This part deals with systematic methods and techniques for the development of high-performance parallel and distributed applications. The goal of this set of methods and techniques is to achieve a good trade-off between two contrasting requirements: programmability and portability, on one side, and performance and efficiency, on the other side. The methodology is based on two interrelated issues: structured parallelism paradigms and cost models. Structured parallelism paradigms aim to provide standard and effective rules for composing parallel computations in a machine independent manner. Cost models are defined for the performance evaluation and prediction, and are a fundamental tool for reducing the complexity in parallel software design. Cost models will take into account all aspects related to calculation and communication parameters of the applications and of the underlying architectures. For this purpose, mathematical techniques in the area of Queueing Theory and Queueing Networks are adopted.

Contents

1. Introduction to High Performance Computing ......................................................... 4

2. Structured parallel computations ............................................................................. 6
   2.1 Level structuring, cost model, and abstract architecture ..................................... 6
   2.2 Issues in parallel program design: computation graphs and parallelism paradigms ........... 9

3. Communication mechanisms and their run-time support .......................................... 13
   3.1 Concurrent language definition ............................................................................ 13
   3.1.1 Structure of parallel programs .......................................................................... 13
   3.1.2 Typed communication channels ....................................................................... 14
   3.1.3 Communication forms ...................................................................................... 16
   3.1.4 Non-determinism control in communications .................................................. 16
   3.2 Interprocess communication run-time support ...................................................... 18
   3.2.1 Process support and shared data structures ...................................................... 19
   3.2.2 “Generic” implementation .............................................................................. 20
   3.2.3 Direct copy into the target variable in asynchronous communications with suspended receiver .................................................. 21
   3.2.4 Synchronous communication: “peer” implementation .................................... 23
   3.2.5 “Zero-copy” communication .......................................................................... 24
   3.3 Communication latency ........................................................................................ 26
4. Parameters for performance evaluation ................................................................. 30
   4.1 Service time and bandwidth, latency, and completion time .................................. 30
   4.2 Efficiency and scalability .................................................................................. 32
   4.3 Masking communication latency: communication-calculation overlapping ................ 34

5. Stream computations studied as queueing systems and queueing networks ................. 37
   5.1 Queueing systems and utilization factor ............................................................ 37
   5.2 Fundamental properties of computations studied as queueing networks ................ 40
      5.2.1 Interdeparture time .................................................................................. 40
      5.2.2 The “multiple clients” theorem ................................................................... 40
      5.2.3 The “server partitioning” theorem ............................................................... 41
      5.2.4 Examples of acyclic graph analysis ............................................................. 44
   5.3 Ideal and effective bandwidth of modules and graph computations .......................... 46
   5.4 Transformation of bottlenecks and optimal parallelism degree ............................... 48
   5.5 Detailed example of acyclic graph analysis and optimization .................................. 50

6. General characteristics of parallel paradigms ........................................................... 54

7. Pipeline paradigm .................................................................................................... 56
   7.1 Cost model ......................................................................................................... 56
   7.2 Partitioning of functions and data ...................................................................... 58
   7.3 Function replication and data partitioning: loop unfolding .................................... 59

8. Stream-equivalent computations ............................................................................... 62

9. Optimal communication grain ................................................................................... 63

10. Data-flow paradigm .................................................................................................. 65

11. Farm paradigm ......................................................................................................... 68
    11.1 Farm cost model .............................................................................................. 70
    11.2 Farm vs pipeline and data-flow ....................................................................... 73
    11.3 Farm with internal state .................................................................................. 74

12. Functional partitioning with independent workers .................................................... 76

13. Collective communications ....................................................................................... 77
    13.1 Multicast .......................................................................................................... 77
    13.2 Scatter ............................................................................................................. 78
    13.3 Gather .............................................................................................................. 78

14. Data parallel paradigm ............................................................................................. 79
    14.1 The Virtual Processors approach to data parallel program design ....................... 80
    14.2 Map cost model .............................................................................................. 86
1. **Introduction to High Performance Computing**

High Performance Computing (HPC) is an ICT area that studies hardware-software architectures and applications characterized by requirements for high processing bandwidth, low response time, high efficiency and scalability.

Many scientific, commercial and technological disciplines benefit from HPC systems and tools: physics, chemistry, earth sciences, biology, medicine, engineering, environmental control, emergency management, high-bandwidth communication and networking, intelligent sensors, image and signal processing, multimedia, finance and economy, and so on.

HPC is synonymous with parallel and distributed processing. Any HPC product is manly based on parallelism exploitation:

- at the instruction level (IPL): high-performance pipeline, superscalar and multithreading CPUs,
- at the process level: shared memory multiprocessors (SMP, NUMA, COMA) and distributed memory multicomputers (PC/workstation clusters, Massively Parallel Processors, Processor Farms, Data Centres, Clouds). Typical configurations range from few processors to tens, hundreds, and thousands processors/computers.

The following figure shows simplified schemes for *shared memory multiprocessors* and for *distributed memory multicomputers*:
Currently, an important technological evolution/revolution is on going: multi-manycore components, or on chip multiprocessors, will replace (are replacing) uniprocessor-based CPUs for both the scientific and the commercial market. The so called “Moore law” - according to which the CPU clock frequency doubles about every 1.5 years - since 2004 has been reformulated in: the number of cores (CPUs) on a single chip doubles every 1.5 years. This fact has enormous implications on technologies and applications: in some measure, all hardware-software products of the next years will be based on parallel processing. In this evolution/revolution, an important role is played by the trends in high-bandwidth, low-latency interconnection networks, especially on-chip networks.

In order to exploit this clear trend, parallel programming and parallel applications development tools are rapidly becoming first-class citizens, although currently a wide gap still exists between parallel architecture and parallel programming maturity.

This course, together with the companion course “Distributes Systems: Paradigms and Models”, provides fundamental methods and tools for parallel programming and parallel applications development. At the same time, parallel architectures are studied at the state-of-the-art and according to the research trends, and a strong relationship is established between parallel application development and parallel architectures.

Start-up bibliography on general sources and on Computer Science Department research:

2. Structured parallel computations

2.1 Level structuring, cost model, and abstract architecture

The following figure is a synthetic view of the course approach:

- Applications should be designed at the highest level by means of formalisms and tools that are fully independent from the machine architecture and from the mechanisms at the process level;
- A concurrent language at the process level is the intermediate language into which the abstract parallel applications are compiled and/or interpreted. It could be considered the “HPC assembler” language. At this level, several optimizations are applied according to the cost model that characterizes the high-level formalism and methodology, in order to evaluate and to predict performance parameters, like processing bandwidth, completion time, latency, response time, efficiency, scalability, and so on.

Currently, the most common situation consists in developing parallel applications directly at the process level by means of sequential languages plus message-passing libraries (e.g. MPI) or shared-variable libraries (e.g. OpenMP). This approach is not suitable for a high-level, portable and “productive” methodology, and there is a clear trend to overcome it. However, even when the only available tools are process level libraries, it is important that the methodology that we’ll study for the application level
is equally applied, i.e., the programmer can adopt the high-level methodology to define parallel applications, and can provide a sort of “compilation by hand” into the process version;

- the run-time support of the process level (concurrent language or libraries) is different for each distinct parallel architecture at the assembler-firmware level. For example, the run-time support for multiprocessors exploits the physically shared memory intensively, while in a multicomputer proper mechanisms are adopted to establish an efficient cooperation among distributed memory nodes. Thus, for each parallel architecture, a proper set of run-time libraries and optimization techniques is used by the compiler of the concurrent language.

A key concept of our methodology is the following:

- the cost model for parallel applications depends in large part on the paradigms adopted for structuring the parallel computation, that are architecture independent,
- while the impact of the underlying architecture can be concentrated on some parameters expressing the cost of process cooperation mechanisms.

For example, the interprocess communication latency $L_{com}$ is a key parameter in our cost models. It measures the latency for completing an interprocess communication, i.e., to copy the message into the target variable and to perform all the needed synchronization and low-level scheduling actions. We’ll see that $L_{com}$ captures a large number of characteristics of the underlying architecture, like

- shared vs distributed memory
- interconnection structures
- process scheduling
- memory hierarchies
- features of CPUs, coprocessors, and other processing units

and so on.

Other parameters of the cost model, notably the calculation time, depend on the above characteristics.

In conclusion, the architectural impact can be concentrated in few, well-recognized parameters of the cost model (communication latency, calculation time, and few others).

Moreover, a strength point of the structured parallelism methodology is the parametric nature of the cost model. That is, the parameters of the cost model are functions of other parameters depending not only on the application and on the architecture, but also on the amount of resources dedicated to the application execution. Notably, the effective parallelism degree of the program is a parameter that impacts other key parameters (communication latency, calculation time, etc). The dependence of the performance parameters of a structured parallel program on the parallelism degree is known in the cost model. In conclusion:

- we can design and compile a parallel application once for all in a parametric way;
- in order to execute the application on a given machine configuration, it is sufficient to allocate the resources (e.g., processing nodes) at loading time, without modifying the application and without recompiling it;
• the performance parameters will be parametrically dependent on the current system configuration.

In order to take into account all the concepts and features discussed above, we utilize a typical concept in Computer Science: **abstract architecture**.

An abstract architecture is able to capture the essential characteristics and features of several, different physical parallel architectures:

The specificity of each individual physical architecture is expressed by the value of some parameters of the cost model (communication latency, calculation time, etc.).

As it is typical of the concept of abstract architecture, the compiler “sees” the abstract architecture, and the related cost model, to transform and to optimize the parallel applications.

If the interprocess cooperation model is the local environment (message-passing) one, a reasonable definition of the abstract architecture, both for shared and for distributed memory parallel machines, is the following:
1. *a distributed memory architecture*, consisting of as many processing nodes as the processes of the parallel program are. That is, *each process of the parallel program is allocated onto an independent processing node*;

2. every node has the same characteristics of the (abstract) uniprocessor architecture of the processing nodes;

3. nodes interact through a *fully interconnected network*, in which each node is connected by a dedicated link to any other node. An interprocess communication channel is implemented by the physical channel connecting the corresponding processing nodes;

4. if the concrete architecture support *calculation-communication overlapping*, then it is supported by the abstract architecture too;

### 2.2 Issues in parallel program design: computation graphs and parallelism paradigms

The following figure illustrates a computation graph example for a complex application consisting in data stream acquisition from several sources, data processing (Filters, Convolution), and result visualization:

![Computation Graph Example](image)

- **Processing and visualization of images acquired from several sources: satellites, sensors, plants, instruments, internet, etc.**
- **Application examples:** prediction of natural emergencies, e.g. landslips, floods, fires, pollutions, etc.

Several tasks shown in the *application graph* (workflow) are complex and time consuming, and require a high processing capability in order to be exploited in real-time, or within a given deadline imposed by the application specifications.

The goal of our methodology is:
to understand which sequential modules are bottlenecks for the application performance,

to transform each bottleneck modules into a parallel computation which is functionally equivalent, i.e. without modifying the semantics nor the interfaces with respect to the sequential version,

to compare some alternative versions of the parallel transformation,

to evaluate the performance according to a cost model,

to design the executable version of the parallel application for the target parallel architecture(s).

An application can operate on values organized as streams or as single data values. A stream is a possibly infinite sequence of values with the same type, e.g. a stream of images represented as matrices. For stream-based computations, we are interested in parameters such as service time $T$ (inverse of processing bandwidth, or throughput) and completion time $T_c$ (the time needed to complete the computation on all the stream elements, provided that the stream is of finite length $m$). We know that the following approximate relation holds:

$$T_c \sim m T$$

provided that the stream length is much greater than the parallelism degree.

It is possible that the application operates on a single data value, instead of on a sequence of values, e.g. on a single image. This case occurs also when, though in a stream-based computation, the temporal distance between two consecutive stream elements is much greater than the computation latency on each element. In this case, service time (bandwidth) is not defined, while completion time is still the performance parameter of interest.

This example shows the main characteristics of our parallelization methodology:

a. applications are expressed as computation graphs, whose nodes are processing modules interacting through streams or single values;

b. we have to recognize possible bottleneck modules in the graph;

c. in order to eliminate, or at least to reduce the effects of, the bottlenecks, we have to parallelize each bottleneck module according to some parallelism paradigms that are typical of the methodology;

d. the final result is a computation graph, functionally equivalent to the given one, in which some nodes are transformed through an internal structured parallelization, or intra-node parallelism. Moreover, also inter-node parallelism is exploited;

e. parallelism paradigms are characterized by a cost model that allow us to evaluate the performance measures of the single modules and of their graph composition.

These issues are characterized by several parallelization problems that are NP-hard, like:

- decomposition of a sequential computation into an equivalent collection of (many) modules able to operate in parallel,

- load balancing of the parallel modules,

- overlapping of calculation and communication,
• mapping of modules onto processing resources (processing nodes, memories, communication facilities, and so on),
• scheduling of processing resources at run-time,
and others.
The goal of the methodology is to reduce the complexity of the parallelization problems in such a way that, through the introduction of some constraints, they become tractable.

The most powerful approach to this methodology is based on the concept of parallelism forms, also called parallelism paradigms, or skeletons. They are schemes of parallel computations that recur in the realization of many real-life algorithms and applications. Parallelism paradigms have the following features:

1. they are characterized by constrains in the parallel computation structure,
2. they have a precise semantics,
3. they are characterized by a specific cost model,
4. they can be composed each other to form complex graph computations.

We consider the following parallel paradigms:

1) stream-parallel paradigms
   - Farm
   - Pipeline
   - Data-flow

2) data-parallel paradigms
   - Map
   - Reduce, parallel prefix
   - Stencils
   - Divide & Conquer

Class 1) is defined to operate on data streams only, while class 2) is able to operate on single values and also on streams. For example,

• the farm paradigm corresponds to the replication of the same function (only “pure” functions are acceptable) so that distinct stream elements can be processed by distinct modules in parallel;
• the data-parallel paradigms correspond to the replication of the same functionality (also a computation with state is acceptable) and to the partitioning of data, so that distinct modules are able to apply the same operations to distinct data partitions in parallel.
As far as **composability** is concerned, let us consider the following graph computation:

![Graph Diagram]

Each module can be parallelized according to *its own parallelism paradigm*. Stream parallel and data-parallel paradigms are composable in stream computations. The *semantics* and the *cost model* of the whole computation are obtained as proper compositions of the individual semantics and cost models.

Some parallel paradigms can be expressed as compositions of other parallel paradigms. In our case, the Divide & Conquer paradigm will not be assumed as primitive, since it will be implemented by compositions of stream- and data-parallel paradigms with better performance results.

The programming approach based on parallel paradigms is called **structured parallel programming**, at the light of the analogy with well known sequential languages that allow the programmer to express a program as the composition of a limited amount of constructs with the same properties 1–4.

As in sequential programming, once that the universality of the adopted paradigms has been proved (see the Cole’s definition of skeletons), the basic performance problem arises: *how to be reasonably sure that a given set of parallelism paradigms is “optimal”?* Here the design experience plays a central role. The research on structured parallel programming has shown that the considered paradigms are reasonably optimal, especially when their composition has the form of a generic *graph* and each graph node is internally parallelized by a parallel paradigm (ASSIST experience, University of Pisa).
3. Communication mechanisms and their run-time support

In order to develop and to exemplify our methodology, we need to refer to some specific mechanisms for process cooperation. We’ll adopt a simple, didactic concurrent language based on the local environment, or message-passing, cooperation model. This choice is not limiting, because of the proved duality of local environment and global environment models. Moreover, the didactic nature of the language is just for clarity and for simplification of concept presentation and understanding. Such a language is defined according to the general semantics of Hoare’s Communicating Sequential Processes (CSP), which is adopted by any existing message-passing library, e.g. MPI.

Let us call LC this concurrent language (“Linguaggio Concorrente”).

3.1 Concurrent language definition

3.1.1 Structure of parallel programs

A LC parallel program is a collection of processes, declared as:

\[
\text{parallel} \ < \ \text{list of unique process names} >; < \ \text{possible declarations of parametric names}>; \\
< \ \text{possible declarations of parametric data types}> \\
< \ \text{process definition} >; \\
\ldots \\
< \ \text{process definition} >; \\
\]

The parallel command cannot be used inside process definitions, i.e. processes cannot be nested each other.

A unique process name can be any character string. The process definition has a simple structure:

\[
< \ \text{unique process name} > :: \\
< \ \text{declarations} > \\
< \ \text{code} > \\
\]

Processes can be defined parametrically using free variables. For example, a unidimensional process array is expressed as:

\[
\text{parallel} \ \text{PROC}[N]; \ \text{int} \ A[N]; \\
\text{PROC} \ [j] :: \ldots \ \text{int} \ A[j]; \ldots; \ A[j] = \ F (\ldots, \ A[j], \ldots); \ \ldots \\
\]

In this case the A[N] components are partitioned among processes PROC[0], ..., PROC[N−1], and each of them declares and uses, as local variable, the array element corresponding to its own index.
Process arrays are a powerful, yet simple, mechanism to express computation that are replications of the same code, possibly with a parametric partitioning of data types: a typical property of structured parallel paradigms.

The sequential part of LC is a typical imperative C-like language. In the following we’ll use an algorithmic pseudo-language with assignment (=), control constructs if-then-else, case, for, while, command sequences enclosed in brackets {...}, procedures and functions, as well as typical static data types.

### 3.1.2 Typed communication channels

In LC semantics, communication channels have a type: it is the type of the messages, which can be sent over the channels, and the type of the target variables, to which received messages can be assigned:

Channel types are recognized and checked at compile time. Messages cannot contain pointers. Channels have unique names, expressed as character strings.

The LC commands for operating on communication channels are the communication mechanisms we wish to study: send and receive commands are the basic message-passing primitives, whose syntax is shown in the figure. Two corresponding send and receive commands refer the same channel.

Messages and target variables can be expressed in the form of tuples. For example (operation, value1, value2, index) is a legal message or target variable.

Communication channels are declared by the process that use them, distinguishing between input channels and output channels. For example:

```plaintext
parallel A, B;
A :: … channel in go, ch2; channel out compute; …
    { … send (compute, …); …; receive (go, …); …; receive (ch2, …); … }
B :: … channel in compute, ch3; channel out go, ch4, ch5; …
    { send (go, …); …; receive (compute, …); … }
```

In this example, channel names are constant. Alternatively, in LC a channel can be identified by a variable: the language types include the channlename type, declared using the var keyword. For example:
channel in ch1, var ch2; channel out var ch3, var ch4, ch5…

The possible values that can be assumed by a channelname variable are determined at compile-time through a static analysis of the parallel program.

Assignment operations are defined on channelname variables. This allows the programmer to manage and control the communication channels parametrically, notably using channel names in messages and in target variables.

For example:

```
parallel A, B, C, D;
A :: … channel in chA; channel out chD; … { … send (chD, (chA, valore)); …; receive (chA, variabile); … } …
B :: … channel in chB; channel out chD; … { … send (chD, (chB, valore)); …; receive (chB, variabile); … } …
C :: … channel in chC; channel out chD; … { … send (chD, (chC, valore)); …; receive (chC, variabile); … } …
D :: … channel in chD; channel out var ch_out; … { … receive (chD, (ch_out, valore)); …; send (ch_out, variabile); … }
```

corresponding to the following computation graph:

In this example, processes A, B, C act as “clients” of a “server” process D. Each client sends onto chD channel a message (the service “request”) including the name of the channel from which the client wishes to receive a message from the server (the service “reply”). The ch_out variable of D is a target variable that is assigned the value chA, or chB, or chC. In this example ch_out is used in a send command to parametrically identify an output channel.

The following example shows how channelname variable and structured channels can be used to parametrically select the input and output channels:
parallel CLIENT[N], SERVER; channel request [N];

CLIENT[j] :: … channel in ch_result, …; channel out request[j], …;
  { … send (request[j], (ch_result, x)); …; receive (ch_result, y); …}

SERVER:: … channel in request [N], …; channel out var ch_out; …
  { … for (j = 0; j < N; j++)
    { receive (request [j], (ch_out, x)); y = F(x); send (ch_out, y) }; …}

3.1.3 Communication forms

The LC communication forms are:

1) symmetric or input asymmetric, which are not distinguished by specific declarations. The channelname mechanism expresses a form of symmetric or asymmetric parametric communication;

2) synchronous or asynchronous The asynchrony degree $k$ of a channel is a non-negative integer constant $k \geq 0$, where the case $k = 0$ corresponds to synchronous communication. The asynchrony degree of a channel is associated to the channel declaration in the destination process only; if $k = 0$ it can be omitted. For example:

channel in ch1 (4), ch2 (1), ch3, ch4(0); channel out …

Channels ch1, ch2, ch3, ch4 have asynchrony degree equal to 4, 1, 0, 0 respectively.

In an asymmetric channel with asynchrony degree $k$, every sender process has an asynchrony degree $k$ with respect to the destination process.

3.1.4 Non-determinism control in communications

Alternative guarded commands (ECSP concurrent language, University of Pisa) are used for this fundamental task in message-passing computations.

An alternative command expresses the possibility to receive a message from any channel belonging to a given channel set, which in general can be variable and controllable by program. Moreover, a priority mechanism can be applied when more than one channel belonging to the specified set are ready for communication.

The syntax is:

```
alternative
  { priority (pr_1), pred_1 (…), receive (…) do { … command_list_1 …}}
or priority (pr_2), pred_2 (…), receive (…) do { … command_list_2 …}
...
or priority (pr_n), pred_n (…), receive (…) do { … command_list_n …}
}
```

A guard as

```
priority (pr_i), pred_i (…), receive (…)
```
contains a *priority* (value of an integer variable), a *local guard* consisting in the evaluation of a predicate on the process internal state, and a *global guard* expressed by a *receive* command. *One or more of these three elements may be absent.*

A guard is said to be

- **verified** if $\text{pred}_i$ is true and the *receive* command can be executed;
- **suspended** if $\text{pred}_i$ is true, but the *receive* command cannot be executed;
- **failed** if $\text{pred}_i$ is false.

If one or more guard are verified, the guard having the highest priority is selected, the *receive* command is executed, then the command list is executed, and the command terminates. If more than one verified guard have the same priority, one of them is selected according to a *random* strategy which is invisible to the programmer (it is implemented in the run-time support).

If *all* guards are suspended, the process is suspended in the alternative command. It will be awaked as soon as one or more guard will become verified.

If *all* guards are failed, the alternative command terminates without executing any *receive* command nor command lists.

The so called *repetitive command* is not primitive in LC, however it is emulated by an alternative command inside a loop. For example:

```plaintext
parallel A, B, C, DEMON;
A :: channel in chA (1); channel out chD_A; …
   { … send (chD_A, (chA, valore)); … ; receive (chA, variabile); … } 
B :: channel in chB (1); channel out chD_B; …
   { … send (chD_B, (chB, valore)); … ; receive (chB, variabile); … } 
C :: channel in chC (1); channel out chD_C; …
   { … send (chD_C, (chC, valore)); … ; receive (chC, variabile); … } 
DEMON :: channel in chD_A (1), chD_B (1), chD_C (1); channel out var ch_out; int pr[3]; T x, y ;
   { for (i = 0; i < 3; i++) do pr[i] = i ;
     while (true) do
       alternative
       { priority (pr[1]), receive (chD_A, (ch_out, x)) do
         { y = F(x); send (ch_out, y); for i = 1 to 3 do pr[i] = (pr[i]) mod 3 + 1 } 
       or priority (pr[2]), receive (chD_B, (ch_out, x)) do
         { y = F(x); send (ch_out, y); for i = 1 to 3 do pr[i] = (pr[i]) mod 3 + 1 }         
       or priority (pr[3]), receive (chD_C, (ch_out, x)) do { y = F(x), send (ch_out, y) } 
         { y = F(x); send (ch_out, y); for i = 1 to 3 do pr[i] = (pr[i]) mod 3 + 1 } 
     }
   }
```
The classical example of buffer management is shown in the following. The buffer object and the management policy are implemented by proper data structures, by the functions `empty_buffer, full_buffer, get` and by the procedure `put`.

The example is meaningful since it illustrates the power of non-determinism control through the mechanism of local + global guards.

The first version is an infinite cycle (demon process):

**Version 1: demon process**

```plaintext
BUFFER_MANAGER::

    channel in ch_prod (1), ch_cons (1); channel out var ch_out; item x; …

    { while true do
      alternative
      { not full_buffer ( ), receive (ch_prod, x) do put (x)
      or not empty_buffer ( ), receive (ch_cons, (ch_out)) do
        { x = get ( ); send (ch_out, x) }
      }

    }
```

In the second version, the process termination can be caused by information received by other partner processes. The command list `TERM` is the termination handler:

**Version 2: termination handling**

```plaintext
BUFFER_MANAGER::

    channel in ch_prod (1), ch_cons (1); channel out var ch_out; item x; boolean end;

    …

    { end = false;
      while not fine do do
      alternative
      { not full_buffer ( ), receive (ch_prod, (end,x)) do { put (x); if end then TERM }
      or not empty_buffer ( ), receive (ch_cons, (ch_out)) do
        { x = get ( ); send (ch_out, x) }

      }

    }
```

### 3.2 Interprocess communication run-time support

In order to build the executable version of a parallel program, LC compiler utilizes a set of `run-time libraries` belonging to the interprocess communication run-time support. We’ll study the run-time support of basic `send` and `receive` commands. `send` and `receive` libraries are implemented as `procedures` with the following signatures:

- `send (ch_id, msg_addr)`
- `receive (ch_id, vtg_addr)`
where parameter ch_id is a unique constant channel identifier, and parameters msg_addr, vtg_addr are logical base addresses of the message data structure and of the target variable data structure respectively in the sender logical address spaces and in the receiver logical address spaces. Run-time support procedures are executed with disabled interrupts.

We consider the case of symmetric, synchronous or asynchronous, deterministic channels (i.e. not referred in alternative commands), for uniprocessor architectures. In Part 2 the run-time support for multiprocessors and multicomputers will be derived by proper modifications to the uniprocessor run-time support.

### 3.2.1 Process support and shared data structures

The run-time support operates on proper data structures, which can be private of, or shared by, the sender and receiver processes.

The basic data structure is the channel descriptor (CH), shared by the sender and the receiver process. We’ll see several implementations of send-receive, for each of which a different CH structure will be provided. The compiler has several versions of run-time support libraries and, for each channel, selects one of them in order to introduce optimizations depending on the application and/or on the underlying architecture.

The Channel Table is a private data structure (TAB_CH), used to obtain the CH logical address as a function of the channel identifier ch_id.

As usually, each process has a Process Descriptor (PCB), shared with all the other processes, containing utility information (internal state, pointer to the Ready List, to the Channel Table, to the Relocation Table, and so on).

