CPA – Parallel Computing

Degree in Computer Science

T2. Shared Memory. Basic Parallel Algorithms Design

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

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Year 2024/25

Contents

Section 1

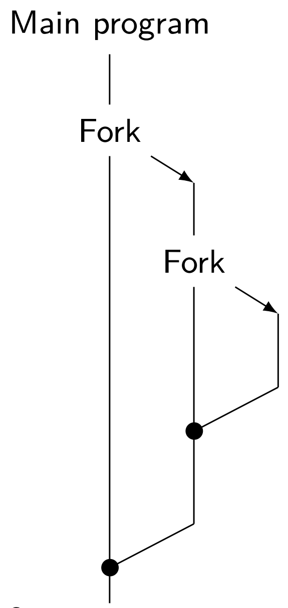
Shared Memory Model

- Model
- Details

Concurrent processes

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

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



This scheme can be implemented at the level of:

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

Shared Memory Model

Features:

- Tasks share a common memory-address space
- Programming quite similar to sequential case
  - Any data are accessible by all
  - No need to exchange data explicitly
- Drawbacks
  - Concurrent memory access may be problematic
    - Need to be coordinated: locks, monitors, ...
    - Unpredictable results if data access is not properly protected
  - Data locality is difficult to control (cache memories)

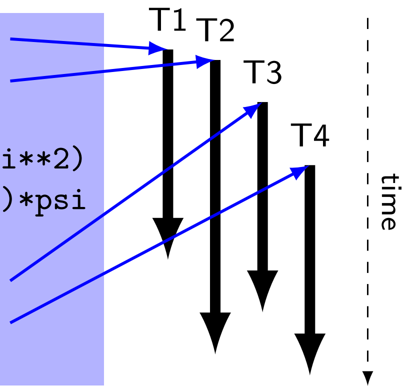
Thread Model

This model is closely related to the shared memory model

Thread: Independent instruction flow that can be scheduled for execution by the operating system

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

```
call subr1
call subr2
do i=1,n
  A(i)=fn(i**2)
  B(i)=A(i)*psi
end do
call subr3
call subr4
...
```



OpenMP

Portable standardization of threads

- Based on compiler directives
- Available in C/C++ and Fortran
- Portable/multi-plataform (Unix, Windows)
- Easy to use: Incremental parallelization

Some directives and functions

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

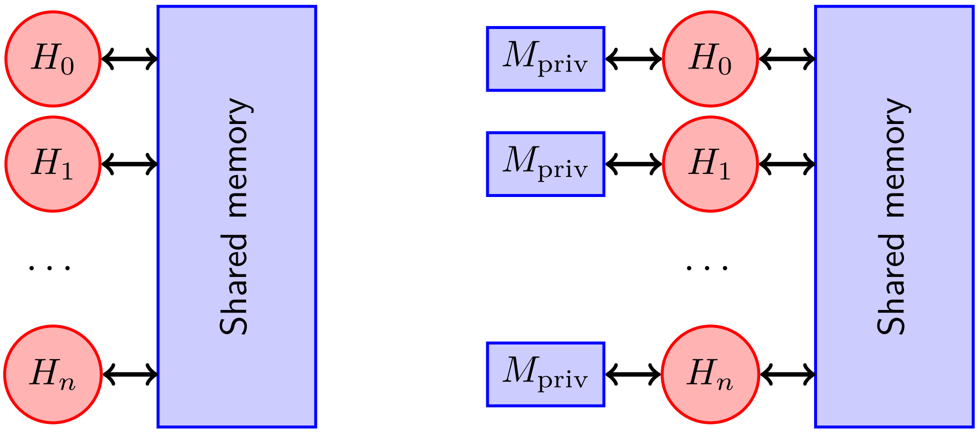
Creation and termination of threads is implicit in some directives

- The programmer does not bother about explicit *fork/join*

Memory Model with Threads

Simple model: single address space

More realistic model: single address space, 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 the another thread's stack 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
- Nondeterministic nature

Example: two threads want to increment variable `i`

Sequence with correct result:	Sequence with incorrect result:
H0 loads `i` in a register: 0	H0 loads `i` in a register: 0
H0 increments register: 1	H1 loads `i` in a register: 0
H0 stores the value in `i`: 1	H0 increments register: 1
H1 loads `i` in a register: 1	H1 increments register: 1
H1 increments register: 2	H0 stores the value in `i`: 1
H1 stores the value in `i`: 2	H1 stores the value in `i`: 1

Mutual Exclusion and Synchronization

How to solve race conditions?

