CPA – Parallel Computing

Informatics Engineering Degree

T2. Shared Memory. Basic Parallel Algorithms Design

J. M. Alonso, F. Alvarruiz, I. Blanquer, D. Guerrero, J. Ibáñez, E. Ramos, J. E. Román

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Year 2016/17

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Content

Section 1

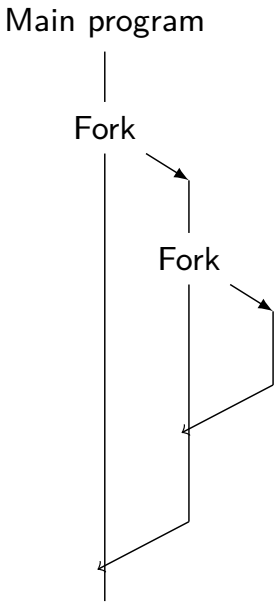
Shared Memory Model

- Model
- Details

Concurrent processes

Concurrent processes are typically defined using *fork-join*-like constructions.

- *Fork* creates a new concurrent task that starts its execution in the same point where the initiator task made the fork.
- *Join* waits for the task to finish.
- Example: system call `fork()` in Unix.



This schema can be implemented at the level of:

- Operating system processes (*heavy processes*).
- Threads (*light processes*).

Shared Memory Model

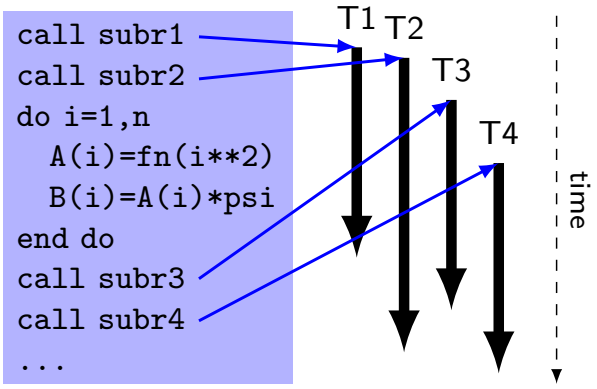
- Tasks share a common memory space.
- Programming quite similar to sequential case
  - There are no task "private" data.
  - We do not need to exchange data explicitly.
- Inconvenients:
- Memory access need to be coordinated.
  - Locks, monitors, ...
  - Unpredictable results if data access is not properly protected.
- Data locality is difficult to control.

Thread model

This model is coupled to shared memory architectures.

(*thread*): Independent instruction flow that can be scheduled by the operating system.

- A process may have multiple concurrent execution threads.
- Each thread has local data.
- Threads share resources/memory of the process.
- Synchronization is needed.



Java Threads

Object-oriented model

```
public class HelloThread extends Thread {
  public void run() {
    System.out.println("Hello from a thread!");
  }
  public static void main(String args[]) {
    (new HelloThread()).start();
  }
}
```

Synchronized methods

```
public class SynchronizedCounter {
  private int c = 0;
  public synchronized void increment() {
    c++;
  }
  public synchronized int value() {
    return c;
  }
}
```


POSIX Threads (pthreads)

A standard for managing threads in Unix systems (standard IEEE POSIX 1003.1c, 1995).

- Software-library based (API to access Operating System calls)
- Only for the "C" language.
- Explicit parallelism: Important programming effort.

Some operations

- Creations: `pthread_create`, `pthread_join`.
- Locking: `sem_wait`, `sem_post`.
- Mutual exclusion: `mutex_lock`, `mutex_unlock`.
- Condition variables: `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`.

Drawbacks:

- Portability (Windows has its own threads).
- Task-oriented parallelism, rather than data-oriented parallelism.

OpenMP

Portable standardization of threads (1997-1998).

- Based on compiler directives.
- Available in C/C++ and Fortran.
- Portable/multi-plataform (Unix, Windows).
- Easy to use: Incremental parallelisation.
- Sequential code can be preserved.

Some directives and functions

- `#pragma omp parallel for`.
- `omp_get_thread_num()`.

Creation and termination of threads is implicit in some directives

- The programmer does not bother about explicitly calling *fork/join*.

Unix Processes

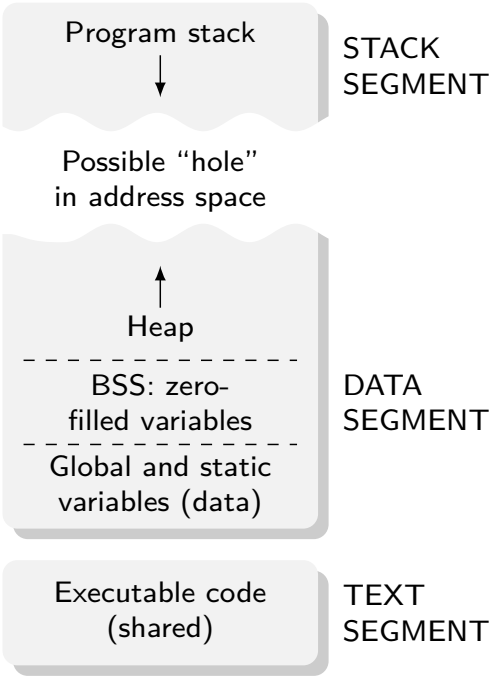
Each process comprises information about resources and its execution status

- Program code (read-only, can be shared).
- Variables (global, *heap* and *stack*).
- Execution context: registres, stack pointer, etc..
- System resources (only accessible through the O.S.)
  - Identifiers (process, user, group).
  - Environment, work directory, signals.
  - File descriptors.

