IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Principles of Parallel and Concurrent Programming

Introduction to Concurrency and Parallelism

Rocco De Nicola

January 2026

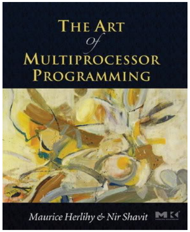
GSSI - L'Aquila and IIT CNR - Pisa

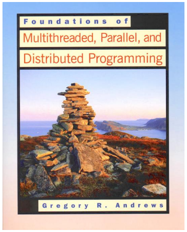
Covered Topics

- Concurrent Programming Styles: Iterative vs Recursive Parallelism
- Mutual exclusion and critical sections: Algorithms for guaranteeing mutual exclusion and implementing critical sections
- Linguistic constructs for concurrent programming: Semaphores and Monitors
- Indirect and Direct Communication: Shared Memory vs Message Passing.
- Linguistic constructs for distributed programming: Rendez-Vous e Remote Procedure Calls.
- Controlled Communication: Shared tuple spaces and pattern matching.

Reading Material

Lectures and slides are based on the two books below.





- Maurice Herlihy & Nir Shavit: The Art of Multiprocessor Programming, Morgan Kaufmann, 2008
- Gregory Andrews: Multithreaded Parallel and Distributed Programming, Addison Wesley, 2000

It might be useful studying also

- Mordechai Ben-Ari: Principles of Concurrent and Distributed Programming, Addison Wesley, 2006

Moore's Law versus Clock Speed

Gordon Moore, the co-founder of Intel in 1965 predicted:

- The number of transistors on an integrated circuit would double each year (later revised to doubling every 18 months).
- This laid the groundwork for another prediction: that doubling the number of transistors would also double the performance of CPUs every 18 months.

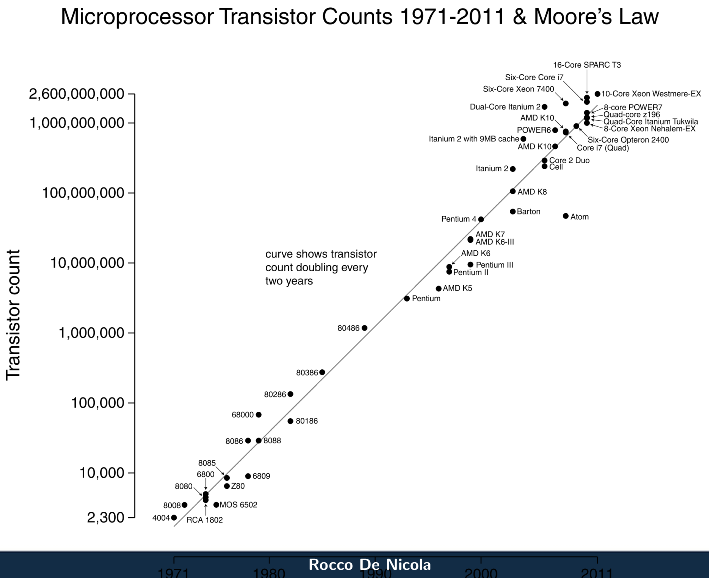
Robert Dennard (IBM) in 1974 explained why this was possible.

- As transistors shrink, voltage and current scale down proportionally.
- Power density stays approximately constant, while clock frequency can increase.
- Result: higher single-core performance without higher power.

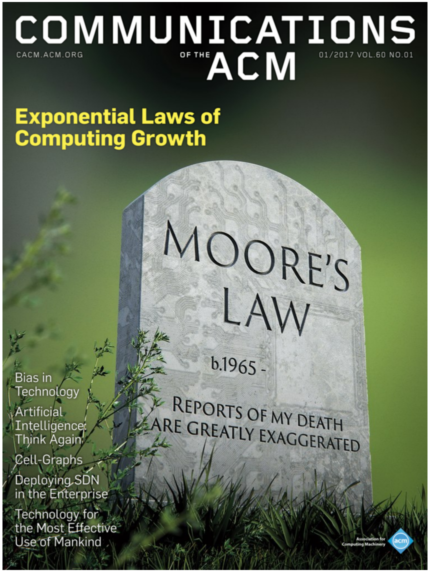
In the mid-2000s, Dennard scaling broke down. Voltage stopped scaling, leakage power increased, and power density began to rise sharply. This created the power wall: frequency scaling and increasingly complex single cores became impractical.

