CPA – Parallel Computing

Informatics Engineering Degree

T3. Message Passing. Advanced 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

Content

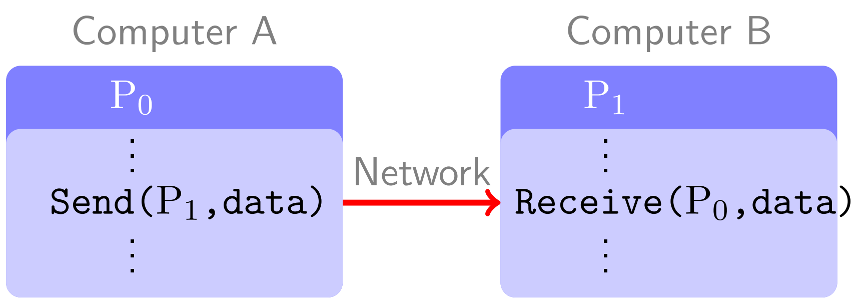
Section 1

Message Passing Model

- Model
- Details

Message Passing Model

- Tasks manage their own private memory space.
- Data are exchanged through messages.
- Communication normally requires coordinated operations (e.g. sending and receiving).
- Complex and costly programming, but total control of the parallelisation.



MPI: Message Passing Interface

Process Creation

The parallel program comprises several processes

- Parallel processes are normally related to O.S. processes.
- Typically one process per processor.
- Each one has an index or identifier (integer number).

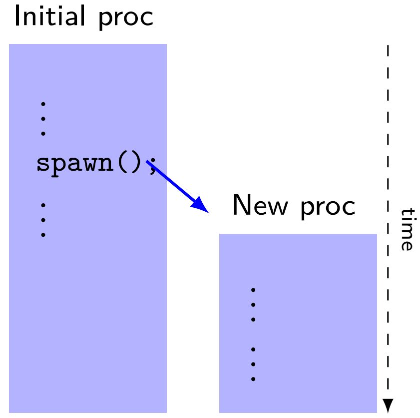
Process creation can be:

Static: At the startup of the program

- Details defined in the command line (`mpiexec`).
- Alive during the whole execution.
- Most common approach.

Dynamic: During the execution

- API `spawn()`.



Communicators

Processes are organized into groups.

- Key in collective operations, such as broadcast communication (1 to all).
- Defined by using indexes or operations on sets (union, intersection, etc.).

More general concept: Communicator = group + context

- The communication in a communicator cannot interfere with the communications taking place in other..
- Useful for isolating the communication within a library.
- Communicators are defined from other groups or communicators.
- Predefined communicators:
  - World (*world*): Comprising all processes created by `mpiexec`.
  - Self (*self*): Formed by a single processor.

Basic Send/Receive Operations

The most common operations are point to point communications.

- One process sends a message (`send`) and other receives it (`recv`).
- Each `send` needs a corresponding `recv` .
- The message includes the content of one or more variables.

```
/* Process 0 */
x = 10;
send(x,1);
x = 0;
```

```
/* Process 1 */
recv(y,0);
```

Operation `send` is semantically safe if it is guaranteed that process 1 receives the value that `x` had before the send operation was performed (`10`).

There are different modalities of send and receive.

Example: Vector Sum

$x = v + w$, $v \in \mathbb{R}^n$, $w \in \mathbb{R}^n$, $x \in \mathbb{R}^n$

- We assume $p = n$ processes
- Initially `v`, `w` are stored in P_0, and the result `x` must also be stored in P_0

```
function sum(v, w, x, n)
distribute(v, w, vl, wl, n)
parsum(v, w, vl, wl, x, xl, n)
combine(x, xl, n)
```

```
function distribute(v, w, vl, wl, n)
foreach P(i), i=0 to n-1
  if i == 0
    for j=1 to n-1
      send(v[j],j)
      send(w[j],j)
    end
  else
    recv(vl,0)
    recv(wl,0)
  end
```

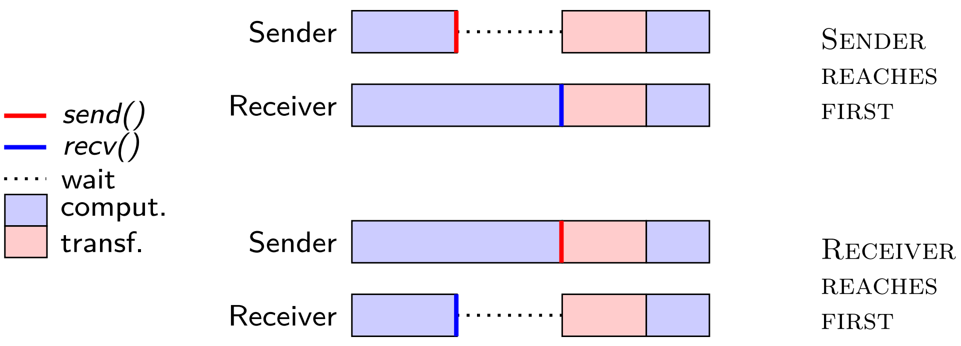
```
function parsum(v, w, vl, wl, x, xl, n)
foreach P(i), i=0 to n-1
  if i == 0
    x[0] = v[0] + w[0];
  else
    xl = vl + wl
  end
```

```
function combine(x, xl, n)
foreach P(i), i=0 to n-1
  if i == 0
    for j=1 to n-1
      recv(x[j],j)
    end
  else
    send(xl,0)
  end
```

Sending with synchronization

In the synchronous mode, the `send` operation does not conclude until the other process has posted the matching `recv`

- Along with the data transfer, process get synchronized.
- It requieres a protocol to create the communication context (this is transparent to the programmer).



Sending/Receiving Modalities

Buffered send/synchronous send

- A *buffer* stores a temporary copy of the message
- The buffered `send` finishes when the message has been copied from the program memory to a system buffer
- The synchronous `send` does not finish until a matching `recv` has been done in the other process

Blocking/non-blocking operations

- When a call to a blocking `send` returns it is safe to modify the send buffer
- When a call to a blocking `recv` returns it is guaranteed that the buffer contains the message
- Non-blocking calls simply initiate the operation

Operation ending

Non-blocking operations need an ending criteria

- In `recv` in order to start reading the message.
- In `send` in order to start overwriting the variable.

Non-blocking `send` and `recv` provide an operation number (*req*).

Primitives:

- `wait(req)` the process gets blocked until the operation *req* is finished.
- `test(req)` indicates if the *req* operation has finished or not.
- `waitany` and `waitall` can be used when several operations are pending.

It can be used to overlap communications and computing.

Selection of Messages

The `recv` operations requires an identifier of the process (*id*).

- It does not finsh until a message from *id* is received.
- Messages from other processes are ignored

For more flexibility, an “accept-any” code is provided to enable receiving from any process.

Moreover, a label (*tag*) is used to differentiate among messages

- An ”accept-any” code is also possible to match any tag.

Example: `recv(z,any_src,any_tag,status)` it will accept the first message received

- Primitive `recv` has a `status` argument where the receiver and the label are filled-in.
- Messsages not matching a receive operation are not lost, they remain in a ”message queue”.

Problem: Dead-lock

An incorrect usage of `send` and `recv` can lead to dead-locks.

Synchronous communication case:

```
/* Process 0 */
send(x,1);
recv(y,1);
```

```
/* Process 1 */
send(y,0);
recv(x,0);
```

- Both get blocked in the sending

Buffer-based communication:

- Previous example will not cause deadlocks.
- There may be other situations leading dead-locks.

```
/* Process 0 */
recv(y,1);
send(x,1);
```

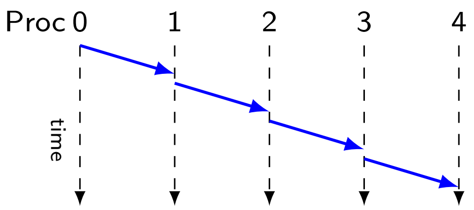
```
/* Process 1 */
recv(x,0);
send(y,0);
```

Potential solution: exchange the order from one of them.

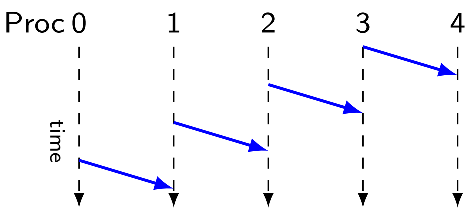
Problem: Serialization

Each process hast to send a data to its right neighbour

```
/* Process i */
recv(y,i-1);
send(x,i+1);
```




```
/* Process i */
send(x,i+1);
recv(y,i-1);
```



Potential solutions:

- Even-odd protocol: Even and odd processes make different operations
- Non-blocking `send` or `recv`.
- Combined operations: `sendrecv`

Collective communication

Collective operations involve all the processes from a communicator (in most of the cases, one has a special role) – root process).

- Synchronization (`barrier`): each process wait for the rest to come.
- Data transfer: one or several processes send to one or several ones.
- Reductions: along with the communication an operation is performed on them.

These operations can be realized using point to point communication, but is recommendable to use the proper primitive.

- There are several algorithms for each case (linear, tree).
- The optimal solution often depends on the architecture (network topology).