In multi-threaded processes

- Each thread has its own execution context.
- Each thread has its own independent stack.
- System resources are shared.

Memory Model of Unix Processes



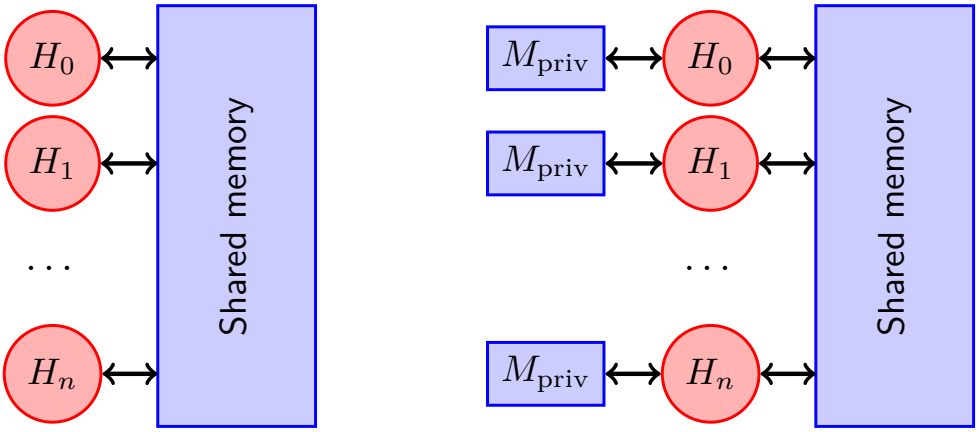
Information in the kernel of the operating system (PCB: *process control block*)

- Program counter
- Stack pointer
- Registers
- Process state
- Process ID
- User ID
- Group ID
- Memory limits
- Open files, sockets
- ...

Threaded-memory model

Simple model: Single address memory space

More realistic model: unique space of addresses with private variables for each thread.



Each thread has its own stack

- Some variables are created in the stack (local variables)
- A thread cannot know if a stack from other thread is active

Memory Access Coordination

The exchange of information among threads is performed by reading and writing on variables in the shared memory space.

Simultaneous access can produce a race condition

- Final result could be incorrect.
- Undeterministic nature.

Example: two threads want to increment variable `i`

Sequence with the correct result:

H0 loads `i` in a register: `0`
H0 increments register: `1`
H0 stores the value in `i`: `1`
H1 loads `i` in a register: `1`
H1 increments register: `2`
H1 stores the value in `i`: `2`

Sequence with incorrect result:

H0 loads `i` in a register: `0`
H1 loads `i` in a register: `0`
H0 increments register: `1`
H1 increments register: `1`
H0 stores the value in `i`: `1`
H1 stores the value in `i`: `1`

Mutual Exclusion and Synchronization

How race conditions can be addressed?

Atomic operations

- Force problematic operations to be performed atomically (without being interrupted).
- Special instructions of the processor: *test-and-set* or *compare-and-exchange* (CMPXCHG in Intel).

Critical Sections

- Code fragments with more than one instruction.
- Only one thread can execute the section simultaneously.
- It requires synchronization mechanisms: semaphores, etc.
- Risk of dead-locks.

Event-based Synchronization

- Barriers: all threads wait in the barrier for the last one to arrive.
- Ordered execution and others.

Section 2

Fundamentals of Parallel Algorithm Design

- Dependency Analysis
- Dependency Graph

Parallelization of Algorithms

Paralellizing an algorithm implies finding concurrent tasks (parts of the algorithm that can be run in parallel)

Almost always, there are dependencies between tasks

- When a task can only start when another one has finished

```
a = 0
FOR i=0 TO n-1
  a = a + x[i]
END
b = 0
FOR i=0 TO n-1
  b = b + y[i]
END
FOR i=0 TO n-1
  z[i] = x[i]/b + y[i]/a
END
FOR i=0 TO n-1
  y[i] = (a+b)*y[i]
END
```

Example:

- The first two loops are independent from each other
- The third loop uses the values of `a` and `b`, that are computed in the previous two loops

Data Dependencies

It is possible to determine if there exist dependencies between two tasks from the input/output data of each task

Bernstein Conditions:
Two tasks T_i and T_j (T_i precedes T_j sequentially) are independent if

1. $I_j \cap O_i = \varnothing$
2. $I_i \cap O_j = \varnothing$
3. $O_i \cap O_j = \varnothing$

I_i and O_i stand for the set of variables read and written by T_i

Dependency types:

- Flow dependencies (condition 1 is not fulfilled).
- Anti-dependency (condition 2 is not fulfilled).
- Output dependency (condition 3 is not fulfilled).

Data Dependencies: Examples

Flow dependency

```
double a=3,b=5,c,d;
c = T1(a,b);
d = T2(a,b,c);
```

T_2 cannot start until T_1 ends, since it reads variable c, that is written by T_1

Anti-dependency

```
// T1,T2 modify 3rd argument
double a[10],b[10],c[10],y;
T1(a,b,&y);
T2(b,c,a);
```

T_2 cannot start until T_1 ends, otherwise T_2 would overwrite the contents of a that is input to T_1

Output dependency

```
// T1,T2 modify 3rd argument
double a[10],b[10],c[10],x[5];
T1(a,b,x);
T2(c,b,x);
```

Both tasks modify array x

Data Dependencies in Loops

Sometimes data dependencies may be eliminated modifying the algorithm.