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Moore's Law



The death of Moore's Law


COMMUNICATIONS OF THE ACM

CACM.ACM.ORG

01/2017 VOL.60 NO.01

Exponential Laws of Computing Growth

Bias in Technology

Artificial Intelligence: Think Again

Cell-Graphs

Deploying SDN in the Enterprise

Technology for the Most Effective Use of Mankind

MOORE'S LAW
b.1965 -
REPORTS OF MY DEATH ARE GREATLY EXAGGERATED
Association for Computing Machinery
acm

Moore's Law versus Clock Speed

Moore's Law and Performances:

- The number of transistors that can be placed on a chip doubles every two years.
- Performance do not grow consistently, because of:
  - Heating: As chip geometries shrink and clock frequencies rise, leakage current increases, leading to excessive power consumption and heat.
  - Memory Wall: Memory access times haven't kept up with increasing clock frequencies.
  - Transmission Delays: Due to limitations in the means of producing inductance within solid state devices, delays in signal transmission grow as feature sizes shrink.

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Moore's Law and Dennard's Scaling

- Moore's law continues to deliver more transistors, but they are used to replicate cores rather than speed up one core. Performance gains depend on parallelism and energy efficiency.
- Performance growth does not come from higher clock frequency but primarily from
  - Parallelism
  - Energy efficiency
  - Specialization and system-level design: GPUs, TPUs, AI accelerators
- Moore's law slows, but useful performance per watt continues to improve.

A new law:

Parallelism (number of CPUs on a single board) doubles every two years!

Computer Core Types and LLM

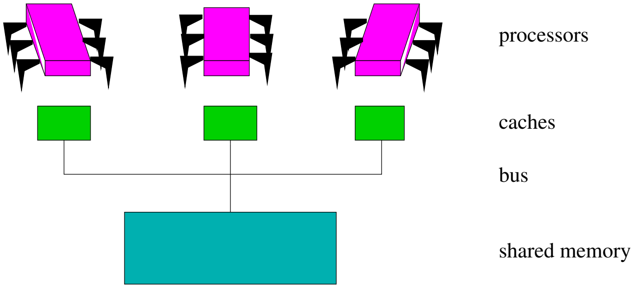
- Large Language Models (LLM) need
  - efficient linear algebra (matrix multiplication, dot products, . . . ,
  - high memory bandwidth (billions of parameters),
  - efficient core interconnections (to share intermediate results efficiently),
  - low-precision arithmetic (not many numerical computation).
- LLM rely on specialized hardware cores designed for high-throughput matrix and tensor operations, rather than traditional general-purpose CPU cores.
- The hardware design balances massive parallelism, high memory bandwidth, and energy efficiency to handle billions of parameters.
  - GPUs (Graphics Processing Units) - Originally for graphics, now optimized for SIMD-style operations
  - TPUs (Tensor Processing Units) - Designed for deep learning workloads.
  - AI accelerators - Specialized cores for tensor operations.

Concurrency is Everywhere

The end of Moore's law has had a major impact on the practice of programming:

- BEFORE: CPUs got faster without significant architectural changes
  - one could program as usual, and wait for the program to run faster
  - concurrent programming was a niche skill (for operating systems, databases, . . .)
- NOW: CPUs do not get faster but add more and more parallel cores
  - It is necessary to program with concurrency in mind, otherwise programs remain slow
  - concurrent programming is pervasive
- Very different systems all require concurrent programming:
  - desktop PCs, smart phones, video-games consoles, embedded systems, Raspberry Pi, cloud computing, . . .