Collective Communication: Types

- One to all broadcast
  - All processes receive what the root process send.
- Reduction all to one
  - Symmetric operation to broadcast.
  - Data are combined using an associative operator.
- *Scatter*
  - Root splits the data into different processes, each one receiving a different part.
- Collection (*gather*)
  - Symmetric operation to scatter.
  - Results from the different processes are piled-up in the root.
- All to all broadcast
  - p concurrent proadcasts, with different root processes.
  - At the end, all processes will have received all the data.
- All to all reduction
  - Reduction operation in which all processes receive the final result.

Section 2

Algorithmic Schemes (II)

- Data Parallelism
- Tree Schemes
- Other Schemes

Data Parallelism / Data Partitioning

In algorithms with many data treated in a similar way (typically, matrix algorithms).

- In shared memory, loops are parallelized (each thread works on a part of the data)
- In message passing, an explicit data partitioning is performed.

In message passing it may be inconvenient to parallelize if:

- the computational volume is not at least one order of magnitude higher than the communication cost
  - ✗ Vector-vector: cost $\mathcal{O}(n)$ with respect to $\mathcal{O}(n)$ communication
  - ✗ Matrix-vector: cost $\mathcal{O}(n^2)$ with respect to $\mathcal{O}(n^2)$ communication
  - ✓ Matrix-matrix: cost $\mathcal{O}(n^3)$ with respect to $\mathcal{O}(n^2)$ communication
- Except if data are already distributed.

Case 1: Matrix-vector product

Message-passing solution with ($p = n$ processors)

- Assuming that `v`, `A` are initially in P_0
- The result `x` should be stored in P_0

```
SUB matvec(A,v,x,n,m)
distribute(A,Al,v,n,m)
mvlocal(Al,v,xl,n,m)
combine(xl,x,n)
```

```
SUB distribute(A,Al,v,n,m)
foreach P(i), i=0 to n-1
  if i == 0
    for j=1 to n-1
      send(A[j,:],j)
      send(v[:],j)
    end
    Al = A[0,:]
  else
    recv(Al,0)
    recv(v[:],0)
  end
```

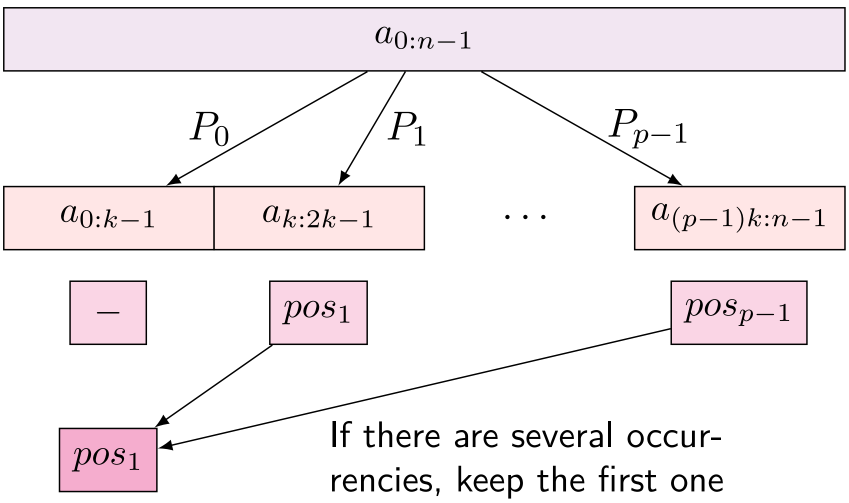
```
SUB mvlocal(Al,v,xl,n,m)
foreach P(i), i=0 to n-1
  xl = 0
  for j=0 to m-1
    xl = xl + Al[j] * v[j]
  end
```

```
SUB combine(xl,x,n)
foreach P(i), i=0 to n-1
  if i == 0
    x[0] = xl
    for j=1 to n-1
      recv(x[j],j)
    end
  else
    send(xl,0)
  end
```


Case 2: Lineal searching

Given a vector $a \in \mathbb{R}^n$ and a number $x \in \mathbb{R}$, find the index i that $x = a_i$ (there could be more than one occurence).

Assuming $n = k \cdot p$, each processor will search in a sub-vector of k elements



Case 2: Lineal searching - Pseudocode

Message-passing Parallel Solution.

- Initially x, a are in P_0, the resul

```
function searching(a, x, pos, n, p)
distribute(a, apr, x, n, p)
searchlocal(apr, x, pos, n, p)
combine(pos, n, p)
```

```
function distribute(a, apr, x, n, p)
foreach P(i), i=0 to p-1
  /* collective operations */
  /* Process 0 sends a part of a (n/p)
     elements, received in apr */
  split(a,n,apr,n/p,p,0)
  /* Process 0 sends x to all, all
     receive in x0 */
  broadcast(x,0)
```

```
function searchloc(apr, x, pos, n, p)
foreach P(pr), pr=0 to p-1
  pos = n;  i = 0
  while (i<n/p) AND (pos==n)
    if apr[i] == x
      pos = i
    end
    i = i+1
  end
```

```
fubction combibe(pos,n,p)
foreach P(pr), pr=0 to p-1
  if pr == 0
    for i=1 to p-1
      recv(aux,i)
      if aux+(n/p*i)<pos
        pos = aux+(n/p*i)
      end
    end
  else
    send(pos,0)
  end
```


Case 3: Sum of the elements of a vector

Message-Passing Parallel Solution

```
function sum(v, s, n, p)
distribute(v, vloc, n, p)
parsum(vloc, sl, n, p)
reduce(sl, s, p)
```

```
function distribute(v, vloc, n, p)
foreach P(i), i=0 to p-1
  k = n/p
  if i == 0
    for j=1 to p-1
      send(v[j*k:(j+1)*k-1],j)
    end
    vloc = v[0:k-1]
  else
    recv(vloc[0:k-1],0)
  end
```

```
function sumapar(vloc, sl, n, p)
foreach P(i), i=0 to p-1
  sl = 0
  for j=0 to n/p-1
    sl = sl + vloc[j]
  end
```

```
function reduce(sl, s, p)
foreach P(i), i=0 to p-1
  if i == 0
    s = sl
    for j=1 to p-1
      recv(saux,j)
      s = s + saux
    end
  else
    send(sl,0)
  end
```

There are more efficient ways to implement the reduction: *recursive doubling*

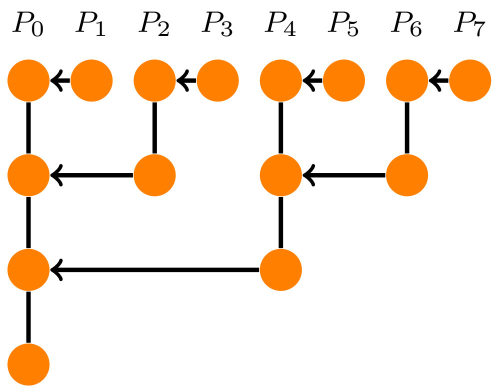
Tree Schemes

Reduction using Recursive Doubling:

There are $\log_2(p)$ communicaion stages.

- The number of processors participating are divided by two after each stage.
- Once a process receives a data, it accumulates its value in the local sum s.

```
foreach P(pr), pr=0 to p-1
  s = sl;
  for j=1 to log2(p)
    if remainder(pr,2^j)==0
      recv(aux)
      s = s + aux
    else
      if remainder(pr,2^(j-1))==0
        send(s,pr-2^(j-1))
      end
    end
  end
```



Task parallelism

Sometimes task-based approaches create more tasks than processors, or when a task spawns new tasks.

- Static allocation of tasks is not feasible or leads to load um-balancing.
- Dynamic allocation: tasks are being allocated to processors as they become idle.

Usually implemented by means of an asymmetric schema: master-slave

- Master process manages the tasks already performed or pending.
- Workers receive tasks and notify master process when they have finished them.

Sometimes, a symmetric solution is feasible: replicated workers.

Master and workers

Example: Fractals with message passing (np processes)

Master

```
count=0; row=0;
for (k=1; k<np; k++) {
  send(row, k, data_tag);
  count++; row++;
}
do {
  recv({r,color}, slave, res_tag);
  count--;
  if (row<max_row) {
    send(row, slave, data_tag);
    count++; row++;
  }
  else
    send(row, slave, end_tag);
  display(r,color);
} while (count>0);
```

Workers

```
recv(y, master, src_tag);

while(src_tag == data_tag ) {
  /*
   * compute row colors
   */
  send( {y,color}, master,
       res_tag);
  recv(y, master, src_tag);
}
```

count stands for the number of processes that have a task assigned

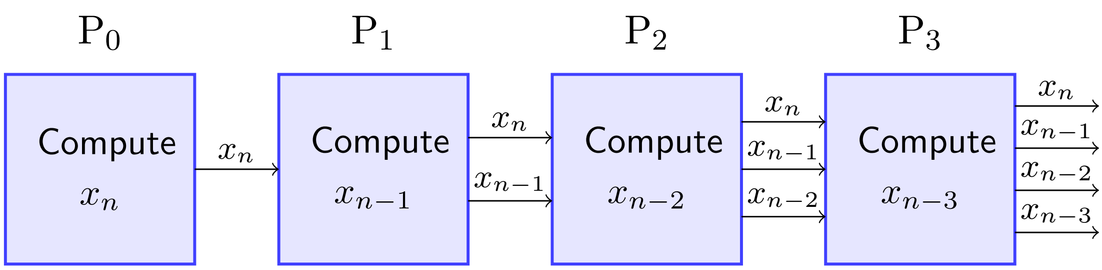
The problem requires processing `max_row` independent lines from the image.

