CPA – Parallel Computing

Degree in Computer Science

T3. Message Passing. Advanced Parallel Algorithms Design

J. M. Alonso, P. 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 2019/20

Content

1 Message Passing Model
- Model
- Details

2 Algorithmic Schemes (II)
- Data Parallelism
- Tree Schemes
- Other Schemes

3 Performance Evaluation (II)
- Parallel Time
- Relative Parameters

4 Algorithm Design: Task Assignment
- The Problem of Assignment
- Strategies for Merging and Replication

5 Assignment Schemes
- Schemes for Static Assignment
- Dynamic Workload Balancing

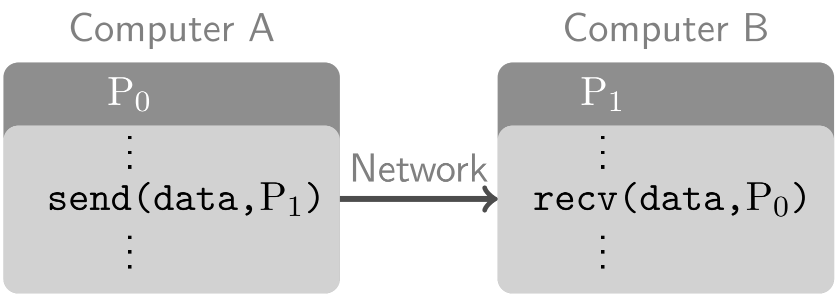
Section 1

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 parallelization



MPI: Message Passing Interface

Process Creation

The parallel program comprises several processes

- Usually correspond to O.S. processes
- Typically one process per processor
- Each one has an index or identifier (integer number)

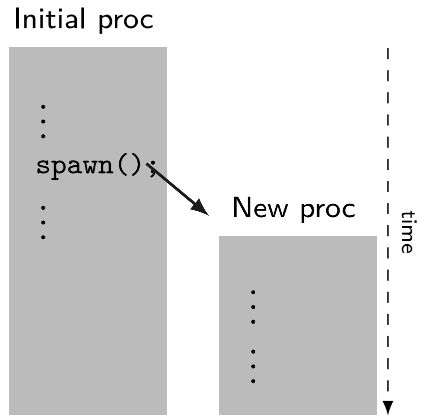
Process creation can be:

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

- `spawn()` primitive



Communicators

Processes are organized into groups

- Key in collective operations, such as broadcast (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 communication taking place in another
- Useful for isolating the communication within a library
- Communicators are defined from groups or other communicators
- Predefined communicators:
  - World: includes all processes created by `mpiexec`
  - Self: includes only the calling process

Basic Send/Receive Operations

The most common operation is point-to-point communication

- One process sends a message (`send`), another 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);
```

The `send` operation 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

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

```
SUB 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
```

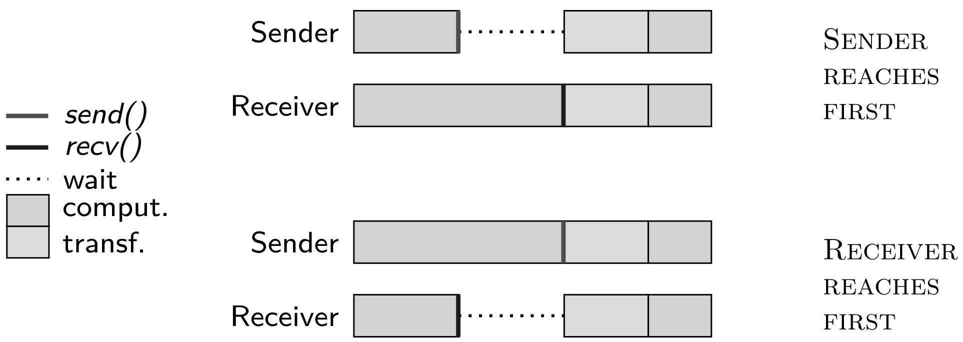
```
SUB parsum(v,w,vl,wl,x,xl,n)
FOREACH P(i), i=0 TO n-1
  IF i == 0
    x[0] = v[0] + w[0];
  ELSE
    xl = vl + wl
  END
```

```
SUB 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 done the matching `recv`

- Along with the data transfer, processes get synchronized
- It requieres a protocol so that sender and receiver know that transmission can begin (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/nonblocking operations

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

Operation Finalization

In nonblocking operations we need to know if the operation is complete

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

Nonblocking `send` and `recv` provide a request number (*req*)

Primitives:

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

This can be used to overlap communications and computing

Selection of Messages

The `recv` operations requires a process identifier *id*

- It does not finish until a message from *id* arrives
- Messages from other processes are ignored

For more flexibility, it is possible to use a wildcard value to receive from any process

Moreover, a tag is used to distinguish among messages

- A wildcard is also possible to match any tag

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

- The `recv` primitive has a `status` argument containing the sender id and the tag
- Unselected messages are not lost, they remain in a “message queue”

Problem: Deadlock

An incorrect usage of `send` and `recv` can lead to deadlocks

Case of synchronous communication:

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

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

- Both get blocked in the sending

Case of buffered send:

- The above example would not cause deadlock
- There may be other situations leading to deadlocks

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

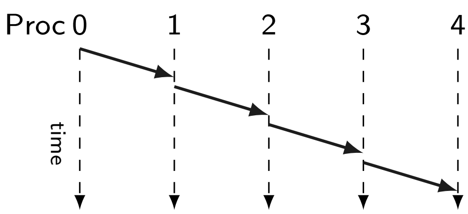
```
/* Process 1 */
recv(x,0);
send(y,0);
```

Potential solution: exchange the order in one of them

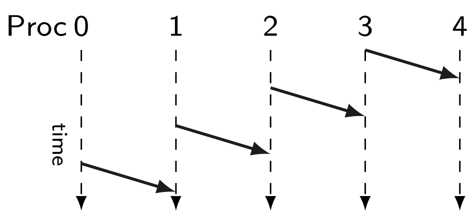
Problem: Serialization

Each process has to send data to its right neighbor

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




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



Potential solutions:

- Odd-even protocol: odd processes perform one variant, even processes the other
- Nonblocking `send` or `recv`
- Combined operations: `sendrecv`

Collective Communication

Collective operations involve all processes in a communicator (in many cases, one of them has a special role – root process)

- Synchronization (`barrier`): all processes wait for the rest to arrive
- Data transfer: one or more processes send to one or more
- Reductions: along with the communication an operation is performed on data

These operations can be realized using point-to-point communication, but it is recommended to use the corresponding 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
- All-to-one reduction
  - The dual of broadcast
  - Data are combined using an associative operator
- Scatter
  - The root sends an individualized message to the rest
- Gather or concatenation
  - The dual of scatter
  - Similar to reduction but without operation
- All-to-all broadcast
  - p simultaneous broadcasts, with different root processes
  - At the end, all processes will have received all the data
- All-to-all reduction
  - The dual of all-to-all broadcast

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

- The computational volume must be at least one order of magnitude higher than the communication cost
  - ✗ Vector-vector: cost $\mathcal{O}(n)$ vs. $\mathcal{O}(n)$ communication
  - ✗ Matrix-vector: cost $\mathcal{O}(n^2)$ vs. $\mathcal{O}(n^2)$ communication
  - ✓ Matrix-matrix: cost $\mathcal{O}(n^3)$ vs. $\mathcal{O}(n^2)$ communication
- Often data are already distributed

Case 1: Matrix-Vector Product

Message-passing solution ($p = n$ 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
```

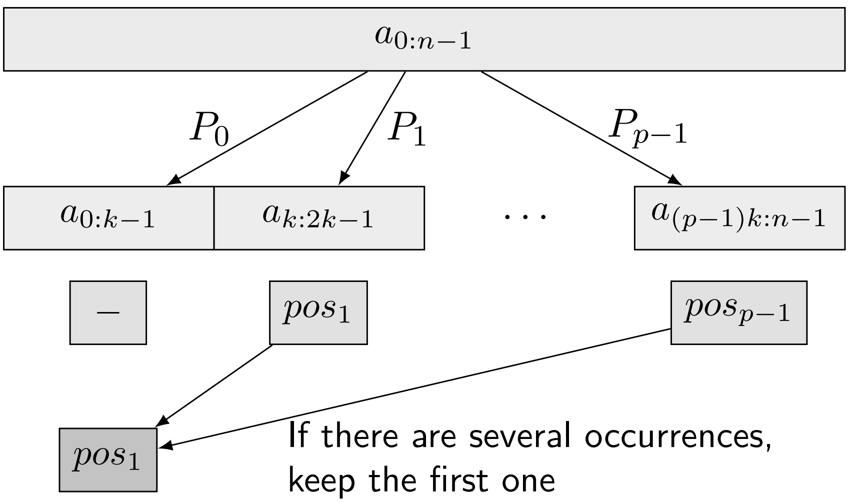
```
SUB mvlocal(Al,v,xl,n,m)
FOREACH P(i), i=0 TO n-1
  xl = 0
  FOR j=0 TO m-1
    xl = xl + Al[j] * v[j]
  END
```

```
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: Linear Search

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

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



Case 2: Linear Search - Pseudocode

Message-passing solution

- Initially x, a are in P_0, the result pos should be stored in P_0

```
SUB search(a,x,pos,n,p)
distribute(a,apr,x,n,p)
searchloc(apr,x,pos,n,p)
combine(pos,n,p)
```

```
SUB distribute(a,apr,x,n,p)
FOREACH P(i), i=0 TO p-1
  /* collective operations */
  /* process 0 sends a portion
     of n/p elements of a,
     received in apr */
  scatter(a,n/p,apr,n/p,p,0)
  /* process 0 sends x to all,
     all receive in x */
  broadcast(x,0)
```

```
SUB 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
```

```
FUNCTION combine(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 solution

```
SUB sum(v,s,n,p)
distribute(v,vloc,n,p)
sumloc(vloc,sl,n,p)
reduce(sl,s,p)
```

```
SUB 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
```

```
SUB sumloc(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
```

```
SUB 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

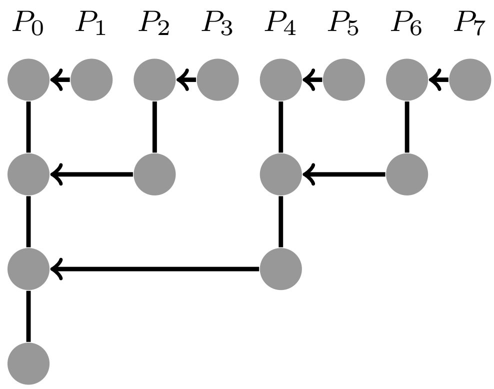
Tree Schemes

Reduction using Recursive Doubling:

There are $\log_2(p)$ communicaion stages

- The number of participating processes is divided by two after each stage
- The receiver accumulates the value on its local sum s

```
FOREACH P(pr), pr=0 TO p-1
  s = sl;
  FOR j=1 TO log2(p)
    IF rem(pr,2^j)==0
      recv(aux)
      s = s + aux
    ELSE
      IF rem(pr,2^(j-1))==0
        send(s,pr-2^(j-1))
      END
    END
  END
```



Task Parallelism

In cases where decomposition creates more tasks than processes, or when solving a task generates new tasks

- Static assignment of tasks is not feasible or leads to load imbalance
- Dynamic assignment: tasks are being assigned to processes as they become idle

---

Usually implemented by means of an asymmetric scheme: manager-worker (or master-slave)

- The master keeps a count of finished and pending tasks
- Workers receive tasks and notify the master when they have finished them

Sometimes, a symmetric solution is feasible: replicated workers

Manager-Worker (Master-Slave)