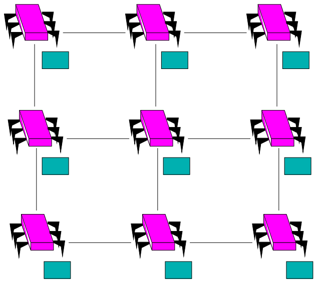
Symmetric Multiprocessing Architecture



- One address space, shared by all processors.
- Uniform Memory Access (UMA): cpus have all memories equidistant from all processors.
- OpenMP is an API (a set of compiler directives, library routines, and environment variables) that can be used to specify shared memory parallelism for programming in C, C++ and Fortran.

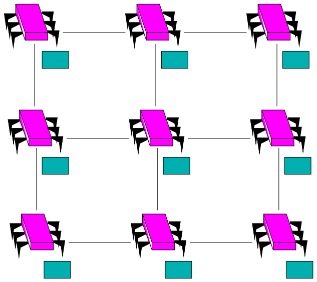
IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Non-Uniform Memory Access Architecture



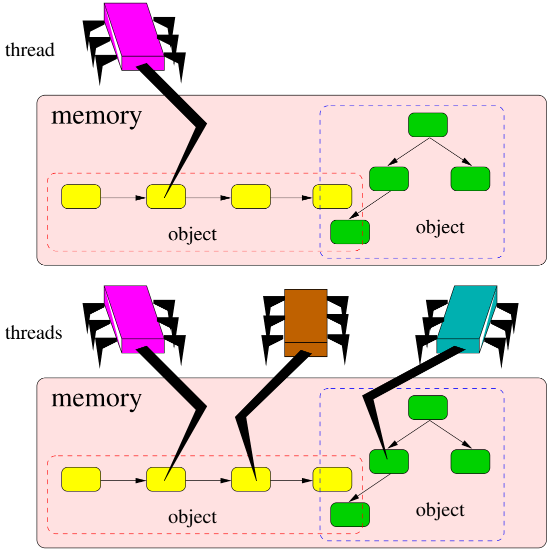
- NUMA (N for non) architectures are such that performance varies with data location.
- NUMA is sometimes confusingly called Distributed Shared Memory architecture: the memory is physically distributed but logically shared.
- MPI (Message Passing Interface) is a specification of an API for programming on such architectures.

Multicomputer Architecture



- The same diagram as for NUMA shared memory! The difference is the lack of any hardware integration between cache/memory system and the interconnect cpus.
- Each processor only accesses its own physical address space.
- Information is shared by explicit, co-operative message passing.
- No memory consistency issues.

Sequential vs Concurrent Computations



Basic Concurrency Model

Our basic assumptions

- multiple threads
- single shared memory
- objects live in memory
- unpredictable asynchronous delays

Jargon: hardware $\rightarrow$ processors, software $\rightarrow$ threads

Road Map

We focus first on principles of multiprocessor programming:

- We start with *idealized* models.
- We look at *simplistic*, *fundamental* problems.
- *Correctness* is emphasized over *pragmatism*.

Our motto

Principled understanding is the foundation of effective practice.

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Basic Concurrency Model

Our basic assumptions

- multiple threads
- single shared memory
- objects live in memory
- unpredictable asynchronous delays

Jargon: hardware $\rightarrow$ processors, software $\rightarrow$ threads

Road Map

We focus first on principles of multiprocessor programming:

- We start with *idealized* models.
- We look at *simplistic*, *fundamental* problems.
- *Correctness* is emphasized over *pragmatism*.

Leonardo Da Vinci

Chi s'innamora di pratica senza scienza è come il nocchiere che entra in naviglio senza timone o bussola, che mai ha certezza dove si vada.

SCHOOL FOR ADVANCED STUDIES LUCCA

Amdahl's Law

With n processors that can run in parallel what is the speed up we can achieve?

$$\frac{\mathit{1\_thread}\ \mathit{sequential\ execution\ time}}{\mathit{n\_threads}\ \mathit{parallel\ execution\ time}}$$

Amdahl's law shows that the impact of introducing parallelism in a program is limited by the fraction p that can be parallelized.
Given a job, that is executed on n processors

- Let $p \in [0, 1]$ be the fraction of the job that can be parallelized (over n processors).
- Let sequential execution of the job take 1 time unit.