Pipelines parallelism

Each process performs a partial processing and forwards the result to the following process

Example: Solving a triangular system of equations

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{ij} x_j}{a_{ii}}$$



An efficient implementation overlaps the computation of x_i with the sending of $x_{i+1}, \ldots, x_n$

A cyclic distribution may be convenient.

Section 3

Performance Evaluation (II)

- Parallel Time
- Relative Parameters

Parallel Execution Time

Time spent by a parallel algorithm with p processors

- From the start of the first one until the last finishes

It is composed of arithmetic and communication time

$$t(n,p) = t_a(n,p) + t_c(n,p)$$

t_a corresponds to all computing times

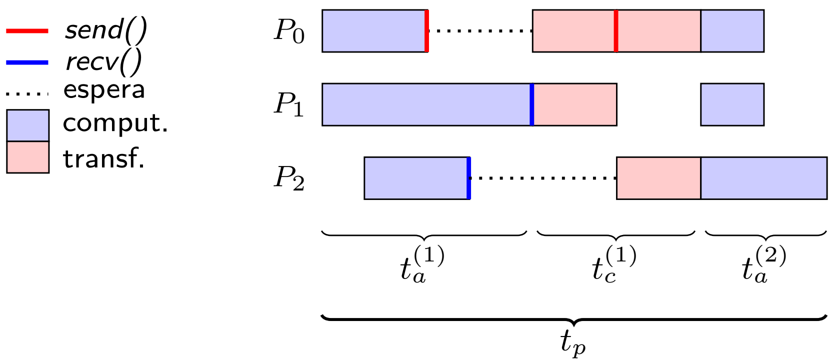
- All processes compute concurrently
- It is equal or higher than the maximum arithmetic time

t_c corresponds to times associated to data transfers

- In distributed memory t_c=time of sending the messages
- In shared memory t_c=synchronization time

Parallel Execution Time: Components

Ex.: message passing with three processes, P_0 sends to P_1 and P_2



In practical terms

- There is no clear splitting between computation and communication stages (P_1 does not need to wait).
- Sometimes communication and computation can be overlapped using non blocking operations, such in the case of P_2.

$$t_p = t_a + t_c - t_{\text{overlap}}$$

t_{overlap}: tiempo de solapamiento

Modelling Communication Time

Assuming message passing and P_0 and P_1 running on two different nodes with direct communication.

The time needed to send a message of n bytes: $t_s + t_w n$

- set-up communication time, t_s
- Bandwidth, w (Maximum number of bytes per second.)
- Sending time for 1 byte, $t_w = 1/w$

In practical terms, it is much more complex

- Switched networks, non-uniform latencies, collisions, ...

Recommendations:

- Grouping serveral messages into one (n big, single t_s)
- Avoiding many simultaneous communications.

In shared memory, considerations are different.

Example: Matriz-vector product (1)

$x = A \cdot v$, $A \in \mathbb{R}^{n \times n}$, $v \in \mathbb{R}^n$, $x \in \mathbb{R}^n$

Sequential time:

$t(n) = 2n^2$ flops

Parallelisation using $p = n$ processors

Parallel time in shared-memory:

$t(n, p) = 2n$ flops

Parallel time in distributed-memory:

- `distributing`: $2 \cdot (n-1) \cdot (t_s + t_w \cdot n)$
- `mvlocal`: $2n$ flops
- `combine`: $(n-1) \cdot (t_s + t_w \cdot 1)$

$t(n, p) = 3 \cdot (n-1) \cdot t_s + (n-1) \cdot (2n+1)t_w + 2n$ flops

$\boxed{t(n, p) \approx 3nt_s + 2n^2 t_w + 2n \text{ flops}}$

Example: Matriz-vector product (2)

Version for $p < n$ proc. (Row-wise distribution)

```
SUB matvec(a,v,x,n,p)
distribute(a,aloc,v,n,p)
mvlocal(aloc,v,x,n,p)
combine(x,n,p)
```

```
SUB distribute(a,aloc,v,n,p)
forall P(i), i=0 to p-1
  nb = n/p
  if i == 0
    aloc = a[0:nb-1,:]
    for j=1 to p-1
      send(a[j*nb:(j+1)*nb-1,:],j)
      send(v[:],j)
    end
  else
    recv(aloc,0)
    recv(v,0)
  end
```

```
SUB mvlocal(aloc,v,x,n,p)
foreach P(pr), pr=0 to p-1
  nb = n/p
  for i=0 to nb-1
    x[i] = 0
    for j=0 to n-1
      x[i] += aloc[i,j] * v[j]
    end
  end
```

```
SUB combinar(x,n,p)
foreach P(i), i=0 to p-1
  nb = n/p
  if i == 0
    for j=1 to p-1
      recv(x[j*nb:(j+1)*nb-1],j)
    end
  else
    send(x[0:nb-1],0)
  end
```

Example: Matriz-vector product (3)

Parallelisation using $p < n$ processors

Parallel time using message passing:

- `distribution`: $(p-1)\cdot\left(t_s + t_w \cdot \frac{n^2}{p}\right) + (p-1)\cdot(t_s + t_w \cdot n) \approx 2pt_s + n^2 t_w + pnt_w$
- `mvlocal`: $2\frac{n^2}{p}$ flops
- `combine`: $(p-1)\cdot(t_s + t_w \cdot n/p) \approx pt_s + nt_w$

$$\boxed{t(n,p) \approx 3pt_s + (n^2 + pn + n)t_w + 2\frac{n^2}{p} \text{ flops}}$$

Relative Parameters

Relative parameters are used to compare different parallel algorithms.

- Speed-up: $S(n,p)$
- Efficiency: $E(n,p)$

Usually, these are applied in the experimental analysis, although speed-up and efficiency can also be obtained in the theoretical analysis.

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 $t(n) could be$:

- 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)

Speed-up: Possible Cases

$S(n,p) < 1$

"Speed-down"

The parallel algorithm is slower than the sequential algorithm

$1 < S(n,p) < p$

Sublinear Case

The parallel algorithm is faster than the sequential, although it does not benefit from all of the processors.

$S(n,p) = p$

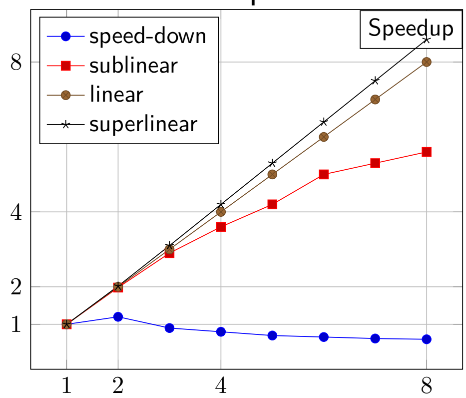
Lineal Case

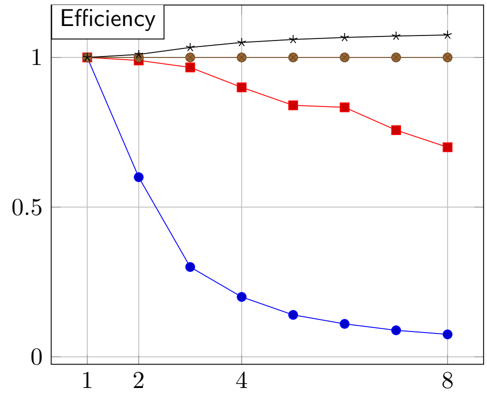
The parallel algorithm is as fastest as possible, using all the processors at 100%.

$S(n,p) > p$

Superlinear Case

Anomalous situation, when the parallel algorithm has less cost than the sequential.





