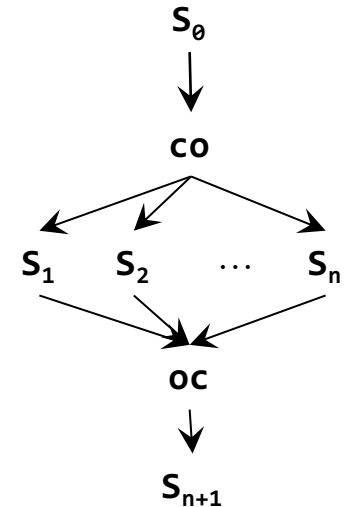
Two Forms of co-Statement

1. Different statements in parallel arms

```

S0;
co S1; // thread 1
|| ...
|| Sn; // thread n
oc;
Sn+1;

```

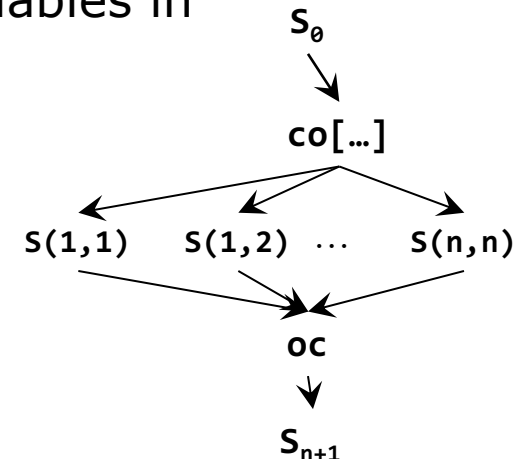


2. With quantifiers: Same sequence of statements in parallel arms (for every combination of variables in quantifiers)

```

S0;
co [i=1 to n, j=1 to n] { // n x n threads
    S(i,j);
}
oc;
Sn+1;

```



process Declaration

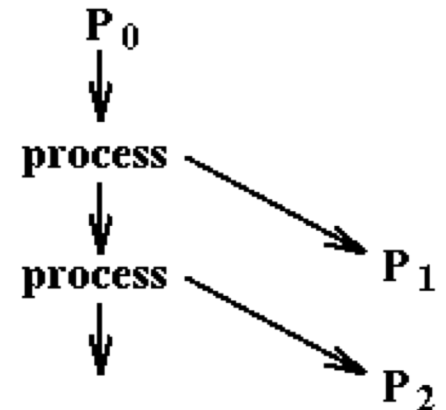
- Syntax is similar to **co** but with one arm and /or one quantifier
- Start a single process in background

```
process p  
    { body }
```

- Start array of processes in background

```
process p[quantifier]  
    { body }
```

- May declare local variables and access global variables
- Can appear where procedure declarations can appear
- Forked when its declaration is encountered
- No synchronization upon termination – detached processes



Parallel Programming Paradigms (Basic Application Patterns)

- Iterative parallelism
- Recursive parallelism
- Producers and consumers (pipelines), a.k.a. dataflow
- Clients and servers
- Interacting peers

1. Iterative Parallelism

- Parallelism of *independent iterations*.
 - An iterative program uses loops to examine data and compute results. Some loops can be parallelized.
- Example: Matrix multiplication $C = A \cdot B$

– Sequential version:

```
double a[n,n], b[n,n], c[n,n];
for [i=0 to n-1] {
    for [j=0 to n-1] {
        c[i,j]=0.0;
        for [k=0 to n-1]
            c[i,j]= c[i,j]+a[i,k]*b[k,j];
    }
}
```

– Parallel version (by rows):

```
double a[n,n], b[n,n], c[n,n];
co [i=0 to n-1] {
    for [j=0 to n-1] {
        c[i,j]=0.0;
        for [k=0 to n-1]
            c[i,j]= c[i,j]+a[i,k]*b[k,j];
    }
} oc;
```

- Replace **for** with **co**
- **n** threads in **co** are executed concurrently for different values of **i**
- To increase concurrency (by elements) **co [i=0 to n-1, j=0, n-1]**

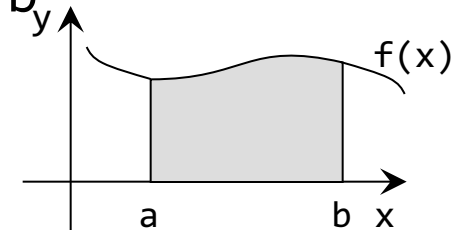
2. Recursive Parallelism

- Parallelism of **independent recursive calls**
 - Assume, a recursive procedure calls itself **more than once** in its body.
 - If the calls are independent, they can be executed in concurrent threads
- **“Divide and conquer” (domain decomposition)**
- Examples: Quick sort, adaptive quadrature
 - Based on domain decomposition: Split a data region (e.g. list, interval) into several sub-regions to be processed in parallel recursively using the same algorithm