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 deadlocks

Other type of synchronization

- Barrier: threads wait until all have reached a certain point
- Ordered execution

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

- A task can only start after 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`

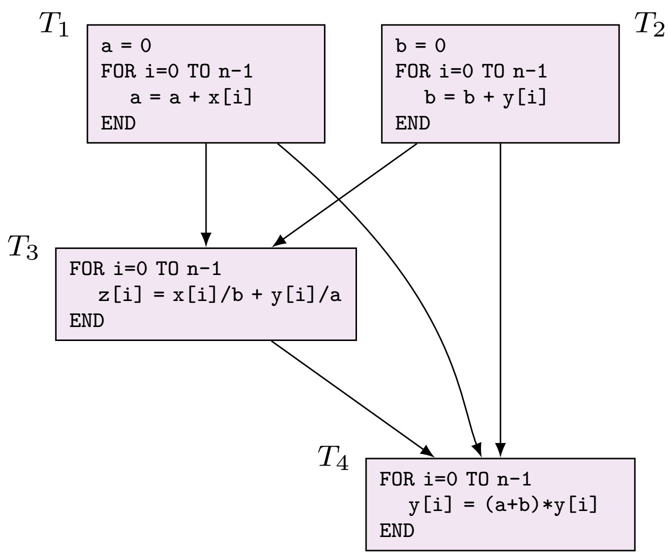
Removal of the dependency by *loop skewing*:

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: $T_1 \to T_3$, $T_2 \to T_3$, $T_1 \to T_4$, $T_2 \to T_4$
Anti-dependencies: $T_2 \to T_4$, $T_3 \to T_4$

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

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

Task Dependency Graph

It is an abstraction used to express the dependencies among the tasks and their relative execution order

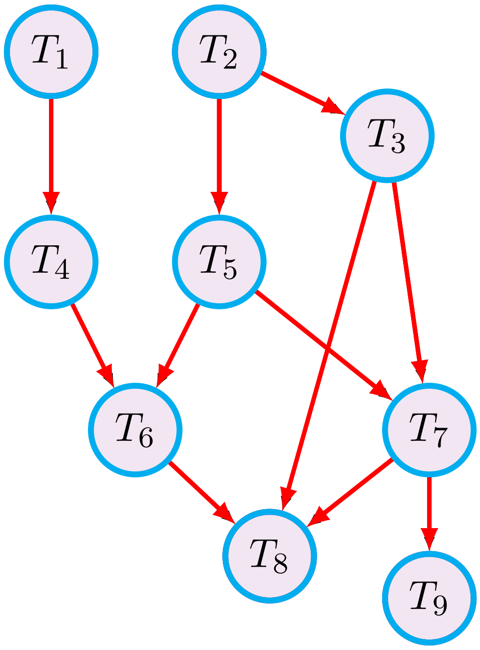
- It is a Directed Acyclic Graph (DAG)
- Nodes denote the tasks (may have an associated cost)
- Edges represent the dependencies among tasks

Definitions:

- Length of a path: sum of the costs c_i of each node contained in the path
- Critical path: longest path between a starting and a final node
- Maximum concurrency degree: largest number of tasks that can be executed concurrently
- 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: $L = 4$

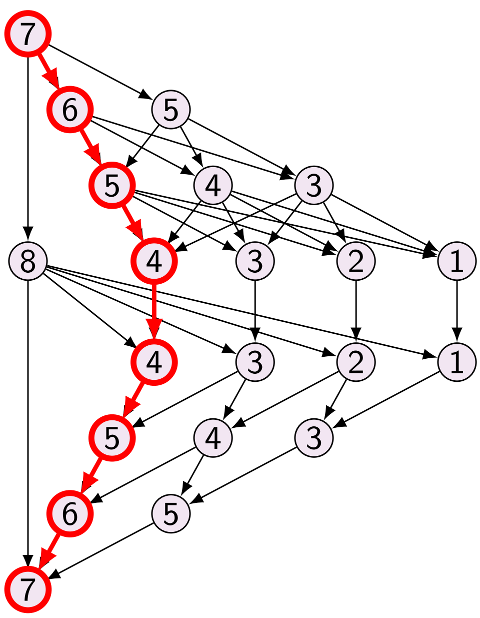
Concurrency:

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

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

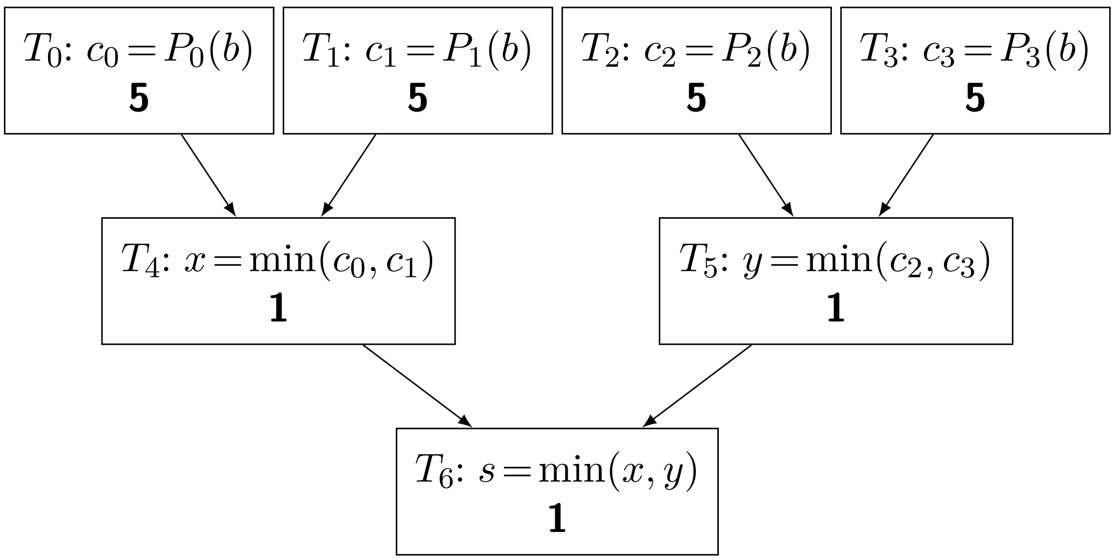
and a value b, compute

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

Possible task decomposition:

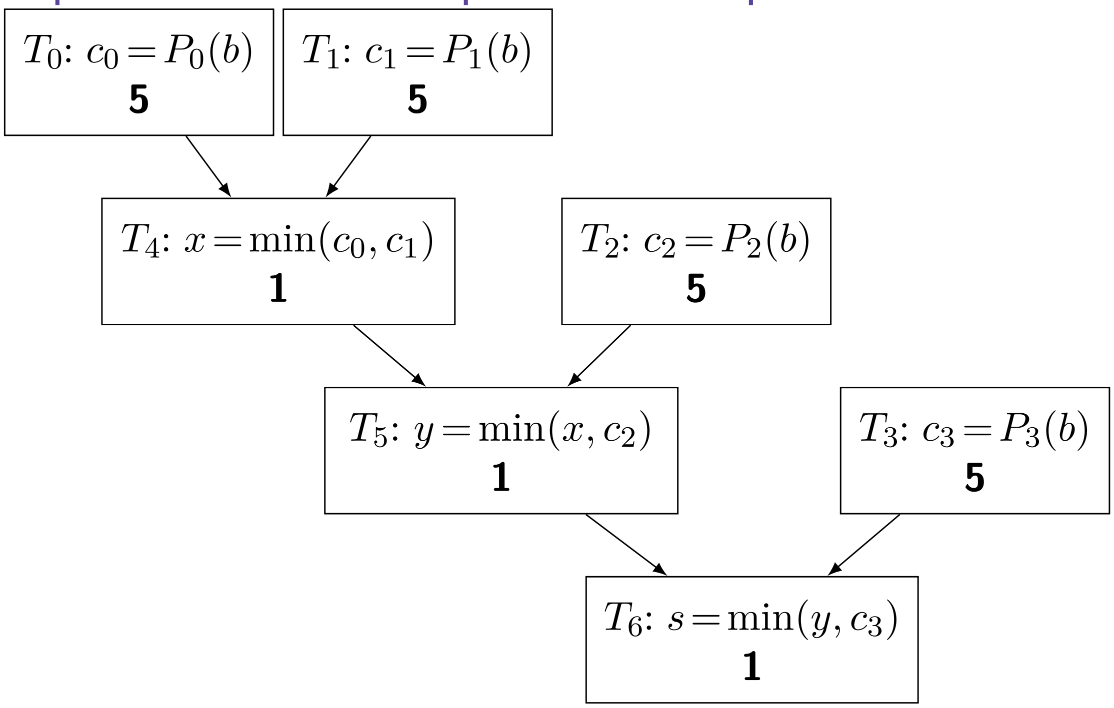
- One task per each polynomial evaluation
  $\rightarrow$ independent from each other
- Several tasks to compute minimum values two by two (recursively)