Example: Matrix-Vector Product

Secuencial Time: $t(n) = 2n^2$ flops

Parallelisation by rows ($p = n$ processors)

In shared memory:
$t(n,p) = 2n$
$S(n,p) = n$
$E(n,p) = 1$

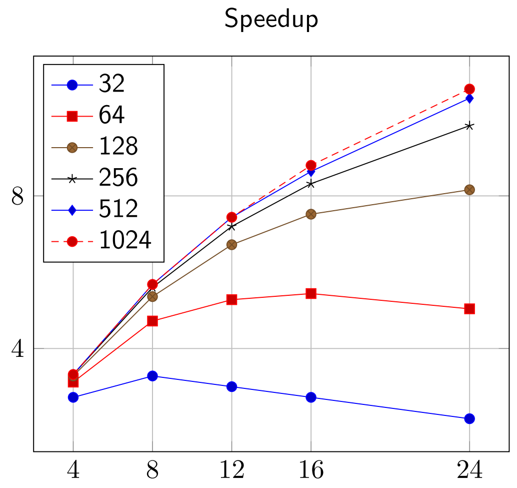
In Message-Passing:
$t(n,p) = 2n^2 t_w + 3nt_s + 2n$
$S(n,p) \rightarrow 1/t_w$
$E(n,p) \rightarrow 0$

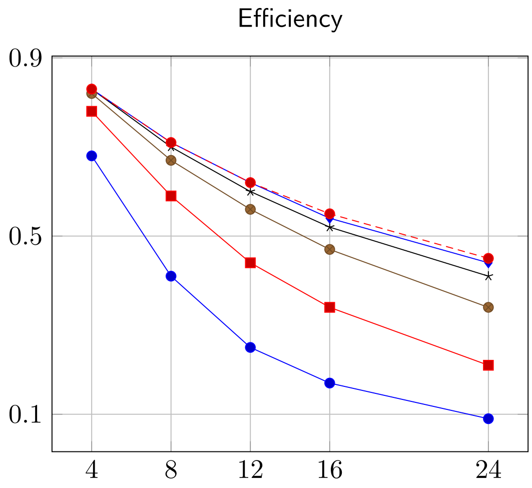
Row-blocks parallelisation ($p < n$ processors)

In Message-Passing:
$t(n,p) = 3pt_s + (n^2 + pn + n)t_w + 2\frac{n^2}{p}$
$S(n,p) \rightarrow \frac{2p}{pt_w+2}$
$E(n,p) \rightarrow \frac{2}{pt_w+2}$

Performance Variation

- Usually, the efficiency decreases as the number of processors is increased.
- The effect is normally less important for larger problem sizes.





Amdahl's Law

Often, a part of the problem cannot be executed in parallel
$\rightarrow$ The Amdahl's Law estimates the maximum Speed-up possible

Given a sequential algorithm, the execution time can be split accordingly: $t(n) = t_s + t_p$, where

- t_s is the time requested for the intrinsically sequential part.
- t_p is the time requested for the part that can be efficiently parallelised using p processors.

The parallel time will be then limited by: $t(n,p) = t_s + \frac{t_p}{p}$

Maximum Speed-up

$$\lim_{p \to \infty} S(n,p) = \lim_{p \to \infty} \frac{t(n)}{t(n,p)} = \lim_{p \to \infty} \frac{t_s + t_p}{t_s + \frac{t_p}{p}} = 1 + \frac{t_p}{t_s}$$

Section 4

Algorithm Design: Task Assignment

- The Problem of Task–Process Assignment
- Strategies for Merging and Replication

Task assignment

- The decomposition phase has produced a set of tasks.
- An abstract and *platform-independent* parallel algorithm is obtained, potentially *inefficient*.
- The domain decomposition must be *adapted* to a specific architecture.

Task assignment and task scheduling consist of defining

- the processing units and
- the order

in which the tasks will be executed.

Processes and processors

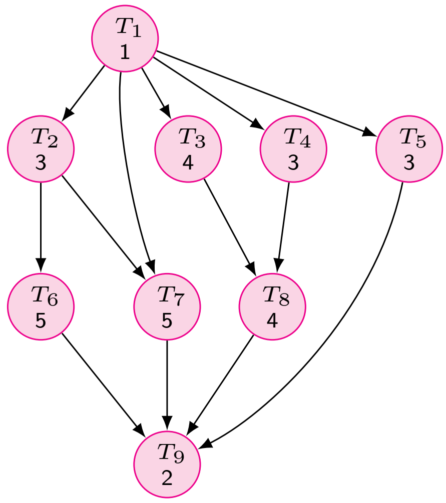
- Process: Computational logic unit that can execute tasks.
- Processor: Hardware unit that runs processes.

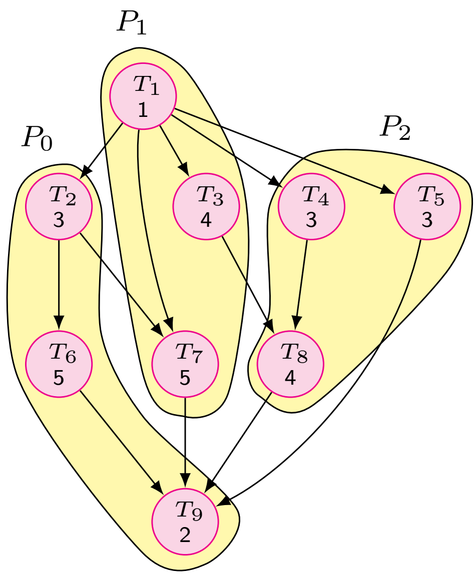
A parallel algorithm is composed of processes that execute tasks.

- The assignment that defines the relation between each task and process in the design phase.
- The assignment defining these relations is done after the design and typically at execution time.

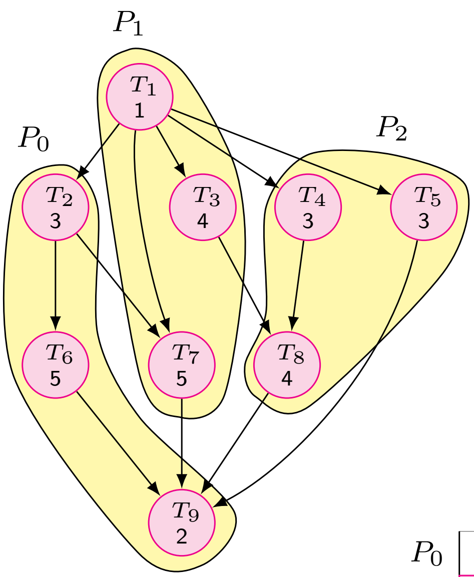
The problem of task–process assignment. Example (1)

Assignment: To define the task–process relation and the execution order. Example:

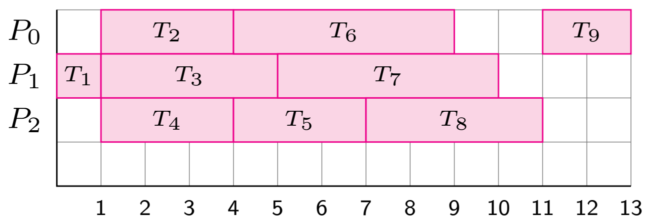




The problem of task–process assignment. Example (2)



Tasks execution order according to the assignment



Objectives of the Assignment

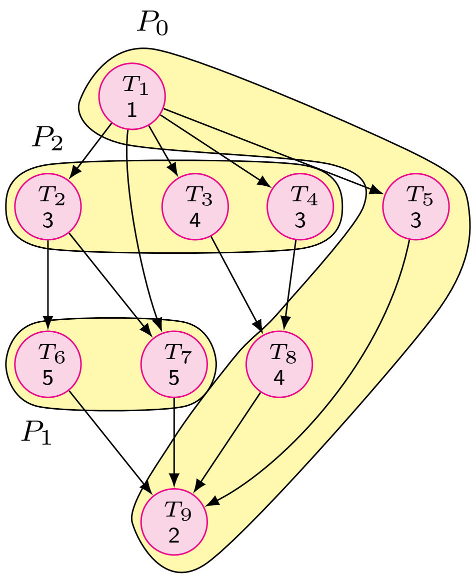
Main objective: To minimize execution time.

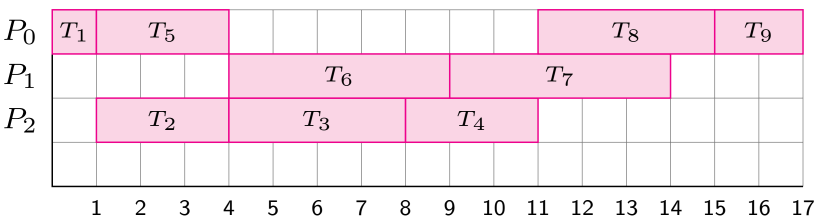
Factors of the execution time of a parallel algorithm and minimization strategies:

- Computing time: To maximize the concurrency by assigning tasks to processes.
- Communication time: To assign tasks that communicate among them to the same process.
- Idle time: To minimize the two main causes:
  - Load unbalancing: Computations and communications costs should be balanced among processes (previous diagram).
  - Waiting time: To minimize the waiting time of tasks which are not yet ready.

Objectives of the Assignment. Example

Example of load balanced assignment with higher *waiting time*





General strategies of assignment (1)

Static assignment or *deterministic scheduling*: The assignment decisions are taken before the execution time.
The typical steps involved are:

1. The number of tasks, their execution time and their communication costs are estimated.
2. Tasks are merged to reduce communication costs.
3. Tasks are associated to processes.

The optimal static assignment problem is *NP-hard* in the general case [1]. Advantages:

- Static methods do not add any overload to the execution time.
- Design and implementation are generally simpler.

[1] There are no algorithm able to solve the problem in polynomical time.

General strategies of assignment (2)

Dynamic assignment: Computational workload is shared at execution time.
This kind of assignment is used when:

- Tasks are dynamically generated
- When task size is not known a priori

In general terms, dynamic techniques are more complex. Main drawback is the overload due to

- load and workload information transfer among processes.
- Decisions for moving load among processes are taken at execution time.