Code with flow dependency

```
for (i=1; i<n; i++) {
  b[i] = b[i] + a[i-1];
  a[i] = a[i] + c[i];
}
```

Iteration `i` modifies `a[i]` which is read in the iteration `i+1`.

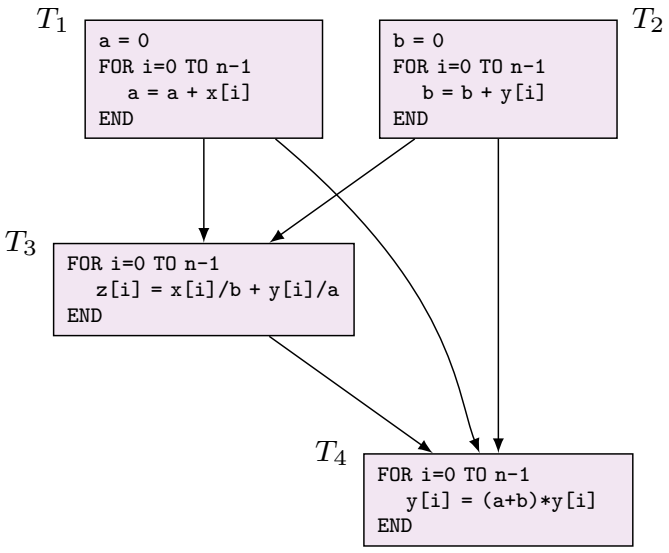
Removal of the dependency by reordering:

Code without dependencies

```
b[1] = b[1] + a[0];
for (i=1; i<n-1; i++) {
  a[i] = a[i] + c[i];
  b[i+1] = b[i+1] + a[i];
}
a[n-1] = a[n-1] + c[n-1];
```


Parallelization of Algorithms: Example

```
a = 0
FOR i=0 TO n-1
  a = a + x[i]
END
b = 0
FOR i=0 TO n-1
  b = b + y[i]
END
FOR i=0 TO n-1
  z[i] = x[i]/b + y[i]/a
END
FOR i=0 TO n-1
  y[i] = (a+b)*y[i]
END
```



Flow dependencies: $T_1 \rightarrow T_3$, $T_2 \rightarrow T_3$, $T_1 \rightarrow T_4$, $T_2 \rightarrow T_4$
Anti-dependency: $T_3 \rightarrow T_4$

Design of Parallel Algorithms: General Idea

Basically two phases:

1. Task decomposition
   - Requires a detailed analysis of the problem
     $\longrightarrow$ Task Dependency Graph
2. Task assignment
   - Which thread/process executes each task
   - Often implies agglomeration of several tasks

Usually there are several possible parallelization strategies

- Use one decomposition or another may have a great impact on performance
- We must try to maximize the degree of concurrency

Task Dependency Graphs

TDGs are an abstraction used to express the dependencies among the tasks and their relative execution order.

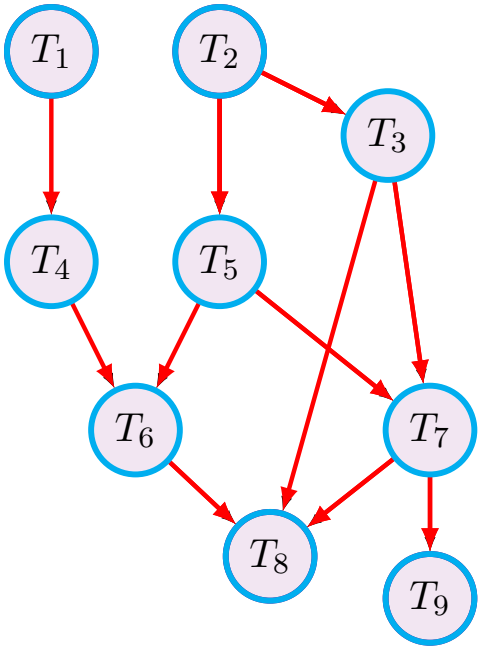
- It can be represented by using a Directed Acyclic Graph (DAG)
- Nodes denote the tasks (may have an associated cost)
- Edges define the dependencies among tasks

Definitions:

- Length of a path: sum of the costs c_i of each node contained in the path
- Critical path: longest path between the starting and final nodes
- Maximum concurrency degree: Larger number of tasks that can be concurrently executed.
- Average concurrency degree: $M = \sum_{i=1}^{N} \frac{c_i}{L}$
  (N = total nodes, L = length of the critical path)

Task Dependency Graphs: Example

Graph with $N = 9$ tasks (suppose all of them have cost $c_i = 1$)

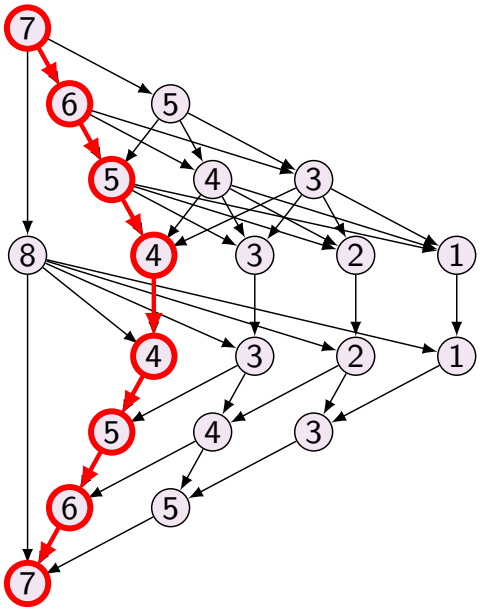


Initial nodes: T_1, T_2
Final nodes: T_8, T_9
Paths:
- $T_1 - T_4 - T_6 - T_8$ (length 4)
- $T_2 - T_5 - T_6 - T_8$ (length 4)
- $T_2 - T_5 - T_7 - T_8$ (length 4)
- $T_2 - T_3 - T_8$ (length 3)
- $T_2 - T_3 - T_7 - T_8$ (length 4)
- $T_2 - T_5 - T_7 - T_9$ (length 4)
- $T_2 - T_3 - T_7 - T_9$ (length 4)