Then the speedup is:

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

Amdahl's Law: Examples

$n = 10$

$p = 0.6$ gives speedup of $\frac{1}{0.4+\frac{0.6}{10}} = 2.2$

$p = 0.9$ gives speedup of $\frac{1}{0.1+\frac{0.9}{10}} = 5.3$

$p = 0.99$ gives speedup of $\frac{1}{0.01+\frac{0.99}{10}} = 9.2$

Conclusion:
To make efficient use of multiprocessors, it is important to

- minimize sequential parts, and
- reduce idle time in which threads (processors) wait.

Amdahl's Law: Examples

$n = 10$

$p = 0.6$ gives speedup of $\frac{1}{0.4+\frac{0.6}{10}} = 2.2$

$p = 0.9$ gives speedup of $\frac{1}{0.1+\frac{0.9}{10}} = 5.3$

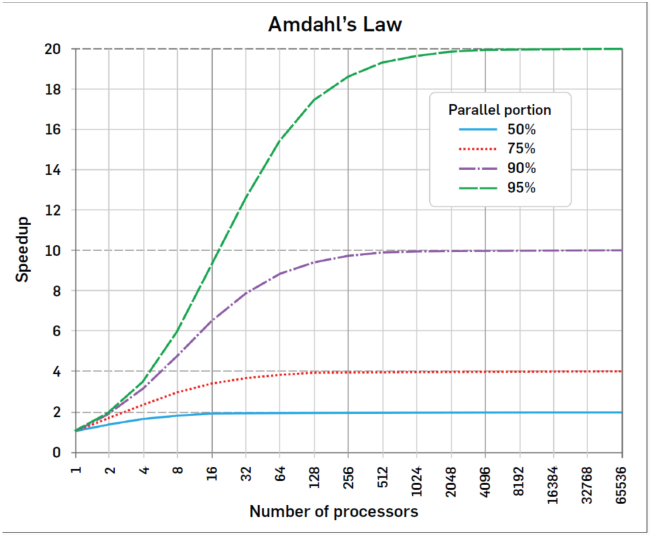
$p = 0.99$ gives speedup of $\frac{1}{0.01+\frac{0.99}{10}} = 9.2$

Conclusion:
To make efficient use of multiprocessors, it is important to

- minimize sequential parts, and
- reduce idle time in which threads (processors) wait.

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Amdahl's Law: examples



Source: Communications of the ACM, Dec. 2017

Our Basic Assumptions

- The CPU time is shared between several programs by time-slicing
- The Operating Systems controls and schedules processes.
- If a time slice period has expired or the process blocks for some reason (e.g. I/O operation, blocking synchronization operation), the CPU is assigned to another process and a context switch is performed.

Context Switch

Process switching suspends the current process and restore a new process:

1. Save the current process state to the memory;
2. Add the process to the tail of the ready queue or to the tail of a wait queue;
3. Pick a process from the head of the ready queue;
4. Restore the process state and make it running.

A parallelization problem and 5 attempts

Write a program that terminates if and only if the total function f has a (positive or negative) zero and proceeds indefinitely otherwise.

Assume to have a program that looks for positive zero:

```
S1 =
   found := false; x := 0;
   while (not found)
       do x:= x+1; found := (f(x) = 0) od
```

From this we can build the program looking for a negative zero.

```
S2 =
     found := false; y := 0;
     while (not found)
         do y:= y−1; found := (f(y) = 0) od
```

Attempt 1

An obvious solution would be running `S1` and `S2` in parallel:

```
S1 || S2
```

If f has a positive zero and not a negative one, and `S1` terminates before `S2` starts, the latter sets found to false and starts looking indefinitely for the nonexisting zero.

The problem is due to the fact that found is initialized to false twice.