Advantage: We do not need to know a priori the behaviour of the tasks, they are flexible and convenient for parallel architectures.

Merging (1)

Merging is used to reduce the number of tasks aiming at:

- limiting the task creation and termination costs, and
- minimizing the delays due to the interaction among tasks (local versus remote communications).

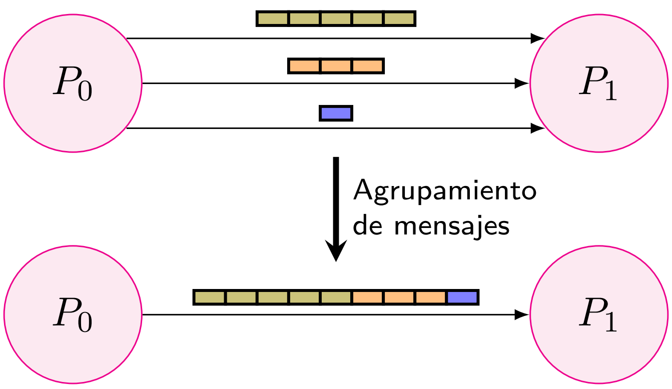
Merging strategies:

- *Data transfer volume minimisation*. Data-blocks task distribution (matrix algorithms), merging of sequential tasks (static task graphs), temporal storage of intermediate results (p.e. scalar product of two vectors).
- *Reduction of the interaction frequency*. To minimize the number of transfers and to increase the volume of data to be exchanged.

Merging (2)

Reduction of the frequency of interactions.

- In *distributed memory*, it means to reduce latency (number of messages) and to increase the volume of data per message.



- In *shared memory*, it means to reduce the number of cache misses.

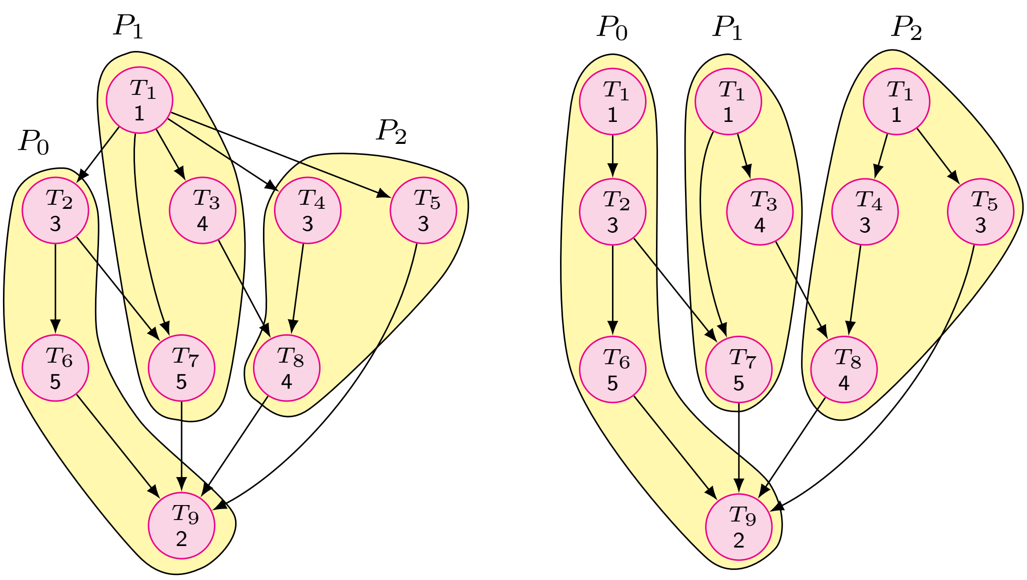
Replication

The Replication implies that part of the computations or data from a problem are not split but they are executed or managed by all or several processes.

- Data replication: It consists on copying common access data in different processes with the objective of reducing communication.
  - In *shared memory* it is implicit since it only affects cache memory.
  - In *distributed memory* it may lead to a considerable improvement of performance and a simplification of the design.
- Computation and communication Replication: It consists on repeating a computation in each one of the processes that need the result. It is convenient in the case that the computation cost is smaller than the communication cost.

Replication. Example

Example of replicationón of computing and communications. Given the next graph, and considering that communications have an associated cost. The T1 task is replicated



Section 5

Assignment Schemes

- Schemes for Static Assignment
- Dynamic Workload Balancing Schemes

Static Assignment Schemes

Static schemes for domain decomposition:

- They focus on the global large-scale data structure.
- The assignment of tasks to processes consists on splitting data among processes.
- Mainly two types:
  - Block-oriented matrix distributions
  - Static splitting of graphs

Schemes on static dependency graphs

- They are normally obtained using a functional decomposition of the data flow or recursive decomposition

Block-oriented distributions of matrices

In matrix computations, typically the computation of an entry depends on the neighbouring entries (spatial locality).

- The assignment considers neighbouring portions (blocks) in the data domain (matrix).

Most typical block distributions:

1. Uni-dimensional block distribution of a vector
2. Uni-dimensional distribution by blocks of rows of a matrix
3. Uni-dimensional distribution by blocks of columns of a matrix
4. Bi-dimensional block distribution of a matrix

We will also see the cyclic variants

Uni-dimensional Block Distribution

The global index i is assigned to process $\lfloor i/m_b \rfloor$ where $m_b = \lceil n/p \rceil$ is the block size

The local index is $i \mod m_b$ (reminder of interger division)

Example: for a vector of 14 elements among 3 processes

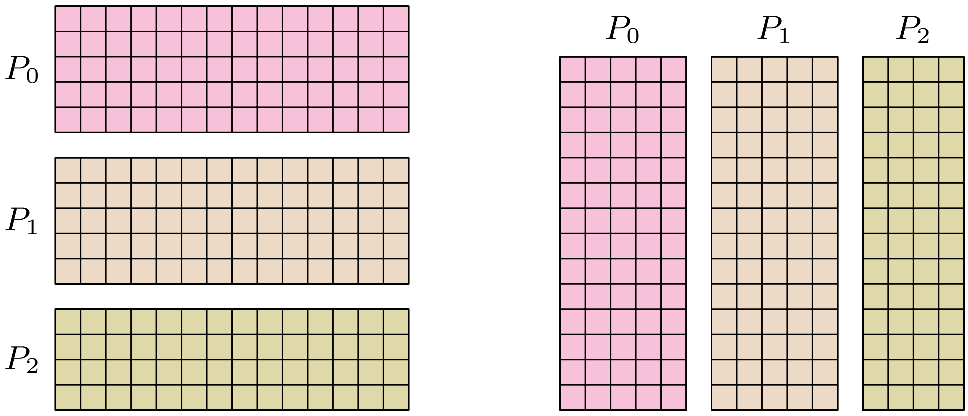
	P_0					P_1					P_2			
Global indices $\longrightarrow$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Local indices $\longrightarrow$	0	1	2	3	4	0	1	2	3	4	0	1	2	3

$m_b = \lceil 14/3 \rceil = 5$

Each process has m_b elements (except the last one)

Uni-dimensional Block Distribution. Example

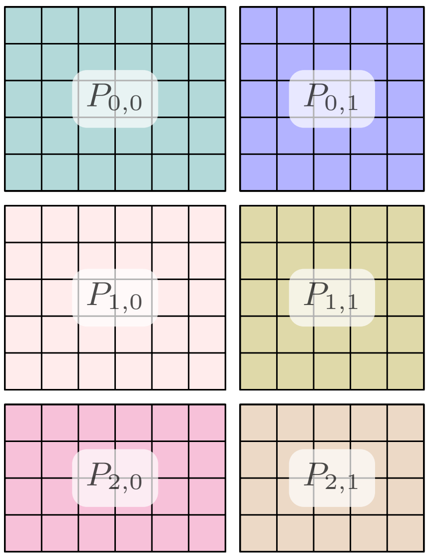
Example for a bi-dimensional matrix of 14×14 elements among 3 processes using row blocks and block columns.



- Each process has $m_b = \lceil n/p \rceil$ rows.

Bi-dimensional Block Distribution

Example for a bi-dimensional matrix of $m \times n = 14 \times 11$ elements among 3 processes by blocks organized in a grid 3×2.