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

`max_row` (independent) lines of the image are processed

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

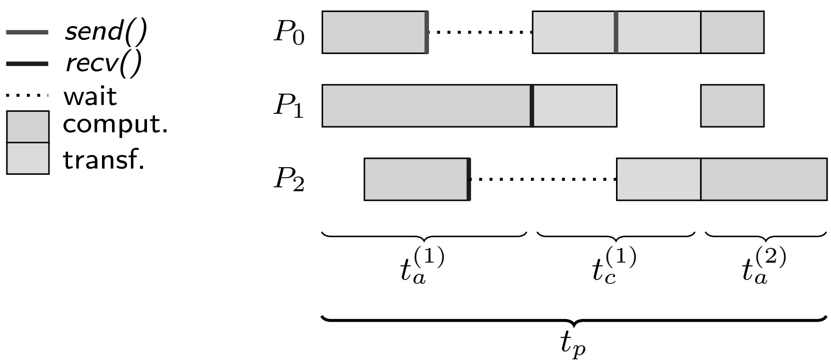
- All processors compute concurrently
- It is equal or higher than the maximum arithmetic time

t_c corresponds to times associated with 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, P_2



In practice:

- There is no clear separation between computation and communication stages (P_1 does not need to wait)
- Often communication and computation can be overlapped (with nonblocking operations, e.g. P_2)

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

Modelling Communication Time

Assuming message passing, P_0 and P_1 running on two different nodes with direct link

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

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

In practice, things get complicated:

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

Recommendations:

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

In shared memory, considerations are different

Example: Matrix-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

Parallelization using $p = n$ processors

Parallel time in shared-memory:
$t(n,p) = 2n$ flops

Parallel time in message-passing:

- `distribute`: $2 \cdot (n-1) \cdot (t_s + t_w \cdot n)$
- `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: Matrix-Vector Product (2)

Version for $p < n$ proc. (block row 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)
FOREACH 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 combine(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: Matrix-Vector Product (3)

Parallelization using $p < n$ processors

Parallel time using message passing:

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

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

Relative Parameters

Relative parameters are used to compare different parallel algorithms

- Speedup: $S(n, p)$
- Efficiency: $E(n, p)$

Usually, these are applied in the experimental analysis, although speedup and efficiency can also be obtained in the theoretical analysis

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

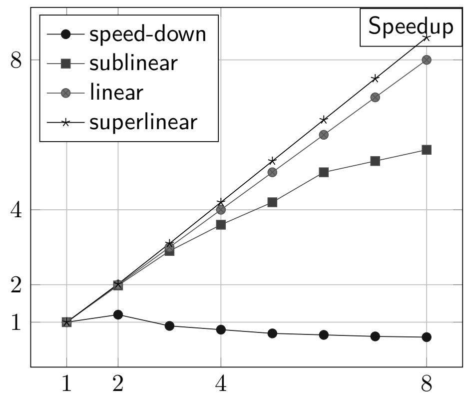
The efficiency measures the degree of utilization of the parallel computer by the parallel algorithm

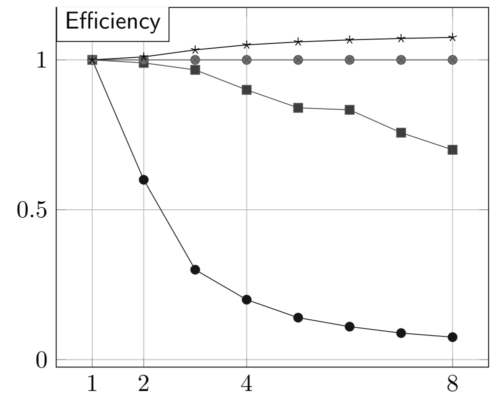
$$E(n, p) = \frac{S(n, p)}{p}$$

Usually expressed as a percentage (or parts per unit)

Speedup: Possible Cases

$S(n,p) < 1$	$1 < S(n,p) < p$	$S(n,p) = p$	$S(n,p) > p$
"Speed-down"	Sublinear case	Lineal case	Superlinear case
The parallel algorithm is slower than the sequential algorithm	The parallel algorithm is faster than the sequential one, but does not exploit all the capacity of procs	The parallel algorithm is as fast as possible, exploiting all the processors up to 100%	Anomalous situation, when the parallel algorithm has less cost than the sequential one





Example: Matrix-Vector Product

Sequential time: $t(n) = 2n^2$ flops

Parallelization by rows ($p = n$ processors)

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

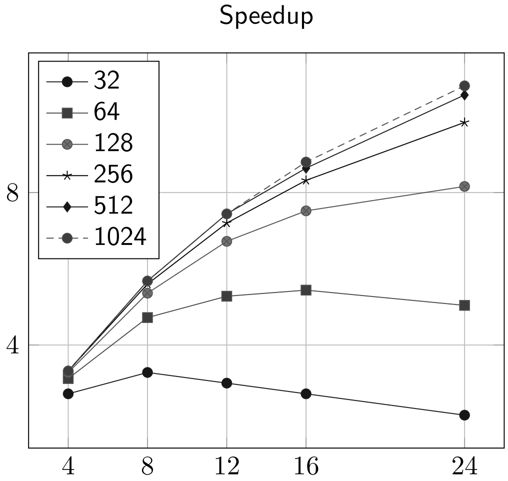
In message passing:
$t(n,p) = 2n^2 t_w + 3nt_s + 2n$
$S(n,p) \to 1/t_w$
$E(n,p) \to 0$

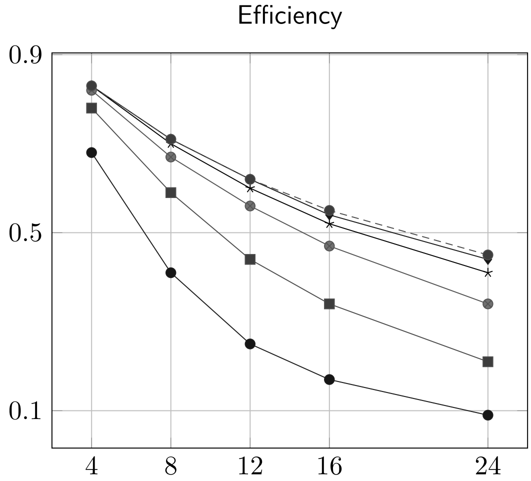
Block row parallelization ($p < n$ processors)