LESSON 1

USING SHARED VARIABLES MAY LEAD TO PROBLEM

Attempt 2

Let us consider a solution that initializes found only once.

```
found := false; (R1 || R2)        where
R1 =  x := 0; while (not found)
                 do x:= x+1; found := (f(x) = 0) od
R2 =  y := 0 while (not found)
                 do y:= y-1; found := (f(y) = 0) od
```

If f has (again) only a positive zero assume that:

1. R2 proceeds just up to the while body (after the do) and is preempted
2. R1 computes till a zero is found
3. R2 gets the CPU back

When R2 restarts it executes the while body and may set found to false with found $:= (f(y) = 0)$. The program then would not terminate because it would look for a non existing negative zero.

Attempt 2 ctd

The problem is due to the fact that found is set to false (by means of found := f(y) = 0) after it has already got the value true.

LESSON 2

NO ASSUMPTION ABOUT THE MOMENT A PROGRAM IS INTERRUPTED CAN BE MADE (It can only be programmed).

Attempt 3

Let us see what happens if we do not perform "unnecessary" assignments and only assign true when we find a x or a y such that f(x) = 0 or f(y) = 0.

```
found := false; (T1 || T2) where
 T1 = x := 0; while not found
        do x:= x+1; if f(x) = 0 then found := true fi od
 T2 = y := 0; while not found
        do y:= y-1; if f(y) = 0 then found := true fi od
```

Assume that f has only a positive zero and that T2 gets the CPU to keep it until it terminates. Since this will never happen, T1 will never get the chance to find its zero.

Attempt 3 - ctd

This problem is due to the considered scheduler of the CPU, to avoid problems we would need a *non fair* scheduler; but this is a too strong assumption.

LESSON 3

NO ASSUMPTION CAN BE MADE ON THE SCHEDULING POLICY OF THE CPU.

Attempt 4

To avoid assumptions on the scheduler, we could think of adding control to the programs and let them "pass the baton" once they have got their "chance" to execute for a while.

```
turn:= 1; found := false; (P1 || P2) where
 P1 = x := 0; while not found do wait turn:= 1 then
      turn:= 2; x:= x+1; if f(x) = 0 then found := true fi od
 P2 = y := 0; while not found do wait turn:= 2 then
      turn:= 1; y:= y-1; if f(y) = 0 then found := true fi od
```

If P1 finds a zero and stops when P2 has already set turn:= 1, P2 would be blocked by the wait command because nobody can change the value of turn.

Attempt 4 - ctd.

The program does not terminate because of the waiting of an impossible event.

LESSON 4

ON TERMINATION CARE IS NEEDED FOR OTHER PROCESSES.

A CORRECT Solution!

Pass (again) the baton before terminating.

```
turn:= 1; found := false; (P1; turn:= 2  || P2; turn:= 1)
where
P1 = x := 0; while not found do
                  wait turn:= 1 then
                      turn:= 2; x:= x+1;
                      if f(x) = 0 then found := true fi
                                                         od
P2 = y := 0; while not found do
                  wait turn:= 2 then
                      turn:= 1; y:= y-1;
                      if f(y) = 0 then found := true fi
                                                          od
```

Multithreaded Programming

Different Kinds of Multithreading Programming

1. Parallel Programming
2. Concurrent Programming
3. Distributed Programming

SCHOOL FOR ADVANCED STUDIES LUCCA

Multithreaded Programming

Different Kinds of MultithreadingProgramming

1. Parallel Programming
2. Concurrent Programming
3. Distributed Programming

1. Parallel Programming

- Challenge: Solve a specific problem in which some parts can be executed in parallel.
- Motivation: Higher performance computing, i.e. solving a problem faster or/and solve a larger problem
- Assumption: Some form of parallel HW is available for exploitation.

Multithreaded Programming

2. Concurrent Programming

- Challenge: Design a protocol to share a resource between independent tasks.
- Motivations: Maximize throughput of the resource or minimize waiting times of individual tasks, or some compromise of these.
- Assumption: Independent tasks compete asynchronously for the resources (CPU, memory) that are available.

3. Distributed Programming

- Challenge: Design protocols to maintain the consistency of a distributed system distributed over a network
- Motivation: Keep the system operational in presence of changes and faults.
- Assumption: The system has no central hub, i.e. its time and state are relativistic and it is based on unreliable machines and connections.