The following figure illustrates the concept of shared data structures at the run-time support level:
It is worth noting the definition of shared objects:

- let $S$ be an object shared by process A and by process B. This means that $S$ belongs to both logical address spaces of A and B;
- let $S_{log\_addr\_A}$ and $S_{log\_addr\_B}$ the logical addresses of $S$ in the logical address space of A and B respectively. In general, $S_{log\_addr\_A} \neq S_{log\_addr\_B}$, i.e. $S$ can be allocated in different areas of A and B virtual memories;
- let $S_{phys\_addr}$ the unique physical address of $S$. That is, in order to be shared, $S$ exists in single copy in main memory;
- let $\mu_A$ and $\mu_B$ the address translation functions of A and B respectively;
- then, the following relation holds:

$$\mu_A(S_{log\_addr\_A}) = \mu_B(S_{log\_addr\_B}) = S_{phys\_addr}$$

Finally, let us remember the typical organization of process low-level scheduling, based on the classical process progress states and transitions, as shown in the following figure:

![Process states and transitions diagram](image)

### 3.2.2 “Generic” implementation

The first version we study is valid both for synchronous and for asynchronous channels. For this reason, with this version we are not able to introduce optimizations, which instead will be introduced in the other versions.

The channel descriptor is a data structure containing the following fields:

- **sender_wait, receiver_wait**: boolean variables, denoting the possible waiting condition of the sender or of the receiver process;
- **message length, $L$**, expressed as number of words. This number corresponds to information size for that channel type.
- **buffer**: a FIFO queue of $N = k+1$ positions, each one of $L$ words. The queue is implemented as a circular vector with insertion and extraction pointers (indexes). The sender process inserts the message into the queue and, if the queue becomes full, then it passes into the waiting state. The solution with $N = k+1$ avoids that the sender process re-executes the send primitive once awakened and resumed into
execution; instead, it is resumed at the return logical address of the send procedure. In this first version, the receiver, when the empty buffer condition is found, is suspended at the logical address of the receive procedure itself (i.e., the receive procedure is re-executed when the receiver execution is resumed);

- sender_PCB_ref, receiver_PCB_ref: references to sender PCB and receiver PCB. The implementation of such references (logical addresses, or unique identifiers, or physical addresses, or capabilities), as in any other case of indirectly referred shared data structures (“shared pointers”), will be studied in a subsequent Section.

The pseudo-code of send and receive procedures is the following:

send (ch_id, msg_address) ::
  CH address =TAB_CH (ch_id);
  put message value into CH_buffer (msg_address);
  if receiver_wait then { receiver_wait = false;
    wake_up partner process (receiver_PCB_ref) ;
  }
  if buffer_full then { sender_wait = true;
    process transition into WAIT state: context switching }

receive (ch_id, vtg_address) ::
  CH address =TAB_CH (ch_id);
  if buffer_empty then { receiver_wait = true;
    process transition into WAIT state: context switching }
  else get message value from CH buffer and assign it to target variable (vtg_address);
  if sender_wait then { sender_wait = false;
    wake_up partner process (sender_PCB_ref) }

It is worth noting that the interprocess communication run-time support does not consist merely in message buffering and copying (“real communication”): it contains also synchronization operations and low-level scheduling operations.

3.2.3 Direct copy into the target variable in asynchronous communications with suspended receiver

The initial version can be optimized in several respects.

First of all, it is complicated by the fact that the same “generic” algorithms covers both synchronous and asynchronous communication. In the asynchronous communication at most one of the partner process can be in waiting state, while in the synchronous case a awakened process can find the partner itself in the waiting state.

From now on, we will distinguish distinct libraries, one for synchronous and the other for the asynchronous communication.

Let us consider the asynchronous communication (k ≥ l).
The most evident source of inefficiency is when the send procedure finds the receiver process in the waiting state. In this case, the double copy of message – first into the buffer queue (during send), then into the target variable (during receive) – is unnecessarily time-consuming, especially for long messages. In this situation (suspended receiver), the optimization is simply the following: the send procedure provides to copy directly the message into the target variable, i.e. the send procedure performs also the receive task. Moreover, the receiver is suspended at the return address of receive procedure.

Notice that target variable becomes a shared data structure for the sender and the receiver process.

The channel descriptor has now the following structure:

- **wait**: boolean variable, assuming the true value if one of the two partners is suspended;
- **message length, L**: expressed as number of words.
- **buffer**: FIFO queue, as in the initial implementation;
- **vtg_ref**: reference to the target variable (indirectly referred shared data structure): written dynamically by the receiver when it is suspended;
- **PCB_ref**: reference to the PCB of the waiting process: written dynamically.

The pseudo-codes are the following:

```plaintext
send (ch_id, msg_addr)
    CH_address = TAB_CH (ch_id);
    if wait then { wait = false;
        copy message into the target variable (msg_addr, vtg_ref);
        partner wake-up (PCB_ref) }
    else { put message value into CH_buffer (msg_addr);
        if buffer_full then { wait = true;
            copy sender PCB_ref into CH;
            process transition into the waiting state}
    }

receive (ch_id, vtg_addr)
    CH_address = TAB_CH (ch_id);
    if buffer_full then { wait = true;
        copy receiver PCB_ref and vtg_ref into CH;
        process transition into the waiting state}
    else { get message value from CH_buffer and assign it to target variable (vtg_addr);
        if wait then { wait = false;
            partner wake-up (PCB_ref) }
    }
```
3.2.4 Synchronous communication: “peer” implementation

The previous solution is generalized to the synchronous communication, leading to a very efficient and elegant implementation. Now, the send and receive behaviour are dual: as soon as one of the two partners tries to execute the respective command, it is suspended; subsequently the other process copies directly the message into the target variable and awakes the suspended partner:

Now both the message and the target variable are shared data structures.

The channel descriptor does not contain any buffer - only synchronization and low level scheduling information - : 

- *wait*: as in the previous solution;
- *message length*: as in the previous solution;
- *val_ref*: reference to the message or to the target variable (indirectly referred shared data structures): written dynamically by the sender or by the receiver, respectively, when it is suspended;
- *PCB_ref*: as in the previous solution.

The dual algorithms for the synchronous send and receive run-time support are:

### send (ch_id, msg_addr) ::

```plaintext
CH_address = TAB_CH (ch_id);
if wait then {
    wait = false;
    Copy message into target variable (msg_address, VAL_REF);
    partner wake-up { PCB_REF } }
else { wait = true;
    copy message_reference and PCB_reference into VAL_REF and PCB_REF;
    process transition into WAIT state: context switching }
```

### receive (ch_id, vtg_addr) ::

```plaintext
CH_address = TAB_CH (ch_id);
if wait then {
    wait = false;
    Copy message into target variable (vtg_address, VAL_REF);
    partner wake-up { PCB_REF } }
else { wait = true;
    copy vtg_reference and PCB_reference into VAL_REF and PCB_REF;
    process transition into WAIT state: context switching }
```
3.2.5 “Zero-copy” communication

The optimizations studied in the previous Sections can be applied, in the most general case, to any asynchronous communication channel, as demonstrated by some advanced research libraries (VIA, Fast Messages): now we are able to reduce the number of message copies to just one, i.e. to the minimum, independently of the receiver progress state.

The basic principle for “zero copy” communication is shown in the following figure:

Multiple copies of every target variable are provided. That is, each time the receiver process refers the same target variable VTG, in fact it refers to a different instance of VTG in a circular way. The programmer or the compiler is able to structure the process according to this principle: in some cases, it is equivalent to realize a $k+1$ loop-unrolling in the receiver process.

A FIFO queue of $k+1$ target variable instances is defined in the receiver process address space, and they are shared (statically or dynamically) with the sender process. Equivalently, the channel descriptor contains, besides the information for synchronization and low level scheduling, a FIFO queue of references to the $k+1$ target variables:

- **wait**: as in the previous solution;
- **message length**: as in the previous solution;
- **buffer**: FIFO queue of tuples (references to target variable, validity bit);
- **PCB_ref**: as in the previous solution.

Notice that, by definition of this method, the receiver process works directly on the target variable instances (without copying them, of course). For this reason, a validity bit is associated to every target variable instance, in order to grant the mutual exclusion between sender and receiver. In fact, a critical race could occur when the sender tries to copy a message into a target variable instance and the receiver is still utilizing it.
the validity bit is reset in the receive command and is set again when the receiver process is no more willing to utilize the target variable instance. Thus a set_validity_bit primitive must be provided in the concurrent language;

- the sender is suspended if the validity bit of the target variable referred by the CH buffer is set.

The pseudo-code of zero-copy send run-time support is the following:

```plaintext
send (ch_id, msg_address) ::
  CH_address = TAB_CH (ch_id);
  if (CH_Buffer [Insertion_Pointer].validity_bit = 0) then
    { wait = true;
      copy reference to Sender_PCB into CH.PCB_ref
      process transition into WAIT state: context switching
    };
  copy message value into the target variable referred by
  CH_Buffer [Insertion_Pointer].reference_to_target_variable;
  modify CH_Buffer. Insertion_Pointer and CH_Buffer_Current_Size;
  if wait then
    { wait = false;
      wake_up partner process (CH.PCB_ref) }
  if buffer_full then
    { wait = true;
      copy reference to Sender_PCB into CH.PCB_ref
      process transition into WAIT state: context switching }
```

Notice that the validity bit can be eliminated if the following constraint is imposed: after each receive only the received target variable instance is used (and not any other instance).

Otherwise, the zero-copy implementation reduces the communication latency at the expense of the validity bit overhead, i.e. while the validity bit manipulation has a negligible effect, possible additional context-switchings can occur in the sender process. This effect can be minimized by adopting some proper strategies:

- a) a general strategy is to increase the asynchrony degree in order to reduce the probability of finding the validity bit set;
- b) moreover, there are some parallel program structures that are able to minimize (to eliminate) this problem owing to their implicit behaviour. For example, consider a farm structured parallel program: a scheduler process distributes the input stream tasks to a collection of executor processes according to an “on demand” load balancing strategy, i.e. an executor explicitly signals the scheduler its availability to receive a new task. Thus, it is impossible that the scheduler tries to write into a target variable instance while the executor is still working on it.

**Exercise**

Write an equivalent representation of the following benchmark, using zero-copy communication on a k-asynchronous channel:
Receiver ::

```c
int A[N]; int v; channel in ch (k);
for (i = 0; i < N; i++)
    { receive (ch, v);
```

The answer includes the definition of the `receive` procedure, and the receiver code before and after the `receive` procedure for a correct utilization of zero-copy communication.

### 3.3 Communication latency

The interprocess communication latency, $L_{com}$, is the average time needed to execute a complete communication, that is the time between the beginning of the `send` execution and the copy of the message into the target variable, including synchronization and low level scheduling operations.

Let us consider the implementation of a `send` command. Its latency, for a message of length $L$, can be expressed by the following formula:

$$T_{send} = T_{setup} + L \times T_{transm}$$

where

- $T_{setup}$ is the average latency of all the actions that are independent of the message length: synchronization, manipulation of buffer indexes, low level scheduling;
- $T_{transm}$ is the latency for a single iteration of the message copy loop, i.e. the latency to copy one word of the message.

A similar formula can be written for the `receive` latency in any implementation, except the zero-copy ones. We can assume that the $T_{setup}$ and $T_{transm}$ values are comparable, and very similar indeed, for `send` and `receive`. Thus, we can approximately write:

$$L_{com} = 2 \times T_{send} = 2 \times (T_{setup} + L \times T_{transm})$$

except for cases in which the message is copied directly into the target variable.

This latency is reduced to about half in the zero-copy run-time support. In fact, consider that the `receive` latency is negligible compared to the `send` latency, because the receive run-time support does not perform message copies. This approximation is as more valid as long the message is.

In the following, we assume that the **zero-copy communication latency** is given by the `send` latency only:

$$L_{com} = T_{send} = T_{setup} + L \times T_{transm}$$

Typical values for uniprocessor architectures are of the following orders of magnitudes:

- $T_{setup} = (10^2 - 10^3) \tau$
- $T_{transm} = (10^1 - 10^2) \tau$

For parallel architectures we must add one or more orders of magnitude, according to the characteristics of memory hierarchy, interconnection network, and so on (this analysis will represent a key issue in Part 2).
3.4 Design issues for interprocess communication run-time support

3.4.1 Implementation in user-space

All described versions of send-receive run-time libraries have been designed according to a basic principle: they are executed in user space, i.e. no privileged hierarchical states are exploited. Consider the typical implementation of a primitive in supervisor state: by its nature, a supervisor call must perform the parameter passing by values. This implies several additional copies of the message and other parameters, that nullify any optimization.

On the other hand, if an implementation in supervisor space is adopted, this means that the run-time support is implemented on top of an operating system (or similar virtual machine), thus interposing a set of interpretation mechanisms that, in general, have been defined for quite different purposes.

In the HPC word, the most efficient run-time libraries of industrial quality are implemented in user space.

3.4.2 Capability-based addressing

Another important design issue concerns the clear and efficient implementation of the so-called problem of indirectly referred shared data structures, or of the shared pointers.

Consider a situation that we have implemented in several run-time versions: direct copy of the message into the target variable. In this implementation, the target variable VTG is a shared objects which is pointed to by another shared object, i.e. the channel descriptor CH. An important difference exists between the shared natures of CH and VTG:

- CH is shared directly by the sender and the receiver, which use their independent and distinct logical addresses to refer it;
- instead, because VTG is pointed to by a CH field (vtg_ref), a mechanism is needed in order that the sender and the receiver can correctly refer VTG in their respective logical address spaces.

Static solutions to this problem exist (coincident logical addresses, unique identifiers, physical addresses) that suffer from several disadvantages in terms of time or space efficiency, or of protection.

The most general, efficient and elegant solution is the capability-based addressing. Referring to the VTG example, this technique allows the sender to allocate dynamically the VTG object in its own address space. The sender process will maintain VTG in its logical address space for the time strictly needed to perform the message copy into VTG, after that it will deallocate VTG from its address space. In other words, the capability-based addressing is a form of dynamic allocation of the virtual memory of a process.

The most evident effect is a minimization of the logical address space size. Think about other cases of shared objects that, if allocated statically, cause an unnecessary and inefficient increase of the logical address space size, for example all the PCBs of created processes.

Moreover, it can be shown that capability-based addressing is a very powerful mechanism for protection of address spaces.

The technique is based on the following principle, described with reference to the generic case in which a process A is willing to pass the reference of an object X to a process B dynamically (e.g., the receiver process passes the VTG reference to the sender process):
i. a shared data structure $S$, referred directly by $A$ and $B$, is provided, through which the $X$ reference is passed (e.g. the channel descriptor $CH$);

ii. the information $C$ ($X$ reference), passed by $A$ into $S$, is such that $B$ can transform it into a logical address in its address space. $C$ is called $X$-capability;

iii. in a very efficient implementation, information $C$ is the entry of the Page Relocation Table of process $A$ (possibly with modified protection rights); for the moment being, we are assuming that $X$ consists of a single page of the Virtual Memory – Main Memory hierarchy;

iv. process $B$ is now able to copy $C$ into its own Page Relocation Table, in a position that has been left free. This means that the corresponding logical page has not yet been allocated (a set of “free pages” have been provided by the compiler). Assume that $C$ is copied into the $J$-th entry of $B$’s Page Relocation Table: this means that the $X$ base logical address in $B$’s space has the logical page identifier equal to $J$, i.e. the $X$ base address is obtained as “$J$ concatenated to an offset equal to zero”;

v. now $B$ is able to refer $X$ with its own logical address. When it has finished to utilize $X$, $B$ renders again “free” the page and the relative Page Relocation Table entry.

The technique is illustrated in the following figure.

---

**X**: $A$’s page to be allocated dynamically in $B$’s space (e.g. PCB$_A$)

**A** copies the $X$'s Relocation Table entry into a shared structure $S$.

**S**: shared structure through which $X$-capability is passed from $A$ to $B$ (e.g. Channel Descriptor, field Sender_PCB_ref)

---

**B** copies $X$-capability into a free position of its Relocation Table. The logical page identifier of $X$, in $B$’s space, is equal to the Relocation Table position: thus the logical address of $X$ in $B$’s space is now determined.
A Page Relocation Table entry, i.e. a capability, contains the following information (one or two words):

- a bit indicating the presence of the logical page in main memory,
- some bits for the page replacement algorithm, e.g. LRU,
- a bit indicating that the page has been modified,
- some bits for the protection rights, i.e. which operations the process is authorized to execute on this page,
- the physical page identifier.

The following operations are defined on a capability:

- Transmit_capability \((C, S, \text{new\_rights})\): copy capability \(C\), with the protection rights field modified according to the new\_rights value, from the process Page Relocation Table into a shared structure \(S\),
- Read_capability \((C1, S, C2)\): read capability \(C1\) from a shared structure \(S\) and save it in a local variable \(C2\),
- Acquire_capability \((C, Addr)\): write capability \(C\) into the process Page Relocation Table in to the “Free” position, generate a base logical address \(Addr = \text{concatenation (logical page identifier = Free, offset = zero)}\), and update Free,
- Release_capability \((C)\): cancel capability \(C\) in the process Page Relocation Table (e.g. put the presence bit at false and the protection rights at null), and update Free.

Capability-based addressing is efficient because the dynamically allocated object is already in physical memory. Few Load/Store instructions are sufficient to perform the dynamic allocation.

Capability-based addressing is “efficiently protected”, because the permission on \(X\) has been passed explicitly by the owner. Moreover, as for any other object, the access to \(X\) is filtered by the MMU where the address translation and the protection check are performed in a single clock cycle.

Finally, the importance of this technique lies in the absolute necessity (not merely convenience) to allocate objects dynamically in some specific situations, i.e. for the compiler is impossible to statically predict the request of some kinds of run-time support objects. Notable examples will be met in Part 2.

Some industrial-level run-time supports exploit the capability-based addressing for implementing the dynamic allocation of virtual memory, e.g. for \texttt{new} or \texttt{malloc} mechanisms.
4. Parameters for performance evaluation

In this section the main parameters for the design and evaluation of parallel computations will be defined. They include, as particular cases, parameters introduced for sequential and parallel architectures, like performance, efficiency and benchmark completion time.

As introduced, the various parallel paradigms will be characterized by cost models that allow the designer and/or the compiler to evaluate such parameters as functions of other parameters that are typical of the application (e.g., calculation time, data size) and of the architecture (e.g., memory access time, communication latency).

4.1 Service time and bandwidth, latency, and completion time

Let us first consider a single sequential module operating on streams, M. M is defined to implement a set of \( k \) operations, each one with average service time \( T_i \) and occurrence probability \( p_i \), where

\[
\sum_{i=1}^{k} p_i = 1
\]

Thus, the average service time (service time, in the following) is

\[
T = \sum_{i=1}^{k} p_i T_i
\]

The average completion time (completion time, in the following) for finite stream length \( m \) is given by

\[
T_c = m T
\]

The processing bandwidth, or throughput, expresses the average number of operations executed by M in the time unit, and it is evaluated as

\[
B = \frac{1}{T}
\]

measured in number of operations per second (ops).

Observations

1) These definitions (that have been adopted at the firmware and assembler level) refer to the “maximum stress” situation for M, or “saturated module” situation, in which as soon as M has completed the execution of one input stream element, the next input stream element is already present and a new execution can start without waiting times. In other words, we evaluate the maximum processing capacity of M.

2) For the time being, we are not considering communications explicitly, nor the impact of interactions with other cooperating modules. In subsequent sections we’ll see how to take such aspects into account. In order to evaluate the various \( T_i \), we don’t take the other cooperating modules and communications into account: i.e. we evaluate the module in isolation. This means that, in order to evaluate \( T_i \), we compute the calculation time of the corresponding operation. Thus, \( T \) is computed as the average calculation time of M. More formally, the calculation time expresses the ideal service
time $T_{id}$ of M. If we consider the real situation of M being included in a system with other cooperating modules, in general the effective service time $T$ of M is greater or equal to $T_{id}$ because of the interactions and communications impact.

3) Therefore, the above definitions of $B$ and $T_c$ refer to the ideal bandwidth, $B_{id}$, and to the ideal completion time, $T_{c-id}$, of the single sequential module M operating on streams.

4) We have referred to one input and output stream. However, in general all the definitions and considerations apply to more than one stream as well. Several streams can be controlled in a non-deterministic (OR graph) or in a data flow (AND graph) manner.

The latency of M is defined as the average time needed to execute the computation on just one stream element. Since M is sequential, the latency coincides with the service time.

Let us now consider the transformation of the sequential module M, operating on streams, into a parallel version $\Sigma$ composed of $n$ modules.

The parallelism degree of $\Sigma$ is equal to $n$, independently of the effective capability of all the modules to work in parallel at any time.

From now on, when we need to characterize a certain parameter or a system or an application $X$ in terms of the parallelism degree $n$, we’ll use the notation $X(n)$. For example, the parallelization problem consists of in transforming a system or an application $\Sigma^{(i)}$ into a system or an application $\Sigma^{(n)}$, which is functionally equivalent.

The latency of $\Sigma^{(n)}$, $L^{(n)}$, is defined as the average time needed by $\Sigma^{(n)}$ to execute the computation on a single stream element.

The service time of $\Sigma^{(n)}$, $T^{(n)}$, is defined as the average time interval between the beginning of the executions on two consecutive stream elements. This is a more general definition with respect to the definition given above, in order to take the existence of parallel activities into account.

The processing bandwidth of $\Sigma^{(n)}$, $B^{(n)}$, or throughput, is defined as:

$$B^{(n)} = \frac{1}{T^{(n)}}$$

average number of operations per second

We have seen that, for a single sequential module:

$$L^{(1)} = T^{(1)} = T$$

while, in parallel, latency and service time are quite different:

- the latency $L^{(n)}$ measures the average time needed for a stream element “to cross” the graph of $\Sigma^{(n)}$ from inputs to outputs channels;
- the service time $T^{(n)}$ measures the average time interval after which $\Sigma^{(n)}$ is again able to accept a new input stream element (or subset of stream elements) without waiting for the output result on the previous element, i.e. without waiting the latency time.

Only in particular cases it is possible to establish a relation between latency and service time. Moreover:

- some parallel paradigms exist which decrease the latency compared to the sequential module,
while for other parallel paradigms the latency is greater (or equal in the ideal situation) compared to the sequential module.

It is worth noting that the second case is not necessarily negative or disadvantageous – far from it. In fact, though few situations exist in which meaningful performance measures depend mainly on latency, in the majority of cases the key parameter is the service time, because of its impact on completion time. Interesting situations exist (e.g., parallel client-server computations) in which the service time has the main impact, but the latency plays a significant role as well.

The bandwidth vs latency dichotomy is typical of many computations and communication systems. It is frequent to find network structures characterized by high latency and large bandwidth, i.e. propagation on links is relatively slow, but several messages can be transmitted in parallel. In parallel architectures, proper solution are needed to achieve both high bandwidth and low latency especially for interconnection structures.

The completion time of $\Sigma^{(n)}$, $T_{c}^{(n)}$, is still defined, for finite length streams, as the average time to complete the execution of all the stream elements. Given the stream length $m$, the following approximate relation holds in many meaningful situations:

$$T_{c}^{(n)} \sim m T^{(n)}$$

on condition that:

$$m \gg n$$

For all the defined parameters, also in the parallel case, we distinguish between the ideal value and the effective value, e.g. $B_{id}^{(n)}$ and $B^{(n)}$. The degradation with respect to the ideal value will be due to various overheads, including communications, and to bottlenecks in the parallel computation graph.

### 4.2 Efficiency and scalability

In this Section, utilizing the service time as the key measure, we define quantitative parameters to estimate “how good” a parallel solution is.

Consider a sequential computation $\Sigma^{(1)}$ with known calculation time $T^{(1)}$ and bandwidth $B^{(1)} = 1/T^{(1)}$. If it is parallelized with parallelism degree $n$, we obtain a computation $\Sigma^{(n)}$ characterized as follows in the ideal case:

$$T_{id}^{(n)} = \frac{T^{(1)}}{n} \quad B_{id}^{(n)} = n B^{(1)}$$

The effective measures will be $T^{(n)} \geq T_{id}^{(n)}$, $B^{(n)} \leq B_{id}^{(n)}$.

The relative efficiency is defined as:

$$\varepsilon = \frac{B^{(n)}}{B_{id}^{(n)}} = \frac{T_{id}^{(n)}}{T^{(n)}}$$

where the effective and ideal parameters are considered for the same parallelism degree. The relative efficiency is a universal and very meaningful parameter, telling us how close (how far) the effective performances are to (from) the ideal ones. From another point of view, it
gives a synthetic idea of the “utilization” degree of the modules composing the parallel system.

The **scalability** (sometimes called **speed up**) is defined as:

\[
s = \frac{B^{(\alpha)}}{B^{(1)}} = \frac{T^{(\alpha)}}{T^{(1)}}
\]

Thus a simple, yet meaningful, relation exist between scalability and efficiency:

\[
s = \varepsilon \cdot n
\]

Informally, scalability provides a measure of the relative “speed” of the parallel computation with respect to the sequential one.

During the course we’ll see that, while efficiency is always applicable, there are some cases in which scalability is of scarce or null value.

It is worth noting that bandwidth is an **absolute** measure of parallel performance, while efficiency is a **relative** one. Expressed as functions of various system parameters, bandwidth and efficiency may have the same qualitative shape for some parameters or quite different shapes for others. For example, by increasing a certain parameter, \( B \) could decrease tending to zero, while \( \varepsilon \) could increase tending to one: this means that the system is as much utilized as it is slow (though this is disappointing from the absolute performance viewpoint). For other parameters, \( B \) can have a monotonically and asymptotically increasing shape tending to the ideal value, while \( \varepsilon \) will monotonically decrease tending to zero.

Relative efficiency and scalability can be expressed as **functions of the completion time** too:

\[
\varepsilon = \frac{T^{(\alpha)}_{c-id}}{T^{(\alpha)}_{c}} \quad s = \frac{T^{(1)}_{c}}{T^{(\alpha)}_{c}} = \varepsilon \cdot n
\]

Finally, some spurious case can sometimes occur: \( \varepsilon \) may assume values greater than 1, \( s \) greater than \( n \) (“hyperscalability”). This result, which apparently is in contrast with the definition, may depend on a wrong application of the formulas, i.e. \( B \) and \( B_{id} \) must be evaluated with the same parallelism degree. However, there are cases in which the formulas are applied correctly, but the system configuration related to \( B^{(\alpha)} \) and \( B^{(1)} \) behave differently, e.g. they exploit the memory hierarchy in a different way (the cache level could be more effective for the parallel system owing to the allocation of smaller data structures). In such situations, the parallel system performance grows more than linearly with the parallelism degree.