In message passing:
$t(n,p) = 3pt_s + (n^2 + pn)t_w + 2\frac{n^2}{p}$
$S(n,p) \to \frac{2p}{pt_w+2}$
$E(n,p) \to \frac{2}{pt_w+2}$

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
$\rightarrow$ Amdahl's Law estimates the maximum attainable speedup

Given a sequential algorithm, split $t(n) = t_s + t_p$, where

- t_s is the time of the intrinsically sequential part
- t_p is the time of the perfectly parallelizable part (can be solved using p processors)

The minimum attainable parallel time will be $t(n,p) = t_s + \frac{t_p}{p}$

Maximum speedup

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

Section 4

Algorithm Design: Task Assignment

- The Problem of 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 obtained decomposition must be *adapted* to a specific architecture

---

Task assignment and task scheduling consist in determining

- the processing units and
- the order

in which the tasks will be executed

Processes and Processors

- Process: Logical computing agent that performs tasks
- Processor: Hardware unit that physically performs computations

A parallel algorithm is composed of processes that execute tasks

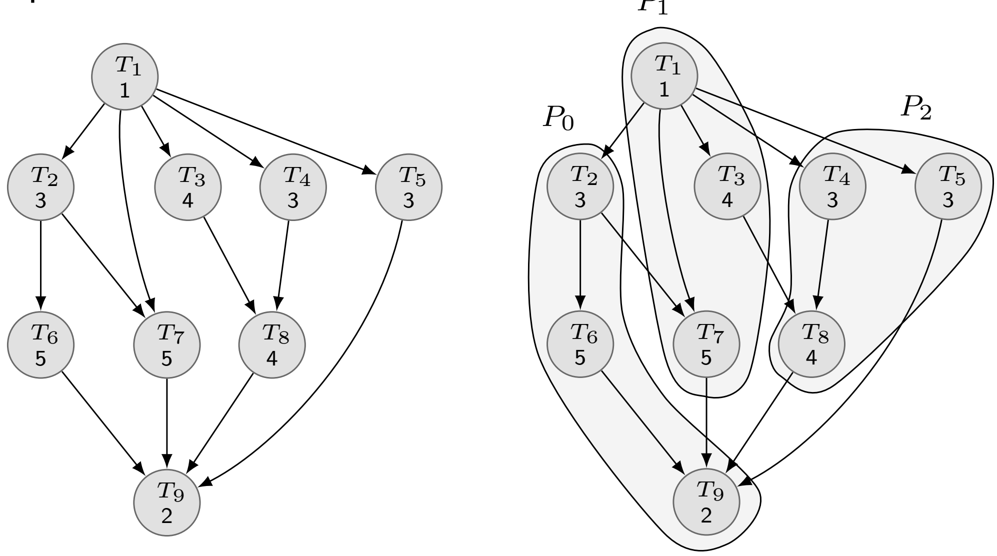
- The assignment establishes the correspondence between tasks and processes in the design phase
- The correspondence between processes and processors is done at the end and typically at execution time

The Assignment Problem. Example (1)

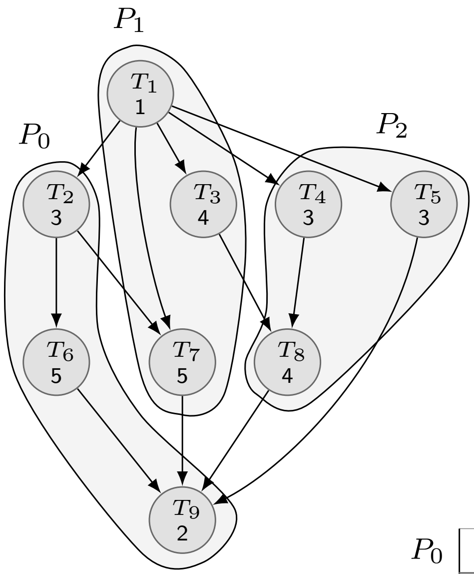
Assignment: to establish the task–process correspondence and select the execution order

Usually also includes the previous grouping of some tasks

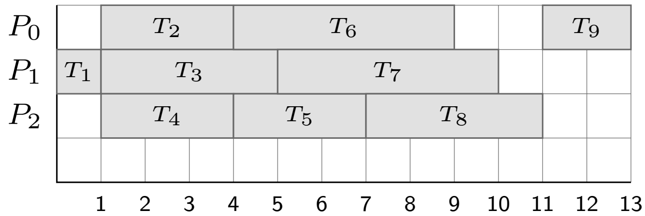
Example:



The Assignment Problem. Example (2)



Task execution order according to the selected assignment



Objectives of the Assignment

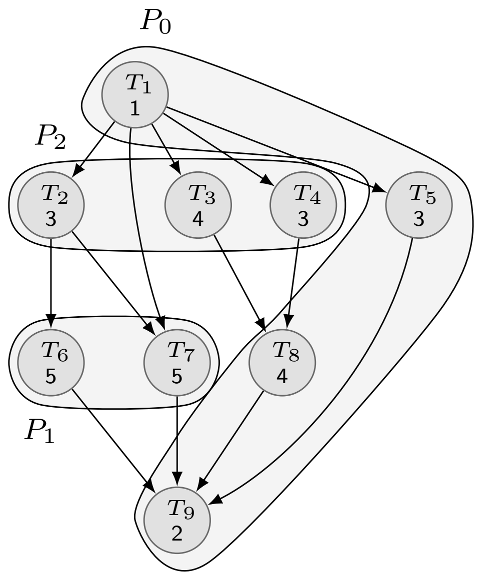
Main objective: To minimize execution time