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

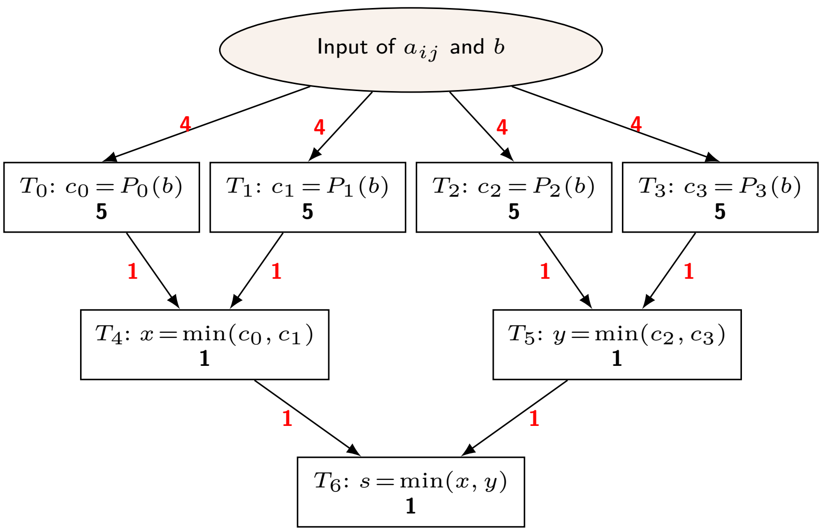


$$L = 8, \quad M = \frac{5+5+5+5+1+1+1}{8} = 2.875$$

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
- Allows to identify the best approach to implement a parallel program
- Allows to determine 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
- Allows to analyze bottlenecks and detect conditions not foreseen during the design

Theoretical Analysis

The cost is analyzed in terms of the problem size: n

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

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

- Cost of the best case
- Cost of the worst case
- Average cost
  By averaging the times of each of the possible inputs weighted by the probability of their appearance

In practice, asymptotic bounds are used (lower and upper)

Concept of 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 integer operations is considered negligible
- The cost of other operations in floating point is expressed in terms of the Flop unit
  → 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

Big O notation, $\mathcal{O}$

- Defines an (asymptotic) upper bound for the growth of a function, disregarding constants
- In practice, it is the highest-order term of the cost expression without considering its coefficient
  - Example: the cost of the matrix-vector product is $\mathcal{O}(n^2)$

Small O notation, o

- Also takes into account the coefficient of the highest-order term
- Appropriate to compare two algorithms of the same $\mathcal{O}$ order
  - Example: the product of a triangluar matrix by a vector can be performed with the conventional algorithm with cost $o(2n^2)$ or an optimized algorithm with cost $o(n^2)$

Parameters to Evaluate the Performance

Absolute parameters

- Allow us to know the real cost of parallel algoritms
- They 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

- Allow us to compare parallel algorithms among them and with respect to the sequential implementation
- They provide information about the degree of utilization 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, $t(n,p)$)

- The a priori cost is measured in flops
  - We will take into account only the number of floating point operations
- Experimentally the cost will be measured in seconds