Example: The Quadrature Problem

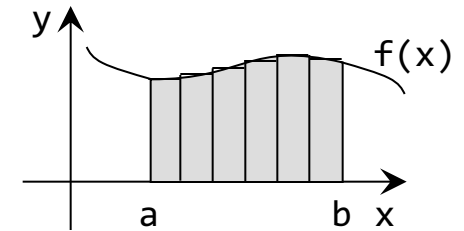
- Compute an approximation of the integral of a continuous function $f(x)$ on the interval from a to b

$$\text{area} \approx \int_a^b f(x)$$

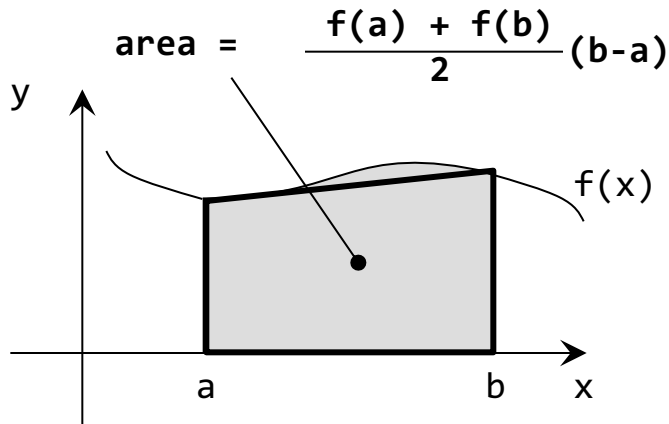


- Sequential iterative quadrature program
 - using the trapezoidal method:

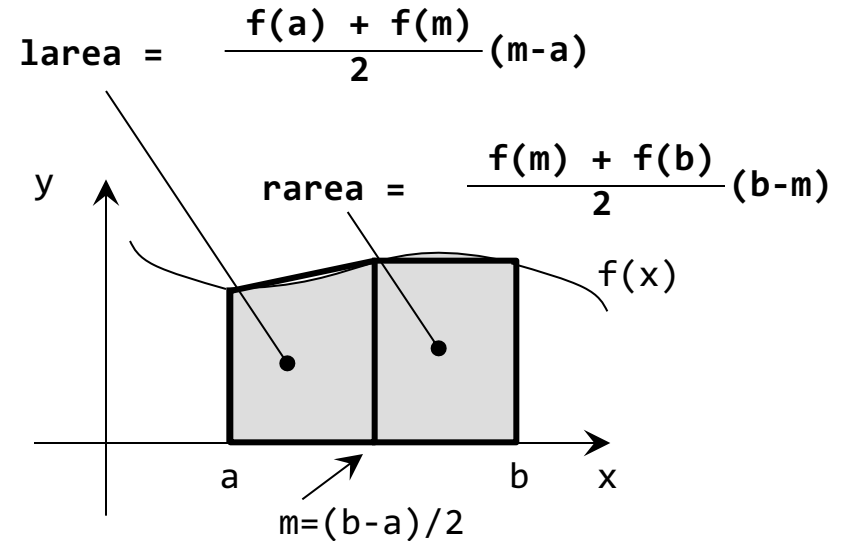
```
double fl = f(a), fr, area = 0.0;
double dx = (b-a)/ni;
for [x = (a + dx) to b by dx] {
    fr = f(x);
    area = area + (fl + fr) * dx / 2;
    fl = fr;
}
```



Recursive Adaptive Quadrature Procedure



(a) First approximation (**area**)



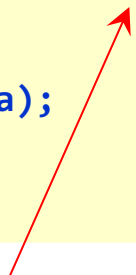
(b) Second approximation (**larea + rarea**)

- The second approximation can be computed for ≥ 2 subintervals
- If $|\text{larea} + \text{rarea} - \text{area}| > e$, repeat computations for each of the intervals $[a, m]$ and $[m, b]$ in a similar way until the difference between consecutive approximations is within a given e

Recursive Adaptive Quadrature Procedure


- Sequential procedure:

```
double quad(double l,r,fl,fr,area) {  
    double m = (l+r)/2;  
    double fm = f(m);  
    double larea = (fl+fm)*(m-l)/2;  
    double rarea = (fm+fr)*(r-m)/2;  
    if (abs((larea+rarea)-area) > e) {  
        larea = quad(l,m,fl,fm,larea);  
        rarea = quad(m,r,fm,fr,rarea);  
    }  
    return (larea + rarea);  
}
```