The co Notation

We introduce some simple notation (the so called co-notation). It is not a real programming language, but a concise way of expressing what we will need to express in real languages. It indicates creation of a set of activities, for the duration of the enclosed block, with synchronization across all activities at the end of the block. This is sometimes called `fork-join` parallelism.
The parallel activities are separated by //, comments preceded by ##

```
    co
    a=1; // b=2; // c=3; ## all in parallel.
    oc
```

We will also use co statements with indices.

```
    co [i = 0 to n-1] {
        a[i] = a[i] + 1; ## all in parallel.
    oc  }
```

The co notation

Things get more interesting when the statements within a co access the same location.

```
co
a=1; // a=2; // a=3; ## What is a afterwards?
oc
```

To resolve this, we need to define our memory consistency model. For our toy language examples we assume sequential memory consistency (SC).

SC - Sequential Memory Consistency Model

A program executed with an SC model will produce a result which is consistent with the following rules:

1. ordering of atomic actions (particularly reads and writes to memory) from a single thread have to occur in normal program order
2. atomic actions from different threads are interleaved arbitrarily (i.e. in an unpredictable sequential order)

Sequential Memory Consistency Model

SC executions are like a random switch, allowing processes to access memory one at a time. This does not mean that instructions are executed sequentially. It means that the result must be the same as if they were executed sequentially.

What is a after the execution of

```
co
a=1; // a=2; // a=3; ## all in parallel.
oc
```

It can result in `1, 2 or 3`. Why?

What is the value of b after the execution of

```
a=0;
co
a=1; // a=2; // b=a+a; ## all in parallel.
oc
```

If reads and writes of single variables are atomic and each access to value is a read while each assignment is a write, in the example, `b` could be `0, 1, 2, 3, or 4`.

Atomic Actions

The simple example we have just seen can illustrate the complications introduced for real languages, compilers and architectures.
A sensible compiler would implement `b=a+a` with one read of `a`, so the outcomes which produce an odd value for `b` would never happen. We cannot rely on such unknown factors.

It is therefore useful to have means for specifying that certain blocks of code are to be treated as atomic. In our notation, statements enclosed in `< >` must appear to be atomic, i.e. they must appear to execute as a single indivisible step with no visible intermediate states.

```
    a=0;
    co
    a=1; // a=2; // <b=a+a;>
    oc
```

Now the only outcomes for `b` are `0, 2 or 4`.

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Atomic Actions and Interleavings

As another example, consider this attempt to increment the count twice

```
    co
    count++; // count++;
    oc
```

where each statement corresponds to a sequence of three actions:

read-modify-write or LOAD - EXECUTE - STORE.

Even with sequential consistency, there are twenty possible interleavings, of which only two match the intended semantics.

```
    co <count++>; // <count++>; oc
```

Exercise

Consider this attempt to reverse the contents of an array in parallel. Can you see what might go wrong?

```
    co [i = 0 to n-1] {a[i] = a[n-i-1]; } oc
```

Atomic Actions and Interleavings

As another example, consider this attempt to increment the count twice

```
co
count++; // count++;
oc
```

where each statement corresponds to a sequence of three actions:

read-modify-write or LOAD - EXECUTE - STORE.

Even with sequential consistency, there are twenty possible interleavings, of which only two match the intended semantics.

```
co <count++>; // <count++>; oc
```

Exercise

Consider this attempt to reverse the contents of an array in parallel. Can you see what might go wrong?

```
co [i = 0 to n-1] {a[i] = a[n-i-1]; } oc
```