Useful expressions for computing the cost:

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

Computational Cost: Examples

```
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 + 3.0*a[i,j]
  END
END
```

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

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

```
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

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

Speedup and Efficiency

The speedup indicates the speed gain of a parallel algorithm with respect to its sequential version

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

The reference time $t(n)$ could be:

- The best sequential algorithm
- The parallel algorithm run on 1 processor

The efficiency measures the degree of utilization of the parallel computer by the parallel algorithm

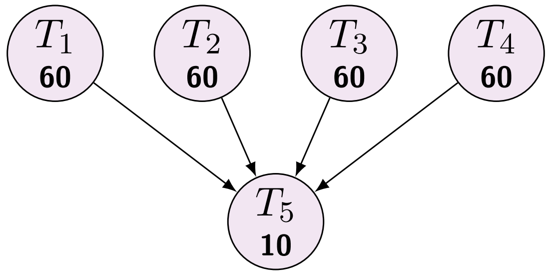
$$E(n,p) = \frac{S(n,p)}{p}$$

Usually expressed as a percentage (or parts per unit)

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 time: $t_1 = 60 + 60 + 60 + 60 + 10 = 250$

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

Speedup and efficiency:

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

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

How to Obtain Good Performance

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

Which factors determine that we get more or less closer?

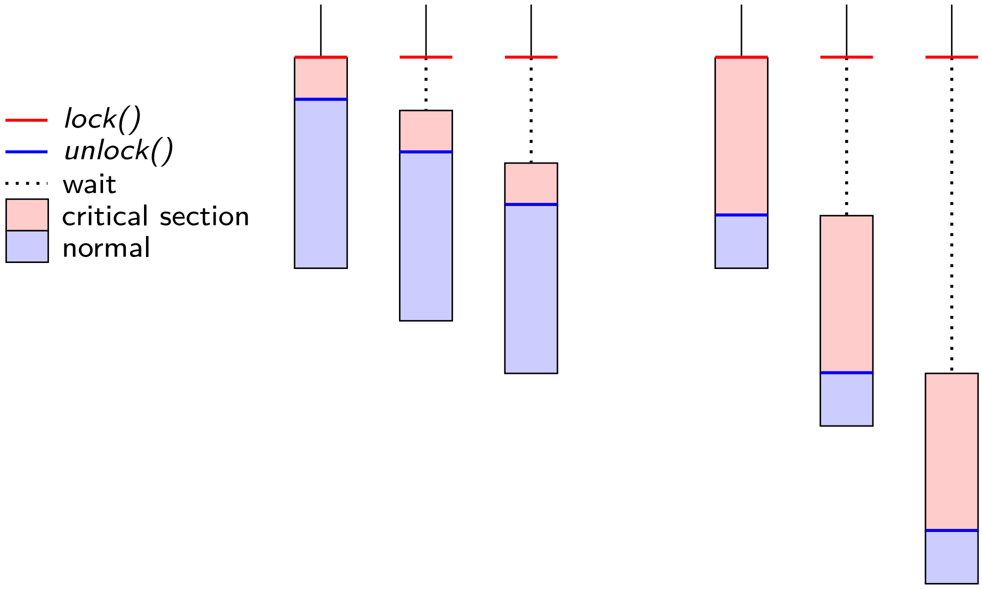
- Appropriate parallelization design
  - Well balanced load distribution
  - Minimize time in which processors are idle
  - Minimum possible overhead
- Specific aspects of the architecture where it runs
  - Different in shared memory or message passing
  - Data access time is not considered in the theoretical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

Synchronization: Efficiency

Synchronization may have a negative impact on efficiency

The critical section should be as small as possible

- Otherwise a "serialization" occurs



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

Section 4

Algorithm Design: Task Decomposition

- Domain Decomposition
- Other Decompositions

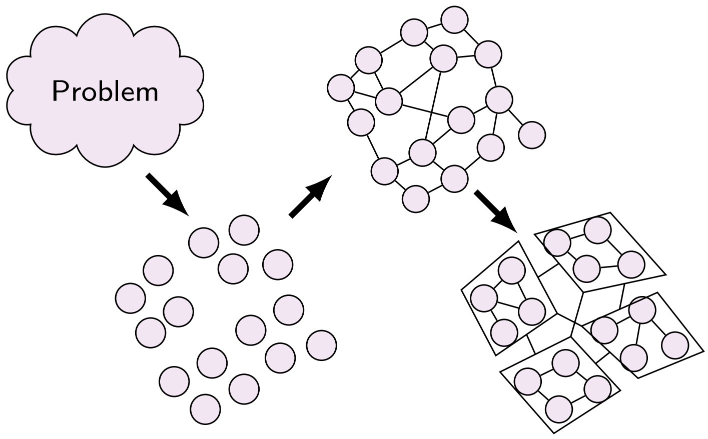
Parallel Algorithms Design

Parallel algorithms have a higher design complexity than sequential ones

- Concurrency (implies communication and synchronization)
- Assignment of data and code 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 of the computation units defined by the programmer that can potentially be executed in parallel

- The process of splitting a computation/program in tasks is called decomposition

Granularity