**Example**

Let’s consider the following computation graph, having the form of a **pipeline** computation:

M₀ generates a stream as a result of the application of function \( F₀ \) to a set of \( m \) local values. M₁ applies a function \( F₁ \) to each element received from M₀ and sends the results to M₂. M₂ applies a function \( F₂ \) to each element received from M₁ and stores the results into a local data structure.
Let’s assume that:

a) functions \(F_0, F_1, F_2\) have the same average calculation time, denoted by \(t\),
b) the impact of communication latency on performance is negligible.

The ideal and effective service time of every module is equal to \(t\), and it is also the service time of the whole computation

\[ T^{(3)} = t \]

Informally, any module is able to start the execution on the next stream element every \(t\) time units. This is just an intuitive explanation. Formally, we will prove this results systematically by applying the results of subsequent sections.

Therefore, the processing bandwidth and completion time are:

\[ B^{(3)} = \frac{1}{t} \text{ elements processed per second } \quad , \quad T_{c}^{(3)} \sim m \ t \]

An equivalent sequential module operating on streams is characterized as follows:

\[ T^{(1)} = L^{(1)} = 3 \ t \quad , \quad B^{(1)} = \frac{1}{3t} \text{ elements processed per second } \quad , \quad T_{c}^{(1)} = 3 \ m \ t \]

Under the assumptions a) and b), the parallel computation with parallelism degree equal to three is able to achieve a bandwidth which is three times higher, and a completion time three times lower, than the equivalent sequential computation. These performances correspond to the ideal case which, in this case, is actually realizable owing to assumptions a) and b). Under the same assumptions, the latency is given by:

\[ L^{(3)} = L^{(1)} = 3t \]

However, if assumption a) and/or b) do not hold, we could have:

\[ T^{(3)} \geq t \]
\[ L^{(3)} > 3t \]

4.3 Masking communication latency: communication-calculation overlapping

The parallel architecture can provide some facilities able to overlap the interprocess communications with the internal calculation of processes, at least in part. The following figure shows feasible temporal situations of a module behavior:
Assume that the message to be sent by the module is the results of the phase indicated by “calculation 1”:

1) this is the traditional situation, in which the order of calculation phases and communication phases is strictly sequential. The communication latency is entirely paid, i.e. the module service time is increased by the communication latency $L_{com}$. The sequential order of calculation and communication phases can be forced by semantics reasons (data dependences exist), or by the absence of proper architectural supports (see case b);

2) this is the situation in which the communication latency is fully masked by calculation phase 2, which can start in parallel to the send execution as soon as the send is invoked. In this case, the communication impact on the module service time is null, i.e. the module service time is equal to the module calculation time. This situation is feasible if: 1) the communication form is asynchronous, and 2) suitable architectural supports are provided, notably every node is associated an independent communication processor to which the send execution is delegated as soon as it is invoked:

3) this is the situation in which, though the architecture provides asynchronous communication overlapping, the duration of calculation phase 2 is less than the communication latency. In this conditions, calculation is partially overlapped to communication, which has a non null impact on the service time.

Case c) could be considered the most general one. Let us denote by $T_{com}$ the average communication time not overlapped to internal calculation, where:

$$0 \leq T_{com} \leq L_{com}$$

That is, the communication time can be fully masked by the internal calculation ($T_{com} = 0$), or partially masked ($0 < T_{com} < L_{com}$), or sequentially ordered ($T_{com} = L_{com}$).

Therefore, in general we can say that the service time of the communicating module is increased by $T_{com}$:

$$T = T_{calc} + T_{com}$$

where $T_{calc}$ is the average calculation time (the ideal service time).

We have:
Thus, equivalently we can write:

\[ T = T_{\text{calc}} + T_{\text{com}} = \max (T_{\text{calc}}, L_{\text{com}}) \]

This important, yet simple, formula characterizes the cost model in presence of communications.

The latency of the module is always affected by the communication latency:

\[ L = T_{\text{calc}} + L_{\text{com}} \]

**Example**

Let us re-consider the example of Sect. 4.2. Relaxing assumption b), because of the communication impact we have:

\[ T_{\text{id}}^{(3)} = t \quad T^{(3)} = t + T_{\text{com}} \quad \varepsilon = \frac{t}{t + T_{\text{com}}} = \frac{1}{1 + \frac{T_{\text{com}}}{t}} \]

For example, assume that:

i) all the message types are integer arrays of size \( M \), thus the message length is \( L = M \);

ii) all the communication are asynchronous and zero-copy;

iii) the parallel architecture is characterized by \( T_{\text{setup}} = 10^2 t \) and \( T_{\text{transm}} = 10^3 t \), and contains a communication processor for every processing node.

Therefore:

\[ L_{\text{com}} = T_{\text{setup}} + L * T_{\text{transm}} = 10^2 t + 10^3 L t \]

The setup time is negligible compared to the calculation time \( T_{\text{calc}} = t \). Thus:

\[ L_{\text{com}} \sim 10^3 L t \]

For \( L \leq 10^3 \), we have \( L_{\text{com}} \leq T_{\text{calc}} \), i.e. \( T_{\text{com}} = 0 \) owing to the existence of communication processor and asynchronous communications. In this case (“relatively fine grain” communications, and proper computational and architectural conditions) the service time of every module and of the whole computation is equal the ideal service time \( T_{\text{calc}} = t \).

The latency of the whole computation is:

\[ L^{(3)} = 3 t + 2 L_{\text{com}} \]

where the term \( 2 L_{\text{com}} \) could be \( 3 L_{\text{com}} \) or \( 4 L_{\text{com}} \) if we take into account also possible communications to \( M_0 \) and from \( M_2 \).
5. Stream computations studied as queueing systems and queueing networks

The cost models for parallel computations will be expressed in terms of fundamental results in the area of Queueing Theory. For our purposes, this theory allows us to formalize important issues related to the evaluation of cooperating modules, e.g.: how to evaluate the service time of a graph structured computation starting from the knowledge of the service times of the component modules, how to detect bottlenecks in a parallel computation, and so on.

Queueing Theory is a very general approach based on mathematical tools, especially Probability Theory and Stochastic Processes. A detailed treatment of this subject, including mathematical proofs, is out of the scope of this course; however, we will acquire the basic elements which are strictly needed for studying cost model of parallel computations.

We will exploit also a branch of Queueing Theory, called Queueing Networks, aiming to solve complex compositions of queueing systems according to a simplified, yet rigorous, mathematical treatment.

5.1 Queueing systems and utilization factor

A queueing system models the behavior of a system composed of a server $S$ (service centre) and some clients $C_1, ..., C_n$. The clients requests are logically ordered into a waiting queue $Q$:

![Diagram of a queueing system]

This scheme is a logical one, not necessarily corresponding to the real structure of the computation or system to be modeled. However, it captures the essential elements of the problem at hand. For example, in some real cases there are distinct multiple communication channels between each client and the server in both directions: a single queue in front of $S$ does not exist physically, however it is emulated by the set of channels and by the nondeterministic behavior of $S$. From the performance evaluation viewpoint, the logical scheme reduces the complexity of the analysis and makes it possible to obtain an approximate evaluation, which is quite acceptable provided that the adopted mathematical and stochastic assumptions are validated.

The queueing systems is analytically defined if the following characteristics are known.

- The service discipline: if not explicitly defined, especially in computing systems the FIFO discipline is assumed, though some priority schemes could be modeled.
• The queue size, that is the number of elements available for storing the clients requests. Many results in Queueing Theory have been derived for infinite length queues. Though in a computing system this size is fixed or is limited, for our purposes the utilization of results for the infinite length queues often represents a sufficient approximation.

• The probability distribution of the random variable service time, \( t_S \)

\[
F_{t_S}(t) = \Pr(t_S \leq t)
\]

with average value \( T_S \) and variance \( \sigma_S \); the average frequency of services is denoted by \( \mu = 1/T_S \).

• The probability distribution of the random variable interarrival time \( t_A \) (time interval between two consecutive arrivals of requests)

\[
F_{t_A}(t) = \Pr(t_A \leq t)
\]

with average value \( T_A \) and variance \( \sigma_A \); the average frequency of interarrivals is denoted by \( \lambda = 1/T_A \).

The queue utilization factor, or equivalently the server utilization factor, is defined as

\[
\rho = \frac{T_S}{T_A} = \frac{\lambda}{\mu}
\]

a very meaningful parameter in performance evaluation models based on Queueing Theory and Queueing Networks. It expresses a global, average measure of the congestion degree, or traffic intensity, of the requests to the server.

When \( \rho \geq 1 \), the server represents a bottleneck with respect to the clients requests: during the steady-state behavior, the average number of queued request grows indefinitely. This situation occurs only when the client does not need an explicit request from the server - for example in a computation described by an acyclic graph - because, on the average, the server is not able to satisfy the client requests.

In real systems and computations, because the queue size is of finite length, the effect is that the client execution is periodically blocked, thus the client service time adapts to the server service time.

Notice that the situation \( \rho \geq 1 \) is a transient one. However, for our purposes, it is meaningful, because it is the condition for the existence of a bottleneck.
Let us consider now a system in which the client needs an explicit request from the server. We are in a typical request-reply computation (cyclic subgraph), in which the client performs the request and then waits for the result from the server.

In this case, it is always \( \rho < 1 \), because the system has a self-stabilizing behavior, i.e. a temporary increase in the interarrival time has the effect of a decrease in the server response time that tends to lower the interarrival time itself, so that on the average \( T_s < T_A \). In a request-reply system we cannot speak of bottlenecks; however, though \( \rho < 1 \), higher values of \( \rho \) (tending to one) cause a greater average response time, that lowers the client processing bandwidth compared to the ideal value.

In both cases – acyclic graphs or presence of cyclic subgraphs with a request-reply pattern – server parallelization is the main technique to reduce \( T_S \) compared to \( T_A \), thus to reduce the \( \rho \) value.

1. In an acyclic graph (sub)computation, the situation of \( \rho = 1 \), though not acceptable from a theoretical viewpoint, is acceptable in practice. We can say that \( \rho \) values less than one and very close to one correspond to the elimination of bottlenecks and, at the same time, to the optimal utilization of the server. A further parallelization of the server, thus a further reduction of \( T_S \), is not beneficial for the client and implies a “higher cost” of the server (lower server efficiency). A proper asynchrony degree of communication channels can be provided in order to reduce the occurrences of client blocking situations. In the design of parallel computations expressed by acyclic (sub)graphs, we try (where possible) to eliminate all the bottlenecks (or to eliminate as many bottlenecks as possible) by imposing utilization factor values equal to one (formally, less than one and very close to one), so achieving the best server efficiency. In this class of computations the server service time is the main parameter to be optimized, while the server latency has no direct effect on the clients bandwidth (it has effect on the global system latency only).

2. In a request-reply (sub)computation, the \( T_S \) reduction decreases the utilization factor, thus decreases the response time: the effect is an improvement of the clients bandwidth. However, in this case also the server latency has a certain impact on the response time. Thus, in designing a parallel version of servers, we are interested in parallel paradigms able to reduce both the server service time and the server latency.

The analytical treatment of Queueing Systems in terms of probability distributions is necessary mainly for request-reply computations. This case will be studied in a subsequent Section.
For acyclic graph computations, Queueing Networks are a sufficiently powerful methodology. They do not utilize an explicit analytical treatment in terms of probability distributions, instead they are based on some basic results about the information flows in a network of queues. For our purposes, the Queueing Networks results make it possible to analyze a graph computation in order to discover bottlenecks and critical paths. This issue will be dealt with in the following subsection.

5.2 Fundamental properties of computations studied as queueing networks

5.2.1 Interdeparture time

Let S be a server with interarrival time $T_A$ and service time $T_S$.

The interdeparture time of S, $T_p$, is defined as the average time between two consecutive results onto the output stream. According to the definition of utilization factor, if S produces one output stream value for each input stream value, we have:

$$T_p = \begin{cases} T_A & \text{if } \rho < 1 \\ T_S & \text{if } \rho \geq 1 \end{cases}$$

Equivalently:

$$T_p = \max (T_A, T_S)$$

5.2.2 The “multiple clients” theorem

Let us consider a queue with clients $C_1$, ..., $C_n$, and server S that accepts service requests according a nondeterministic discipline (OR graph). Let us denote by $T_{p1}$, ..., $T_{pn}$ the interdeparture times of the clients.

It can be shown that the interarrival time to the server is given by:

$$T_A = \frac{1}{\lambda} = \frac{1}{\sum_{i=1}^{n} \lambda_i} = \frac{1}{\sum_{i=1}^{n} \frac{1}{T_{pi}}}$$
That is, the global interarrival frequency is equal to the sum of the individual interarrival frequencies from the clients.

Equivalently:

- the global interdeparture frequency of a set of clients is equal to the sum of the individual interdeparture frequencies.

Not only this theorem is as basic result for queueing networks analysis, it also expresses the following fundamental result of parallel processing:

- given a set of independent subsystems \( \Sigma_1, ..., \Sigma_n \) with known individual bandwidths \( B_1, ..., B_n \), the global bandwidth \( B \) of subsystem \( \Sigma = \{ \Sigma_1, ..., \Sigma_n \} \) is equal to the sum of the individual bandwidths:

\[
B = \sum_{i=1}^{n} B_i
\]

In other words, this is the formal key to prove an intuitive / naïve fact: by exploiting a parallelism degree \( n \), the bandwidth increases by \( n \) times. Of course, this is the bandwidth of \( \Sigma \) considered in isolation: not necessarily the whole system, in which \( \Sigma \) is possibly included, is able to exploit such bandwidth. Subsequent Sections will discuss these important issues in detail.

5.2.3 The “server partitioning” theorem

Let a system composed by a client \( C \) and some servers, \( S_1, ..., S_n \), in general able to provide distinct services (OR graph):

![Diagram of server partitioning theorem](image)

Let \( T_p \) the interdeparture time of \( C \), and \( p_1, ..., p_n \) the probabilities that a generic request of \( C \) is directed to \( S_1, ..., S_n \) respectively, where

\[
\sum_{i=1}^{n} p_i = 1
\]

**Version 1: no bottlenecks**

- if all the utilization factors \( \rho_1, ..., \rho_n \) are less than one, then the interarrival time to the generic server \( S_i \) is given by:

\[
T_A = \frac{T_p}{\rho_i}
\]

**Proof**
Let a module A, with service time $T_A$, be the client of a server module B with probability $p$, and client of a set of other modules with probability $1 - p$. Because B and the other servers have utilization factors less than one, the interarrival times from A do not depend on the servers service time. For this reason, the interarrival time $t_{AB}$ from A to B is a random discrete variable with the following distribution:

<table>
<thead>
<tr>
<th>$t_{AB}$</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_A$</td>
<td>$p$</td>
</tr>
<tr>
<td>$2T_A$</td>
<td>$p(1-p)$</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>$nT_A$</td>
<td>$p(1-p)^{n-1}$</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Thus, the average interarrival time is:

$$T_{AB} = \sum_{n=0}^{\infty} nT_A p(1-p)^{n-1} = \frac{pT_A}{1-p} \sum_{n=0}^{\infty} n(1-p)^n$$

According to the general property:

$$\text{for } x < 1: \sum_{n=0}^{\infty} nx^n = \frac{x}{1-x^2}$$

we have:

$$T_{AB} = \frac{T_A}{p}$$

CVD.

**Version 2: existence of bottlenecks**

If the assumptions of Version 1 are verified (no bottlenecks), all the interarrival times to the various servers can be derived independently each other. This is no more true if at least one server is a bottleneck: the bottleneck influences the interarrival times to the other servers. Let us consider the following example:

![Diagram](image)

where A, B, C have service times $T_A$, $T_B$, $T_C$, and $p$ is the probability that A requests a service to B. Let B be a bottleneck:
\[ \rho_B = \frac{T_B}{T_{AB}} > 1 \]

where \( T_{AB} \) is evaluated as in the Version 1:

\[ T_{AB} = \frac{T_A}{p} \]

The existence of this bottleneck introduces a delay in the A activity with probability \( p \), i.e. the service time of A is increased. This has impact on the interarrival time to C too.

The corrected service time of A is a new value \( T'_A \) such that:

\[ \frac{T'_A}{p} = T_B \]

Thus:

\[ T'_A = p T_B \]

which is > \( T_A \), i.e. the delay introduced in A activity (“bubble”) with probability \( p \) is:

\[ T'_A - T_A = p (T_B - T_{AB}) \]

The real interarrival time to C is now:

\[ T'_{AC} = \frac{T'_A}{1-p} = \frac{p}{1-p} T_B \]

Moreover, it can be shown that, because the influence of a bottleneck on the interarrival times of the other servers, there cannot be more than one bottleneck, that is the module having the highest utilization factor.

Considerations on the application of Version 1 and Version 2

1) In order to apply the “server partitioning” theorem, in Version 1 or in Version 2, first we must evaluate all the utilization factors.

2) If Version 1 can be applied, the analysis of an acyclic graph is rather straightforward: starting from the modules generating the initial streams, we apply properties 4.2.1, 4.2.2, 4.2.3 until we determine the interdeparture time of the graph, i.e. the service time. Because of the absence of bottlenecks, the service time is always equal to the ideal one, though some modules are underutilized.

3) If Version 2 has to be applied, the analysis of an acyclic graph is complicated. For example, consider the following graph:
where B is a bottleneck. The corrected value of A service time implies a new analysis of the interarrival time to F. In general, it is possible that the analysis of a graph, in presence of bottlenecks, implies a procedure that is *iterative until convergence is reached* (it is always achievable with a finite and limited number of attempts).

A formal algorithm is not provided here. Students can solve problems of this kind by an informal iterative procedure, which can easily be derived for small graphs. The interested student can propose a formal algorithm as an advanced exercise.

### 5.2.4 Examples of acyclic graph analysis

**Example 1**

This example shows that, in a computation without bottlenecks, the service time is equal to the ideal one. Let the computation $\Sigma$ be described by the following acyclic OR graph:

Let $t$ be a standardized time unit. M1 generates a stream of $N = 10K$ integer values with interdeparture time equal to $30t$. Each element of the stream is sent to M2 with probability equal to $3/4$, otherwise to M3. The calculation times of M2, M3, M4 is $20t$, $100t$, $15t$ respectively.

$\Sigma$ is a computation with parallelism degree $n_\Sigma = 4$, whose processing bandwidth cannot be greater than the stream generation bandwidth of M1. Therefore the *ideal service time* of $\Sigma$ is

$$T_{id}^{(4)} = 30t$$

The interarrival times to M2 and M3 respectively are given by:

$$T_{A2} = \frac{T_{n1}}{1/4} = 40t$$

$$T_{A3} = \frac{T_{n1}}{1/4} = 120t$$

The utilization factors of M2 and M3 are:

$$\rho_2 = \frac{T_2}{T_{A2}} < 1$$
\[ \rho_3 = \frac{T_3}{T_{A3}} < 1 \]

M2 and M3 are not bottlenecks. Their interdeparture times are:

\[ T_{p2} = T_{A2} = 40t \]
\[ T_{p3} = T_{A3} = 120t \]

The interarrival time to M4:

\[ T_{A4} = \frac{1}{\frac{1}{T_{p2}} + \frac{1}{T_{p3}}} = 30t \]

Thus:

\[ \rho_4 = \frac{T_4}{T_{A4}} < 1 \]

Finally:

\[ T_{p4} = T_{A4} = T^{(4)} = 30t \]

which is equal to the ideal service time, as expected.

The relative efficiency and scalability have the maximum values:

\[ \varepsilon^{(4)} = \frac{T_{d}^{(4)}}{T^{(4)}} = 1 \]
\[ s^{(4)} = \varepsilon^{(4)} \times 4 = 4 \]

The completion time of \( \Sigma \) is given by:

\[ T^{(4)}_{c} \approx N T^{(4)} = 300Kt \]

The computation latency is evaluated as:

\[ L_{\Sigma} = T_1 + \frac{3}{4}T_2 + \frac{1}{4}T_3 + T_4 = 85t \]

Example 2

Consider the computation of Example 1 with the only difference that the calculation time of M3 is equal to 200t.

Now M3 is a bottleneck:

\[ \rho_3 = \frac{T_3}{T_{A3}} > 1 \]

Therefore, the corrected service time of M1 is:

\[ T_{1}^{'} = \frac{1}{4}T_{3} = 50t \]

However, the ideal service time of the computation is still:
\[ T_{id}^{(4)} = 30t \]

The corrected interarrival time to M2 is:
\[ T_{A2}^{'} = \frac{T_1^{'}}{1/4} = 200t \]

thus M2 is not a bottleneck, as expected.

We have:
\[ T_{p2} = T_{A2}^{'} = 200t \]
\[ T_{p3} = T_3 = 200t \]

The interarrival time to M4:
\[ T_{A4} = \frac{1}{T_{p2}^{-1} + T_{p3}} = 100t \]

Thus:
\[ \rho_4 = \frac{T_4}{T_{A4}} < 1 \]

Finally:
\[ T_{p4} = T_{A4} = T^{(4)} = 100t \]

which is greater to the ideal service time, as expected.

The relative efficiency and scalability have rather low values:
\[ \varepsilon_{\Sigma} = \frac{T_{id}^{(4)}}{T^{(4)}} = 0.3 \]
\[ s^{(4)} = \varepsilon_{\Sigma} * 4 = 1.2 \]

The completion time of \( \Sigma \) is given by:
\[ T_c^{(4)} \approx N T^{(4)} = 1000 K t \]

The computation latency is:
\[ L_{\Sigma} = T_1 + \frac{3}{4} T_2 + \frac{1}{4} T_3 + T_4 = 110t \]

5.3 Ideal and effective bandwidth of modules and graph computations

The next step in our methodology is a set of general results about 1) how to evaluate ideal and effective bandwidths of acyclic graph computations, and about 2) the optimal degree of parallelism. Let us start with issue 1).
Consider an acyclic system $\Sigma$ consisting in a module, or a subsystem, $\Sigma_1$ with one or more input streams with interarrival time $T_A$.

This simple configuration allows us to understand how to evaluate the ideal and the effective bandwidth in any complex system.

Let us first evaluate the ideal and the effective service time of $\Sigma_1$.

The ideal service time of $\Sigma_1$ is evaluated by considering $\Sigma_1$ in isolation, i.e. the ideal service time is related to its internal calculation time. If $\Sigma_1$ is a single module, it is characterized by an average calculation time $T_{\text{calc}}$, thus the ideal service time of $\Sigma_1$ is given by

$$T_{\Sigma_1}^{\text{id}} = T_S = T_{\text{calc}}$$

If $\Sigma_1$ is subsystem with parallelism degree $n$, the ideal service time of $\Sigma_1$ is given by

$$T_{\Sigma_1}^{\text{id}} = T_S = \frac{T_{\text{calc}}}{n}$$

Because $\Sigma_1$ belongs to a system $\Sigma$, the effective service time of $\Sigma_1$ is given by its interdeparture time:

$$T_{\Sigma_1} = T_p = \max(T_S, T_A)$$

In other words, considering $\Sigma_1$ as part of a more complex system, the interarrival and the interdeparture time of $\Sigma_1$ must be considered as well in order to evaluate its performance.

Notice that, even if an explicit output stream is not present, in a stream-based computation the interdeparture time can always be determined by considering a fictitious stream onto which the results are sent (in the real system, it is possible that the results are assigned to internal variables only).

At this point, we are able to evaluate the relative efficiency of $\Sigma_1$, considered as part of a complex system. By definition:

$$\varepsilon_{\Sigma_1} = \frac{T_{\Sigma_1}^{\text{id}}}{T_{\Sigma_1}} = \frac{T_S}{\max(T_S, T_A)}$$

Therefore:

$$\varepsilon_{\Sigma_1} = \begin{cases} \rho_{\Sigma_1} & \text{se} & \rho_{\Sigma_1} < 1 \\ 1 & \text{se} & \rho_{\Sigma_1} \geq 1 \end{cases}$$

This is a very meaningful, yet simple, result: the relative efficiency of a module, or a subsystem, included in a system, is equal to its utilization factor if it is not a bottleneck for the whole system, otherwise it is equal to one.

In other words, if the module, or subsystem, is a bottleneck it is fully utilized, otherwise it is utilized according to the ratio ideal service time / interarrival time.

Let us now evaluate the whole system $\Sigma$. We have:

$$T_\Sigma = T_{\Sigma_1} = T_p$$

$$T_{\Sigma}^{\text{id}} = T_A$$
In fact: 1) the effective service time is the interdeparture time of the graph (that coincides with the interdeparture time of $\Sigma_1$); 2) no parallel realization of $\Sigma_1$ exists able to achieve a service time lower than the interarrival time.

The relative efficiency of the whole system $\Sigma$ is given by:

$$\varepsilon = \frac{T_{\Sigma-id}}{T_\Sigma} = \frac{T_A}{T_p}$$

This means that the system is able to achieve the ideal bandwidth, and the maximum efficiency, if it does not contain bottlenecks (in this case $T_p = T_A$, otherwise $T_p > T_A$).

This results are applicable to any acyclic system, for example the indicated one, where the module, or subsystem, $\Sigma_{in}$ generating the stream is explicitly represented. By induction, any acyclic system can be derived by expanding $\Sigma_{in}$.

5.4 Transformation of bottlenecks and optimal parallelism degree

The results of Section 4.3 allow us to determine the optimal parallelism degree of modules, or subsystems, that represent bottlenecks in acyclic graph computations.

Let $\Sigma_1$, with service time $T_s$ e interarrival time $T_A$, and $T_s > T_A$. Thus, $\Sigma_1$ is a bottleneck for the whole system. Let us parallelize $\Sigma_1$ with parallelism degree $n$.

- The following value of $n$:

$$\bar{n} = \left\lceil \frac{T_s}{T_A} \right\rceil$$

is the potentially optimal parallelism degree. That is, the bottleneck can be potentially eliminated, on condition that a parallelism paradigm exists able to actually exploit this parallelism degree.

- If this parallelization is possible, then the effective service time of $\Sigma_1$ is given by:

$$T_s^{(\bar{n})} = T_A$$

and the relative efficiency:

$$\varepsilon_{\Sigma_1} = \rho_{\Sigma_1} = \frac{T_s}{\bar{n}} \frac{T_A}{T_A} = \frac{T_s}{\left\lceil \frac{T_s}{T_A} \right\rceil}$$

- If the optimal parallelization is not possible, the parallelized $\Sigma_1$ is still a bottleneck, thus

$$\varepsilon_{\Sigma_1} = 1$$

In this situation, let

$$n_0 < \bar{n}$$
the best parallelism degree achievable. The effective service time is given by:

\[ T_s^{(n_0)} = \frac{T_s}{n_0} > T_A \]

Notice that if the ratio \( T_s/T_A \) is not integer, and we choose \( n = \bar{n} = \left\lfloor \frac{T_s}{T_A} \right\rfloor \), the utilization factor (the relative efficiency) is inevitably less than one; however this is the best solution that can be found: some additional “resources” are needed in order to minimize the service time. On the other hand, if we choose \( n = \left\lceil \frac{T_s}{T_A} \right\rceil \), some resources are saved, but the service time is not optimal.