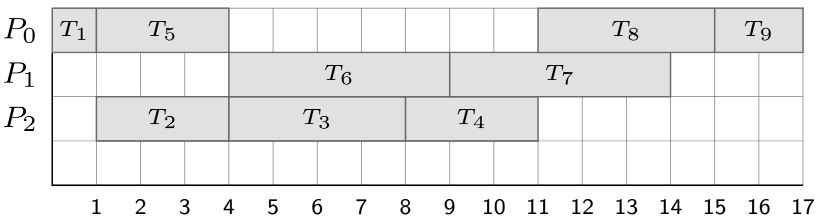
Factors of the execution time of a parallel algorithm and minimization strategies:

- Computing time: To maximize concurrency by assigning independent tasks to distinct processes
- Communication time: To assign tasks that communicate between them a lot to the same process
- Idle time: To minimize the two main causes:
  - Load imbalance: computation and communication costs should be balanced among processes (previous diagram)
  - Waiting time: to minimize the waiting time of tasks that are not yet ready

Objectives of the Assignment. Example

Example of well balanced assignment, but with longer waiting time





General Strategies of Assignment (1)

Static assignment or deterministic scheduling: The assignment decisions are taken before the execution time. Steps:

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

The optimal static assignment problem is *NP-complete* in the general case[1]

Advantages of static assignment:

- Does not add any overhead at run time
- Design and implementation are generally simpler

[1] There is no known algorithm to solve the problem in polynomial time

General Strategies of Assignment (2)

Dynamic assignment: The distribution of computational workload is done at execution time

This kind of assignment is used when:

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

In general, dynamic techniques are more complex. The main drawback is the induced overhead due to:

- Information transfer among processes regarding load and work
- Decision making to move load among processes (done at run time)

Advantage: No need to know the behaviour a priori, they are flexible and appropriate for heterogeneous 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 data access)

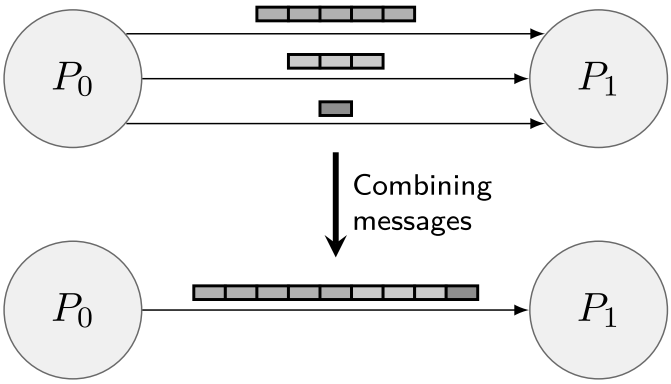
Merging strategies:

- *Volume minimization of the transferred data*. Distribution of tasks based on data blocks (matrix algorithms), merging of non-concurrent tasks (static task graphs), temporal storage of intermediate results (e.g. 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

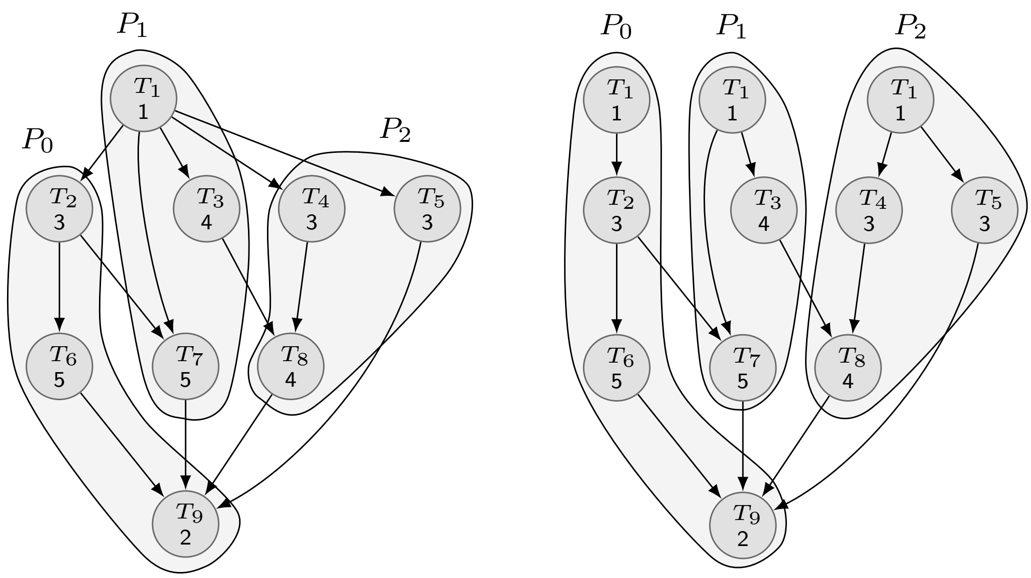
Replication

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

- Data replication: consists in copying commonly accessed data in the different processes aiming at reducing communication
  - In *shared memory* it is implicit since it only affects cache memory
  - In *distributed memory* it may lead to a considerable performance improvement and design simplification
- Computation and communication replication: consists in 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 replication 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

Static Assignment Schemes

Static schemes for domain decomposition:

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

Schemes on static dependency graphs

- They are normally obtained by 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 neighboring entries (spatial locality)

- The assignment considers contiguous 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)