Critical path: $L = 4$
Concurrency:
- Maximum degree: 3
- Average degree: $M = \sum_{i=1}^{9} \frac{1}{4} = 2.25$

Task Dependency Graphs: Example

Graph with $N = 21$ tasks (the cost c_i is indicated in each task)



Critical path
$L = 7 + 6 + 5 + 4 + 4 + 5 + 6 + 7 = 44$

$$M = \sum_{i=1}^{N} \frac{c_i}{L} = \frac{7 + 6 + 5 + 5 + \cdots}{44} = 2$$

Example of Task Decomposition

Given m polynomials

$$P_i(x) = a_{i,0} + a_{i,1}x + a_{i,2}x^2 + \cdots + a_{i,n}x^n, \;\; i = 0 : m-1$$

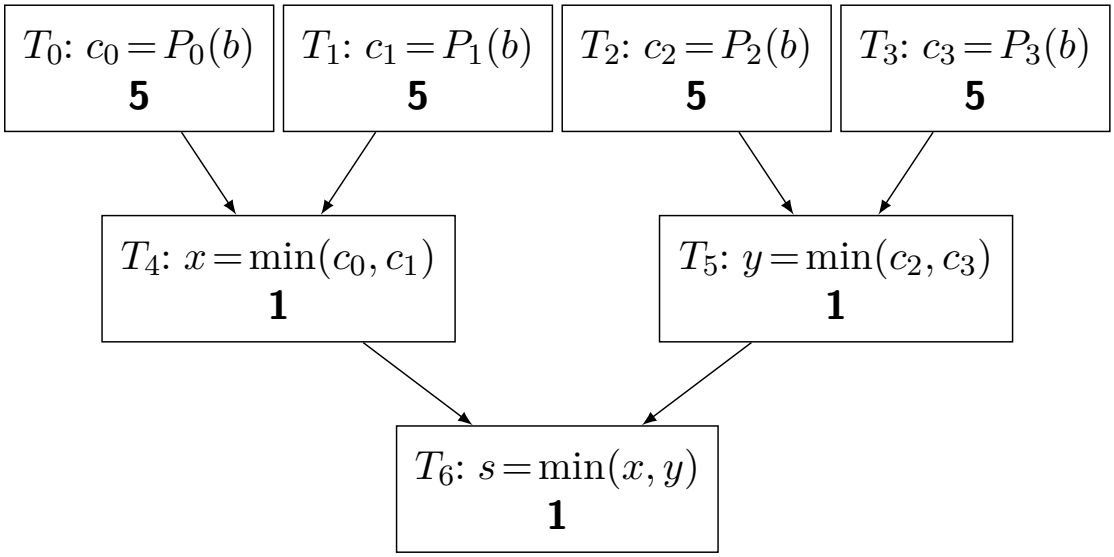
and a value b, compute

$$s = \min_{i=0:m-1} \{P_i(b)\},$$

Possible task decomposition:

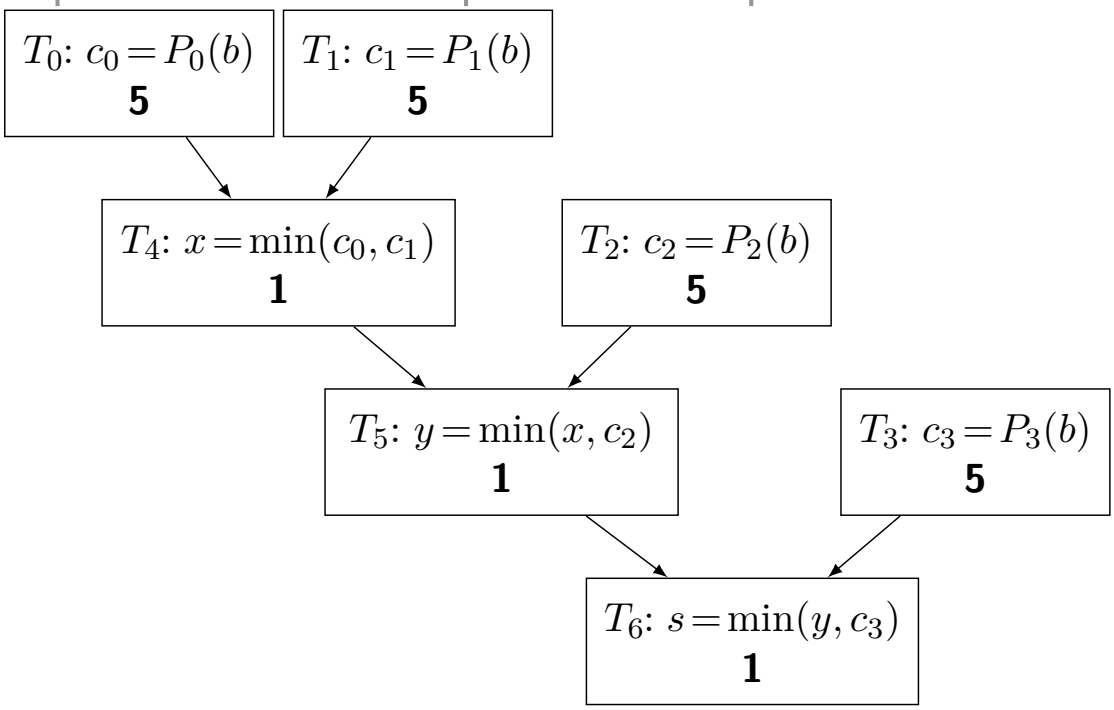
- One task per each polynomial evaluation
  $\rightarrow$ independent from each other
- Several tasks to compute minimum values two by two (recursively)