The "await" Notation

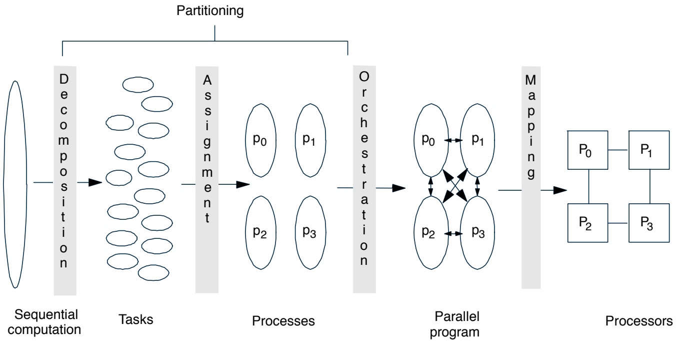
The await notation `< await (B) S >` allows us to indicate that `S` must appear to be delayed until `B` is true, and must be executed within the same atomic action as the successful check of `B`, i.e. behaving like

```
while (true) { < if (B) {S; break;} > }
```

For example, the code below results in x having a value of 25, because of the semantics of await and sequential consistency.

```
a=0; flag=0;
co
{a=25; flag=1;}
//
<await (flag==1) x=a;>
oc
```

Four Steps in Creating a Parallel Program



Parallelization

Starting from a sequential program it is possible to execute subparts in parallel if they are independent.

Read and Write Sets

Read set: the set of variables that an operation uses but does not change

Write set: the set of variables that an operation does changes and possibly uses.

Actions Independence

Two actions are independent if the write set of each of them is disjoint from the union of the read and write set of the other:

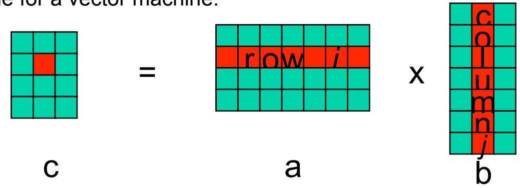
1. $write_{set}(op_1) \cap (write_{set}(op_2) \cup read_{set}(op_2)) = \emptyset$
2. $write_{set}(op_2) \cap (write_{set}(op_1) \cup read_{set}(op_1)) = \emptyset$

Parallel Programming Paradigms

1. Iterative parallelism
2. Recursive parallelism
3. Producers-consumers pipelines
4. Clients and servers
5. Interacting peers

Iterative Parallelism

An example for a vector machine.



- For each element c_{ij} of the product matrix c the task is to compute the sum of the products of the elements on row a_i and column b_j: $c_{ij} = \sum_k (a_{ik} * b_{kj})$
- A set of tasks can be executed independently, if: no variable written by a task is read or written by any other task.
- The computation of each c_{ij} is independent and there are several alternatives for actually performing it:
  1. execute 1 sequential run
  2. compute rows or columns of c with n parallel runs
  3. compute all elements of c with n^2 parallel runs

Iterative Parallelism

- An iterative program uses loops to examine data and compute results.
- Some loops can be parallelized to execute concurrently independent iterations.

Matrix multiplication $C = A \times B$

– 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 i

Iterative Parallelism

- It is interesting to consider the case we have P processors and have, e.g., to manipulate matrices $n \times n$, with $n \geq P$.
- Assuming that n is a multiple of P we design a program with P processes, each of which manages a group of rows in C.

```
process worker[w = 1 to P] { # strips in parallel
  int first = (w-1)* n/P; # first row of strip
  int last = first + n/P - 1; # last row of strip
  for [i = first to last] {
     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];
       }
  }
}
```

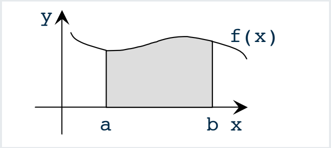
Recursive Parallelism

Parallelism of independent recursive calls

- When a procedure calls itself more than once in its body.
- The different calls can be executed in concurrent threads

An example: adaptive quadrature

Split a data region (e.g. list, interval) into several sub-regions to be processed recursively using the same algorithm.

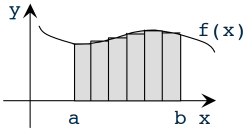


Compute an approximation of the integral of a continuous function `f(x)` on the interval from `a` to `b`.