In subsequent Sections, we will study the problem “verify the existence of parallelism paradigms able to exploit the optimal parallelism degree”.

The following figures illustrate the application of the above results: they show typical shapes of the bandwidth and the relative efficiency for a system \( \Sigma \) consisting of a subsystem \( \Sigma_1 \), as functions of the parallelism degree \( n \) of \( \Sigma_1 \):

In many real-life applications, these shapes are valid as first approximations. Because of some performance degradation reasons, in general linear segments are replaced by asymptotic curves, for example:
5.5 Detailed example of acyclic graph analysis and optimization

Let us consider a computation $\Sigma$ described by the following graph:

![Graph Diagram]

Modules functionalities are specified in a LC-like formalism:

parallel $M_0, M_1, M_2$;

$M_0 :: \text{int } A[N][N]; \text{int } C[N];$

$\{ \forall i = 0..N-1: C[i] = F_0(A[i][*]); //function applied to the i-th row/;$

$send C to M_1 \}$

$M_1 :: \text{int } C[N]; \text{int } X;$

$\{ receive C from M_0; X = F_1(C); send X to M_2 \}$

$M_2 :: \text{int } D[N]; \text{int } X;$

$\forall i = 0..N-1: \{ receive X from M_1; D[i] = F_2(X) \}$
where $N = 10^2$.

The average processing times of $F_0$, $F_1$, $F_2$ are $10^2\tau$, $10^6\tau$, $10^5\tau$, respectively, where $\tau = 1$ nsec is the clock cycle of the processing nodes.

The parallel program is executed on a machine with 64 nodes with communication processor. The interprocess communication run-time support is zero-copy and the channels are asynchronous. The communication cost model is characterized by $T_{\text{setup}} = T_{\text{transm}} = 10^3\tau$.

a) Evaluate the ideal and effective bandwidth $B$, and the relative efficiency $\varepsilon$.

b) Parallelize the bottlenecks, and evaluate the ideal and effective bandwidth $B$, and the relative efficiency $\varepsilon$, of the optimal parallel configuration, assuming that a proper parallelism paradigm exists able to exploit the optimal parallelism degree.

c) Referring to case b), study the functions $B = B(n)$, $\varepsilon = \varepsilon(n)$, where $n$ is the $M_1$ parallelism degree, for $M_1$ and for the whole computation $\Sigma$.

To solve this problem we apply the methods and techniques of Sections 3.3, 4.3, 5.2.1, 5.3, 5.4.

a) The calculation time of $M_0$ is evaluated as

$$T_{\text{calc-0}} \sim N * T_{F0} = 10^4\tau$$

since the loop control has a negligible impact. This is the value of the *ideal* service time of $M_0$.

The communication latency for the channel $M_0$-$M_1$ is

$$L_{\text{com-0}} = T_{\text{send}}(N) = T_{\text{setup}} + N * T_{\text{transm}} \sim 10^5\tau$$

Therefore, the *effective* service time of $M_0$ is dominated by the communication latency:

$$T_0 = L_{\text{com-0}} = 10^5\tau$$

and it is the value of $M_0$ interdeparture time, thus of $M_1$ interarrival time:

$$T_{\text{p0}} = T_{\text{A1}} = 10^5\tau$$

This is also the *ideal service time of the whole computation* $\Sigma$:

$$T_{\Sigma,\text{id}} = 10^5\tau$$

Notice that $T_{\text{calc-0}}$ is the ideal service time of $M_0$ (not of $\Sigma$), thus the $M_0$ relative efficiency is given by $\varepsilon_0 = T_{\text{calc-0}} / T_0 = 0.1$. However, as far as $\Sigma$ is concerned, the service time cannot be lower than the interdeparture time of $M_0$, i.e. the interdeparture time from the module that generates the stream.

The ideal service time (calculation time) of $M_1$ is equal to $10^6\tau$ (*receive* latency is negligible in zero-copy communication), thus $M_1$ is a *bottleneck*. Its communication latency (channel $M_1$-$M_2$):

$$L_{\text{com-1}} = T_{\text{send}}(1) = T_{\text{setup}} + T_{\text{transm}} = 2 * 10^5\tau$$

is fully masked by the calculation time, thus the effective service time of $M_1$ is:

$$T_1 = T_{\text{p1}} = T_{\text{A2}} = T_{\text{calc-1}} = 10^6\tau$$
The ideal service time of M₂ is given by the calculation time of a single iteration of the for loop, because for each input steam element X a result is computed (a fictitious output stream of integer type has to be considered). Therefore:

\[ T_{\text{id}} = T_{\text{calc}} = 10^5 \tau \]

and M₂ is not a bottleneck, thus the effective service time, which is equal to the whole system service time, is given by:

\[ T_2 = T_{p2} = T_{A2} = T_\Sigma = 10^6 \tau \]

The relative efficiency is:

\[ \varepsilon = \frac{T_\Sigma - \text{id}}{T_\Sigma} = 0,1 \]

The processing bandwidth is:

\[ B_\Sigma = \frac{1}{T_\Sigma} = \frac{10^9}{10^6} = 10^3 \text{ processed arrays per second} \]

b) A parallel version of the bottleneck module M₁ has the optimal parallelism degree:

\[ n_{\text{opt}} = \frac{T_1}{T_{A1}} = 10 \]

The whole parallelism degree is 12, which can be effectively exploited, being less than the number of physical processing nodes.

Assuming that a proper parallel paradigm exists (we’ll see that both farm and data-parallel solutions can be applied), the performance measures, with parallelism degree equal to 12, are:

\[ T_\Sigma = T_{A1} = 10^5 \tau = T_{\Sigma, \text{id}} \]

\[ B_\Sigma = \frac{1}{T_\Sigma} = \frac{10^9}{10^5} = 10^4 \text{ processed arrays per second} \]

\[ \varepsilon_\Sigma = \frac{T_\Sigma - \text{id}}{T_\Sigma} = 1 \]

c) Let us evaluate the performance measures of M₁, considered as a part of Σ, as a function of the M₁ parallelism degree n:

\[ B_{1,\text{id}} = \frac{1}{T_{1,\text{id}}} = \frac{1}{T_{\text{calc}}} = 10^3 \frac{n}{n} \text{ for any } n \]

\[ B_1 = \begin{cases} \frac{1}{T_1} = \frac{1}{T_{\text{calc}}} = 10^3 \frac{n}{n} & \text{for } n < 10 \\ \frac{1}{T_{A1}} = 10^4 & \text{for } n \geq 10 \end{cases} \]
\[ \varepsilon_1 = \frac{B_1}{B_{1-id}} = \begin{cases} 1 & \text{for } n < 10 \\ \frac{10}{n} = \rho_1 & \text{for } n \geq 10 \end{cases} \]

while for the whole computation \( \Sigma \) we have:

\[ B_{\Sigma-id} = \frac{1}{T_{\Sigma-id}} = 10^4 \text{ for any } n \]

\[ B_{\Sigma} = \begin{cases} \frac{1}{T_1} = 10^3 n & \text{for } n < 10 \\ \frac{1}{T_{\Sigma-id}} = 10^4 & \text{for } n \geq 10 \end{cases} \]

\[ \varepsilon_{\Sigma} = \frac{B_{\Sigma}}{B_{\Sigma-id}} = \begin{cases} n/10 & \text{for } n < 10 \\ 1 & \text{for } n \geq 10 \end{cases} \]

We recognize the shapes that have been anticipated, in a general and qualitative way, at the end of Sect. 5.4.

Finally, notice that the bandwidth of this computation cannot be increased by further parallelizations, because \( M_0 \) service time is dominated by the communication latency. Thus, a parallelization of \( M_0 \) has no effect. The same is true for \( M_2 \), which is underutilized (\( \varepsilon_2 = 0.1 \)).
6. General characteristics of parallel paradigms

Starting from this Section, we apply the general methodology to the study of structured parallelism paradigms and to the design of complex parallel applications.

We can individuate some general characteristics of parallel paradigms.

1. Streams vs single data structures. Some paradigms are meaningful only on streams, notably pipeline and farm, while data-parallel and data-flow can be applied to streams and/or to single data structures. For exploiting the data-parallel paradigm, data must be organized into, hopefully large, data structures (uni- or multi-dimensional arrays), while the other paradigms can be applied also on scalar values.

As far as streams are concerned, there are many applications in which the input streams are primitive, because they are generated by external sources or I/O (e.g. sensors, satellites, networks). However, if streams are not primitive, it is possible that they can be generated by program. For example, given the sequential computation:

\[
M :: \text{int } A[m], B[m]; \ldots \;
\]

\[
\text{for } (i = 0; i < m; i++) \;
\]

\[
B[i] := F(A[i])
\]

we can transform it into a three-stage pipeline, where the first encapsulates A and generates a stream of A elements (“unpacking”), the second executes function F and sends the results to the third stage, which encapsulates B and produces the final B value step by step (“packing”). The second stage is the stream computation which is a candidate for parallelization.

2. Partitioning vs replication. This is a key characteristic of our methodology. Functions and data can be partitioned or replicated among the parallel computation modules. Farm paradigm exploits replication only, applied to functions and to non-modifiable data; only stateless computations (“pure” functions) are candidate for farm parallelization. Data-parallel is characterized by replication of functions and partitioning of data; computations can have an internal state. Often data-flow and pipeline use partitioning of both functions and of data. However there are some notable cases of pipeline computations with pure function replication or with function replication and data partitioning (also with state). In fact, we’ll see that pipeline paradigm could also be viewed as a special implementation of farms or of data-parallel computations, with specific advantages and disadvantages.

3. Stateless computations vs computations with internal state. See the previous point.

4. Degree of knowledge of the sequential computation form. The farm paradigm can be applied to any pure function, operating on any data type, without any knowledge of the internal form of the function itself. Provided that it is stateless, the function to be parallelized can be assumed as a “black box”, since it is merely replicated into some worker modules. To be applied, all the other paradigms need a certain degree of knowledge of the sequential computation form. For example, pipeline requires that the sequential function can be expressed as (or transformed into) the linear composition of (sub)functions; data-flow and data-parallel requires the application of formal conditions to detect data dependences among distinct parts of the sequential computation.
5. **Memory capacity.** The whole memory capacity of a parallel program is the sum of the memory capacities of the components modules (nodes). Farm paradigm tends to increase the memory capacity compared to the sequential implementation (\(n\) times replication), the others, being based on data partitioning, tend to decrease it. This might also have a “hyperscalable” effect, owing to the better exploitation of local memories and caches of processing nodes.

6. **Latency.** Farm and pipeline paradigms tend to increase the latency compared to the sequential implementation (in the pipeline case, the increase is proportional to the parallelism degree), while data-parallel and data-flow tend to decrease it (though the effect of communication latency might smooth the difference).
7. Pipeline paradigm

The pipeline paradigm is defined on streams only. It is a very simple and didactic, yet sometimes “natural” and effective, solution to some parallelization problems. However, it is worth noting that, unless some application or architectural constraints could force/recommend the utilization of this paradigm, there are other paradigms which, for the same problem, will prove to be more powerful.

The application of the pipeline paradigm to a sequential module requires some knowledge of the form of the sequential computation, that is the sequential computation must be expressed as the composition of n functions:

\[ F(x) = F_n ( F_{n-1} ( \ldots F_2 ( F_1 (x) ) \ldots ) ) \]

for each element \( x \) of the input stream. In this case, a possible parallelization is a linear graph of \( n \) modules, also called pipeline stages, each one corresponding to a specific function.

A notable example is in the area of image reconstruction, where at least four phases can be recognized: smoothing, feature extraction, object recognition, and display. They can correspond to four stages of a pipeline applied to a stream of images. Of course, if the input is a single image, the pipeline is not a meaningful solution (however, data-parallel solutions will apply).

7.1 Cost model

As usual, let \( T_{\text{calc}} \) the average calculation time of the whole function \( F(x) \):

\[ T_{\text{calc}} = \sum_{i=1}^{n} T_{\text{calc}_i} \]

Initially we assume that the component functions \( F_1, \ldots, F_n \) have the same calculation time:

\[ t = \frac{T_{\text{calc}}}{n} \]

Moreover, assume that all stages have the same \( T_{\text{com}} \) (\( \geq 0 \)). We can obtain a well-balanced pipeline structure, i.e. all stages have the same service time and this is the service time of the whole pipeline computation:

\[ T^{(n)} = T_{\text{stage}} = \frac{T_{\text{calc}}}{n} + T_{\text{com}} \]

while the ideal service time is:

\[ T^{(n)}_{\text{id}} = \frac{T_{\text{calc}}}{n} \]

In the most general case, stages might be unbalanced in terms of calculation time and/or of communication time. Therefore, because one or more stages are bottlenecks for the whole computation, we have:

\[ T^{(n)} = \max_i (T_i) = \max_i (T_{\text{calc}_i} + T_{\text{com}_i}) \]
\[ T_{id}^{(n)} = \frac{T_{calc}}{n} = \frac{\sum_{i=1}^{n} T_{calc_i}}{n} \]
\[ \varepsilon = \frac{T_{id}^{(n)}}{T^{(n)}} = \frac{\sum_{i=1}^{n} T_{calc_i}}{\max_i (T_{calc_i} + T_{com_i})} \]

The pipeline latency is:
\[ L^{(n)} = \sum_{i=1}^{n} (L_i + L_{com}) \]
greater than the sequential module latency.

As a general comment, we can say that the simplicity of a pipeline structure is, at the same time, the reason for its pros and cons: a linear chain of \( n \) modules is characterized by the minimal number of channels, however the latency – proportional to \( n \) – is its Achilles’ heel.

**Formal development of pipeline cost model**

The intuitive formulas above can be proved formally by exploiting the methodology for solving a computation graph as a queueing network. We will also take the pipeline interarrival time \( T_A \) into account (the formulation above is valid when the first stage generates the stream, as in the example \( M_0 - M_1 - M_2 \) of previous Sections).

For the first stage we have:
\[ T_{p_0} = \max(T_A, T_0) = T_{h_0} \]
\[ T_{p_i} = \max(T_{A1}, T_1) = \max(T_A, T_0, T_1) = \max(T_A, T_0, T_1) \]

Applying the induction reasoning, if
\[ T_{p_i} = \max(T_A, T_0, ..., T_i) \]
then:
\[ T_{p_{i+1}} = \max(T_A, T_0, ..., T_{i+1}) \]
for any \( i \).

In conclusion:
\[ T_{\Sigma} = T_{p_{n-1}} = \max(T_A, T_0, ..., T_{n-1}) = \max(T_A, \max_{j=0}^{n-1} (T_j)) \]

**Completion time**

Let \( m \) be the finite stream length. For our purposes, we consider a balanced pipeline, or an equivalent pipeline in which each stage has service time \( T^{(n)} \). It is sufficient to observe a pipeline with \( n = 2 \) and \( m = 2 \), whose temporal behavior is:
We recognize a \textit{filling transient phase} of duration 
\[ (n - 1) T^{(n)} \]
followed by a \textit{steady-state phase} and an \textit{emptying transient phase}, globally having a duration 
\[ m T^{(n)} \]
Therefore:
\[ T_c = (m + n - 1) T^{(n)} \]
We find the well known approximate formula:
\[ T_c \approx m T^{(n)} \]
that will be applied to all the other computations on stream, because a sort of “pipeline effect” is always present: the steady-state situation (i.e., in well balanced computations all modules are simultaneously “active”) is preceded by a transient phase which has negligible impact if \( m \gg n \).

\textbf{Example}

Let \( M \) be a module operating on an input stream of triples \((a, b, c)\) and an output stream of values \( y \), defined as follows:

\begin{align*}
M :: & \forall (a, b, c); \{ y_0 = F_0(a, b); y_1 = F_1(a, c); y = F(y_0, y_1) \}
\end{align*}

Conceptually, the parallelism degree is equal to three. In the next Section we’ll see that a partial ordering data-flow graph can easily be derived owing to the data dependences induced by \( F_0 \) and \( F_1 \) on \( F \). However, a three-stage pipeline can be defined as well, by providing that some unmodified input values are sent onto intermediate streams:

- \( S_0 :: \text{receive } (\text{input\_stream}, (a, b, c)); \text{send } (\text{stream\_01}, (F_0(a, b), a, c)) ; \)
- \( S_1 :: \text{receive } (\text{stream\_01}, (y_0, a, c)); \text{send } (\text{stream\_12}, (y_0, F_1(a, c))) ; \)
- \( S_2 :: \text{receive } (y_0, y_1); \text{send } (\text{output\_stream}, F(y_0, y_1)). \)

We will see that, with respect to a data-flow solution, the pipeline service time is the same, while the pipeline latency is higher.

\section*{7.2 Partitioning of functions and data}

In general, each pipeline stage has its own local variables, possibly modifiable. Thus, function partitioning can be coupled with data partitioning, provided that data partitions are \textit{disjoint}. If the whole \textit{n-stage} pipeline computation operates on a data structure \( S \), we must be able to obtain:

\[ S = (S_0, ..., S_n) | S_i \cap S_j = 0 \quad \forall i, j ; i \neq j \]

where distinct data partitions are encapsulated into distinct stages:
**Example**

Let us consider the following sequential computation:

```plaintext
int A[k], B[k]; input stream: int x; output stream: int y;

∀ x : { index = search (x, A);  y = \sum_{i=1}^{index} B[i] }
```

A 2-stage pipeline can be defined, where stage $M_{A}$ and $M_{B}$ encapsulates $A$ and $B$ respectively. $M_{A}$ searches $x$ in $A$ and passes the result $index$ to $M_{B}$ (assuming every $x$ is present in $A$).

### 7.3 Function replication and data partitioning: loop unfolding

In several sequential programs, we can transform an iterative computation into a sequential composition of functions by applying a loop unfolding transformation. The obtained functions are identical and, in general, operate on distinct partitions of the data structures that are manipulated by the loop. This suggest a pipeline implementation with function replication and data partitioning.

For example, the following sequential module operating on streams:

```plaintext
M :: int A[m]; input stream: int s_0; output stream: int s;
{ receive $s_0$ from input_stream;
  $s = s_0$;
  for (i = 0; i < m; i++)
    $s = F (s, A[i])$;
  send $s$ onto output_stream
}
```

can be transformed by a $k$-unfolding:

```plaintext
M :: int A[m]; input stream: int s_0; output stream: int s;
{ receive $s_0$ from input_stream;
  $s = s_0$;
  for (i = 0; i < m/k; i++)
    $s = F (s, A[i])$;
  for (i = m/k; i < 2m/k; i++)
    $s = F (s, A[i])$;
  ...  
  for (i = (k-1)*m/k; i < m; i++)
    $s = F (s, A[i])$;
  send $s$ onto output_stream
}
```

$k$ times
A *k*-stage balanced pipeline computation can be defined, where all stages execute $F$ and each stage contains a distinct partition of $A$. Intermediate values of $s$ are passed from each stage to the next one (a particular implementation of a multicast, or one-to-many, communication).

The parametric LC description is the following:

```plaintext
parallel PIPE[k]; int A[m]; channel stream[k-1];
PIPE[0] :: … channel in input_stream (1); …
PIPE[j | 0 < j < k-1] :: int A[j*m/k .. (j+1)*m/k]; int s;
    channel in stream[j-1] (1); channel out stream[j];
    while (true) do
        { receive (stream[j-1], s);
          for (i = j*m/k; i < (j+1)*m/k; i++)
            s = F (s, A[i]);
          send (stream[j], s)
        }
PIPE[k-1] :: … channel out output_stream; …
```

The value of $k$ is determined according to the cost model, i.e. imposing that the service time is equal to the interarrival time of $s_0$ values.

As anticipated in Sect. 6, this a simple, yet meaningful, data-parallel implementation of the given problem. In a subsequent Section it will be compared to the most general data-parallel solutions.

**Example**

A module $M$ operates on an input integer stream $x$ and on an output integer stream $y$. $M$ contains an integer array $A[100]$. For each $x$, $y$ is equal to the number of times a given predicate $F(x, A[i])$ is true. Let $T_{calc} = 100 \, t$ and $T_A = L_{com} = 10 \, t$.

We wish to transform $M$ according to a pipeline solution.

As we know, the optimal value of parallelism degree $n$ is given by:

$$\bar{n} = \frac{T_{calc}}{T_A} = 10$$

Correspondingly, we obtain:

$$T^{(10)}_\Sigma = T^{(10)}_{\Sigma-id} = 10 \, t \quad \varepsilon^{(10)}_\Sigma = 1$$

With the optimal parallelism degree, communications are perfectly masked.

A balanced pipeline solution, able to achieve these performance measures, exists since the $M$ loop can be transformed according to a 10-unfolding, with $A$ partitioned statically:
The $i$-th stage $S_i$ (it can be written parametrically in LC):

- contains the $i$-th partition $A^i$ of 10 elements;
- receives from $S_{i-1}$ ($S_0$: from input stream) the value of $x$ and of the partial sum $s_{i-1}$ ($S_0$: zero), initializes $s_i$, and executes the 10-iteration loop;
- sends to $S_{i+1}$ ($S_9$: output stream) the unchanged value of $x$ and the new computed value $s_i$ ($S_9$: $y$).

Notice that the whole memory capacity is equal to that of the sequential implementation.

The latency is given by:

$$L^{(10)}_\Sigma = T_{calc} + 10 \cdot L_{com} = 200 \, t$$
8. Stream-equivalent computations

In Sect. 6 we introduced the possibility to transform sequential computations, not operating on streams, into *stream-equivalent* computations by means of a 3-stage pipeline implementing an “unpack – pack” strategy. For example, given the sequential computation:

\[
M :: \quad \text{int } A[M], B[M]; \ldots;
\]
\[
\text{for } (i = 0; i < M; i++)
\]
\[
B[i] := F(A[i])
\]

the stream-equivalent “initial” pipeline computation can be defined as follows (using LC notation):

```
parallel unpack; compute; pack;

unpack:: \quad \text{int } A[M]; \textbf{channel out} \text{ input\_stream};
\quad \text{for } (i = 0; i < M; i++)
\quad \quad \textbf{send} \text{ (input\_stream, } A[i])\text{;}

compute:: \quad \text{int } x; \quad \textbf{function } F \ldots; \textbf{channel in} \text{ input\_stream (1); channel out} \text{ output\_stream};
\quad \text{for } (i = 0; i < M; i++)
\quad \quad \{ \quad \textbf{receive} \text{ (input\_stream, } x)\text{; } \textbf{send} \text{ (output\_stream, } F(x)) \quad \}

pack:: \quad \text{int } B[M]; \textbf{channel in} \text{ output\_stream (1)};
\quad \text{for } (i = 0; i < M; i++)
\quad \quad \textbf{receive} \text{ (output\_stream, } B[i])\text{;}
```

The *compute* module can be further parallelized according to one or more parallel paradigms. Consider the *unpack* module: the calculation phase is of very fine grain (*for* loop control and read A[i]) compared to the communication latency. Thus, its service time is:

\[
T_{\text{unpack}} = T_{\text{arrival-compute}} = L_{\text{com}}
\]

That is, unpack acts just as a *communicator*.

Potentially, a parallel transformation of *compute* has an optimal parallelism degree:

\[
n = \frac{T_F}{L_{\text{com}}}
\]

(the *for* loop control instructions are neglected, provided that \( F \) is of sufficiently coarse grain).

Consequently \( T_F > L_{\text{com}} \), thus \( T_{\text{com-compute}} = 0 \).

Moreover, in the zero-copy communication model the *pack* service time is negligible. Of course, according to the \( F \) definition, some computational tasks of *compute* could be delegated to *unpack* and/or to *pack*. 
9. Optimal communication grain

The formulation of Sect. 8 assumes that “unpacking” is applied to just one component of the original array. However, we could also think about “L-unpacking”, with \( L \geq 1 \) components belonging to each stream element.

This consideration introduces the more general problem of finding the optimal communication grain in any stream-based computation. Notably, the objective function to be optimized can be the completion time:

\[
T_c \sim m T
\]

In fact, the communication cost model

\[
L_{\text{com}}(L) = T_{\text{setup}} + L T_{\text{transm}}
\]

suggests to build messages whose length is able “to write off” \( T_{\text{setup}} \) (which is spent once for all). At the same time, the calculation time grain increases of \( L \) times.

Let us generalize the 3-stage pipeline unpack-compute-pack for streams of length \( L \). The interdeparture time from unpack, thus the interarrival time to compute, is expressed by:

\[
T_A = L_{\text{com}}(L) = T_{\text{setup}} + L T_{\text{transm}}
\]

Since

\[
T_{\text{calc}} = L T_F
\]

the optimal parallelism degree of compute is given by:

\[
n = \frac{L T_F}{T_{\text{setup}} + L T_{\text{transm}}}
\]

Rigorously, to optimize \( T_c \) we should refer to a specific parallel paradigm with its own cost model. However, in order to have an easy and approximate, yet general, technique to find the optimal \( L \), we can solve the equation:

\[
T_c \sim \frac{M}{L} T = \frac{M}{L} L_{\text{com}}(L)
\]

with the constraint:

\[
\frac{M}{L} \gg n
\]

This condition can be expressed as:

\[
\frac{M}{L} = a n
\]

where \( a \) is a sufficiently large constant, e.g. order of \( 10^1, 10^2 \).

Thus:

\[
\frac{M}{L} = a \frac{L T_F}{T_{\text{setup}} + L T_{\text{transm}}}
\]

from which the second order equation in \( L \):

\[
a T_F L^2 - M T_{\text{transm}} L - M T_{\text{setup}} = 0
\]
The approximation is also due to the fact that the last message has to be “padded” with some fictitious values for integer rounding reasons. However, it is acceptable.

For example, let us consider a computation with \( M = 10^3 \), \( T_F = 10^3 \tau \), \( T_{\text{setup}} = 10^3 \tau \), \( T_{\text{transm}} = 10^2 \tau \). For \( L = 1 \), no parallel version can be found because \textit{compute} is not a bottleneck, thus \( T_c \) is over \( 10^6 \tau \).

Adopting the optimal \( L \) technique, for \( a = 10 \) we obtain a good parallel version with

\[
L \sim 16 \quad n \sim 6 \quad T_c \sim 1.6 \times 10^5 \tau
\]

with a stream length of \( m \sim 60 \) and a service time \( T = L_{\text{com}} = 2.6 \times 10^3 \tau \).
10. Data-flow paradigm

The data-flow computational model is a rather intuitive parallelization strategy. On the other hand, this model has a very great significance in general, owing to the formal definition of data dependences and to the conceptual impact on parallel architectures (Part 2).