Each process has a block of size $m_b \times n_b = \lceil m/p_m \rceil \times \lceil n/p_n \rceil$, where p_m and p_n are the first and second dimension of the grid respectively (3 and 2 in the example)

Example: Finite Differences (1)

Iterative computation on a matrix $A \in \mathbb{R}^{n \times n}$

- At the beginning it has a given value $A^{(0)}$
- At the k-th iteration ($k = 0, 1, \dots$) a new value is obtained $A^{(k+1)} = \left(a_{i,j}^{(k+1)}\right)$, $i, j = 0, \dots, n-1$, where

$$a_{i,j}^{(k+1)} = a_{i,j}^{(k)} - \Delta t \left(\frac{a_{i+1,j}^{(k)} - a_{i-1,j}^{(k)}}{0.1} + \frac{a_{i,j+1}^{(k)} - a_{i,j-1}^{(k)}}{0.02} \right)$$

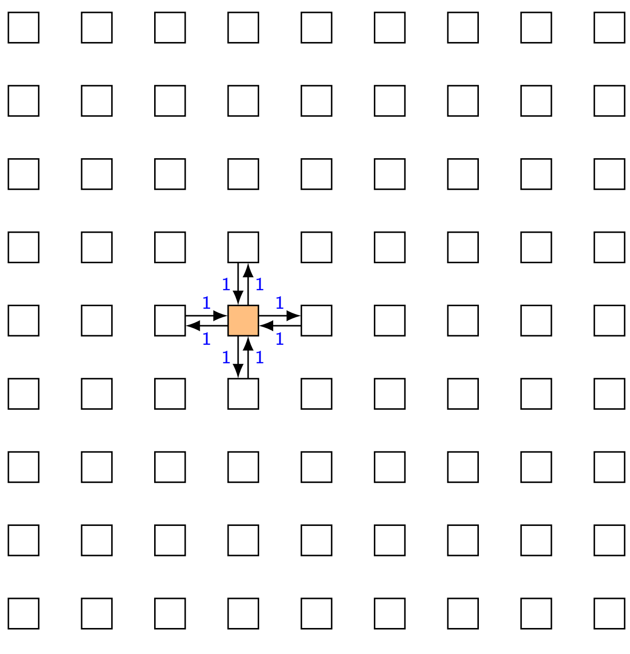
  and certain boundary conditions

We will next see the communication scheme of the algorithm for different distributions (for $n = 9$)

Example: Finite Differences (2)

Without merging

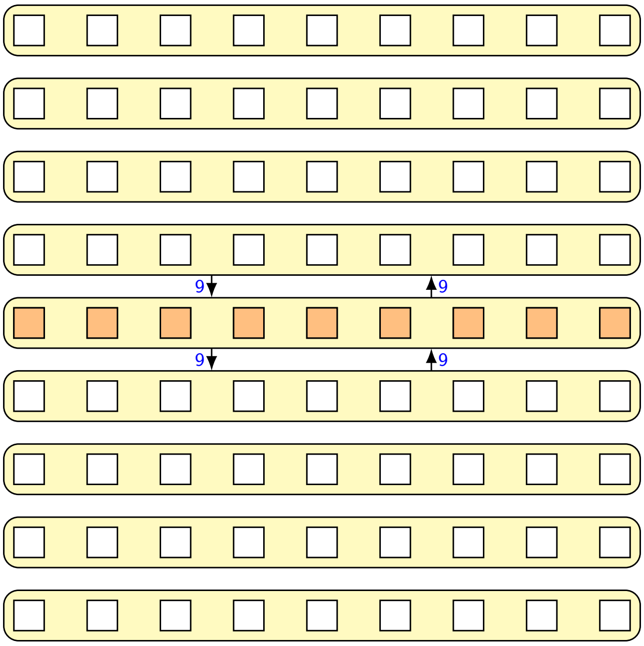
- 4 messages per task (1 element each)
- 288 total messages, 288 elements transferred



Example: Finite Differences (3)

Uni-dimensional merging

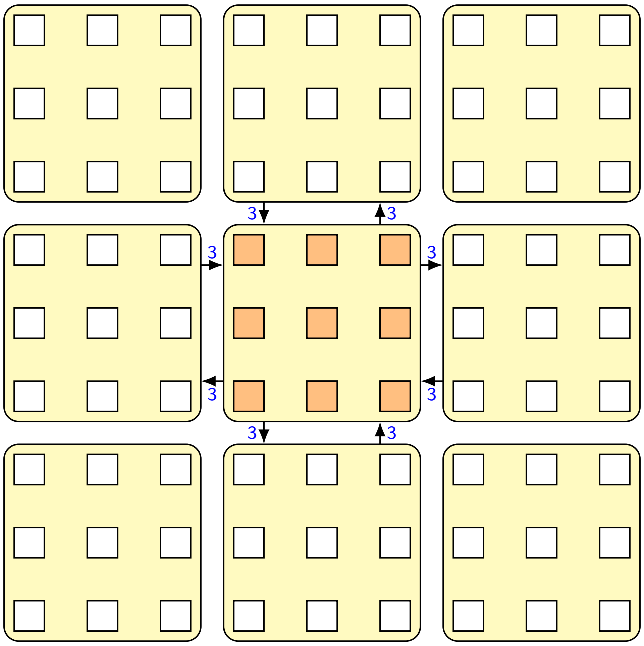
- 2 messages per task (9 elements each)
- 16 total messages, 144 elements transferred



Example: Finite Differences (4)

Bi-dimensional merging:

- 4 messages per task (3 elements each)
- 24 total messages, 72 elements transferred



Volume-surface effect

Task merging improves locality

- Reduces the volume of communication
- Try to merge task to maximize computation and minimize communication

Volume-surface effect

- The computational load increases proportionally with the number of elements assigned to a task (volume in 3D matrices)
- The communication cost increases proportionally to the perimeter of the task (surface in 3D matrices)

This effect grows as the number of dimensions of the matrix is increased

Cyclic Distributions

Objective: to balance the load during all execution time

- Larger communication cost since locality is reduced
- Usually combined with block schemes
- An equilibrium between load balancing and communication costs should be kept: most appropriate block size

Uni-dimensional cyclic distribution (block size 2):

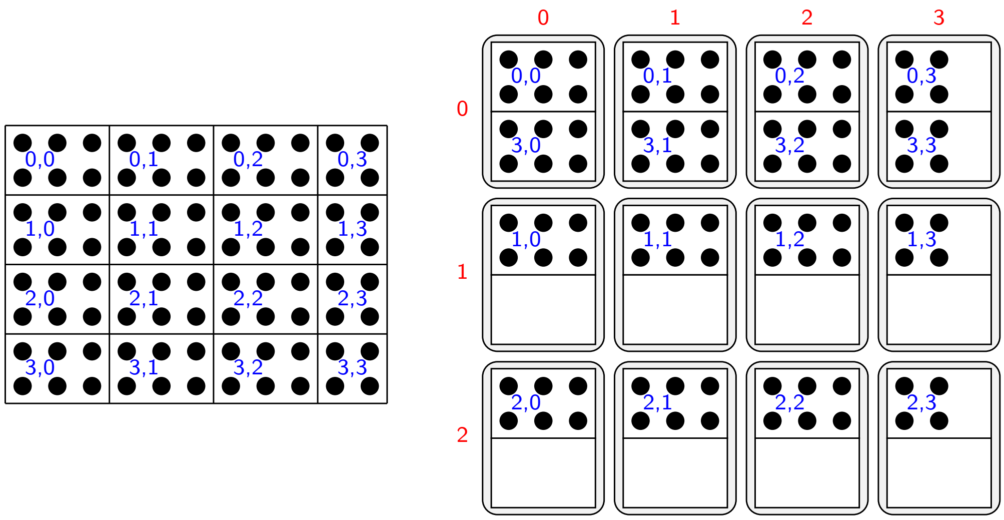
P_0		P_1		P_2		P_0		P_1		P_2		P_0	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

P_0						P_1				P_2			
0	1	6	7	12	13	2	3	8	9	4	5	10	11

Similarly, it applies to matrices (by rows or columns)

Bi-dimensional Cyclic Distribution

Exemple of a bi-dimensional block cyclic distribution:
Matrix of 8×11 elements in blocks of 2×3 in a grid of 3×4 processes



Assignment based on static dependency graphs

Case of functional decomposition

- We assume a static dependency graph and task costs known a priori

The problem of finding optimal assignments is NP-complete

However, there are cases in which optimal algorithms and heuristic approaches are known

Examples:

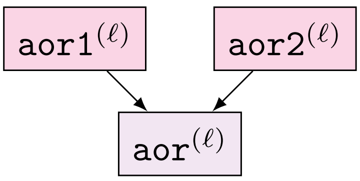
- *Binomial Tree Structure.*
- *Hypercube Structure.*

Example: Vector Sorting (1)

Given a vecor of numbers (a) sort it and store it on (aor)

```
function mergesort(a, aor, n)
if n<=k
  aor = sort(a, aor, n)
else
  m = n/2
  a1 = a[0:m-1]
  a2 = a[m:n-1]
  aor1 = mergesort(a1, aor1, m)
  aor2 = mergesort(a2, aoir2, n-m)
  aor = merge(aor1, aor2, aor)
end
```

Simplified dependency graph:

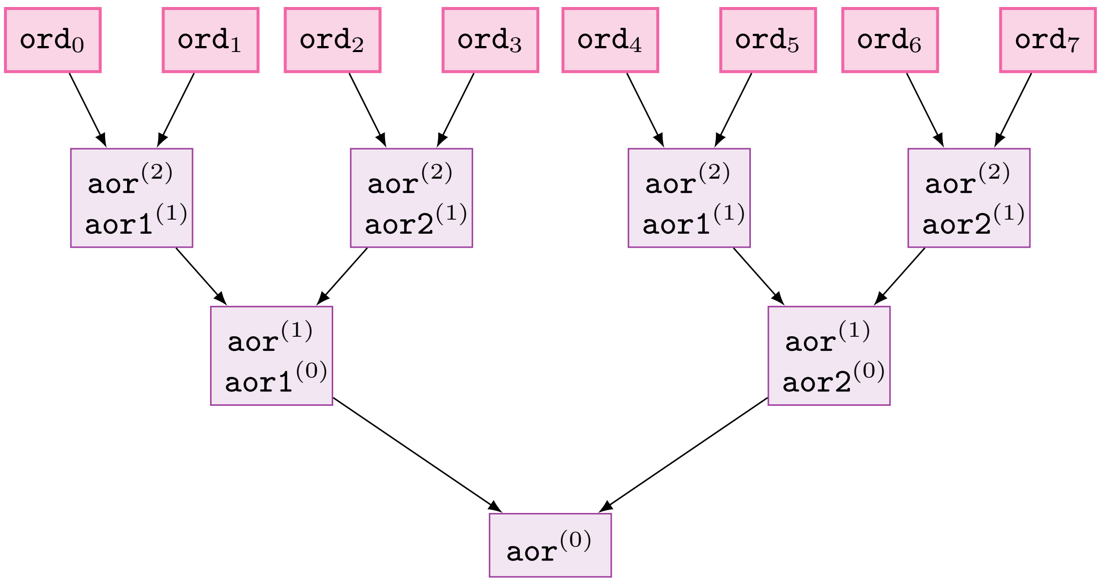


Parallelisation strategy:

- $\log_2(n/k)$ recursion levels
- Distributing tasks
  - Tree leaves (sort)
  - merge in each level ℓ

Example: Vector Sorting (2)

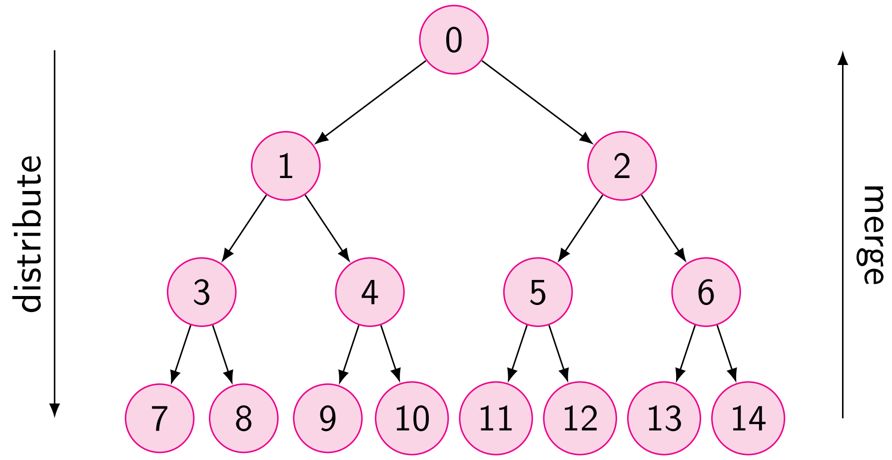
Complete dependency graph for $n = 8k$



Example: Vector Sorting (3)

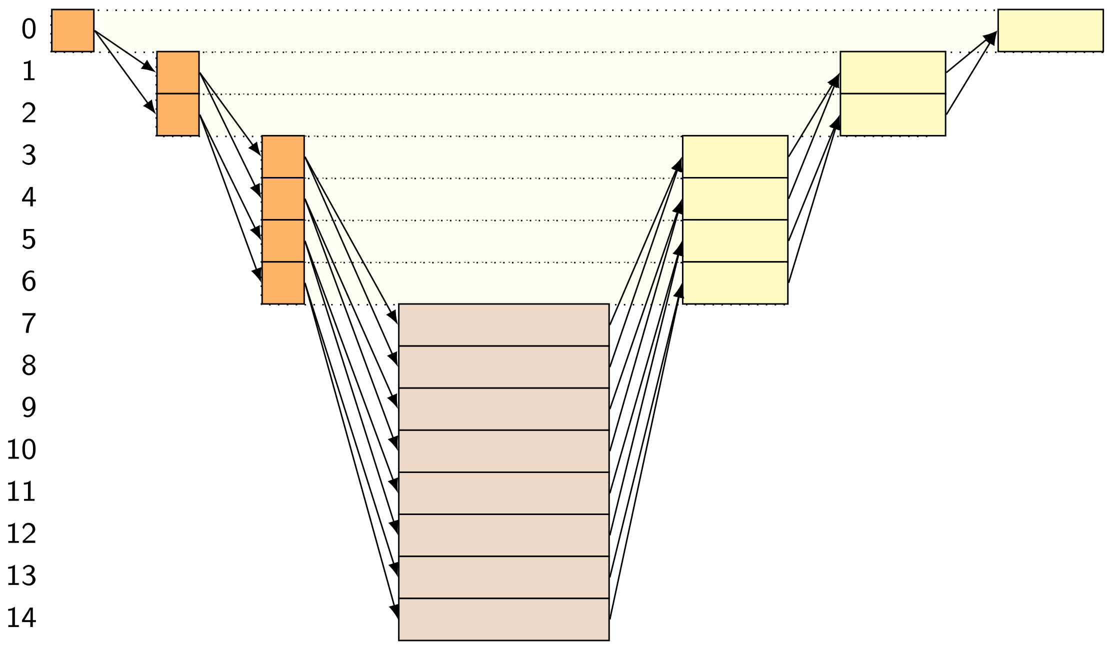
Assuming p processors with a topology of a binary tree

- Last level orders, rest of the stages merge.
- The maximum size is $k * (p + 1)/2$



If the processors are not linked by a tree-like topology, it can be mapped on other ones.

Example: Vector Sorting (4)



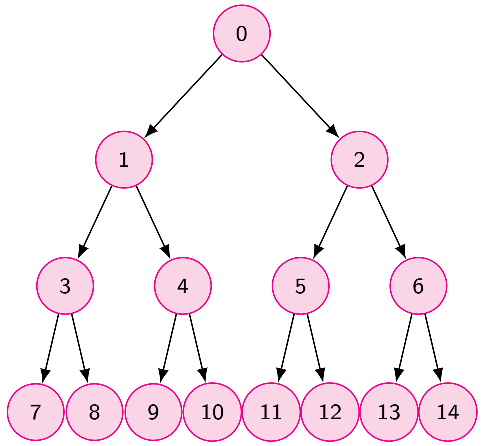
Better efficiency if processors are "reused"

- For example, as many processors as leaves in the tree.

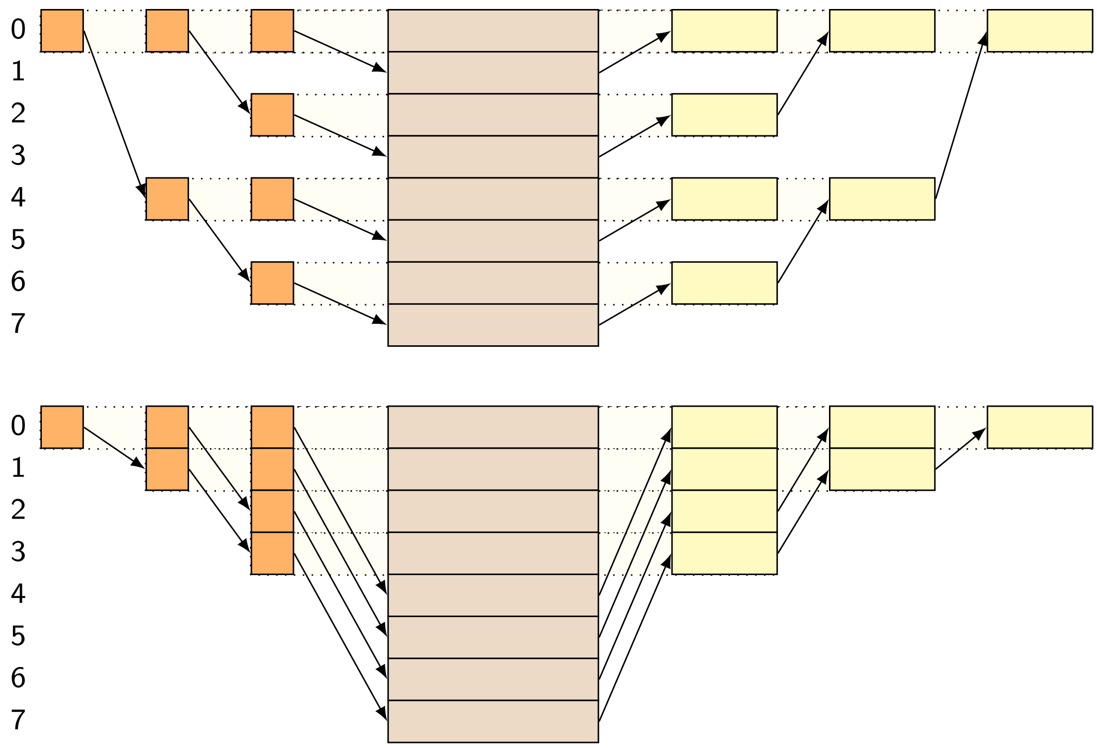
Example: Vector Sorting (5)