Example of Task Decomposition: Graph 1



$$L = 7, \quad M = \frac{5+5+5+5+1+1+1}{7} = 3.28$$

Example of Task Decomposition: Graph 2

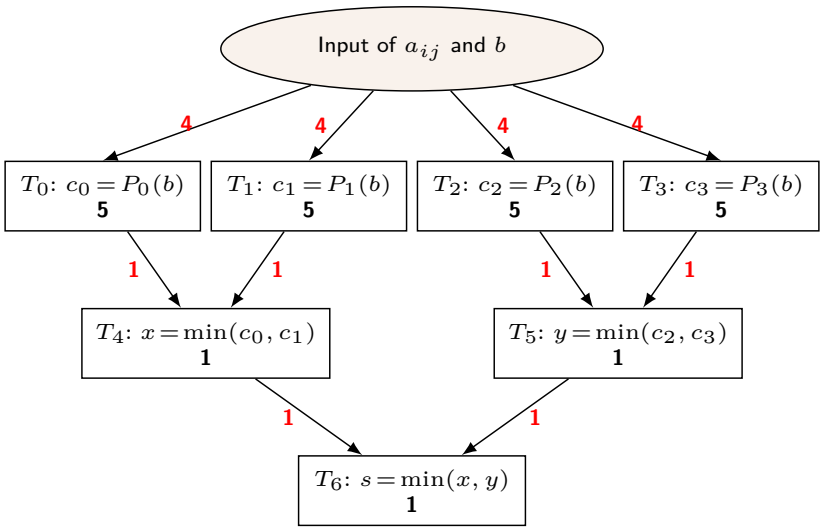


$$L = 8, \quad M = \frac{5+5+5+5+1+1+1}{8} = 2.875$$

Graph with Communication

Sometimes the graph incorporates information related to communication

- Possibility of adding auxiliary nodes (without cost)
- Edges with weight: denote the communication between tasks (value proportional to the amount of data)



Section 3

Performance Evaluation (I)

- Absolute Parameters
- Performance in Shared Memory

Performance Evaluation

The main objective of parallel computing is to increase performance.

- Is very important to know how the different parts of a parallel program behave.
- Is also important to know how they will behave when the number of processors and the size of the program change.

This section describes different measures and technics to detect where a parallel program reduces its performance and to compare it with sequential implementations and other configurations.

Analysis types

A priori Analysis

- It is performed on the pseudocode and the program design, before the implementation of a program.
- Independent of the machine where it is executed.
- It enables identifying the best approach to implement a parallel program.
- It enables determines the best size of the problem and the features of the hardware used.

A posteriori Analysis

- It is performed on a specific implementation and machine, and using a defined set of input data.
- It enables analysing bottlenecks and detecting unpredicted conditions during the design.

Theoretical Analysis

The cost is analyzed depending on the problem size: n

In many cases the cost depends only on n: $t(n)$.

However, sometimes given the same problem size, different behaviour may be observed depending on the input data.

- Cost of the most favourable case
- Cost of the less favourable case
- Average case
  Reasonable when the cost only depends on the problem size.

In practical terms, asymptotic values are defined (inferior and superior).

Concept of a FLOP

FLOP: *floating point operation* - measurement unit for:

- Cost of algorithms
- Performance of computers (flop/s)

1 flop = cost of an elemental floating point operation (product, sum, division, subtraction)

- The cost of the elemental integer operations are neglectable.
- The cost of other operations in floating point are evaluated depending on the FLOP unit.
  $\rightarrow$ for example, a square root may be equal to 8 flops.

The flop represents a machine-independent cost measurement unit (the time elapsed in a flop varies from one processor to another)

Asymptotic Notation

$\mathcal{O}$ Notation

- It defines an upper bound, except for constants and asitotically the shape of the function growing.
- In practical terms it is the highest order term of the cost expression without considering its coefficient.
  - Example: Matrix - vector product: $\mathcal{O}(n^2)$

o (o-small) Notation

- It also considers the coefficient of the highest-order term.
- Adecuado cuando comparamos dos algoritmos que tienen el mismo orden $\mathcal{O}$
  - Example: the product of a triangluar matrix by a vector can be performed with the algorithm for the full matrix with cost $o(2n^2)$ or an optimized algorithm with cost $o(n^2)$.

Parameters to evaluate the performance

Absolute parameters

- They enable knowing the real cost of parallel algoritms.
- The are the basis for the computation of relative parameters that are used to compare algorithms.
- They are the most important ones for real-time problems.

Relative parameters

- They enable comparing parallel algorithms among them and with respect to the sequential implementations.
- They provide information about the rate of usage of processors.