Let us remember the so-called Bernstein conditions. Consider a sequential computation consisting of two function applications:

$$\text{R}_1 = F_1 (\text{D}_1); \text{R}_2 = F_2 (\text{D}_2)$$

where "\text{;}" is the sequencing operator, and D and R denote function domain and co-domain values respectively.

The sequential computation can be transformed into an equivalent parallel computation:

$$\text{R}_1 = F_1 (\text{D}_1) \parallel \text{R}_2 = F_2 (\text{D}_2)$$

if all the following conditions hold:

$$\begin{cases}
\text{R}_1 \cap \text{D}_2 = \emptyset \\
\text{R}_1 \cap \text{R}_2 = \emptyset \\
\text{D}_1 \cap \text{R}_2 = \emptyset
\end{cases}$$

(They are sufficient conditions, that sometimes are also necessary under specific assumptions. Moreover, remember that, when Bernstein conditions are applied to parallelism in microinstructions, the third condition is eliminated, since it is unnecessarily limiting in a synchronous model of computation).

Given a sequential computation, a partial ordering of operations is built through the computation-wide application of Bernstein conditions (quadratic complexity procedure). The precedence relations express the strictly needed data dependences.

The partial ordering is represented by a data-flow graph, that is the computation graph in this model.

Example

Re-consider the module operating on streams (already implemented in pipeline, Sect. 7.1):

$$\text{M} :: \forall (a, b, c); \{ y_0 = F_0(a, b); y_1 = F_1(a, c); y = F(y_0, y_1) \}$$

Through the application of Bernstein conditions we obtain the corresponding data-flow graph (AND graph):
Data flow through the graph nodes in a pipeline-like fashion. Parallelism is naturally exploited among operations on the same stream element, for example \( F_0(a, b) \) and \( F_1(a, c) \), and among operations on different stream elements, for example \( F_0(a, b) \) and \( F(y_1, y_2) \).

The cost model is analogous to the pipeline case:

\[
T_\Sigma = \max(T_A, \max_{i=0}^{n-1}(T_i))
\]

Moreover, unlike the pipeline paradigm, the data-flow paradigm:

- is potentially able to reduce the latency compared to the sequential computation, owing to the partial ordering of operations,
- is meaningful for computations operating on single data values too.

As in the pipeline case, data are partitioned, and computations with state are compatible with this paradigm.

**Example**

A module \( M \) operates on an input integer stream \( x \) and on an output integer stream \( y \). Let \( s \) be an initialized integer variable. \( M \) is specified as follows:

\[
M:: \forall x: \{ y_1 = F_1(x); y_2 = F_2(y_1); y_3 = F_3(y_1, s); s = F_4(x, s); y = F_5(y_2, y_3) \}
\]

All integer functions have average calculation time equal to \( t \). Let \( L_{com} = t/10 \), and the architecture contains communication processors. Let the interarrival time be \( T_A = t/2 \).

We have:

\[
T_{calc} = 5t \\
T_{\Sigma-id} = T_A = \frac{t}{2}
\]

The optimal parallelism degree is:

\[
\bar{n} = \frac{T_{calc}}{T_A} = 10
\]

By applying the Bernstein conditions, we obtain the data-flow graph of figure, where node \( F_4 \) encapsulates the state variable \( s \). Notice that the computational equivalency with the sequential version is respected, since the \( s \) value in \( F_3 \) and \( F_4 \) domains corresponds to the present state, while the \( s \) value in \( F_4 \) co-domain corresponds to the next state (i.e. \( F_4 \) sends the present state value to \( F_5 \)).

All communications are masked by internal calculations.

This parallel computation is not able to exploit the optimal parallelism degree (10), because the parallelism degree of the graph, in a stream-based behavior, is equal to 5. That is, \( M \) is a bottleneck.

We could try to eliminate the bottleneck by further parallelizing all the data-flow nodes. However this is not possible for \( F_4 \) (operating on integers, and having an internal state). Thus, it is not possible to do better than \( n = 5 \).

The performance measures are:
\[ T^{(n)}_\Sigma = \begin{cases} 
\frac{5t}{n} & \text{for } n < 5 \\
t & \text{for } n \geq 5 
\end{cases} \]

\[ \varepsilon^{(n)}_\Sigma = \begin{cases} 
\frac{n}{10} & \text{for } n < 5 \\
\frac{1}{2} & \text{for } n \geq 5 
\end{cases} \]

\[ \varepsilon^{(n)}_M = 1, \text{ for any } n, \text{ since M remains a \textit{bottleneck}.} \]
11. Farm paradigm

Farm (sometimes called master-worker) is a stream parallel paradigm based on the replication of a pure function, without knowing the internal structure of the function itself. Only under particular conditions the paradigm will be extended to computations with state.

The following figure shows the conceptual scheme of a farm computation:

- a function $F$ is replicated into a set of identical modules, called workers, $W_1, \ldots, W_n$.
- emitter (E): a scheduling functionality of input stream values to be distributed to the workers,
- collector (C): a collection functionality of workers results to be sent onto the output stream.

This scheme can be the real implementation in terms of communicating modules, in which emitter and collector are single modules themselves. However, in general the conceptual scheme can be implemented in different ways, notably: emitter and collector could be decentralized (e.g. tree or ring structures), or emitter and collector could be the same module (the master-worker terminology is adopted in this case), or emitter and collector could be even implemented at a different underlying level.

Unless stated otherwise, we will assume that emitter and collector are single independent modules, as in the figure.

A pipeline effect exists among emitter, set of workers, and collector (except in the master-worker case, in which emitter and collector are interleaved in the same module).

Emitter provides to schedule each input stream value to a worker. The objective is to balance the workers load, i.e. in such a way that their processing capabilities are exploited at best, thus in order to optimize the service time. This is the fundamental feature characterizing this paradigm.

At first sight, a scheduling strategy could be the round-robin one, i.e. circular distribution. However, this strategy is not able to assure load balancing if the calculation time of $F$ has a significantly high variance depending on the data values. An on-demand strategy is much more effective: its implementation is based on the availability of workers to accept a new task. For example, each worker can communicate to emitter that it has finished to compute the
assigned task; or better, we can exploit a certain asynchrony degree by communicating that
the worker has just started to compute the assigned task.

A simple and efficient way to implement the on-demand strategy is described as follows:

E:: channel in input_stream (k); channel in available (1); channel out new_task [n];
int j; < declaration of input stream variable x type >;

while (true) do
{
  receive (input_stream, x);
  receive (available, j);
  send (new_task[j], x)
}

That is, emitter has a deterministic behavior. In LC description, available is an asymmetric
channel into which the workers communicate their availability. Notice the powerful
application of the channel array mechanism, that renders the emitter description fully
parametric with respect to the worker set parallelism degree.

Collector is basically an output stream interface, with a simple non-deterministic behavior. In
general, in a load balanced strategy, the order of the output values becomes different from the
order of the input value. When necessary, an order-preserving task could be delegated to
collector itself (ordering collector): the algorithm is based on unique identifiers, associated by
emitter to the input values. However, this solution is rather complex and unnecessarily
inefficient. The simplest and most efficient solution is non-ordering collector: it consists in
formatting the input stream elements as couples (unique identifier, value) and in sending the
couple (unique identifier, result) onto the output stream. In this way, the task of utilizing the
results in the correct order is left to the destination module. For example, if the results have to
be stored into an array, then the unique identifier is an array index. In many cases (e.g. image
processing) the unique identifier is implicitly present in the input stream values (e.g. pixel
coordinates).

A good benchmark for on-demand farms is represented by the so-called Mandelbrot set
problem. A Mandelbrot set is defined by the set of complex numbers \( c = x+jy \) such that:

\[
M = \{ c : \big| M_k( c ) \big| < \infty , \forall k \in N \}
\]

where

\[
M_{k0}(c) = 0 \\
M_{k+1}(c) = M_k(c)^2 + c
\]

The following property holds:

\[
\exists k : \big| M_k( c ) \big| > 2 \quad \Rightarrow \quad c \notin M
\]

The outcome of this computation can be graphically visualized as fractals, for example:
This computation is characterized by very high variance: the percentile variation around the mean value is of the order of several $10^2$.

### 11.1 Farm cost model

Let’s evaluate the farm service time by formally solve the farm graph considered as a queueing network.

If $T_{calc}$ is the calculation time of function $F$, the service time of each worker is

$$T_w = T_{calc} + T_{com-w}$$

If the underlying architecture supports communication overlapping, then

$$T_{com-w} = 0$$

*provided that the farm parallelism degree is equal to the optimal one* (determined as an outcome of the cost model), i.e. provided that the workers load is fully balanced.

Both the emitter and the collector service times (for a non-ordering collector) are equal to the communication latency:

$$T_E = T_{coll} = L_{com}$$

for zero-copy communications (i.e, only the emitter *send* towards the generic worker, and the collector *send* onto the output stream, have impact on the communication latency). Verify that the emitter and collector internal calculation times are negligible compared to the communication latency, i.e. emitter and collector act as “smart” *communicators*.

In general, if the input and the output streams have different types, we should write:

$$T_E = L_{com-input} \quad T_{coll} = L_{com-w}$$

For the sake of simplicity, the following analysis will assume $T_E = T_{coll}$, however it could be easily corrected when needed.

Let $T_A$ be the interarrival time to farm $\Sigma$.

The emitter and collector ideal service time is equal to $L_{com}$.

The optimal number of workers is given by the general theory (Sect. 5.4):
\[ n_{\text{opt}} = \frac{T_{\text{calc}}}{T_A} \]

provided that the farm structure is able to exploit it: this is always true if

\[ T_A \geq L_{\text{com}} \]

as it is verified in the large majority of cases.

The only case in which the farm is not able to exploit such a parallelism degree is when:

\[ T_A < L_{\text{com}} \]

Though very unusual, this case cannot be excluded in principle, i.e. the input stream is generated by multiple clients, all having interdeparture time equal to \( L_{\text{com}} \). However, in this case the emitter is a bottleneck: it cannot have an ideal service time less than \( L_{\text{com}} \). In conclusion, the emitter interdeparture time is:

\[ T_{pe} = \max(T_A, L_{\text{com}}) \]

Since the workers are load balanced, the probability that an input stream element is sent to any worker is constant and equal to \( 1/n \). Therefore:

\[ T_A = n \max(T_A, L_{\text{com}}) \quad \forall i = 1 \ldots n \]

\[ T_{pe} = \max(T_w, n \max(T_A, L_{\text{com}})) \quad \forall i = 1 \ldots n \]

The collector interarrival frequency is given by:

\[
\frac{1}{T_A} = \sum_{i=1}^{n} \frac{1}{T_{pe}} = \min\left(\frac{n}{T_w}, \frac{n}{n \max(T_A, L_{\text{com}})}\right) = \min\left(\frac{n}{T_w}, \frac{1}{\max(T_A, L_{\text{com}})}\right)
\]

which represents also the effective farm bandwidth, since collector is not a bottleneck:

\[ B_x = \min\left(\frac{n}{T_w}, \frac{1}{\max(T_A, L_{\text{com}})}\right) \]

Thus, the farm service time is:

\[ T_x = \max\left(\frac{T_w}{n}, \max(T_A, L_{\text{com}})\right) \]

The best number of workers as possible is the \( n \) value that maximizes the bandwidth:

\[ n = \frac{T_w}{\max(T_A, L_{\text{com}})} \]

We have verified the general theorem (Sect. 5.4) in the most frequent case in which emitter is not a bottleneck, thus when the farm structure is really able to exploit the optimal degree of parallelism:

\[ n_{\text{opt}} = \frac{T_{\text{calc}}}{T_A} \]

where \( T_w = T_{\text{calc}} \) since, with the optimal parallelism degree, the worker communications are fully overlappable to the internal calculation time.

Thus

\[ T^{(n_{\text{opt}})} = T_A \]
If the ratio $\frac{T_{\text{calc}}}{T_A}$ is not integer,

$$n_{\text{opt}} = \left\lfloor \frac{T_{\text{calc}}}{T_A} \right\rfloor$$

if the goal to maximize the bandwidth (most frequent case). Instead:

$$n_{\text{opt-e}} = \left\lfloor \frac{T_{\text{calc}}}{T_A} \right\rfloor$$

if the goal is to maximize the efficiency.

In the rare cases in which emitter is a bottleneck, we cannot do better than:

$$n_{\text{sub-opt}} = \frac{T_{\text{calc}}}{L_{\text{com}}}$$

For the **completion** time evaluation, the analysis is quite similar to the pipeline case: similar transient and steady-state phases are recognized. Therefore:

$$\text{for } m >> n : \quad T_c \sim m T^{(n)}$$

The latency is greater than the sequential latency:

$$L_{\text{farm}} \sim T_{\text{calc}} + L_{\text{com-input}} + L_{\text{com-w}}$$

If the definition of function $F$ includes some constant (read-only) data structures, they must be fully replicated in all workers, thus the farm memory capacity is $n$ times the sequential memory capacity.

Latency and, above all, memory capacity are **the main weaknesses** of farm paradigm when applicable (pure functions).

**Example 1**

A stream parallel program consists of a process Q operating on integer arrays $A[N]$, $B[N]$, $C[N]$, with $N = 10^3$. A and B values are received from two distinct input streams, and C is sent onto the output stream. The stream length is $m = 10^3$.

For each couple (A, B), Q is defined as follows:

$$\forall i = 0 \ldots N - 1 : C[i] = F(A[i], B)$$

The program is executed on a parallel architecture, having 128 processing nodes with communication processor and 3,33 GHz clock frequency. The communication cost model is: $T_{\text{setup}} = 10^3 \tau$ and $T_{\text{transm}} = 10^2 \tau$, with zero-copy communications.

Function $F$ has processing time exponentially distributed with mean value $5*10^3 \tau$.

The Q interarrival time is equal to $500*10^3 \tau$.

Let us parallelize Q as the farm structure with the best bandwidth as possible.

Input values A and B are received by the emitter from distinct channels with AND logic (data-flow logic), since both values are needed. Therefore, interarrival time $T_A = 500*10^3 \tau$ refers to both input streams.

The optimal parallelism degree is:
\[ n = \frac{T_{\text{calc}}}{T_A} = \frac{N T_F}{T_A} = \frac{5 \times 10^6}{5 \times 10^5} = 10 \]

In order to verify whether a farm structure is able to exploit it, let us evaluate the emitter ideal service time. Emitter distributes the couple (A, B) in the same message, in order to minimize the communication latency:

\[ T_E = L_{\text{com}}(2N) = T_{\text{send}}(2N) = T_{\text{setup}} + 2N T_{\text{transm}} = 10^3 \tau + 2 \times 10^3 \times 10^3 \tau \sim 2 \times 10^5 \tau \]

Therefore, emitter is not a bottleneck:

\[ T_E < T_A \]

as it is for the collector, for which the message length is \( N \).

The whole parallelism degree:

\[ n_e = n + 2 = 12 \]

is much lower than the number of physical nodes, so one process per node can be allocated.

We are able to achieve the maximum bandwidth and efficiency:

\[ T_{\Sigma}^{(10)} = T_{\Sigma-1d}^{(10)} = T_A = 5 \times 10^5 \tau \]

\[ \varepsilon_{\Sigma}^{(10)} = 1 \]

The completion time is:

\[ T_c \sim m T^{(10)} = 5 \times 10^8 \tau = 5 \times 10^8 \times \frac{10^{-9}}{3,33} = 150 \text{ msec} \]

The latency:

\[ L_{\Sigma} = T_{\text{send}}(2N) + T_{\text{calc}} + 2 T_{\text{send}}(N) = 5,3 \times 10^6 \tau = 1,6 \text{ msec} \]

**Example 2**

Let us parallelize the following sequential program:

```c
int A[M], B[M], C[M];
for (i = 0; i < M; i++)
    C[i] = F(A[i], B);
```

It can be realized as a 3-stage pipeline unpack-compute-pack, where compute has a farm structure. Array B is replicated in all the farm workers, while unpack generates a stream of A components (better, each element contains multiple components, see Sect. 9).

Express this parallel program in LC, and evaluate it.

**11.2 Farm vs pipeline and data-flow**

Given a function to be parallelized, let us compare a farm implementation with a pipeline and a data-flow implementation, provided that they are feasible according to the function internal form.

For example, let us consider a module operating on streams with the following characteristics:
• \( T_A = t \),
• \( L_{com} < t \),
• \( T_{calc} = 16t \),
• function expressed as the sequential composition of four functions \( F_1, F_2, F_3, F_4 \), all with the same processing time equal to \( 4t \).

A 4-stage pipeline solution has effective service time equal to \( 4t \), while the ideal service time is \( T_A = t \). Thus \( \varepsilon = 0.25 \).

A farm solution with parallelism degree
\[
    n_W = \frac{T_{calc}}{T_A} = 16 \quad n = n_W + 2 = 18
\]
is really able to achieve the ideal service time \( T_A = t \), with efficiency equal to one. Also the latency is in favour of the farm solution.

Analogous considerations can be done with respect to a data-flow computation.

In conclusion, if specific constraints are not forced (e.g. the initial application workflow), the farm parallelization of a pure function is always better than the pipeline or data-flow implementations, even when they are feasible according to the function internal form.

Moreover:

a. If each pipeline stage is parallelized with a farm structure, we achieve the same parallelism degree and service time of the pure farm, however with an higher latency.

b. Similar considerations can be applied to solutions data-flow + farm, however in this case it is possible to obtain a lower latency.

c. The same considerations apply if the pipeline or data-flow solutions are not balanced, and the farm paradigm is used to balance them.

d. If the internal parallelism degree of pipeline stages or data-flow nodes is rounded up, the pure farm service time is lower or equal, with the same service time.

### 11.3 Farm with internal state

Consider a farm computation in which a data structure \( S \) is replicated in all workers. A variant of the pure farm paradigm can be defined in the case that \( S \) is modifiable, i.e. an internal state exists, provided that the state modification operations are not too frequent.

Let \( x \) and \( y \) denote the generic input stream and output stream element, respectively. Let us assume that two operations can be executed

- a function with state, such as \( S = F_1(x, S) \).
- a pure function, such as \( y = F_2(x, S) \).

The latter is implemented according to the usual farm structure.

The former must be executed by all workers, in order to maintain the \( S \) copies consistent. The emitter can perform a multicast communication of \( x \) value to all workers. Of course, the operation \( F_1 \) does not contribute to increase bandwidth.
Each input stream element contains \((op, x)\), where \(op\) denotes \(F_1\) or \(F_2\). Let \(p\) the probability of the event \((op = F_i)\).

The service time is not:

\[
\frac{p \, T_{F_1} + (1 - p) \, T_{F_2}}{n}
\]

The correct formula is:

\[
T_s = p \, T_{F_1} + (1 - p) \frac{T_{F_2}}{n}
\]

The optimal degree of parallelism is found by imposing \(T_S = T_A\):

\[
n \ (p) = \left[ \frac{(1 - p) \, T_{F_2}}{T_A - p \, T_{F_1}} \right] \]

In order to have a meaningful farm, it must be

\[
p < \frac{T_A}{T_{F_1}}
\]

For \(p = 0\) we are in the usual stateless farm situation:

\[
n \ (0) = \left[ \frac{T_{F_2}}{T_A} \right]
\]
12. Functional partitioning with independent workers

Consider a sequential module operating on streams defined as follows:

\[
M :: \text{channel in input\_stream (…)}; \text{channel out output\_stream}; \text{int op}; \ldots \\
\text{while (true) do} \\
\quad \{ \text{receive (input\_stream, (op, x));} \\
\quad \quad y = \text{case op of} \\
\quad \quad \quad 1 : F_1(x) \\
\quad \quad \quad \ldots \\
\quad \quad k : F_k(x):
\quad \text{send (output\_stream, y)} \\
\quad \}
\]

Functions \( F_1, \ldots, F_k \) have processing times \( T_1, \ldots, T_k \) and occurrence probabilities \( p_1, \ldots, p_k \). A simple functional partitioning parallelization scheme with independent workers can be defined: \( k \) workers \( W_1, \ldots, W_k \) are provided, where generic \( W_i \) is specialized to execute \( F_i \). An emitter distributes input tasks to workers according to the \( op \) value, and a collector sends the results onto the output stream. In other words, it is a server partitioning scheme (Sect. 5.2.3). Remember that pipeline and data-flow are functional partitioning paradigms too, however they are based on the interaction of processing modules, while now the workers are independent.

As in any paradigm based on (function and/or data) partitioning, a potential load unbalance problem exists which lowers the processing bandwidth. In order to verify it, we apply the server partitioning theorem to detect possible bottlenecks. If \( T_A \) is the interarrival time, then

\[
T_A = \frac{T_A}{p_i}
\]

Emitter and collector have ideal service time equal to \( L_{com} \), thus they are not bottlenecks in all the computations of interest. The validity condition of theorem Version 1:

\[
\forall i : \rho_i \leq 1 \implies \forall i : p_i \cdot T_i \leq T_A
\]

assures that no bottleneck exists, then the whole effective bandwidth is equal to the ideal one:

\[
B^{(k)} = \frac{1}{T_A}
\]

It is not important that all workers are perfectly balanced (i.e. all the utilization factors are equal), it is just sufficient that \( \forall i : \rho_i \leq 1 \).

Instead, if the validity conditions of theorem Version 2 are verified, then the presence of a bottleneck increases the service time that becomes greater than \( T_A \).

As for the other functional partitioning paradigms (pipeline, data-flow), bottlenecks can be eliminated by a farm parallelization. Very similar results are achieved (Sect. 11.2): the pure farm solution, with “general-purpose” workers, is characterized by a greater or equal bandwidth, thus it is preferable unless application/architecture dependent constraints force the adoption of a functional partitioning frame.
13. Collective communications

Besides symmetric and input-asymmetric, deterministic and non-deterministic communications, the implementation of parallel programs requires the adoption of other communication forms, called collective communications because they are based on the cooperation of several modules according to predefined patterns. Data-parallel paradigms will apply collective communications intensively, however other paradigms can use them (for example, see Sections 7.3 and 11.3 for applications of the multicast communication).

The main collective communication primitives are:

1. **multicast**: a module sends the same message to a specified set of destination modules (it is sometimes called broadcast, when the destination set coincides with the partners entirety);

2. **scatter**: given a data structure $A$ local to module $P$, $P$ sends $A$ partitions to a specified set of destination modules. For example, partitions of a matrix $A$ could be blocks of rows (distribution *by rows*), or blocks of columns (distribution *by columns*), or partial square/rectangular blocks (distribution *by blocks*), or blocks composed of interleaved elements according to a “mod n” law, analogous to an interleaved memory (*cyclic* distribution);

3. **gather**: given a set of data structures $A_1$, ..., $A_n$ local to modules $P_1$, ..., $P_n$ respectively, a module $P$ builds a unique data structure $A$ with components $A_1$, ..., $A_n$. For example, $A_1$, ..., $A_n$ are blocks of rows, or blocks of columns, or square/rectangular blocks, or cyclic blocks of a matrix $A$.

In turn, these primitives are implemented by means of deterministic and non-deterministic, symmetric and asymmetric communications, and, in some cases, by proper architectural supports provided by the communication processors and/or by the communication network.

Let us study the collective communication cost model. Let $k$ be the size of a data structure to be communicated, and $n$ the number of destination modules or the number of partitions of size $g = k/n$, according to the various cases.

13.1 Multicast

The best implementation has latency:

$$T_{multicast} = T_{send}(k)$$

on condition that the run-time support exploits $n$ communications *in parallel*. This is possible only if the underlying architecture supports this facility, as it happens in some limited degree interconnection networks, or in some standard protocols (IPv6).

If this architectural support is not available, the most simple implementation is a *sequential* implementation of $n$ point-to-point communications. The latency and the service time are linear in $n$:

$$T_{multicast} = n \cdot T_{send}(k)$$

If a proper number of nodes is available, a *tree structured implementation* is a very interesting alternative, in which *the latency is logarithmic in n and the service time is constant*. For
example, consider a binary tree of $n + n$ communicating modules, where each module executes two point-to-point communications sequentially. The latency is given by:

$$T_{\text{multicast}} = 2 \lceil \log_2(n) \rceil T_{\text{send}}(k)$$

while the service time is just:

$$2 T_{\text{send}}(k)$$

That is, owing to the pipeline effect, the communication bandwidth is low and constant, and comparable to a primitive parallel implementation.

### 13.2 Scatter

In the most simple case the implementation is sequential, i.e. $n$ sequential point-to-point communications to distinct modules, with message size $g = k/n$. The latency and the service time are linear in $n$:

$$T_{\text{scatter}} = n T_{\text{send}}(g) = n T_{\text{setup}} + k T_{t_{\text{rasm}}}$$

Tree structured schemes are possible, where each node implements successive partitionings of the whole data structure. The latency is sensibly improved, since it is logarithmic in $n$:

$$T_{\text{scatter}} = 2 \sum_{i=1}^{\lceil \log_2 n \rceil} T_{\text{send}}(\frac{k}{2^i})$$

while the constant service time is equal to the root service time:

$$2 T_{\text{send}}(k/2) = 2 T_{\text{setup}} + k T_{t_{\text{rasm}}}$$

which is a modest improvement compared to the implementation with linear latency.

### 13.3 Gather

At first sight, the gather implementation could be independent of $n$, since the source nodes send the respective partitions in parallel. However the structure composition in the destination module (node) represents a bottleneck, at the process or a the firmware level. In other words, even with zero-copy point-to-point communications, the receive latency cannot be neglected in the gather cost model.

Though some modest optimizations can be done, a reasonable evaluation of the gather latency is:

$$T_{\text{gather}} \sim T_{\text{scatter}}$$
14. Data parallel paradigm

A data parallel computation, on streams and/or on single data values, is characterized by partitioning of (large) data structures and function replication. Moreover, data replication is sometimes adopted as well.

We will see that data parallelism is a very powerful and flexible paradigm, that can be exploited in several forms according to the strategy of input data partitioning and replication, to the strategy of output data collection and presentation, and to the organization of workers (independent or interacting), though this flexibility is inevitably paid with a more complex realization of programs and of programming tools. The knowledge of the sequential computation form is necessary, both for functions and for data, and this is the main reason for the parallel program design complexity. Besides service time and completion time, latency and memory capacity are optimized compared to farms and other stream parallel paradigms, though a potential load unbalance exists.

As a generic example, let us consider a data parallel computation operating on a bimimesional array $A[M][M]$. Assume that $A$ is partitioned by square blocks among $n$ workers, with $g = \frac{M^2}{n}$ partition size:

All workers apply the same function $F$ to the corresponding partitions in parallel.