The Quadrature Problem

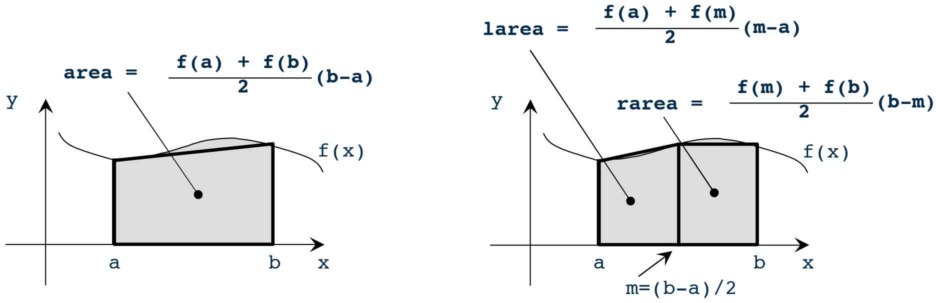
- Sequential iterative quadrature program:

```
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;
}
```



- This program can be parallelized to be executed by `ni` processes. Their intermediate results have finally to be added.
- Unfortunately this approach does not guarantee that the desired approximation of the result is obtained.

Recursive Adaptive Quadrature Procedure



area = $\frac{f(a) + f(b)}{2}$(b-a)
y
f(x)
a
b
x
larea = $\frac{f(a) + f(m)}{2}$(m-a)
rarea = $\frac{f(m) + f(b)}{2}$(b-m)
y
f(x)
a
b
x
m=(b-a)/2

(a) First approximation (`area`)

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

If `absvalue((larea + rarea) - area)` *gt* 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 error e.

IMT SCHOOL FOR ADVANCED STUDIES LUCCA

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);
}
```

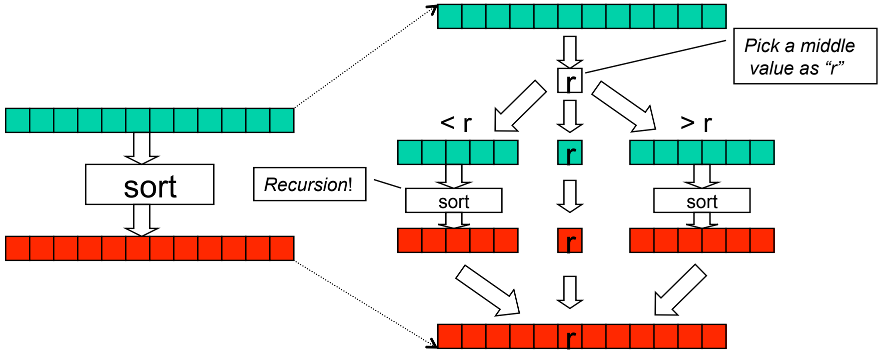
- Parallel procedure:

```
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)
```

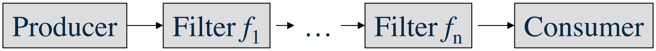
Parallel Quick Sort



- How many tasks could be executed independently, i.e. in parallel ?
- How many independent phases are there, i.e. what is the time complexity ?
- What are the memory requirements ?

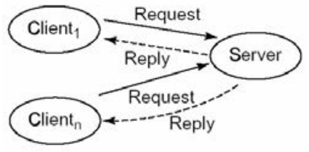
Pipelines of Producers and Consumers

- Parallelism of production (of next data) and consumption (of previous data)
- One-way data stream between Producer and Consumer
- Filters can be placed in between processes organized in a pipeline
- Parallelism of pipeline stages



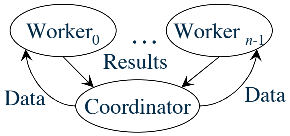
Clients and Servers

- 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)
- With distributed-memory implemented using message passing, RPC, RendezVous, RMI
- With shared-memory implemented using subroutines, semaphores, monitors, . . . (see later)

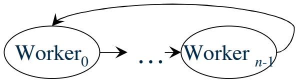


Interacting Peers

- Parallelism of equal peers
  - Each execute the same set of algorithms and communicate with others in order to achieve the goal.
- Configurations
  - Regular Structures: Grid, Mesh, etc.
  - Master (coordinator) and slaves (workers)
  - Circular pipelines
  - Each to each



(a) Coordinator/worker interaction



(b) A circular pipeline