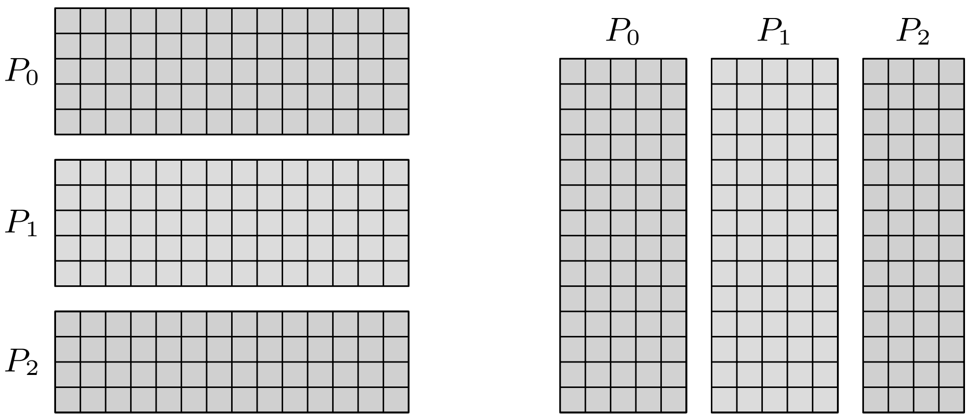
Example: for a vector of 14 elements among 3 processes

	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

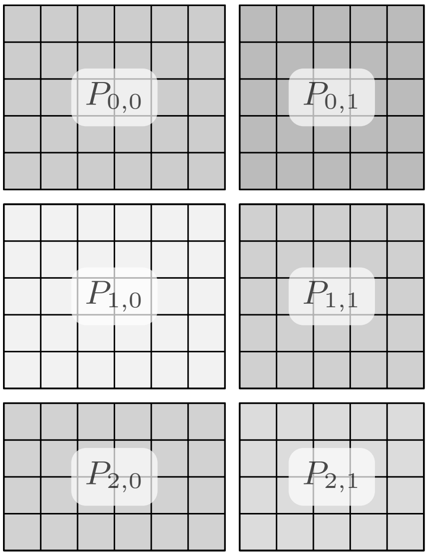
Example for a bidimensional matrix of 14×14 elements among 3 processes by row blocks and block columns



Each process owns $m_b = \lceil n/p \rceil$ rows (or columns)

Bi-dimensional Block Distribution

Example of 2-D distribution by blocks for a matrix of size $m \times n = 14 \times 11$ among 6 processes organized in a 3×2 grid



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 process 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 ($k = 0, 1, \ldots$) a new value is obtained $A^{(k+1)} = \left(a_{i,j}^{(k+1)}\right)$, $i, j = 0, \ldots, n-1$, where

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

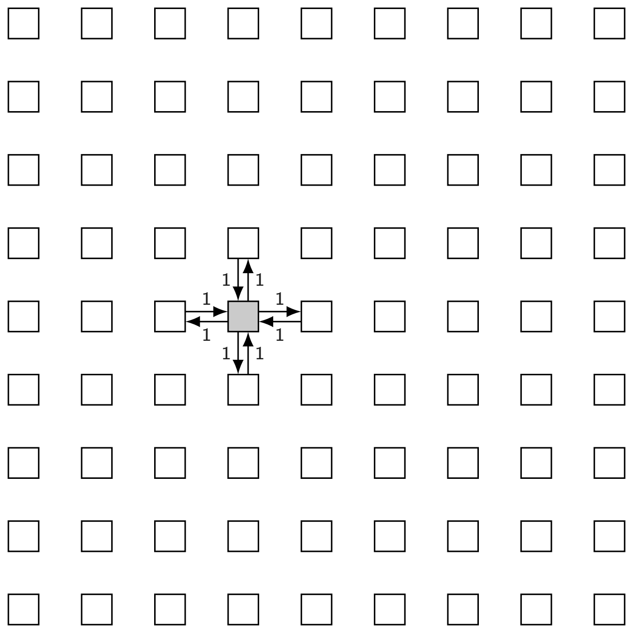
and certain boundary conditions

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

Example: Finite Differences (2)

Without merging

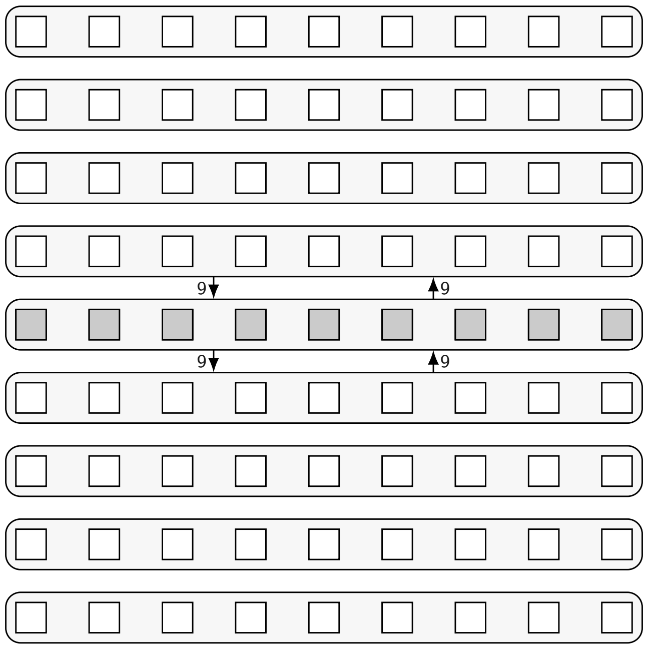
- 4 messages per task (1 element each)
- 288 total messages, 288 elements transferred



Example: Finite Differences (3)

Uni-dimensional merging

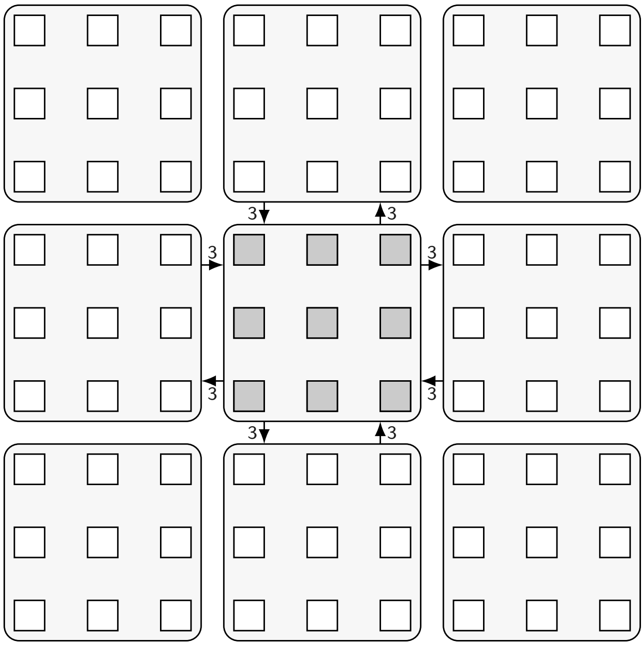
- 2 messages per task (9 elements each)
- 16 total messages, 144 elements transferred