Absolute Parameters

- Execution time of a sequential algorithm: $t(n)$
- Execution time of a parallel algorithm: $t(n, p)$
  - Arithmetic time: $t_a(n, p)$
  - Communication time: $t_c(n, p)$
- Total Cost: $C(n, p)$
- *Overhead*: $t_o(n, p)$

Notation:

- When the problem size is always n, without ambiguity, it will be omitted, for instance: $t(p)$
- Sometimes we will use subindices instead of functions: t_p, C_p

Execution time

Time spent in the execution by the sequential algorithm (using only one processor, $t(n)$) or by the parallel algorithm (in p processors, $t(n,p)$)

- It only takes into account the number of floating point operations
- A priori cost is measured in FLOPs
- Experimentally the cost will be measured in seconds

Useful expressions for computing the cost:

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} i \approx \frac{n^2}{2} \qquad \sum_{i=1}^{n} i^2 \approx \frac{n^3}{3}$$

Example: Computational cost

```
For i=1 to n
  For j=1 to n
    x = x + a(i,j);
  end
end
```

$$t(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2 \text{ flops}$$

```
For i=1 to n
  For j=i to n
    x = x + a(i,j);
  end
end
```

$$t(n) = \sum_{i=1}^{n}\sum_{j=i}^{n} 1 \approx \sum_{i=1}^{n}(n-i) =$$
$$n^2 - \sum_{i=1}^{n} i \approx n^2 - \frac{n^2}{2} = \frac{n^2}{2} \text{ flops}$$

```
For i=1 to n
  For j=i to n
    For k=i to n
      x = x + a(i,k);
    end
  end
end
```

$$t(n) = \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{n} 1 \approx \sum_{i=1}^{n}\sum_{j=i}^{n}(n-i) \approx$$
$$\sum_{i=1}^{n}(n^2 - 2ni + i^2) = \sum_{i=1}^{n} n^2 - 2n\sum_{i=1}^{n} i +$$
$$\sum_{i=1}^{n} i^2 \approx n^3 - \frac{2n^3}{2} + \frac{n^3}{3} = \frac{n^3}{3} \text{ flops}$$

Total cost and *Overhead*

The execution of a parallel algorithm normally implies an extra time with respect to the sequential algorithm

The parallel total cost accounts for the total time employed by a parallel algorithm.

$$C(n,p) = p \cdot t(n,p)$$

The *overhead* indicates which is the added cost with respect to the sequential algorithm

$$t_o(n,p) = C(n,p) - t(n)$$

Speed-up and Efficiency

The *Speed-up* denotes the speed gaining of a parallel algorithm with respect its sequential version.

$$S(n,p) = \frac{t(n)}{t(n,p)}$$

The reference time $t(n) couldbe$:

- The best sequential algorithm at our knowledge
- The parallel algorithm using 1 processor

The Efficiency measures the degree of usage of the parallel units by an algorithm

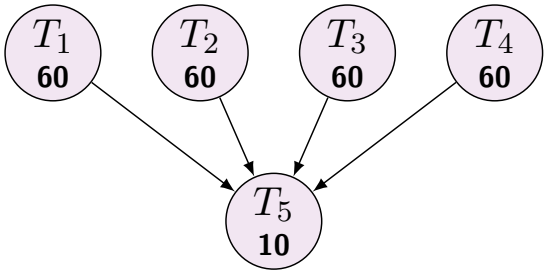
$$E(n,p) = \frac{S(n,p)}{p}$$

It is normally expressed as a percentage (either in the frame 0-100% or 0-1)

Example of Basic Performance Analysis

Consider this dependency graph

(in this example, the cost does not depend on n)



Assume that the sequential alg. does T_1, T_2, T_3, T_4, T_5

Sequential tiem: $t_1 = 60 + 60 + 60 + 60 + 10 = 250$

Parallel time for $p = 4$, where T_1, T_2, T_3, T_4 are executed concurrently: $t_p = 60 + 10 = 70$

Speedup and eficiencia:

$$S_p = \frac{t_1}{t_p} = \frac{250}{70} = 3.57 \qquad E_p = \frac{S_p}{p} = \frac{3.57}{4} = 0.89$$

What will be the speedup for $p = 2$, $p = 3$ and $p > 4$?

How to Obtain Good Performance

Ideally, for p processors we a *speedup* equal to p (efficiency equal to 1)

Which factors determine that we get more or less closer?

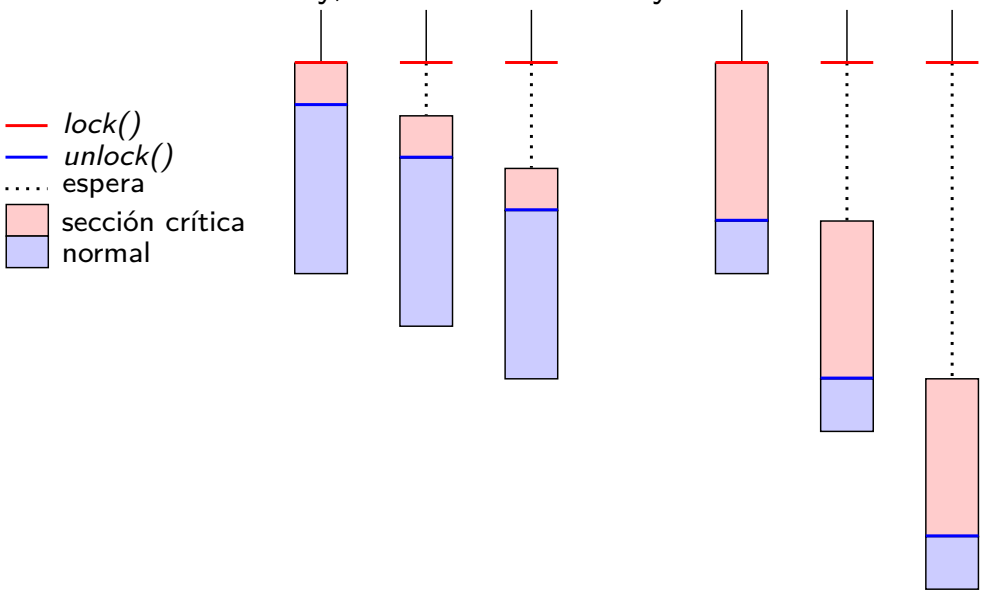
- Appropriate parallelization design
  - Well balanced load distribution
  - Minimize time in which processors are idle
  - Minimum possible *overhead*
- Specific aspects of the architecture where it runs
  - Different in shared memory or message passing
  - Data access time is not considered in the theoretical cost analysis, but it is very important in current architectures