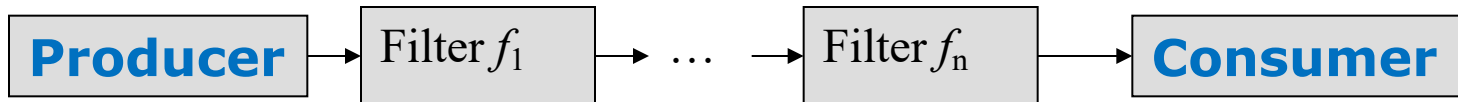
- Parallel procedure:

```
double quad(double l,r,fl,fr,area) {  
    double m = (l+r)/2;  
    double fm = f(m);  
    double larea = (fl+fm)*(m-l)/2;  
    double rarea = (fm+fr)*(r-m)/2;  
    if (abs((larea+rarea)-area) > e) {  
        co larea = quad(l,m,fl,fm,larea);  
        || rarea = quad(m,r,fm,fr,rarea);  
        oc  
    }  
    return (larea + rarea);  
}
```



- Two recursive calls are independent and can be executed in parallel
- Usage: `area = quad(a, b, f(a), f(b), (f(a)+f(b))*(b-a)/2)`

3. Producers and Consumers. Pipeline



- Parallelism of production (of next data) and consumption (of previous data)
- One-way data flow between **Producer** and **Consumer**
 - Functional **filters** can be placed in between
- Processes form a **pipeline**
 - Parallelism of pipeline stages
 - Each consumes the output of predecessor and produces the input for its successor – true data dependence between stages
- **Data buffers** (FIFO queues) are placed between processes

The Producer-Consumer Problem

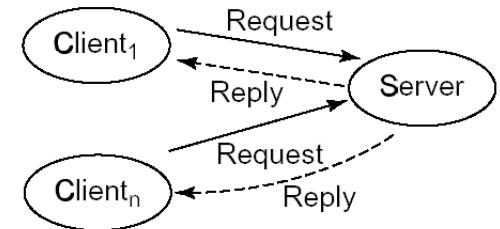
- Coordinate access of multiple processes to a shared (fixed-size) buffer
 - A.k.a. also the **bounded-buffer problem**
 - Consumers should not read the buffer if it's empty
 - Producers should not write to the buffer if it's full
 - For a **single-slot buffer**, read must not overlap with write
 - Producer should not overwrite data which are not read yet (partly read);
 - Consumer should not read data which are not completely written yet.
 - A **multiple-slot buffer** allows concurrent access

The Readers-Writers Problem

- A specific case of the Producer-Consumer problem
- Develop a protocol to access a shared resource that allow concurrent reads but preclude concurrent writes
- Selective mutual exclusion
- Examples:
 - Shared database
 - Concurrent reads but exclusive writes
 - The single-lane bridge problem
 - Cars heading in the same direction can cross the bridge at the same time, but cars heading in opposite direction cannot;
 - Cars cannot pass each other on the bridge
 - The unisex bathroom problem
 - Either men or women but not both at the same time

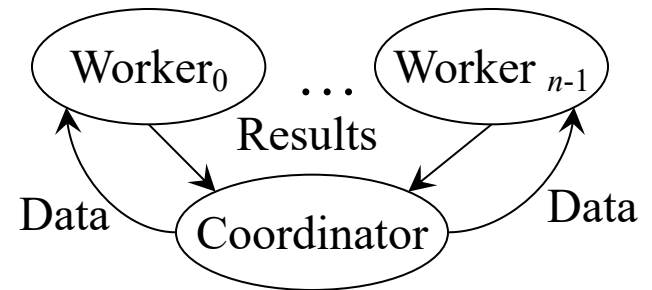
4. Clients and Servers

- Parallelism of client and server processes
 - Client requests a service
 - Server provides the service
 - Two-way communication: request – reply pairs
- Parallelism in servicing of multiple clients in separate threads
 - Multithreaded servers. Synchronization might be required
- Implemented
 - Distributed-memory: using message passing, RPC, rendezvous, RMI
 - Shared-memory: using subroutines, monitors etc.
- Example: (Distributed) file systems
 - open, read, write, close – client requests
 - Data, acknowledgements – server replies

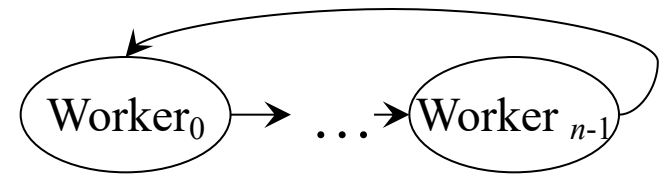


5. Interacting Peers

- Parallelism of “equal” peers
 - Each execute the same set of algorithms and communicate with others in order to achieve the goal
- Configurations
 - An exponential network
 - A hierarchy (tree)
 - Master (coordinator) and slaves (workers)
 - Roles may change
 - A circular pipeline
 - Each to each
 - Mesh
 - Arbitrary



(a) Coordinator/worker interaction



(b) A circular pipeline