Unless statically allocated, input data are distributed by program through a scatter, and possibly a multicast, operation. Output data may be collected through a gather operation, however other possibilities exist, notably reduce operations.

The simplest data parallel computation is the so-called map, in which the workers are fully independent, i.e. each of them operates on its own local data only, without any cooperation during the execution of $F$.

More complex, yet powerful, computations are characterized by workers that operate in parallel and cooperate (through data exchanges): this is because data dependences are imposed by the computation semantics. In this case we speak about stencil-based computations, where a stencil is a data dependence pattern implemented by inter-worker communications. The form of a stencil may be:

- statically predictable, or dynamically exploited according to the current data values,
- in the static case: fixed throughout the computation, or variable from a computation step to the next one (however, it is statically predictable which stencil will occur at each step).

Initially we refer to data parallel computations operating on single data structures, while data parallel computation on streams will be discussed in a subsequent Section.
14.1 The Virtual Processors approach to data parallel program design

The complexity in data parallel program design can be “dominated” through a systematic and formal approach which, starting from the sequential computation, is able to derive the basic characteristics of one or more equivalent data parallel computations. The main characteristics to be recognized are:

1. how to distribute input data (partitioning and possibly replication), and how to collect output data,
2. how to understand that a map solution is possible,
3. how to understand that a stencil-based solution is possible, and, in this case, how to understand the stencil form.

Of course, according to our methodology, the answers to such questions must be accompanied by the a cost model.

It is important to notice that, though formal and systematic, in general the approach is not able to support a fully automatic compilation of data parallel programs starting from the sequential computation, especially taking into account the objective of achieving good performance measures. However:

a) there are some classes of algorithms for which an automatic, efficient solution exists;

b) when it does not exist, the approach is able to give many information and properties about the data parallel program structure, which render much easier the designer work, and which can be effectively exploited by means of a limited, high-level set of annotations. Such annotations are able to drive automatic compilation strategies.

We adopt the so-called Virtual Processors approach to data parallel program design, consisting of two phases:

1) according to the structure of sequential computation, derive an abstract representation of the equivalent data parallel computation, which is characterized by the maximum theoretical parallelism compatible with the computation semantics. The modules of this abstract version are called (for historical reasons) virtual processors. Data are partitioned among the virtual processors according to the program semantics. During the first phase, we are not concerned with the efficiency of the virtual processors computation: in the majority of cases, the parallelism exploited in the virtual processors version is (much) higher than the actually optimal one. However, the goal of the virtual processors version is only to capture the essential features of the data parallel computation (characteristics 1, 2, 3 above). Because it is a maximally parallel version, often the grain size of data partitions is the minimal one, but this is not necessarily true in all cases (however, it is of a grain (much) smaller than the actual one);

2) map the virtual processor computation onto the actual version with coarser grain worker modules and coarser grain data partitions. This executable, process-level program is parametric in the parallelism degree \( n \). In order to effectively execute the program, the degree of parallelism must be chosen at loading time, according to the resources that are allocated. The paradigm cost model is fully exploited during this second phase, both for deriving the parametric version and for characterizing the resource allocation strategy.

Some examples will be used to understand the Virtual Processors approach.
**Example 1**

Let us first consider computation that is suitable for the *map* paradigm. Consider the following sequential loop computation:

```c
int A[M], B[M];
∀ i = 0 .. M – 1:
    B[i] = F(A[i])
```

We are interested in recognizing the potential parallelism between iterations. This can be done by applying the *Bernstein conditions* to the iterations set, in order to detect possible data dependences among iterations. Of course, the loop indexes (and other sequential control values) are *free variables* that are not sources of data dependences.

Formally, the Bernstein conditions are applied to the *loop unrolled* sequential version:

```c
B[0] = F(A[0]);
...
B[M-1] = F(A[M-1])
```

or, more concisely:

```c
B[i] = F(A[i]);
B[j] = F(A[j])
```

for any \(i, j\) with \(i ≠ j\).

The outcome of the Bernstein analysis is that all iterations are *fully independent* each other, thus, the *map* paradigm can be applied.

Formally, a linear array of \(M\) virtual processors is defined, \(VP[M]\), where \(VP[i]\) is capable of executing \(F\) and encapsulates \(A[i]\) and \(B[i]\). A LC abstract representation could be:

```c
parallel VP[M]; int A[M], B[M];
VP[i] :: int A[i], B[i];
B[i] = F(A[i])
```

achieving \(O(1)\) completion time vs \(O(M)\) of the sequential program.

It should be evident, in this simple example, that the virtual processors version allows us to understand *how to exploit parallelism* and *how to distribute data*. The input data distribution is more critical than the output collection one, since data collection depends clearly on the data declaration and on the input data distribution itself.

Data distribution and collection are not written explicitly in this abstract version. It is sufficient that their characteristics are defined in an abstract way.

In this computation, the *maximal parallelism* \((M)\) implies the *minimal data grain* too (one element of each input data structure).

According to the methodology studied in the previous Sections, it is clear that, in general, \(M\) is not the optimal parallelism degree. For this reason, we pass to the *second phase*, i.e. map the virtual processors computation into the parametric actual version with coarser grain.
First of all, the optimal parallelism degree \( n \) (in general < \( M \)) is evaluated according the general methodology and to the cost model of the used data parallel paradigm (the cost model will be studied in Sections dedicated to the specific data parallel forms).

Then, it is easy, and automatic, to map the virtual processors version onto the actual parametric version with \( n \) workers and with data partitions of size \( g = M/n \). In this version we include also data distribution and collection functionalities (in this example: scatter and gather, by linear blocks. In order to focus on the essential concepts, we avoid to express these functionalities):

```
parallel scatter, gather, worker[n]; int A[M], B[M]; param g = M/n;
worker[i] :: int A[i*g .. (i+1)*g-1], B[i*g .. (i+1)*g-1];
for (j = i*g; j < (i+1)*g; j++)
    B[j] = F(A[j])
scatter :: …
gather :: …
```

**Example 2**

Let us consider the following sequential algorithm for the matrix-vector product:

```
int A[M, M], B[M], C[M];
\forall i = 0 .. M - 1:
    \{ C[i] = 0;
        \forall j = 0 .. M-1:
```

This algorithm is suitable for both a map and a stencil parallelization.

The Bernstein analysis of the computation shows that the parallelism can be exploited among the \( M \) iterations of the outermost loop:

```
\forall j = 0 .. M-1:  C[0] = C[0] + A[0, j] * B[j] \}
…
```

Thus a linear array of \( M \) virtual processors, VP[\( M \)], is defined.

They are fully independent, thus a map paradigm can be applied, provided that VP[i] contains

- the single element \( C[i] \),
- all \( M \) elements of the \( i \)-th row of \( A \),
- all \( M \) elements of \( B \).
That is, a map computation is defined by replicating $B$ among all VPs:

```c
parallel VP[M]; int A[M][M], B[M], C[M];
VP[i] :: int A[i][*], B[*], C[i];
    for (j = 0; j < M; j++)
```

achieving $O(M)$ completion time vs $O(M^2)$ of the sequential program.

We have expressed the maximal parallelism of the computation ($M$). The data grain, though not the minimal one (each VP contains also an A row and whole B), is what is needed for the exploitation of such maximal parallelism.

Notice that, in applying the Bernstein analysis, our initial choice has been to verify the existence of a parallelism degree equal to $M$. In principle, in presence of bidimensional arrays, the existence of a parallelism degree equal to $M^2$ cannot be excluded a priori. The reader can verify that, though conceptually possible, the $VP[M][M]$ version has a completion time not greater than the $VP[M]$ version, i.e. it can be proved that only $M$ over $M^2$ virtual processors would be able to be executed in parallel. In fact, the evaluation of this alternative solution (parallelism degree $M^2$) belongs to the Virtual Processor approach. The important fact is that, reasoning according to the systematic approach, the space of possible alternatives is very limited. From this point of view, the area of Parallel Algorithms is of very great help.

Mapping the virtual processors version onto the actual parametric version with $n$ workers (optimal degree of parallelism obtained according to the cost model), we have to describe also the scatter distribution of $A$ ($n$ row blocks of $g=M/n$ rows) and of $C$ ($n$ linear blocks of $g$ elements), the multicast distribution of $B$ (all $M$ elements), and the gather collection of $C$ ($n$ linear blocks of $g$ elements).

```c
parallel distribution, collection, worker[n]; int A[M][M], B[M], C[M];
param g = M/n;
worker[i] :: int A[i*..(i+1)*g-1][*], B[*], C[i*..(i+1)*g-1];
    for (j = 0; j < M; j++)
```

distribution :: …
collection :: …

The detailed description and evaluation is left as an exercise.

Among the possible alternative virtual processors versions, in this case we realize that also the stencil-based paradigm can be applied. Consider again the outcome of the Bernstein analysis, according to which the parallelism can be exploited among the $M$ iterations of the outermost loop:

```
{ C[0] = 0;
    ∀ j = 0 .. M-1:  C[0] = C[0] + A[0, j] * B[j] }
```
\{ C[1] = 0; \

\ldots

\{ C[M-1] = 0; \\

We realize that partitioning of \( C \) (single element) and of \( A \) (single row) is common to any solution with parallelism degree \( M \). However, we have a degree of freedom about \( B \) distribution: instead of \( B \) replication, let us try \( B \) partitioning. Since \( B \) is a unidimensional array, the solution is that the \( single \ B[i] \) is encapsulated into \( VP[i] \).

This means that, though \textit{executable in parallel}, the \( M \) virtual processors are not fully independent. That is, the generic \( VP[i] \) performs an execution sequence like

\ldots

\{ C[i] = 0; \\
C[i] = C[i] + A[i, 0] \ast B[0]; \\
C[i] = C[i] + A[i, 1] \ast B[1]; \\
\ldots \\
C[i] = C[i] + A[i, j] \ast B[j]; \\
\ldots \\
C[i] = C[i] + A[i, M-1] \ast B[M-1]; \\
\}

\ldots

At the beginning of the \( j \)-th iteration, each VP needs the value of \( B[j] \), which is \textit{non-local} (except during the iteration in which \( i = j \)). This means that at the beginning of \( j \)-th iteration, the value of \( B[j] \) must be rendered available to all the other VPs, e.g., in a message-passing environment, \( VP[j] \) multicasts \( B[j] \) to all the other VPs. The corresponding communication pattern, i.e. \( M^2 \) communication channels, is a \textit{static and variable} stencil:

\begin{verbatim}
parallel   VP[M]; int A[M][M], B[M], C[M]; channel stencil [M][M];
VP[i] ::   int A[i][*], B[i], C[i]; int x;
          channel in stencil [*/i] (1), channel out stencil [i]/*;
          for (j = 0; j < M; j++)
             \{ if (j = i) then \{ x = B[i]; < multicast (stencil [i]/*, x) > \} \\
              else (receive (stencil [j][i], x); \\
                     C[i] = C[i] + A[i][j] \ast x \}
\end{verbatim}

where \textit{multicast} has to be expanded in the actual version.

Mapping the virtual processors version onto the actual parametric version with \( n \) workers (optimal degree of parallelism obtained according to the cost model), we have to describe also the \textit{scatter} distribution of \( A \) (\( n \) row blocks of \( g=M/n \) rows), of \( C \) (\( n \) linear blocks of \( g \)
elements), and of B \((n \text{ linear blocks of } g \text{ elements})\), and the \text{gather} collection of C \((n \text{ linear blocks of } g \text{ elements})\):

```c
parallel scatter, gather, worker[n]; int A[M][M], B[M], C[M]; param g = M/n;
channel actual_stencil [n][n];

worker[i] :: int A[i*\(g \cdot (i+1) \cdot g-1\)][*], B[i*\(g \cdot (i+1) \cdot g-1\)], C[i*\(g \cdot (i+1) \cdot g-1\)];
channel in actual_stencil [*][i] (1), channel out actual_stencil [i][*];
for (j = 0; j < M; j++)
    { if (j = i) then \{ x = B[i]; <multicast (stencil [i][*], x) > \}
    else (receive (stencil [j][i], x);
    C[i] = C[i] + A[i][j] * x \}
```

The detailed description and evaluation is left as an exercise, including

- the \text{multicast} implementation, according to the cost model, in order to achieve the best completion time as possible.
- the optimization consisting in the minimization of communications to properly take the actual partition grain into account.

For this algorithm a much cheaper stencil form can be found, i.e. a static and fixed stencil consisting in a linear \text{ring} of channels. At each step, every VP sends its local B element to the neighbour VP, which uses the received value to compute the local result partial value. Though all VPs operate on different B elements at each step, this algorithm works owing to the \text{associative} property of addition.

The virtual processors description is:

```c
parallel VP[M]; int A[M][M], B[M], C[M]; channel stencil [M];
VP[i] :: int A[i][*], B[i], C[i]; int x;
channel in stencil [i] (1), channel out stencil [(i+1) mod M];
for (j = 0; j < M; j++)
    { send (stencil [(i+1) mod M], B[i]);
      receive (stencil [i], B[i]);
```

The actual parametric version is left as an exercise.

\textit{Owner computes rule}

This example allows us to introduce the so-called “\textit{owner computes rule}”, which is usually applied in \textit{stencil-based} computations.

The owner computes rule is based on the concept of \textit{data ownership}. Each module gets the ownership of the input distributed data, i.e. it is the only one that can modify those data. The owned elements are always up-to-date and need no communication or synchronization before
being accessed by the process. Usually the rule is expressed with respect to assignment statements that defines updates of variables in a program: the module that owns the left-hand side element is in charge of performing the calculation. Therefore the owner module is the only one that can update its owned data and moreover is responsible for performing all the operations on them. In case those operations required non-owned data, communications between processes are necessary.

In a local environment cooperation model, this rule is quite “natural”. Non-owned variables are non-local variables, therefore, if needed for the computation, they must be acquired from the respective virtual processors. For example, in a message-passing environment, at each step every VP, encapsulating \( x \), sends \( x \) to those VPs which need \( x \) during such step. In Example 2, \( C[i], A[i][*] \) and \( B[i] \) can be utilized directly by \( VP[i] \) only; at \( j-th \) step \( VP[j] \) multicast \( B[j] \) to all the other VPs.

Besides to be “natural”, the owner computes rule is of great help in finding the virtual processors set: in Example 2, the choice of \( M \) virtual processors is driven by the observation that the distinct \( M \) components of \( C \) are assigned. The owner computes rule is not the only possible strategy, however very rare examples exist for which optimizations can be done according to a different strategy.

**Observation on the variety of data parallel solutions**

Especially for data parallel computations, many forms exists and, for each form, some variants are possible, e.g. with or without replicated data, with or without gather, and so on. In every specific case, we are able to derive the cost model. However, the existence of such variety of solutions is the main reason for the impossibility to conceive an automatic compilation in the general case. Only for some domain-specific systems, an automatic compilation could be implemented.

In the ASSIST programming environment (University of Pisa), parallel computations are expressed in an abstract form by using parallel programming constructs corresponding to the paradigm-oriented annotations. Starting from this parallel abstract version (which is rather close to the sequential one, but it is not the sequential one), compiler transformations and optimizations are now feasible.

### 14.2 Map cost model

Each worker has a calculation time

\[ T_{map-calc} = g \ T_{F} \]

which is the calculation time of the whole workers set. The effective service time must include the impact of data distribution and collection. Because we are assuming to operate on single data structures, initially we do not consider any parallelism exploitation among the scatter-compute-gather phases: Therefore the completion time is given by:

\[ T_c = T_{scatter}(n, g) + T_{map-calc} + T_{gather}(n, g) \sim 2 \ T_{scatter}(n, g) + T_{map-calc} \]

Let us consider a map without replicated data, and let \( k \) the size of the partitioned data structure:

\[ T_c = 2n(T_{setup} + gT_{trans}) + gT_{F} = 2nT_{setup} + 2kT_{trans} + T_{F} \frac{k}{n} \]
The optimal parallelism degree can be determined by imposing the first-degree derivative equal to zero:

\[ \frac{dT_c}{dn} = 2T_{\text{setup}} - T_F \frac{k}{n^2} = 0 \]

We can verify that the minimum exists. Rounding up the result:

\[ n_{\text{opt}} = \left\lceil \frac{kT_F}{\sqrt{2T_{\text{setup}}}} \right\rceil \]

An optimization can be obtained, since a limited pipeline-effect exists: a part of the collective communications is overlapped to calculation. Precisely, all send primitives in the gather implementation, except the last, are overlapped. In this case:

\[ T_c = T_{\text{scatter}}(n, g) + g T_F + T_{\text{send}}(g) = (n + 1) T_{\text{send}}(g) + g T_F \]

from which:

\[ n_{\text{opt}} = \left\lceil \frac{k(T_F + T_{\text{transm}})}{T_{\text{setup}}} \right\rceil \]

For coarse grain calculations and/or data, \( n_{\text{opt}} \) might assume quite large values, often much larger than the processing nodes number. It is worth observing that the function

\[ T_c = T_c(N) \]

is rather “flat” around the minimum, i.e. a wide interval of acceptable \( n \) values around the minimum exist. This allows the designer to find more reasonable quasi-optimal values of \( n \).

**Example**

Consider again Example 1 of previous Section:

```c
int A[M], B[M];
for ( i = 0; i < M; i++ )
    B[i] = F(A[i]);
```

where \( M = 256K \), \( T_F = 1.000 \tau \), and \( T_{\text{send}}(L) = T_{\text{setup}} + L T_{\text{transm}} = 1000\tau + 100L\tau \).

Applying the second formula for \( n_{\text{opt}} \):

\[ n_{\text{opt}} = 387 \quad T_{c_{\text{min}}} = 28,3\times10^6 \tau \]

For \( n \) in the range \((200 - 1000)\), \( T_c \) varies very slowly around the minimum \((28,8\times10^6 \tau - 28,4\times10^6 \tau)\). For example, a machine with 256 nodes offers almost the same completion time of one with 387 nodes.

**Map vs farm**

All the stateless functions that can parallelized through the map paradigm can be parallelized through a farm as well (by generating the stream in a unpack-compute-pack fashion). This equivalence is a strength point for the parallelization methodology:

- few parallel versions exist for the same computation, so we are able to easily compare them from several viewpoints.
In our case, with the same completion time, *farm* has the advantage of load balancing for high variance computations, while *map* is better in terms of latency and memory capacity. Moreover, for stream-based computations, we’ll see that often the data parallel implementation might require a higher parallelism degree due to the implementation of scatter and gather.

### 14.3 A benchmark for static, fixed stencils: nearest neighbours convolution

The *convolution* computational pattern recurs in many scientific application kernels, notably in numerical solution of partial derivative differential equations. The value of every point in a discrete space is updated by the function applied to the point itself and to some *neighbour* points; this is repeated until a given convergence condition is satisfied. For example, in a bidimensional space:

```c
int A[M][M], old_A[M][M]; init old_A = A;
repeat
  \forall i = 0..M-1:
    \forall j = 0..M-1:
      A[i,j] = F (old_A[i, j] , old_A[i-1, j] , old_A[i+1, j] ,
                   old_A[i, j-1] , old_A[i, j+1] );
  old_A[i,j] = A[i,j];
until convergence (A)
```

The convergence condition is a proper predicate that must be satisfied by all the elements of A (e.g., for all points, the absolute value difference between the current and the previous iteration value is less than a given threshold). It can be expressed as a *reduce* computation, which in turn is suitable for parallelization (see a subsequent Section).


The application of the *owner computes rule* implies that $VP[i,j]$ is the only VP authorized to modify $A[i,j]$ and $old_A[i,j]$. To implement the 4-neighbours stencil, at each step $VP[i,j]$ must receive $old_A[i-1,j]$, $old_A[i+1,j]$, $old_A[i,j-1]$, and $old_A[i,j+1]$ from $VP[i-1,j]$, $VP[i+1,j]$, $VP[i,j-1]$, and $VP[i,j+1]$ respectively.

That is, at each step every $VP[i,j]$ knows the four neighbours to which $old_A[i,j]$ has to be sent and from which $old_A[i-1,j]$, $old_A[i+1,j]$, $old_A[i,j-1]$, $old_A[i,j+1]$ have to be received.

Once the virtual processor version is known, we can pass to the actual parallel version with parallelism degree $n$, in which each worker contains a partition of $A$ and $old_A$ of size $g = M^2/n$.

In the figure, we assume that a square block partitioning is adopted. Only the $\sqrt{g}$ **elements on each borders** have to be exchanged.

The figure shows only the exchange with the northern and southern neighbours, after which the exchange with the eastern and western neighbours will occur.
Communication channels are asynchronous to assure deadlock avoidance. Assuming zero-copy communications, the completion time is given by:

\[ T_c = s \left[ 4 T_{send}(\sqrt{g}) + g T_F + T_{reduce}(n) \right] \]

where \( s \) is the actual number of performed iterations.

We can also adopt a partitioning by blocks of rows. In this case, only the “vertical” communications have to be done (one row exchanged with the northern neighbour, and one with the southern neighbour), since the values on the same row are local to the worker. In this case:

\[ T_c = s \left[ 2 T_{send}(M) + g T_F + T_{reduce}(n) \right] \]

The unidimensional space algorithm:

```
int A[M], old_A[M]; init old_A = A;
repeat
    \forall i = 0..M-1:
    \{ A[i] = F (old_A[i], old_A[i-1], old_A[i+1],
              old_A[i] = A[i] ; \}
until convergence (A)
```

is by a linear array of virtual processors, \( VP[M] \), each one encapsulating a single element of \( A \) and \( old_A \). The actual parametric version is implemented by linear blocks, thus:

\[ T_c = s \left[ 2 T_{send}(1) + g T_F + T_{reduce}(n) \right] \]

The detailed implementation of the various versions of this benchmark is left as an exercise.

### 14.4 A benchmark for static, variable stencils

Let us consider the following sequential computation:

```
int A[M, M];
\forall i = 0..M-1:
    \forall j = 0..M-1:
        \forall h = 0..M-1:
            A[i, j] = F (A[i, h], A[h, j])
```

For example, a fine grain computation with this structure may be the Floyd-Warshall shortest path algorithm for weighted graphs, where \( A \) is the weight matrix.

According to the Virtual Processors approach, we are able to recognize a matrix of \( M^2 \) virtual processors, \( VP[M][M] \), each one encapsulating a single element \( A[i, j] \), thus achieving \( O(M) \) completion time vs \( O(M^3) \). Let’s say, “parallelism is applied to \( i \) and \( j \).”

Moreover, according to the Bernstein analysis and to the owner computes rule, at the beginning of each step VPs must interact to implement a static stencil, which is variable at each sequential step according to the \( h \) value.

The following figure exemplify the stencil form for \( M = 3 \) and \( h = 0 \):
It is possible to establish a parametric rule that binds the stencil form to $h$: at each step $h = 0 .. M - 1$,

- each $VP[h, j]$ in row $h$ multicasts $A[h, j]$ to all virtual processors in the same column $VP[* , j]$;
- each $VP[i, h]$ in column $h$ multicasts $A[i, h]$ to all virtual processors in the same row $A[i, *]$.

At each step, all the $2M$ multicast communications are performed in parallel.

Mapping the virtual processors version into the actual parametric version with parallelism degree $n$ and grain size $g = M^2 / n$, the completion time is given by:

$$ T_c = T_{scatter}(n, g) + M \left( T_{multicast}(g) + g T_F \right) + T_{gather}(n, g) $$

The multicast cost model, and in particular the send cost model for the target machine, will take into account the effect of the parallel execution of $2M$ multicast operations, i.e. the conflicts at the architecture level are evaluated in the $T_{send}$ terms (see Part 2).

With a proper choice of the actual worker definition, it is possible to reduce the number of simultaneous multicast operations: for example, $M$ with a data distribution by row.

The detailed implementation of the various versions of this benchmark is left as an exercise.

### 14.5 Reduce operation in logarithmic time

The reduce operation is a second order function which, applied to a vector value $A[M]$ and to any associative operator $\otimes$, returns the following scalar value

$$ x = reduce (A, \otimes) = A[0] \otimes A[1] \otimes \ldots \otimes A[M-1] $$

This operation is very popular in the sequential, as well in the parallel, world. A simple example is the sum of all the elements of an array. More interesting examples are represented by global conditions on set of values, for example guards in while/repeat operations.

Consider the following example:

```plaintext
int A[M]; ...

while ( \exists i \in 0 .. M - 1 : G (A[i]) < 0 ) do ...
```
Let a boolean array \( B[M] \) defined in this way:

\[
\forall i = 0 .. M - 1 : B[i] = \text{if} \ G(A[i]) \geq 0
\]

The computation can be expressed as:

\[
\text{while not reduce (B, and) do ...}
\]

In many parallel programs, reduce is used as a collective operation, that is an operation performed by the cooperation of a set of modules. For example, in the convolution benchmark, the repeat guard is a condition applied to all the array elements, thus it has to be tested by all VPs (all workers): the guard is true if and only if it is verified on all array elements.

Because of its diffusion, an efficient implementation of reduce operation is important. A data parallel version exists with completion time \( O(\log M) \), instead of for the sequential \( O(N) \). The data parallel form is static, variable stencil.

Owing to the associativity of operator \( \otimes \), the Bernstein conditions are satisfied by the following tree-structured virtual processors computation, for a binary tree (figure a):

The virtual processors computation is performed in \( \lg_2 M \) steps (number of tree levels). At generic step \( h \), the number of active virtual processors and of communications is halved with respect to step \( h-1 \).

According to a general property of computational geometry, a \( M \)-leaf tree can always be mapped onto a \( \lg_2 M \) dimension binary cube. This property is exploited in order to implement the virtual processors computation with \( M \) VPs only, as shown in figure b):


This pattern can be generalized to any \( M \) by induction. Thus, at \( i \)-th step all the stencil communications are performed along the \( i \)-th dimension.

Formally, we can write (the LC version can be derived easily):

\[
\forall j = 1 \ldots \lg_2 M:
\]

\[
\forall i = 1 \ldots M \text{ in parallel}:
\]

\[
\text{if } ((i + 1) \mod 2^j = 0) \text{ then } A[i] = A[i - 2^{i-j}] \otimes A[i]
\]
At the end, $VP[M-1]$ contains the reduce results.

Mapping the virtual processors version into the actual parametric version with parallelism degree $n$ and grain size $g = M/n$, during an initial step all the workers in parallel execute the sequential reduce operation on their own partitions. After that, the logarithmic computation is performed, in which every message contains just a single element of $A$. At the end, $worker[n-1]$ contains the reduce result.