Synchronization: Efficiency

Synchronization may have a negative impact in the performance and efficiency.

The critical section should be as smallest as possible

- In the contrary, a serialization may occur.



In the same way, barriers should be used only when necessary.

Section 4

Algorithm Design: Task Decomposition

- Domain Decomposition
- Other Decompositions

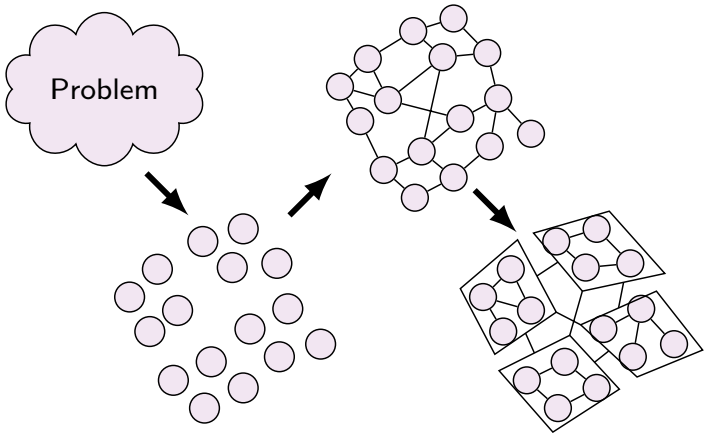
Parallel Algorithms Design

Parallel algorithms have a higher design complexity than sequential ones.

- Concurrency (communication and synchronization).
- Data and code allocation to processors.
- Concurrent access to shared data.
- Scalability for an increasing number of processors.

Main steps in the design are:

- Task decomposition.
- Task Assignment.



Task Decomposition

Task: Each one of the computation units defiend by the programmer which can be potentially be executed in parallel.

- The process of splitting computations in task is called decomposition

Granularity

- The decomposition can be fine-grained or coarse-grained
- Usually a fine grain decomposition is performed and then concurrent operations are grouped into coarser tasks.

Decomposition techniques

- Domain decomposition
- Functional decomposition directed by data flows
- Recursive decomposition
- Other: exploratory decomposition, speculative decomposition, mixed approaches

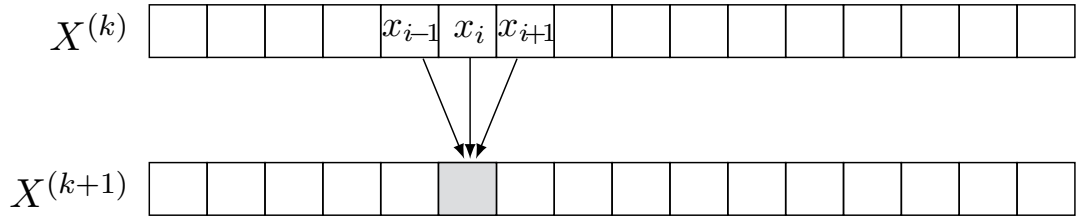
Domain Decomposition Techniques

- Data are split in chunks of similar size (sub-domains).
- A task is assigned to each domain, which will perform the needed operations on the sub-domain's data.
- Typically used when all sub-domains data require the same set of operations.
- These techniques are classified in:
  - Output data centred decompositions,
  - Input data centred decompositions,
  - Block-oriented decompositions (matrix algorithms).

Example of output data domain decomposition

An iterative algorithm that computes the succession of vectors $X^{(0)}, X^{(1)}, \cdots, X^{(k)}, X^{(k+1)}, \cdots, X^{(p)} \in \mathbb{R}^n$, where X^0 is a known vector and the rest of the vectors are obtained following the schema:

$$x_i^{(k+1)} = \frac{x_{i-1}^{(k)} - x_i^{(k)} + x_{i+1}^{(k)}}{2}, i = 0 : n-1$$
$$x_{-1}^{(k)} = x_{n-1}^{(k)}, x_n^{(k)} = x_0^{(k)}$$



Domain decomposition centered in the intermediate data

Example: Scalar product of two vectors

$$\left.\begin{array}{l} x = \left(\begin{array}{cccc} x_0 & x_1 & \cdots & x_{n-1} \end{array} \right) \\ y = \left(\begin{array}{cccc} y_0 & y_1 & \cdots & y_{n-1} \end{array} \right) \end{array}\right\} \Rightarrow x \cdot y = \sum_{i=0}^{n-1} x_i y_i$$

We assume t tasks and n multiple of t. Then the i-th task (i=0:t-1) will compute

$$\sum_{j=i\frac{n}{t}}^{(i+1)\frac{n}{t}-1} x_j y_j$$

and, for example, other task will receive and sum the computations performed by the previous tasks.