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
- Goal is to maximize computation and minimize communication

Volume-surface effect

- The computational load increases proportionally to 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					
0	1	6	7	12	13

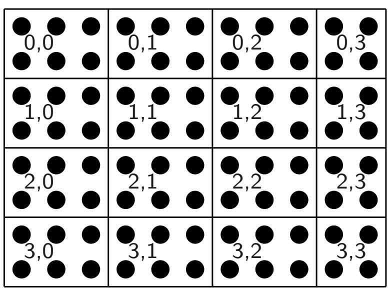
P_1			
2	3	8	9

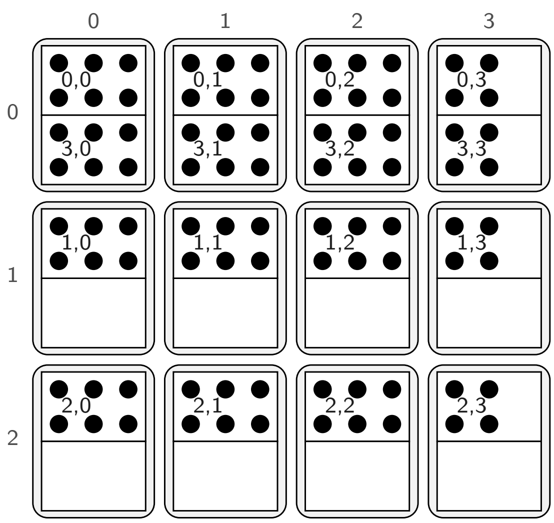
P_2			
4	5	10	11

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

Bi-dimensional Cyclic Distribution

Example 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 or heuristic approaches are known

Examples:

- *Binomial tree structure*
- *Hypercube structure*

Example: Vector Sorting (1)

Given a vector of numbers (a) sort it into another vector (aor)

```
SUB mergesort(a,aor,n)
IF n<=k
  sort(a,aor,n)
ELSE
  m = n/2
  a1 = a[0:m-1]
  a2 = a[m:n-1]
  mergesort(a1,aor1,m)
  mergesort(a2,aor2,n-m)
  merge(aor1,aor2,aor)
END
```

Simplified dependency graph:

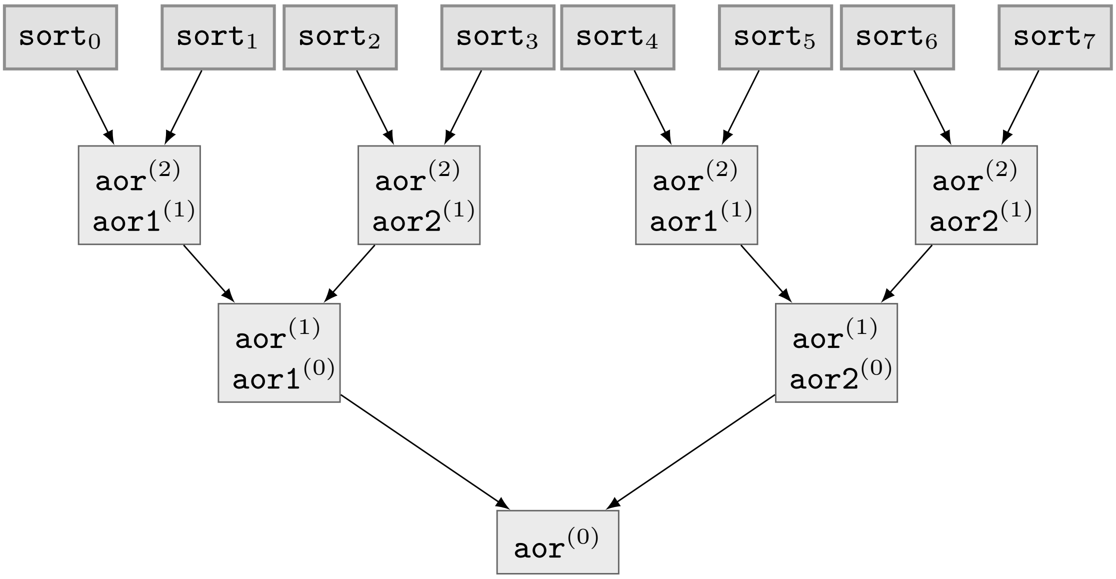
$$\begin{tikzcd} \boxed{\texttt{aor1}^{(\ell)}} \arrow[dr] & & \boxed{\texttt{aor2}^{(\ell)}} \arrow[dl] \\ & \boxed{\texttt{aor}^{(\ell)}} & \end{tikzcd}$$

Parallelization strategy:

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

Example: Vector Sorting (2)