The following figure shows the cube implementation on a linear array of VPs (workers):

![Cube Implementation Diagram]

The reduce completion time is given by:

$$T_c = T_{scatter}(n, g) + T_{reduce}(n, g)$$

where:

$$T_{reduce}(n, g) = (g - 1) T_{\otimes} + \lg_2 n (T_{send}(1) + T_{\otimes})$$

If the reduce is used as a collective operation, the result must be multicast to all VPs (all workers). The same tree of the figure, if traveled in the opposite direction, supports the multicast implementation too.

**Example 1: map + reduce computations**

Let us consider the following sequential computation, where $F$ and $G$ are integer functions:

```plaintext
int A[M];

while ( \exists i \in 0..M - 1 : G(A[i]) < 0 ) do
    \forall j = 0..M - 1:
```

The data parallel solution is a map + reduce composition.

As seen, let a boolean array $B[M]$ defined in this way:

$$\forall i = 0..M - 1 : B[i] = \text{if } G(A[i]) \geq 0$$

The computation can be expressed as:

```plaintext
while not reduce (B, and) do A = map (A, F)
```

In the virtual processors version, the generic $VP[i]$ encapsulates $A[i]$. A scheme of $VP[i]$ behavior is:

```plaintext
VP[i] ::
    compute $B[i] = \text{if } G(A[i]) \geq 0$;
    take part to the reduce collective operation;
```
if \( i = M-1 \), then multicast the reduce result \( x \) to all VPs, otherwise wait the reduce result \( x \) from VP[M-1];

if \( x = \text{true} \) then terminate, otherwise compute \( A[i] = F(A[i]) \) and go back to B[i] evaluation.

Let us map the virtual processors version into the actual parametric version. Let \( m \) denote the average number of iterations, and \( n \) the parallelism degree. Assume that the initial and final value of \( A \) is contained in a separate module, that communicates with the Scatter and Gather modules. The completion time is given by:

\[
T_c = 2 T_{\text{send}}(M) + 2 T_{\text{scatter}}(n,M) + m \left[ T_{\text{map}}(n,M) + T_{\text{reduce}}(n,1) + T_{\text{multicast}}(n,1) \right] = \\
= 2 \left( T_{\text{setup}} + M T_{\text{trans}} \right) + 2 \left( n T_{\text{setup}} + M T_{\text{trans}} \right) + \frac{m (T_G + T_F)}{n} \\
+ m \log_2 e \ln n \left( T_{\text{setup}} + T_{\text{trans}} \right) + m n \left( T_{\text{setup}} + T_{\text{trans}} \right)
\]

By imposing:

\[
\frac{dT_c}{dn} = 0
\]

we obtain a second degree equation in \( n \), having one and only one real and positive solution:

\[
a n^2 + b n + c = 0
\]

where:

\[
a = (2 + m) T_{\text{setup}} + m T_{\text{trans}} \quad b = m \left( T_{\text{setup}} + T_{\text{trans}} \right) \log_2 e \quad c = -m (T_F + T_G)
\]

14.6 Parallel Prefix in logarithmic time

The reduce operation can be generalized to the prefix operation, which computes all the prefixes of a vector \( A[M] \) applying the associative operator \( \otimes \):

\[
A_0 = A_0 \\
A_1 = A_0 \otimes A_1 \\
\vdots \\
A_{M-1} = A_0 \otimes A_1 \otimes A_2 \otimes \ldots \otimes A_{M-1}
\]

In parallel, this computation requires \( \log_2 M \) steps, as shown in the following figure:
We can apply many considerations of the reduce implementation. The virtual processors tree-structured computation can be mapped onto a \( \log_2 M \) dimension cube, although not all communications occur between neighbours.

The VP abstract computation is:

\[ \forall j = 1 \ldots \log_2 M: \]

\[ \forall i = 1 \ldots M \text{ in parallel}: \]

\[ \text{if } (i \geq 2) \text{ then } A[i] = A[i - 2^{j-1}] \odot A[i] \]

The parallel prefix completion time is equal to the parallel reduce one:

\[ T_c = T_{scatter}(n, g) + T_{par-prefix}(n, g) \]

where:

\[ T_{par-prefix}(n, g) = (g - 1) T_\odot + \log_2 n (T_{send}(1) + T_\odot) \]

The parallel prefix computation has some interesting features. Unlike reduce, at each step some apparently redundant operation are executed (for example, \( A_1 \odot A_2 \) during the first step). However, they are useful for the parallelism exploitation in the next steps. This is called speculative parallelism.

Several algorithms, that at first sight appear inherently sequential, can be parallelized by the parallel prefix and speculative parallelism, provided that they use associative operations.

As an example, consider a finite state automaton, e.g. for parsing a regular language. Given the input sequence as a string of \( M \) symbols, let us calculate the generated sequence of internal states. The sequential realization has a completion time \( O(M) \). It can be expressed as a parallel prefix, with completion time \( O(\log M) \), since the state transition function is associative.

**Example**

Let us consider the following sequential computation:

```plaintext
int A[M], B[M];
B[0] = A[0];
\forall i = 0 \ldots M - 1:
```

Verify that it can be implemented as a parallel prefix. In the actual parametric version, with parallelism degree \( n \), initially each worker performs a sequential prefix step on the \( M/n \) element partition, of duration \( T_{iter}(M/n) \), where \( T_{iter} \) is the completion time of a loop iteration. After that, the logarithmic parallel prefix is executed. The completion time is given by:

\[ T_c = T_{scatter} \left( n, \frac{M}{n} \right) + T_{parallel-prefix} \left( \frac{M}{n} \right) + T_{gather} \left( n, \frac{M}{n} \right) \]

\[ T_c \sim 2(n T_{setup} + M T_{transm}) + T_{iter} \frac{M}{n} + (T_{send}(1) + T_{iter}) \log_2 n \]
The optimal parallelism degree is obtained by solving the equation:

\[
\frac{dT_c}{dn} = 2 T_{\text{setup}} - T_{\text{iter}} \frac{M}{n^2} + \frac{T_{\text{send}} (1)}{\log_e 2} \frac{1}{n} = 0
\]

14.7 Data-parallel on streams

The data parallel paradigm, in its various forms, is applicable also to stream computations. This fact significantly improves the range of parallel solutions for a given stream-based application, i.e. how to parallelize bottlenecks in graph structured applications.

Now, the three phases operate in pipeline: distribution (scatter, multicast), calculation, and collection (gather) or reduce.

If \( T_A \) is the interarrival time, the optimal parallelism degree

\[
n = \left[ \frac{T_{\text{calc}}}{T_A} \right]
\]

of the calculation phase can actually be exploited provided that the distribution (collection) phase is not a bottleneck.

If this condition is not verified, the best we can do is to find a parallelism degree value \( n_0 \), with \( n_0 < n \), such that the distribution and the calculation phases are balanced, though the module remains a bottleneck.

**Example 1**

A module operates on a stream of arrays \( A[M] \), applying a function \( F \) to each element of \( A \). Let \( M = 1K, T_F = 1.000 \tau, T_{\text{setup}} = 1.000 \tau, T_{\text{transm}} = 100 \tau, T_A = 150.000 \tau \).

Both a farm and a map parallelization are potentially feasible solutions, with optimal parallelism degree:

\[
n = \left[ \frac{M T_F}{T_A} \right] = 7
\]

In the farm solution, the emitter E is not a bottleneck, because:

\[
T_A > T_{\text{send}} (M) = T_{\text{setup}} + M T_{\text{transm}} = T_E
\]

For the map solution, we verify that the scatter functionality is not a bottleneck:

\[
T_{\text{scatter}} = n T_{\text{setup}} + M T_{\text{transm}} = 109.226 \tau < T_A
\]

just with a sequential implementation.

Therefore both the farm and the map solutions are able to achieve the ideal service time \( T_A \), with the typical known pros and cons (load balancing vs latency and memory capacity) to be evaluated according to other application/architecture requirements and/or constraints.

**Example 2**

Let us consider Example 1 with the following modifications: \( T_F = 10.000 \tau, T_A = 100.000 \tau \).
Now, the optimal parallelism degree
\[ n = 103 \]
can be exploited by the *farm* solution, while in the *map* solution the scatter functionality is a bottleneck.

The best *map* solution has a parallelism degree \( n_0 \), such that:
\[ \text{scatter} (n_0) = \frac{MT_F}{n_0} \]
That is:
\[ n_0 T_{\text{setup}} + M T_{\text{transm}} = \frac{MT_F}{n_0} \]
from which we obtain a second degree equation, whose acceptable solution is:
\[ n_0 = 63 \]

**Example 3**

Let us consider Example 1 of Sect.11.1:

A stream parallel program consists of a process \( Q \) operating on integer arrays \( A[N], B[N], C[N] \), with \( N = 10^3 \). \( A \) and \( B \) values are received from two distinct input streams, and \( C \) is sent onto the output stream. The stream length is \( m = 10^3 \).

For each couple \( (A, B) \), \( Q \) is defined as follows:
\[ \forall i = 0 \ldots N - 1 : C[i] = F(A[i], B) \]

The program is executed on a parallel architecture, having 128 processing nodes with communication processor and 3.33 GHz clock frequency. The communication cost model is: \( T_{\text{setup}} = 10^3 \tau \) and \( T_{\text{transm}} = 10^2 \tau \), with zero-copy communications.

Function \( F \) has processing time exponentially distributed with mean value \( 5 \times 10^3 \tau \).

The \( Q \) interarrival time is equal to \( 500 \times 10^3 \tau \).

The *farm* solution was able to exploit the optimal parallelism degree \( n = 10 \), with two additional modules for emitter and collector functionalities:
\[ n_\Sigma = n + 2 = 12 \]
with latency equal to 1.6 msec.

Let us now parallelize \( Q \) as the data parallel computation with the best bandwidth as possible.

With the virtual processors approach, we find a *map* computation with partitioned \( A \) and \( C \) and replicated \( B \).

A sequential *scatter* functionality has service time:
\[ T_{\text{scatter}} = n T_{\text{setup}} + N T_{\text{transm}} \sim 10^5 \tau < T_A \]
thus it is not a bottleneck, and so is the *gather* functionality.

A linear multicast has a service time of about \( 10^6 \tau \), thus it represent a bottleneck. A *tree-structured multicast* is not a bottleneck, since it has service time:
\[ T_{\text{multicast}} = 2 T_{\text{send}}(N) \sim 2 \times 10^5 \tau < T_A \]
In order to implement such tree structure \( IN \), we need 15 modules, of which the root can also execute the scatter functionality.

Therefore, the whole parallelism degree is:

\[
 n_s = n + 16 = 26
\]

which is necessary to achieve the ideal service time. However, due to the exponential distribution of calculation time, the load unbalance effect could have a significant impact. Thus the achievement of ideal service time is an optimistic evaluation.

In conclusion, the \( farm \) solution utilizes a lower number of modules (nodes) and offers guarantees about the achievable service time.

The latency and memory capacity are significantly better for the \( map \). Despite the tree-structured multicast, the latency is given by:

\[
 L_{map} = L_{scatter}(N) + L_{multicast}(N) + \frac{T_{calc}}{n} + T_{gather}(N)
\]

\[
 \sim 2 L_{scatter}(N) + 4 (T_{IN} + T_{send}(N)) + \frac{T_{calc}}{n} \sim 2 L_{scatter}(N) + 4 T_{send}(N) + \frac{T_{calc}}{n}
\]

\[
 \sim 1.1 \times 10^5 \tau = 0.33 \text{ msec}
\]

while the memory capacity per node is about 10 times lower.

### 14.8 Pipelining and other structures for data parallel implementations

In Sect. 7.3 we have seen that a loop-unfolding pipeline computation, operating on streams of (large) data structures, is an alternative implementation of the data parallel paradigm.

Let us refer to the Example of Sect. 7.3:

A module \( M \) operates on an input integer stream \( x \) and on an output integer stream \( y \). \( M \) contains an integer array \( A[100] \). For each \( x \), \( y \) is equal to the number of times a given predicate \( F(x, A[i]) \) is true. Let \( T_{calc} = 100 \, t \) and \( T_{A} = L_{com} = 10 \, t \).

A general data parallel solution is structured as \( map+reduce+multicast \), where \( multicast \) is applied to \( x \) and \( reduce \) to the partial results on the \( n \) partitions. As said, we can now verify that the pipeline data parallel solution is characterized by higher simplicity (communication channels, implementation of collective communications and operations) and by higher latency.

As an exercise, the reader is invited to formalize how the pipeline data parallel paradigm implements the \( multicast, scatter, gather, \) and \( reduce \) functionalities.

In general, we can say that the data parallel paradigm family is able to exploit many topological structures for data distribution and collection, and for VPS/workers cooperation: uni- and multi-dimensional arrays, cubes, trees, linear chains, and rings are the most usual structures. Other topologies, that will be studied in Part 2 for the interconnection structures (e.g., butterflies), represent interesting cases as well.
15. Reduction of parallelism degree

In order to achieve the best bandwidth as possible, we know how to determine the whole parallelism degree of a graph computation

\[ n_\Sigma = \sum_i n_i \]

where each addend represents the parallelism degree of a component module.

Denoting the number of processing node of the parallel architecture by \( N \), it is possible that

\[ n_\Sigma > N \]

In this case, the parallel program has to be restructured with parallelism degree \( N \). In fact, according to our methodology, a reduction of the number of processes (in order to allocate one process per node) is more effective than a multiprogrammed execution of several processes on a lower number of nodes.

In its generality, the reduction of parallelism degree is of exponential complexity, as it can be assimilated to the optimal mapping problem of graphs onto graphs. In this Section we utilize a heuristic method of low complexity, which is suitable to exploit the characteristics of the parallelization methodology and of parallel paradigms.

This method, called **uniform reduction of parallelism degree**, consists in a reduction of the parallelism degree of each module proportional to the quantity

\[ \frac{N}{n_\Sigma} \]

The main motivations of the method are:

1) parallel paradigms are parametric in the parallelism degree;

2) the bandwidth of parallel paradigm structures is proportional to the parallelism degree.

More precisely, parallel paradigms are parametric in the number of farm or map workers, or pipeline stages, or data-flow modules. In addition, some “service modules” exist whose number is constant and does not affect the true parallelism degree of the computation, notably: stream generators, emitters, collectors, scatter, centralized multicast, gather, centralized reduce, and so on.

Let \( ps \) denote the whole number of service modules. All the parametric quantities are multiplied by the following reduction factor:

\[ \alpha = \frac{N - ps}{n_\Sigma - ps} \]

so that:

\[ \sum_i n_i = N - ps \]

When present, **tree-structured** multicast or reduce computations are included in the parametric structures. In the most general case in which a parametric structure with \( n_{tree} \) intermediate nodes is present, we have:

\[ (\sum_i n_i) + n_{tree} = N - ps \]
Reflecting upon the general methodology, and in particular on the results of Sect. 5.4, this method aims to achieve a proportional de-scalability of the parallel computation.

The method is able to achieve a good approximation for relatively high values of \( N \). Of course, being a heuristic method, small anomalies can occur, especially due to the rounding up effect, which can force some adjustments of \( \pm 1 \) in the parallelism degree of some modules.

**Example**

Let a computation \( \Sigma \) be structured as a 3-stage pipeline, with the following characteristics:

- **\( S_0 \):** a 6-worker *farm*, with centralized emitter and collector, generating a stream of arrays with interdeparture time equal to \( 4 \times 10^5 \tau \) and overlapped communications;
- **\( S_1 \):** *farm* with \( T_{calc} = 4 \times 10^6 \tau \), thus with 10 workers, plus centralized emitter and collector. Consequently, its interdeparture time is equal to \( 4 \times 10^5 \tau \);
- **\( S_2 \):** *map* with \( T_{calc} = 6 \times 10^7 \tau \), thus with 150 workers. Assume that scatter and gather functionalities are not bottlenecks.

The whole service time is equal to \( 4 \times 10^5 \tau \), and the relative efficiency is equal to one. The whole parallelism degree is:

\[
n = 172
\]

with

\[
sp = 6
\]

Assume that the parallel architecture has \( N = 64 \) nodes. The reduction factor is equal to:

\[
\alpha = \frac{N - ps}{n - ps} = 0.35
\]

- The number of workers of \( S_0 \) is decreased from 6 to \( 6 \alpha \), i.e. about 2 workers; its interdeparture time is increased by \( 1/\alpha \), thus becomes about \( 1.1 \times 10^6 \tau \).
- The number of workers of \( S_1 \) is decreased from 10 to 4; its interdeparture time is increased by \( 1/\alpha \), thus becomes about \( 1.1 \times 10^6 \tau \).
- The number of workers of \( S_2 \) is decreased from 150 to 52; its interdeparture time is increased by \( 1/\alpha \), thus becomes about \( 1.1 \times 10^6 \tau \).

In conclusion, the \( \Sigma \) service time decreases from \( 4 \times 10^5 \tau \) to \( 1.1 \times 10^6 \tau \) and the relative efficiency is still \( \sim 1 \), because also the ideal bandwidth has been reduced by \( \alpha \).
16. Queueing systems and client-server computations with request-reply behavior

16.1 Analytical treatment of Queueing Systems

In Sect. 5.1 we have introduced some basic characteristics of Queueing Systems. We saw that for acyclic graphs *Queueing Networks* are a sufficiently powerful methodology. They do not utilize an explicit analytical treatment in terms of probability distributions, instead they are based on some basic results about the *information flows* in a network of queues.

The analytical treatment of Queueing Systems, in terms of probability distributions of interarrival and service times, is necessary mainly for cyclic graph computations with a request-reply computations:

This Section is dedicated to this issue. Only the minimal basic results of Queueing Theory will be studied and applied for our purposes, while, for a complete treatment of this very interesting theory, the reader can refer to the fundamental book by Kleinrock.

The parameters of interest in evaluating a queueing system are:

- **average queue length**, $L_Q$: average number of client requests in the waiting queue;
- **average request number in the system**, $N_Q$: with respect to $L_Q$, it includes the currently served request(s);
- **average waiting time in queue**, $W_Q$: average time spent by a request in the waiting queue;
- **average waiting time in queue**, $R_Q$: with respect to $W_Q$, it includes the time spent on the currently served request(s). It is also called **response time**, as it is the average time needed from a client to receive the result for the requested service.

These parameters are related each other in several ways. First of all, under very general conditions about the service discipline, the **Little Law** holds:

$$L_Q = \lambda \cdot W_Q$$

$$N_Q = \lambda \cdot R_Q$$

where $\lambda = 1/T_A$ is interarrival frequency.
Moreover, the following relations holds for queueing systems with a sequential server with $\rho < 1$:

$$N_Q = L_Q + \rho$$

$$R_Q = W_Q + T_s$$

The former holds since the average number of requests currently in the service phase is equal to the utilization factor $\rho$, if $\rho < 1$.

The latter means that we add, to the average time spent in the waiting queue, the average latency to process the request currently in the service phase. This relation holds for sequential servers only, for which the service time $T_s$ and the latency coincide. For parallel servers, it is corrected as follows:

$$R_Q = W_Q + L_s$$

where $L_s$ is the average server latency. This formula is more general and includes the sequential case too. It is significant that impact of the server service time is on the first added (through $\rho$), while the impact of server latency is on the second one.

A fundamental concept, to understand queueing systems behavior qualitatively, is the following: all parameters are monotonically increasing with $\rho$, and, for $\rho < 1$ and $\rho \to 1$, they tend asymptotically to infinity, for example:

$$\lim_{\rho \to 0} L_Q = 0$$

$$\lim_{\rho \to 1} L_Q = \infty$$

$$\lim_{\rho \to 0} R_Q = L_s$$

$$\lim_{\rho \to 1} R_Q = \infty$$

That is, in order to have prompt reply from a server, this has to be underutilized. Trying to increasing the utilization degree of a server has a negative effect on the clients.

Some notable cases of queueing systems, for which the parameters can be expressed analytically in closed form, are described in the following. For proofs, and for further cases, the reader is referred to the mentioned bibliography on Queueing Theory.

### 16.1.1 M/M/1 queue

The basic case is an infinite FIFO queue with negative exponential distributions of interarrival and services:

$$F_{t_a}(t) = \Pr(t_a \leq t) = 1 - e^{-\lambda t}$$

$$F_{t_s}(t) = \Pr(t_s \leq t) = 1 - e^{-\mu t}$$
As known, this probability distribution model a memory-less behavior, that is the random variable distribution does not depend on the observation time instant. This property is simple enough for solving the analytical model in closed form. Moreover, it is a reasonable approximation of many real cases.

It can be shown that:

\[ N_Q = \frac{\rho}{1 - \rho} \]

Thus:

\[ L_Q = N_Q - \rho = \frac{\rho^2}{1 - \rho} \]

By applying Little Law:

\[ W_Q = \frac{L_Q}{\lambda} = \frac{\rho}{\mu(1 - \rho)} \]

Moreover, as usually:

\[ R_Q = W_Q + L_S \]

These formulas are valid also for finite FIFO queues with good approximation, especially for relatively large values of the physical queue positions. When the approximation is not sufficient, exact formulas exist for the finite M/M/1 queue.

16.1.2 M/G/1 queue

Though frequent, the assumption on exponential service time is not always applicable. Much more frequent is the situation of exponential interarrival time, for example in queues with multiple clients. An important result, known as Pollaczek-Khinchine formula, is valid for exponential interarrival time distribution and general (i.e., any) service time distribution: M/G/I:

\[ N_Q = \left( \frac{\rho}{1 - \rho} \right) \left[ 1 - \frac{\rho}{2} \left( 1 - \mu^2 \sigma_s^2 \right) \right] \]

Formulas for all the other parameters are derived as seen in the previous Section:

Notice that the \( N_Q \) expression \( \rho/(1 - \rho) \) for the M/M/1 queue is multiplied by a corrective factor taking into account the effective service distribution through its mean \( (T_S = 1/\mu) \) and variance \( \sigma_s \).

A particular case, of special significance for our purposes, is the M/D/I queue, with constant service time distribution, for which \( \sigma_s = 0 \):

\[ N_Q = \left( \frac{\rho}{1 - \rho} \right) \left( 1 - \frac{\rho}{2} \right) \]
16.2 Client-server computations with request-reply behavior

In this Section we derive a general method to solve, analytically or numerically, queueing systems that model client-server computation with request-reply behavior.

The client and server computational scheme is:

\[
\begin{align*}
C_i &::= \text{initialize } x \ldots \\
\text{while (true) do } &
  \begin{cases}
    < \text{send } x \text{ to } S>; \\
    < \text{receive } y \text{ from } S>; \\
    x = G(y, \ldots)
  \end{cases} \\
S &::= /\text{possible initialization of state}/;
\end{align*}
\]

Assume that all clients are identical. Let \( T_{cl} \) be the client effective service time, \( T_G \) the client ideal service time, and \( T_S \) the server service time, possibly including communication times.

The computation cost model is described by the following system of equations:

\[
\begin{align*}
T_{cl} &= T_G + R_Q \\
R_Q &= W_Q(\rho, T_S, \sigma_S) + L_S \\
\rho &= \frac{T_S}{T_A} \\
T_A &= \frac{T_{cl}}{n}
\end{align*}
\]

whose solution is subject to constraint \( \rho < 1 \), as discussed in Section 5.1.

For the proper \( W_Q \) expression, a second, or higher, order equation in \( \rho \) is derived by substitution, which admits one and only one real, positive solution satisfying \( \rho < 1 \).

In the following, we report the function of some parameters in graphic form, for the \( M/M/1 \) queue with \( T_S = L_S \). The measures are expressed in relative time units \( t \). Where constant, \( T_G \) and \( T_S \) are assumed equal to 10t.
Tempo di servizio del cliente in funzione del tempo di calcolo e del numero dei clienti, con $T_S = 10$

Fattore di utilizzazione della coda di richieste in funzione del tempo di calcolo e del numero dei clienti, con $T_S = 10$
The qualitative shape of the various functions can be evaluated by reasoning about queueing systems behavior. For example, if $T_G$ increases then $T_A$ increases, thus $\rho$ decreases, and $R_Q$ decreases. If $T_S$ increases, $\rho$ increases, thus $R_Q$ increases. If $n$ increases, then $T_A$ decreases, $\rho$ increases, thus $R_Q$ increases.

Let us study the relative efficiency of generic client:

$$\varepsilon = \frac{B_{cl}}{B_{cl-id}} = \frac{T_G}{T_G + R_Q} = \frac{1}{1 + \frac{R_Q}{T_G}}$$

The following Figure D shows the typical dichotomy bandwidth vs efficiency of a client module: if the client ideal service time $T_G$ decreases, $\rho$ increases, $R_Q$ increases, thus $R_Q/T_G \to \infty$, so $\varepsilon \to 0$; if the client ideal service time $T_G$ increases, $\rho$ decreases, $R_Q$ decreases, thus $R_Q/T_G \to 0$, so $\varepsilon \to 1$. However, for $T_S \to 0$, $\varepsilon \to 1$ and for $T_S \to \infty$, $\varepsilon \to 0$ (Figure E)
Example

Let us consider a client-server computation with request-reply behavior, \( n \) identical clients, and a functional server. Let the client calculation time and the server service time have known exponential distributions.

The server is parallelized as a \textit{farm} with parallelism degree \( m \). Let us study the client service time and client efficiency as functions of \( m \), and let us evaluate the server latency impact.
Qualitative study

First of all, the following relation holds:

\[ 1 \leq m \leq n \]

since more than \( n \) request cannot be executed by a farm server.

Il valore ottimale di \( m \) sarà, in generale, minore di \( n \). La determinazione di questo valore non è effettuabile in maniera semplice risolvendo in forma chiusa il sistema di equazioni che caratterizza il caso cliente-servente, bensì andrà stimato numericamente una volta risolto il sistema.

If \( m \) increases, then \( T_G \) decreases, thus \( \rho \) decreases, so \( W_Q \) decreases: \( T_{cl} \) decreases, and \( \varepsilon_{cl} \) increases. However, the server latency (\( L_s \)) effect could modify this conclusion, at least partially. \( L_s \) includes the communication latency from clients to server and from server to clients. At the process level, the farm latency is constant with \( m \). However, it could be linear or logarithmic in \( m \) in a farm with state implementation, due to the multicast latency.

In conclusion, because of the latency effect, the function \( T_{cl} = T_{cl}(m) \) might have a minimum, or to be monotonically decreasing. If the minimum occurs for \( m > n \), the former situation reduce to the latter. The same considerations apply to the relative efficiency.

Quantitative study

Let us solve the system of equations:

\[
\begin{align*}
T_{cl} &= T_G + R_Q \\
R_Q &= W_Q(\rho, T_z, T_A) + L_s \\
\rho &= \frac{T_z}{T_A} \\
T_A &= \frac{T_G}{n} \\
\rho < 1
\end{align*}
\]

For a M/M/1 queue:

\[
W_Q = \frac{L_Q}{\lambda} = \frac{T_A \rho^2}{1 - \rho} = \frac{T_s^2}{T_A - T_s}
\]

We achieve:

\[
R_Q^2 + (T_G - nT_s - L_s)R_Q - nT_s^2 - L_s(T_G - nT_s) = 0
\]

having always two real solutions, of which one and only one satisfies \( \rho < 1 \).