Functional decomposition directed by the data flow

- It is used when a problem can be split into phases.
- Each phase executes a different algorithm.
- Typically, it involves the next steps:
  1. Different phases are identified.
  2. A task is assigned to each phase.
  3. Data requirements for each task are analysed.
     - If the data overlapping among different tasks is minimum and the data flow among then is relatively small, decomposition will be complete and feasible.
     - If not, a different decomposition approach may be needed.

Recursive Decomposition

A method to obtain concurrency in problems that can be solved using the divide and conquer technique

1. Divide the original problem in two or more subproblems
2. In turn, these subproblems are divided in two or more subproblems, and so on until a base case is reached
3. Obtained data are appropriately combined to obtain the final result

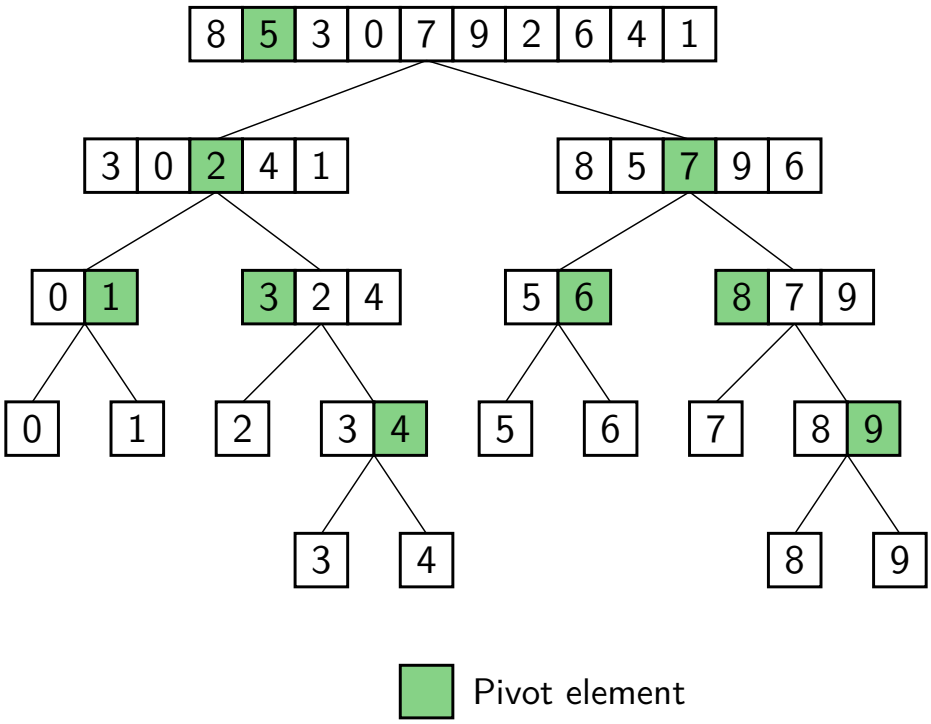
It can be implemented in differents forms:

- Replicated workers with bag of tasks
- Recursive algorithm

We will see these options in section Algorithmic Schemes

Recursive Decomposition

Example: Quicksort



Section 5

Algorithmic Schemes (I)

- Replicated Workers
- Divide and Conquer

Algorithmic Schemes

The Algorithmic schemes are commonly used parallelization approaches.

- A schema is used to solve a wide range of problems.
- A problem used to require several schemes.

Some schemes:

- Data Parallelism / data partitioning.
- Task Parallelism (master-slave, process farm, replicated workers)
- Tree and graph based schemes (divide and conquer)
- (*Pipelining*).
- Synchronous Parallelism.

Replicated Workers in a Bag of Tasks

Bag of tasks: shared data structure that contains the pending tasks

```
int get_next_task() {
    static int index = 0;
    int result;
    #pragma omp critical
    {   if (index==MAXIDX) result=-1;
        else { index++; result=index; }
    }
    return result;
}
...
    int myindex;
    #pragma omp parallel private(myindex)
    {   myindex = get_next_task();
        while (myindex>-1) {
            process_task(myindex);
            myindex = get_next_task();
        }
    }
```

In this example, a fixed number or tasks (`MAXIDX`) is defined.

Divide and Conquer

This method consists on solving a problem by splitting it into a series of similar sub-problems. Sub-problems can be recursively split until a base case is reached. Then it is solved directly and solutions are combined.

$\rightarrow$ Typically implemented in a recursive way (tree)

There are several types of tasks:

- Dividing the problem: it is performed in the inner nodes to create child nodes.
- Solving the base case: only in the leaves of the tree.
- Combining the results: performed in the inner nodes, they collapse the associated sub-tree.

Examples:

- *Quicksort* splitting stage has the largest cost.
- *Merge-sort* focuses work on combination.

Divide and Conquer: Example

Recursive solution with tasks: in each call two recursive calls are performed, each of them generating a new task

Parallel Mergesort

```
void mergesortpar(double *a, int n)
{
  int k;
  if (n<=nsmall)
    mergesortseq(a,n);
  else {
    k = n/2;
    Create two tasks:
      1. recursive call mergesortpar(a,k)
      2. recursive call mergesortpar(a+k,n-k)
    Wait for task finalization
    merge(a,k,n-k);
  }
}
```