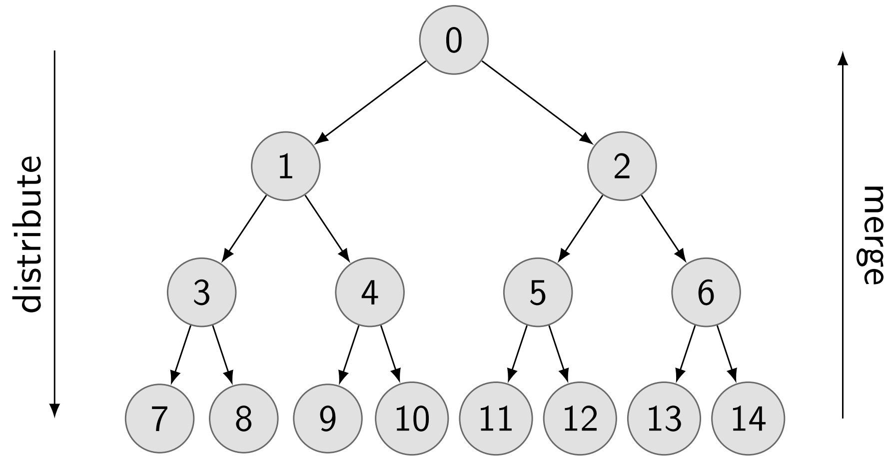
Complete dependency graph for $n = 8k$



Example: Vector Sorting (3)

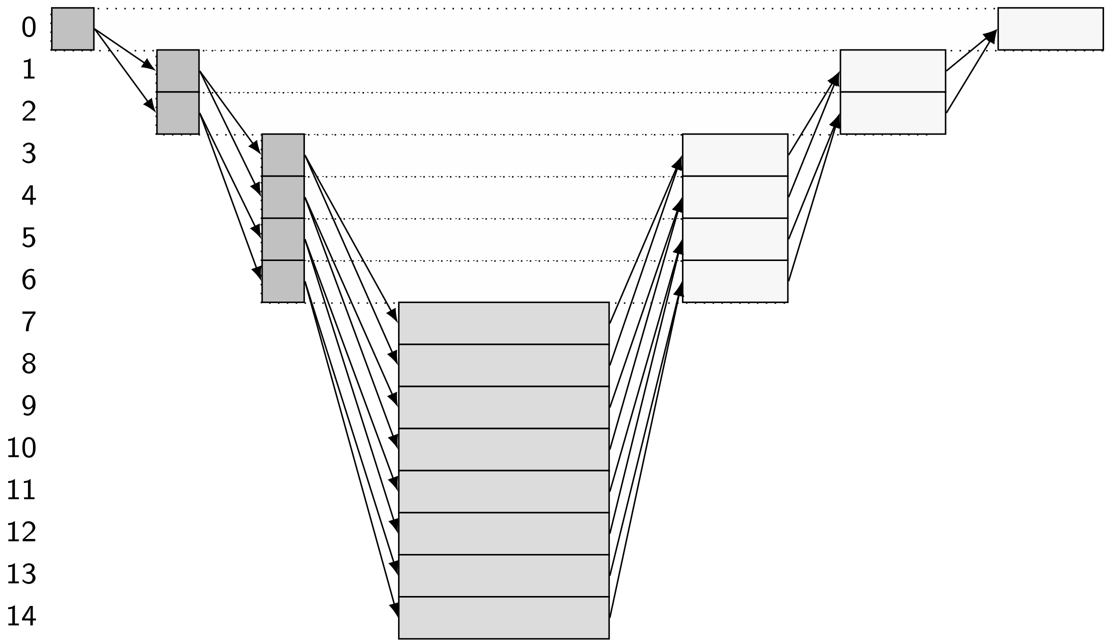
Assuming p processors with a topology of a binary tree

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



No need to have a physical tree topology, it is emulated

Example: Vector Sorting (4)



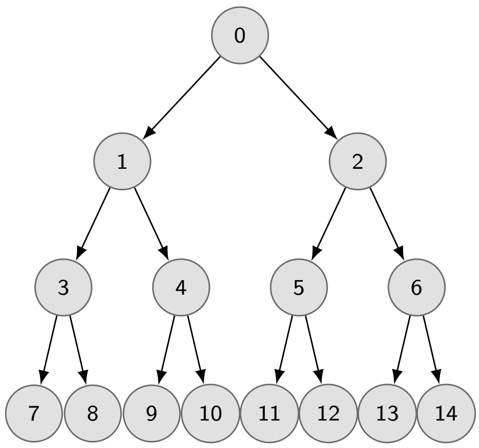
Better efficiency if processors are “reused”

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

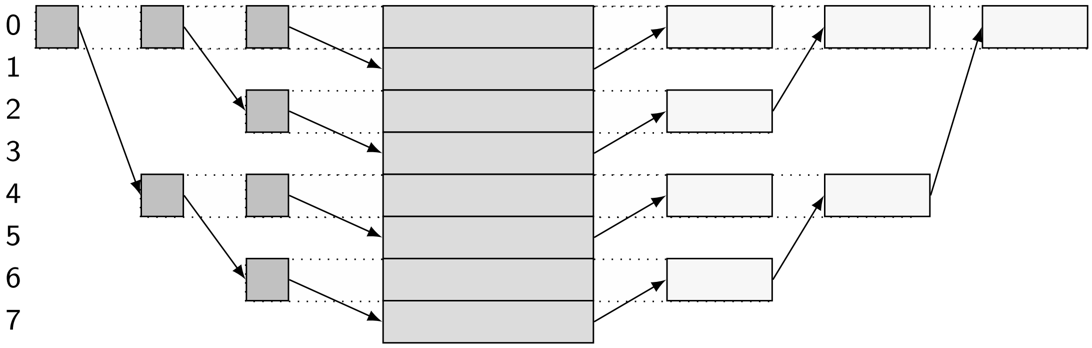
Example: Vector Sorting (5)

Message passing: as many processors 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)
    merge(aor1,aor2,aor)
  ELSE
    sort(a,aor)
  END
  IF pr <> 0
    send(aor,(pr-1)/2)
  END
```



Example: Vector Sorting (6)



Dynamic Workload Balancing Schemes

When the static assignment is not feasible

- The tasks obtained in the decomposition are data structures that represent sub-problems
- Sub-problems are kept in a collection and dispatched to the different processes
- The solution of a sub-problem may lead to the dynamic creation of more sub-problems
- These schemes require a mechanism for detecting termination

Types:

- Centralized scheme, a master process manages the collection of sub-problems and dispatches them
- Distributed scheme, without a master; load balancing is not trivial