For example, the system has been solved for

\[ n = 16 \quad 1 \leq m \leq 16 \quad T_G = 1.000 \ t \quad T_{calc} = 1.000 \ t \]

The latency has been evaluated, respectively in the constant and logarithmic case, as:

\[
L_s = T_{calc} + L_0 \quad L_s = T_{calc} + L_0 (1 + \log_2 (m))
\]

with \( L_0 = 500 \ t \).
The results are reported in the following curves:

**Constant latency**

- $T_{cl}$

**Logarithmic latency**

- $T_{cl}$
16.3 Client-server implementation issues

16.3.1 Critical value of utilization factor
In the typical shape of $R_Q$ as a function of $\rho$ (as reported in the following figure), we can recognize an interval of $\rho$ values with a relatively slow slope of $R_Q$, until a critical value $\rho_c$, after which the slope is very fast:

![Graph showing critical value of utilization factor]

The further constraint

$$\rho \leq \rho_c$$

leads to an implementation for which the response time has almost the minimum value (i.e., the base latency).

It can be applied to the solution of client-server systems analytically or numerically.

16.3.2 Farm and data parallel structures for server implementation
The parallelization of a server, in a client-server computation, can be done by farm or data parallel paradigms, with interesting implications.

In a farm realization, the server parallelism degree is limited by the number of clients. Moreover the latency is greater than the sequential computation, and this has meaningful impact on the response time.

In a data parallel realization, when applicable, the server parallelism degree is not constrained by the number of clients: in general, this can lead to a greater parallelism degree compared to the farm solution.

16.3.3 Client-server transactions
In computations organized as transactions, the interaction between client and server consists in a sequence, in general of variable length, of requests from the same client to the server, and corresponding replies. Often, a transaction has an internal state, which is initialized with the first request, and is updated at each step.

For example, a un web server operates on transactions.
The *open_transaction* (initial state, ...) request is served according to an *on-demand* emitter strategy, in order to balance the load of workers:

![Diagram of the transaction process](image)

Once a worker is scheduled, and the internal state is initialized, *this worker* interact with the client through a dynamic direct connection (a collector could not exist). It is a sort of *dynamic functional partitioning* scheme with independent workers (Sect. 12).

The *close_transaction* request is communicated to the emitter.

Often, the transaction operates on large data structures, thus a *data parallel* structure for each worker is a suitable solution, in order to optimize both bandwidth and latency. The workers can be dynamically allocated from a common pool, according to the transaction requirements.
17. Exercises

The following exercises contain the definition of computations to be parallelized. In general, try possible “reasonable” solutions, give a specification of their structure and how they have been derived, evaluate and compare them. For the sake of simplicity and generality, numerical values of data structure sizes, calculation times, parameters of communication latency, number of nodes, and so on, are not specified: the student is invited to fix some suitable values.

1) \texttt{int A[M]; //G is an associative function//}
   \{ \forall i = 0 .. M - 1 
   \begin{align*}
   & \text{if ( reduce (A, G) > 0 ) then} \\
   & \{ x = A[i] ; \\
   & \forall j = 0 .. M - 1 \\
   & \} ; \\
   \}
   \}

2) Module operating on input stream \texttt{R; S is local.}
   \texttt{int R[M], S[M]; int result;}
   \forall i = 0 .. M-1 
   \forall j = 0 .. M-1 
   S[i] = F (R[j], S[i]);
   result = reduce (S, +)

3) \texttt{int A[M][M], B[M][M], C[M][M];}
   \forall i, j = 0 .. M - 1; C_{ij} = \sum_{k=0}^{M-1} F(A_{ik}, B_{kj})

4) A client-server, request-reply computation, with \( n \) identical clients and one server. The external result is the stream of values produced by the set of clients.
   
   a) Explain how the ideal bandwidth, effective bandwidth, and relative efficiency are evaluated: 1) for the server, 2) for the generic client, 3) for the whole computation.
   
   b) Show graphically the qualitative functions \( B(n), \varepsilon(n) \) for the server and for the whole computation.
   
   c) The same of point b), in function of parallelism degree \( m \) of server.
5) A client-server computation, with 8 identical clients operating on transactions. A transaction consists in \( K \) matrix-vector products: initially the connection is requested by sending the \( A \) matrix, that will be used for all the transaction steps; then, for each step, a vector \( B \) is sent and the corresponding result \( C \) is returned to the client. Evaluate the completion time.
18. Parallel systems at the firmware level

In this Section we apply the parallelization methodology to systems at the firmware level. As an alternative to the process level, any parallel computation can be implemented as a graph of processing units, where some units are parallelized according to the studied parallel paradigms. That is, the methodology can be used to design architectures that are specialized to the execution of specific parallel computations, with the typical advantages of the firmware-level performances: parallelism in microinstructions, reduced communication latencies and communication overlapping.

Initially the firmware communication cost model, that will be reviewed.

The Section contains a summary of the methodology application to some typical parallel general-purpose uniprocessor architectures, i.e. architectures able to exploit Instruction Level Parallelism (IPL), like pipeline and superscalar CPUs. These topics belong to a basic undergraduate course in Computer Architecture (Architettura degli Elaboratori, University of Pisa).

18.1 Cost model of firmware-level communications

Let us consider a symmetric communication between a sender and a receiver processing unit on the dedicated link with Ready-Acknowledgement protocol. Assuming level-transition interfaces, the communication with asynchrony degree equal to one is primitive. Moreover, it does not introduce any overhead on the calculation time, because the protocol actions are executed in parallel in the same clock cycles of microinstructions doing calculation (i.e., null overhead is “free” at the firmware level, while at the process level it requires additional resources, as communication processors). In particular, typically the sender executes the send protocol in the last microinstruction of the calculation phase that produces the message value, while the receiver executes the receive protocol in the first microinstruction of the calculation phase on the target variable.

The next figure shows the execution and evaluation of a communication protocol and the possible overlapping with calculation phases.

The communication cost model has the general characteristics studied in Section 4.3:

The service time of the communicating module is increased by the communication time not overlapped to calculation, $T_{\text{com}}$:

$$T = T_{\text{calc}} + T_{\text{com}}$$

$$T_{\text{com}} = \begin{cases} L_{\text{com}} - T_{\text{calc}} & \text{if } L_{\text{com}} > T_{\text{calc}} \\ 0 & \text{otherwise} \end{cases}$$

Equivalently:

$$T = T_{\text{calc}} + T_{\text{com}} = \max (T_{\text{calc}}, L_{\text{com}})$$

The specificity at the firmware level is given by the communication latency formula:

$$L_{\text{com}} = 2 (\tau + T_p)$$
For example, if $T_{tr} = 10\tau$ (typical value for medium-length *inter-chip* links), a calculation with $T_{calc} \geq 22\tau$ (relatively fine grain) is fully overlapped to communication, i.e., the processing bandwidth cannot be greater than $1/22\tau$.

For many *intra-chip* links, we can assume $T_{tr} = 0$. This is acceptable for all links except for links belonging to some complex intra-chip interconnection networks (e.g., crossbar, butterflies, trees). The result is that calculations with $T_{calc} \geq 2\tau$ (very fine grain) are fully overlapped to communication: *for an on-chip parallel system the ideal bandwidth is given by 1/2\tau, i.e., the numerical value of half the clock frequency.* For example, in a pipeline CPU the ideal performance can be estimated as $1/2\tau$ instructions per second, for a *n*-way superscalar CPU as $n/2\tau$ instructions per second.

### 18.2 An example of firmware-level parallel computation

Let us consider again the Example of Sect. 7.3:

A module $M$ operates on an input integer stream $x$ and on an output integer stream $y$. $M$ contains an integer array $A[100]$. For each $x$, $y$ is equal to the number of times a given predicate $F(x, A[i])$ is true.

Let the predicate be: $x > A[i]$.

Assume that $x$ values are produced by a processing unit $U_1$ with calculation time equal to $1\tau$, and $y$ values are consumed by unit $U_2$ with calculation time $1\tau$.

Let the transmission latency of any link be equal to $4\tau$, and the stream length equal to 1000.

$M$ is realized as a processing unit $U$. Let us study the $U$ parallelization according to the parallel paradigms: *a) pipeline, b) farm, c) data-parallel.*
The communication latency is given by:

\[ L_{\text{com}} = 2 (\tau + T_{tr}) = 10 \tau \]

Therefore, the effective service time of U\(_1\) and U\(_2\) is equal to \( L_{\text{com}} \) (\( \varepsilon_1 = \varepsilon_2 = 1/10 \)).

The interarrival time to U, thus the system ideal service time, is equal to \( L_{\text{com}} \):

\[ T_{\Sigma-id} = T_A = 10 \tau \]

The microprogram of U consists in a loop repeated 100 times. At each iteration \( x \) is compared with \( A[i] \) and possibly the partial sum register is incremented: this requires just one clock cycle, thus:

\[ T_{\text{calc}} = 100 \tau \]

The optimal parallelism degree is:

\[ \bar{n} = \frac{T_{\text{calc}}}{T_A} = 10 \]

For \( n = \bar{n} \) we obtain:

\[ T_{\Sigma}^{(10)} = T_{\Sigma-id}^{(10)} = 10 \tau \quad \varepsilon_{\Sigma}^{(10)} = 1 \quad T_c^{(10)} \sim m T_{\Sigma}^{(10)} = 10,000 \tau \]

provided that the adopted parallel paradigm is able to exploit the optimal parallelism degree.

**a) 10-loop-unfolding pipeline with statically partitioned A (as in Sect. 7.3)**

The result is:

- \( n_{\Sigma} = \bar{n} + 2 = 12 \), able to support the ideal service time,
- perfectly balance stages,
- whole memory capacity equal to the sequential realization,
- the latency, which in general is linear in \( n \), for \( n = \bar{n} \) is equal to:

\[ L_{\Sigma}^{(10)} = (\tau + T_{tr}) + 10 (10 \tau + T_{tr}) + \tau = 102 \tau + 11 T_{tr} = 146 \tau \]

**b) Farm with statically replicated A**
Emitter unit is not a bottleneck, since its service time is equal to $L_{com}$ ($e_E = 1$). The on-demand strategy is easily implemented at the firmware level: the availability of workers is merely signaled by the “ack” value.

The service time is equal to the ideal one, with:

$$n_z = \bar{n} + 4 = 14$$

The memory capacity is 10 times larger than the sequential realization.

Emitter and collector can be realized as single units, since their pin-count is relatively low. In general, tree-structured realizations are able to solve the pin-count problem, without increasing the service time, at the expense of a logarithmic latency.

In our case:

$$L_z^{(10)} = (\tau + T_{cr}) + (\tau + T_{tr}) + (100 \tau + T_{tr}) + (\tau + T_{tr}) + \tau = 104 \tau + 4 T_{tr} = 120 \tau$$

c) Data-parallel: map + reduce with statically partitioned A and multicasted x

IN (service time $L_{com}$) and OUT (service time: $10\tau$ for second phase of reduce) are not bottlenecks. Thus, the service time is equal to the ideal one, with:

$$n_z = \bar{n} + 4 = 14$$

In this computation load balancing is achieved, due to the zero-variance computation.

The whole memory capacity is equal to the sequential realization.

The latency is the minimum for all possible realizations:

$$L_z^{(10)} = (\tau + T_{cr}) + (\tau + T_{tr}) + (10 \tau + T_{tr}) + (10 \tau + T_{tr}) + \tau = 23 \tau + 4 T_{tr} = 39 \tau$$

18.3 Interconnection structures for farm e data-parallel systems

In the firmware realization of farm and data parallel systems, some functionalities are critical from the pin-count problem viewpoint, if implemented by single units: emitter, collector, scatter, gather, multicast.
On the other hand, interconnection structures for highly parallel systems must be based on dedicated links. Traditional bus solutions are not suitable, due to their serial nature, linear latency, synchronous communication protocol, and arbitration overhead.

In Part 2 we will study limited degree interconnection structures for highly parallel multiprocessor and multicomputer architectures. They are based on dedicated links, and each switching node is interconnected to the others by means of a limited number of links. Interesting structures (butterflies, cubes, fat tree) will be defined, many of them with logarithmic communication latency.

A simple example is a tree-structured realization of emitter and collector, as well as of scatter, gather and multicast:

Communication latency is $O(\log n)$, while the service time is just $L_{com} = 2(\tau + T_{tr})$ on a dedicated link, owing to the pipeline effect and to the parallelism at microinstruction level (all the interfaces can be read/written simultaneously).

Another emitter structure is a ring:

If $W_i$ is able to accept a request, $E_i$ routes the incoming request to it, otherwise $E_i$ routes the request to $E_{i+1}$ on the ring. A task that is not temporarily accepted by any worker continues to circulate on the ring until a worker is available “to capture” it. $E_0$ can implement well-known strategies for deadlock avoidance.
18.4 Instruction Level Parallelism architectures

A general-purpose computer with a sequential organization, i.e. able to execute only one instruction at a time, is characterized by low performance and low efficiency. Cache memory on chip (primary and possibly secondary cache) is a very important technique to reduce instruction latency, however a great increase of instruction bandwidth is still necessary. The goal is to achieve an instruction service time of the same order of magnitude of the clock cycle (performance of the same order of magnitude of the clock frequency) or fractions of the clock cycle.

This goal is achieved by Instruction Level Parallelism (IPL) architectures, in which several instructions of the same sequential program are executed in parallel. This parallelism is completely hidden to the programmer, instead it is fully exploited at the firmware level. In general, assembly language annotations and intensive compiler optimizations are needed to achieve a good utilization of the parallel firmware features.

All the main parallel paradigms can be exploited in IPL architectures in various combinations:

- **Pipeline**: for the basic parallelization of the instruction interpreter at the firmware level,
- **Farm**: for the replication of parts of the firmware interpreter,
- **Functional partitioning with independent workers**: for the parallel execution of fixed and floating point arithmetic operations,
- **Data parallel (map and possibly stencil)**: for the parallel execution of vectorized instructions (i.e. instructions operating on arrays or matrixes, instead of on scalars),
- **Data-flow**: for an effective ordering of instructions at compile-time and/or at run-time.

Due to the nature of this parallelization problem, computations are stream-based, i.e. one or more instruction streams is generated and executed.

18.4.1 Pipeline CPU

This is the basic starting point for IPL architectures: each element of the instruction stream is a single assembler instruction.

Figure 1 gives a very elementary idea of the pipeline organization of a CPU. A more detailed pipeline CPU organization is shown in Figure 2.

Because $L_{com} = 2\tau$ on chip (Sect. 17.1), the ideal service time is

$$T_{id} = 2\tau$$

thus the ideal Performance (instruction bandwidth) is:

$$\mathcal{P}_{id} = \frac{1}{2\tau} \cdot \frac{clock \ frequency}{2}$$

The Instruction Memory (IM), which is basically an instruction cache, generates the instruction stream, with an interdeparture time $2\tau$, at consecutive addresses. The Instruction Unit (IU) is able to decode and to prepare any instruction with a service time of $2\tau$. The Data Memory (DM), which is basically a data cache, is able to read/write a word with a service
time $2\tau$. The Execution Unit (EU) is able to execute any arithmetic logic instruction with a service time equal to $2\tau$, though its latency may be greater according to the kind of operation.

In principle, a continuous stream of instructions feeds the pipelined execution of the firmware interpreter; each pipeline stage corresponds to an interpreter phase.

**Figure 1**

**Figure 2**

- IM: Instruction Memory
- MMU: Instruction Memory MMU
- CI: Instruction Cache
- TAB-CI: Instruction Cache Relocation
- DM: Data Memory
- MMU$_d$: Data Memory MMU
- CD: Data Cache
- TAB-CD: Data Cache Relocation
- IU: Instruction preparation Unit; possibly parallel
- EU: Instruction Execution Unit; possibly parallel/pipelined
- RG: General Registers, primary copy
- RG$_1$: General Registers, secondary copy
- IC: program counter, primary copy
- IC$_1$: program counter, secondary copy

(Primary cache only is shown for simplicity)
An abstract architecture is defined in Figure 3 for a Risc machine:

![Diagram of CPU architecture](image)

The main subsystems (IM, IU, DM, EU) are shown. It is assumed that each subsystem, including EU, has a service time and a latency equal to \( t = 2\tau \). Basically, these subsystems cooperate in pipeline. However, this is just the ideal situation, because other interactions are needed: a feedback from IU to IM to communicate target addresses of branch/jump instructions, and a feedback from EU to IU to communicate updated values of general registers. A copy of the program counter register (IC) is present in IM (IC1), and a copy of the General Registers array (RG) is present in IU (RG1). Proper synchronization mechanisms are provided to maintain IC-IC1 and RG-RG1 consistent (these mechanisms imply zero-overhead at the firmware level, as they are executed in parallel in the interpreter microinstructions).

Thus, the so-called Pipeline CPU is not merely a pipeline structure, but a stream-based computation graph containing client-server cycles and operations with internal state too. According to our methodology, we are able to study it and to derive a cost model.

Figure 4 shows a simple execution sequence on general registers, which is actually able to achieve the ideal service time, without degradations.

However, the client-server feedbacks introduce waiting time intervals (“bubbles”) in the clients behavior, thus an instruction service time increase. The two main performance degradations correspond to the existence of the two feedbacks in the graph:

1. **branch degradation**: a bubble is introduced in IU, because IU receives an instruction which is not the branch target one, thus it is not significant and must be discarded. An example is shown in Figure 5;
b) **logical dependency degradation**: a bubble is introduced in IU, because IU needs to read general register contents (e.g. to compute a data address, or to evaluate a predicate) which are not yet been updated by EU. An example is shown in Figure 6.

![Figure 4](image1)

Execution simulation: to determine service time $T$ and efficiency $\varepsilon$

Instruction sequences of this kind are able to fully exploit the pipelined CPU.

In fact, in this case CPU behaves as a “pure” pipeline.

No performance degradation.

“Skipping” DM stage has no impact on service time, in this example.

![Figure 5](image2)

“bubble”:

IM performs a *wasted* fetch (instruction 4), instruction 4 must be **discarded** by IU, *about* one time interval $t$ is wasted and it propagates through the pipeline.

Abstract architecture: *exactly* one time interval $t$ is wasted (*approximate* evaluation of the concrete architecture)
A simple and effective cost model can be derived for Risc machines (undergraduate Computer Architecture course, University of Pisa).

The general problem of performance degradations can be studied according to the general cost model for client-server computations with request-reply behavior (Sect. 16). Let us consider the logical dependency problem: it can be modeled by a queueing system in which (a) the server is the subsystem (DM, EU), and (b) the clients are instances of instructions that request a service and, with a certain probability, are waiting for a reply. Thus, the mean waiting time introduced in IU can be estimated as

\[ d R_Q \]

where \( d \) is the occurrence probability of a logical dependency, and, as usually, \( R_Q \) is the server response time.

**Compile-time optimizations** (code motion, delayed branch, branch prediction and out-of-order behavior: all based on the proper application of Bernstein conditions) are very important in order to exploit the potentials of such IPL machines (undergraduate Computer Architecture course, University of Pisa). In this area, a fundamental contribution has been given by the data-flow computational model, which is able to formalize the most relevant issues in parallelism exploitation at the instruction level.

### 18.4.2 Parallel Execution Unit

The queueing model for evaluating Pipeline CPU logical dependencies introduces an interesting issue: the **Execution Unit realization of arithmetic operations with long latency** (fixed point multiplication/division, any floating point operation). If the CPU system were a pure pipeline, the EU latency would have been no impact on performance (just a marginal
impact on completion time). However, because of the logical dependencies, both the EU service time and the EU latency have a significant impact on $R_Q$ and have to be minimized:

$$R_Q = W_Q(\rho) + L_S$$

The service time can be rendered equal to the ideal one ($2\tau$) through parallelization techniques, while the latency can be reduced by optimized hardware technologies and proper parallelizations. Some solutions to this problem are described in the following.

**Loop-unfolding pipeline**

Let us consider a simple example: integer multiplication. As known, sequential iterative multiplication algorithms are executed in time proportional to the word length, e.g. a multiplication microprogram has a latency of about 50-64 clock cycles for a 32-bit machine. Hardware multipliers (combinatorial circuits) exist with 1-2 clock cycles latency for full word operations; however, they are expensive in terms of chip area. A trade-off solution can be found utilizing the parallelization methodology:

- use a hardware multiplier for few bits operands (8-bit) as building blocks; it has a much lower complexity of a full word (32-bit) multiplier, thus a much smaller chip area;
- implement a loop-unfolding pipeline structure of small multipliers, e.g. four 8-bit multipliers, whose complexity (chip area) is still much lower than the 32-bit multiplier:

Multiply $a$, $b$, result $c$

Let byte $(v,j)$ denote the $j$-th byte of integer word $v$

**Sequential** multiplication algorithm (32 bit) exploiting the hardware implementation applied to bytes:

```
init c;
for (i = 0; i < 4; i++)
    c = combine ( c, hardware_multiplication ( byte (a, i), byte (b, i) ) )
```

**Loop-unfolding pipeline (“systolic”)** implementation:

```
byte (a, 0) byte (b, 0) byte (a, 1) byte (b, 1) byte (a, 2) byte (b, 2) byte (a, 3) byte (b, 3)
```

- $c \rightarrow \text{hw\_mul \& combine}
- \text{hw\_mul \& combine}
- \text{hw\_mul \& combine}
- \text{hw\_mul \& combine}

service time = $2\tau$, latency = $8\tau$

Thus, the service time has been minimized ($2\tau$), while the latency has been only increased logarithmically ($8\tau$ in the example) compared to the monolithic hardware implementation.
**Pipelined floating point operations**

Many other pipeline implementations exist for arithmetic operations, notably for floating point operations, as shown for example in Figure 8.

Now, our parallelization problem is the following: given some pipelined units specialized for classes of arithmetic operations (fixed point multiply/divide, floating point addition, floating point multiply/divide, ..., floating point square root, and so on), realize an Execution Unit with minimal service time ($2\tau$). A cost-effective solution is the *functional partitioning with independent workers*, as shown in Figure 9.

In Sect. 12 we saw that this parallel paradigm is characterized by potential load unbalance, however load unbalance is not a source of degradation if no worker is a bottleneck: this is
exactly our situation, because each pipelined worker has service time equal to $2\tau$. A *farm* realization (i.e. general-purpose workers) could be feasible as well, however in this application it has not a better bandwidth, nor a better latency, while the chip area cost is much greater.

The EU parallel implementation is paid with an increase in logical dependencies, which now exist also “inside the EU” (besides “between IU and EU” as discussed till now).

### 18.4.3 Data parallelism: instruction vectorization and SIMD machines

Another important firmware application of the parallelization methodology is the data parallel (DP) realization of *vectorized instructions*. Let us assume that the assembler machine includes instructions to operate on arrays or matrixes, for example

- `vector_addition  base_address 1, base_address 2, base_address 3, vector_size`
- `reduce  base_address, size, operator_code`
- `matrix_vector_product  base_address 1, base_address 2, base_address 3, size`
- `convolution  base_address, size`

These data parallel computations can be implemented in EU, according to what has been studied in general and, as far as communications are concerned, according to Sect. 17.3.

These concepts is extended to *Single Instruction Steam Single Data Stream* (*SIMD*) architectures, that is data parallel specialized architectures used as input-output *coprocessors*, as shown in Figure 10 and 11. In Figure 10 Instruction Unit implements scatter/gather operations and instruction multicast to all the pipelined Execution Units.

![Figure 10](image)

Although a lot of confusion is created in the commercial world about this subject, SIMD machines cannot be general-purpose computers. This does not means that they are not useful: general-purpose machine (possibly, multiprocessors) equipped with SIMD coprocessors are very powerful solutions in principle.

Currently, this approach has been adopted in *GPUs* (Graphic Processing Units): SIMD machines on single chip, with some programming libraries support.
The main problems of the SIMD approach, including GPUs, are:

- I/O bandwidth and latency for data transfer between Host and SIMD coprocessor might be very critical parameters,
- high-level programmability of composite architectures (Host + coprocessor) is yet an open issue.

18.4.4 Superscalar and VLIW architectures

The concept of ILP in Pipeline CPU is extended to superscalar architecture: *each element of the instruction stream contains more than one instruction.*

For example, it is conceptually easy to pass to a 2-way superscalar CPU. Two consecutive instructions are read simultaneously from the Instruction Memory during a service time $2\tau$, e.g. from a 64-bit or a 2-way interleaved Instruction Cache. IU, DM and EU are realizable with double bandwidth (2 instructions in 2 clock cycles), so

$$\varphi_{id} = \frac{2}{2\tau} = \text{clock frequency value instructions per second}$$

However, performance degradations are greater than in the basic Pipeline CPU, due to increased utilization factors of servers and to the conflicting instructions belonging to the same stream element.

In general, we can think about a *n-way superscalar* CPU, as illustrated in Figure 12. Currently, the high-end ILP CPUs have $n = 4 – 8 – 16$. 

![DIagram](image-url)
In the VLIW (Very Long Instruction Word) variant, the compiler ensures that all the instructions in the same word are independent, and static branch predictions technique are applied.

Though some notable attempts, like VLIW, exist to simplify the hardware-firmware design, the complexity of superscalar CPUs remains very high (too high) with respect to the actually achievable performances (low performance/cost ratio). Notably, some techniques for controlling out-of-order instructions (caused by the intensive applications of code motion and branch prediction optimizations) are very expensive and area-consuming:

- **reordering buffer**: since some inconsistent, out-of-order instruction sequences are executed simultaneously, more copies of general register instances are maintained, associated to a corresponding unique identifier in order to reconstruct the correct order of execution,
- **dynamic register allocation**: general registers are renamed at run-time, in order to reduce the effect of those logical dependencies that cannot be predicted at compile-time, due to the application of out-of-order mechanisms.

In many commercial products, such techniques are implemented at the expense of other important resources, e.g. by reducing the primary / secondary cache capacity, with contradictory results.

The main negative consequence has been a too large increase in power consumption. This phenomenon has been the primary reason for the “Moore law” crisis, i.e about in 2004 the CPU technology evolution (“double clock frequency every 1.5 year”) has been stopped. Currently, 4 GHz is still the highest-end product.

This trend has been replaced by the multi-core evolution/revolution, according to which the new version of “Moore law” is “double the number of CPUs per chip every 1.5 years”, provided that much simpler CPUs are exploited. The clear trend is towards in-order ILP cores, notably Pipelined CPUs or 2-way superscalar CPUs.