- The decomposition can be fine-grained or coarse-grained
- Usually a fine-grained decomposition is performed and later tasks are grouped together into coarser tasks

Decomposition Techniques

- Domain decomposition
- Functional decomposition driven by data flow
- Recursive decomposition
- Other: exploratory decomposition, speculative decomposition, mixed approaches

Domain Decomposition

In case of large, regular data structures

- Data are split in chunks of similar size (sub-domains)
- A task is assigned to each sub-domain, which will perform the required operations on the sub-domain's data

Typically used when it is possible to apply the same set of operations on the data of every sub-domain

The descomposition can be:

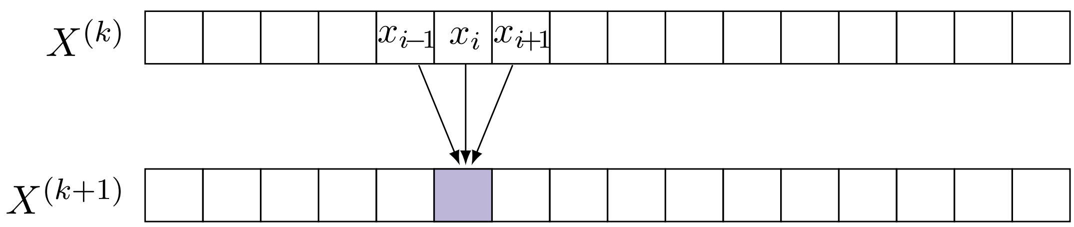
- Centered on output data
- Centered on input data
- Centered on intermediate data
- Block-oriented decompositions (matrix algorithms)

Domain Decomposition Centered on Output Data

Each component of the output data can be computed independently from the rest

Example: design an iterative parallel algorithm to compute a sequence of vectors $X^{(0)}, X^{(1)}, \ldots, X^{(k)}, X^{(k+1)}, \ldots \in \mathbb{R}^n$, where $X^{(0)}$ is a known vector and the rest are obtained as:

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



Domain Decomposition Centered on Input Data

Example: Scalar product of two vectors

$$\left.\begin{array}{l} x = [x_0, x_1, \ldots, x_{n-1}] \\ y = [y_0, y_1, \ldots, y_{n-1}] \end{array}\right\} \Rightarrow \quad x \cdot y = \sum_{i=0}^{n-1} x_i y_i$$

Assuming p tasks and n multiple of p, then the ith task $(i = 0, \ldots, p-1)$ would compute

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

Finally, there would be additional tasks to accumulate partial sums into the global sum

Functional Decomposition

The functional decomposition driven by data flow is used when

- The resolution of the problem can be split into phases
- Each phase executes a different algorithm

Typically, it involves the next steps:

1. The different phases are identified
2. A task is assigned to each phase
3. Data requirements for each task are analyzed
   - If the data overlapping among different tasks is minimum and the data flow among them is relatively small, the decomposition will be complete and feasible
   - Otherwise, 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. Resulting data are appropriately combined to obtain the final result

It can be implemented in differents forms:

- Replicated workers with a task pool
- Recursive algorithm

Recursive Decomposition

Example: Quicksort