Message passing: as nuch processirs as nodes in the tree.

```
foreach P(pr), pr=0 to p-1
  if pr <> 0
    recv(a,(pr-1)/2)
  end
  if log2(pr+1) < log2(p)
    n = len(a)
    a1 = a[0:n/2-1]
    a2 = a[n/2:n-1]
    send(a1,pr*2+1)
    send(a2,pr*2+2)
    recv(aor1,pr*2+1)
    recv(aor2,pr*2+2)
    aor = merge(aor1, aor2)
  else
    sort(a, aor)
  end
  if pr <> 0
    send(aor,(pr-1)/2)
  end
```



Example: Vector Sorting (6)



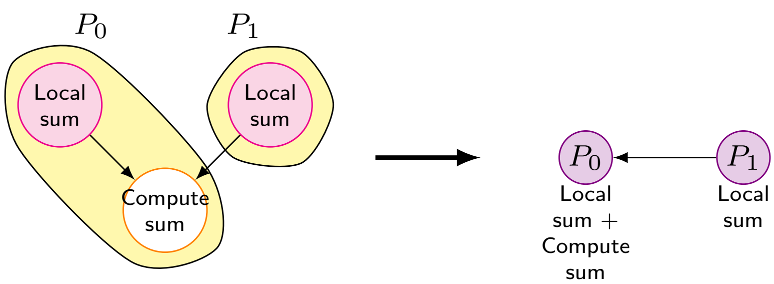
Static Dependency Graphs. Binomial Tree

Example: design of a parallel reduction

- In cases such as the computation of the *scalar product* the dependency graph has a binary tree topology

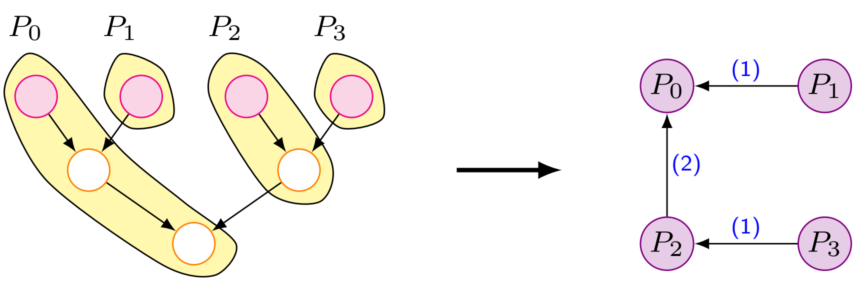
The optimal assignment is obtained by grouping nodes of different levels for which a dependency relation exists $\longrightarrow$ binomial tree

$$p = 2$$

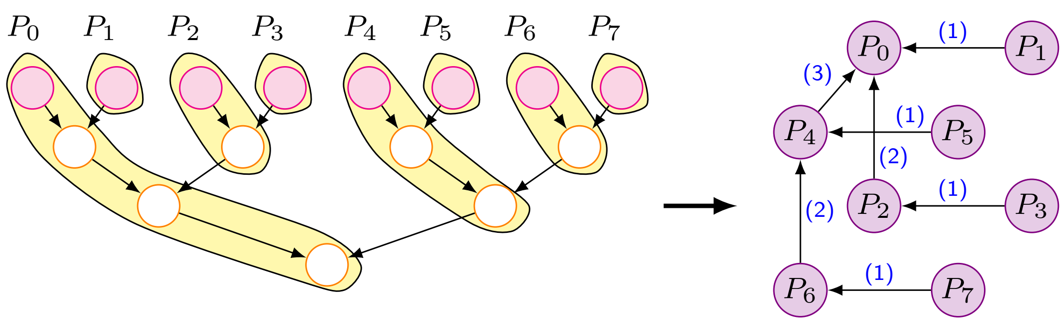


Static Dependency Graphs. Binomial Tree

$$p = 4$$



$$p = 8$$



Static Dependency Graphs. Binomial Tree

- Previous binary tree has $1+\log n$ levels, being $n =$ leaves.
- A binomial tree of order 0 has a single node.
- A binomial tree of order k is form by linking two binomial trees of order $k-1$ taking as root a node that has the two root nodes of the previous trees as leaves.
- A binomial tree of order k has 2^k nodes and a k depth.
- Using a binomial tree, it is possible to compute the sum of p values in $\log p$ steps.
- For a binary tree with $p = 2^k$ leaves, an optimal assignment will consist on a binomial tree with p processes.

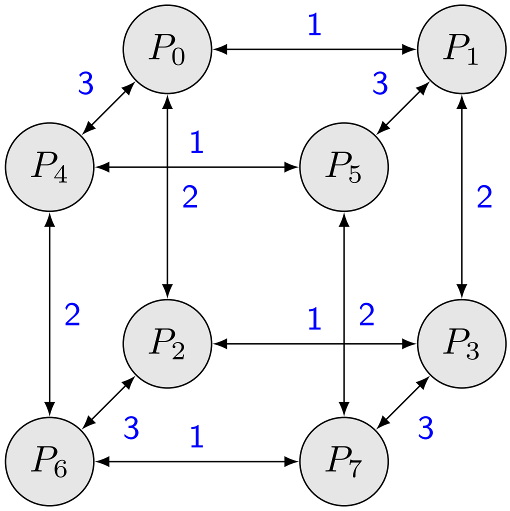
Static dependency Graphs. Hypercube (1)

Example: *Reduction with replication*
Assuming that the result of the scalar product should be available to all the tasks.

- Option 1 ($2\log p$ steps):
  - Phase 1: Obtain the scalar product by going through the binary tree from the bottom to the root .
  - Phase 2: Broadcast the result by going back in the binary tree until all leaves are reached.
- Option 2 ($\log p$ steps): Strategy of computation and communication replication.
  - The binomial tree structure is extended in a way that the maximum distance between each one of the 2^k nodes is k by assigning k neighbours to each node: hypercube of order k.
  - In each step of the algorithm, each process exchanges its value with each one of their neighbours and adds the received value to the local result.
  - After $\log p$ steps, it can be guaranteed that all processes has the sum of the initial values.

Static dependency Graphs. Hypercube (2)

Example: *Reduction with replication of the scalar product of two vectors using 8 neighbours*



Dynamic workload balancing schemes

These schemes are used when static approaches are inefficient.
These schemes consider that:

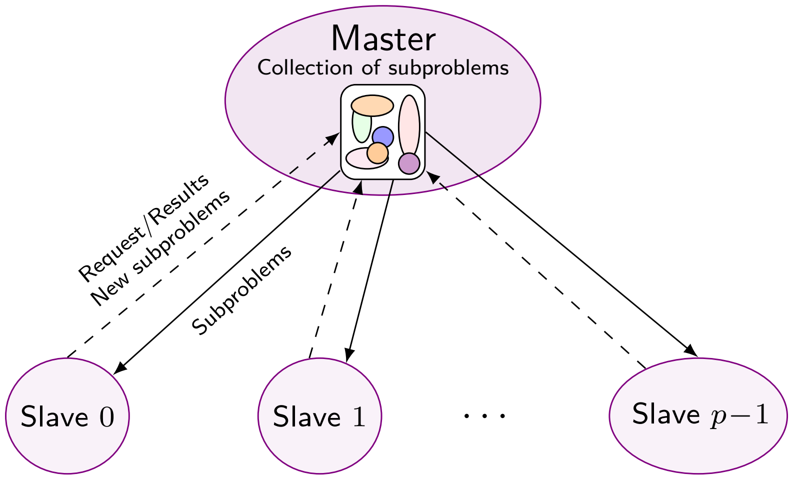
- The tasks obtained in the decomposition are data structures that define sub-problems.
- Processes make the solution of the sub-problems.
- Sub-problems are kept in a collection and dispatched to the different processes.
- The solution of a problem's usualy leads to the dynamic creation of more sub-problems.
- This data collection can be centralized or distributed.
- These schemes require a *stopping criteria* to end.

Centralized dynamic schemes (1)

These schemes are based on

- A master process that manages the collection of tasks and assigns them to the rest of the processes typically called
- slaves, which repeatedly take sub-problems from the master, execute them and potentially generate new sub-problems.

New sub-problems generated by the slaves can be solved by the salve or sent back to the master.



Centralized dynamic schemes (2)

This strategy is effective when

- the number of slaves is reduced, and
- the cost of executing sub-problems is high with respect to the cost of generating them.

This strategy can be improved:

- **Block scheduling**: Slaves take a group of sub-problems at a time.
- **Local collection of sub-problems**: Slaves do not send back sub-problems to the master but keep them on a local list.
- **Overlapped fetching**: Slaves take sub-problems from the master concurrently with their processing.

Stopping criteria is simple since it is centralised in the master process.

Distributed dynamic schemes (1)

In these schemes:

- There is no master,
- Sub-problems are kept distributed in the different local queues of the slaves.

Load distribution is not trivial. There are two main strategies that differ in the way transfer is initiated:

- By the receiver: suitable for high workloads.
- By the sender: suitable for low workloads.

Distributed dynamic schemes (2)

Receiver-initiated transfer: There are two strategies for a process that needs to get sub-problems to select the source processes:

- Random poll: Simple and even.
- Cyclic poll: Balanced but most costly.