Course Notes

High Performance Computing

Marco Vanneschi © - Department of Computer Science, University of Pisa

Part 1

Structuring and Design Methodology for Parallel Computations

This Part deals with systematic methods and techniques for the design of high-performance parallel and distributed computations. The goal 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 Queuing Theory and Queuing Networks are adopted.

The performance evaluation methodology has a much wider application. In Part 2 it will be used for multiprocessors and multicomputer architectures.

The last five sections apply the methodology to the firmware level, that is to structure parallel computations implemented as collections of processing units. In particular, we study *the Instruction Level Parallelism* (ILP) as the main approach to real CPU architectures with scalar and superscalar organization. ILP techniques are applied to hardware *multithreading*, which represents an additional, important feature for the implementation of parallel computations.

Contents

1. Introduction to High Performance Computing

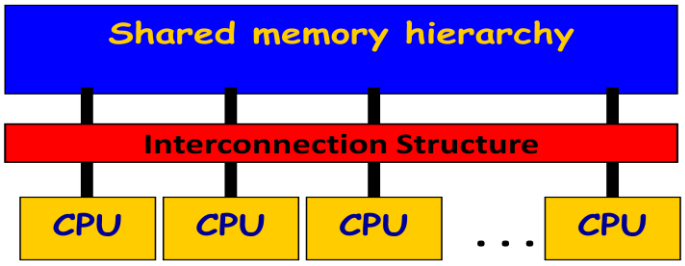
High Performance Computing (HPC) is an ICT area that studies hardware-software architectures and applications characterized by requirements for high processing bandwidth, low response time, high efficiency and scalability.

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.

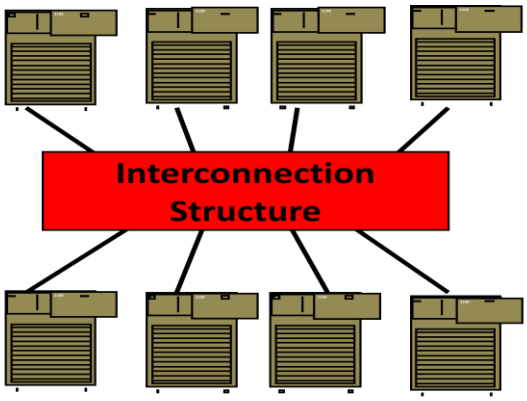
HPC is synonymous with *parallel and distributed processing*. Any HPC product is manly based on parallelism exploitation:

- *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*:



Shared memory multiprocessor



Distributed memory multicomputer: PC cluster, Farm, Data Centre, ...

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. Since 2004 the so called "Moore law" - according to which the CPU clock frequency doubles about every 1.5 years - 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.

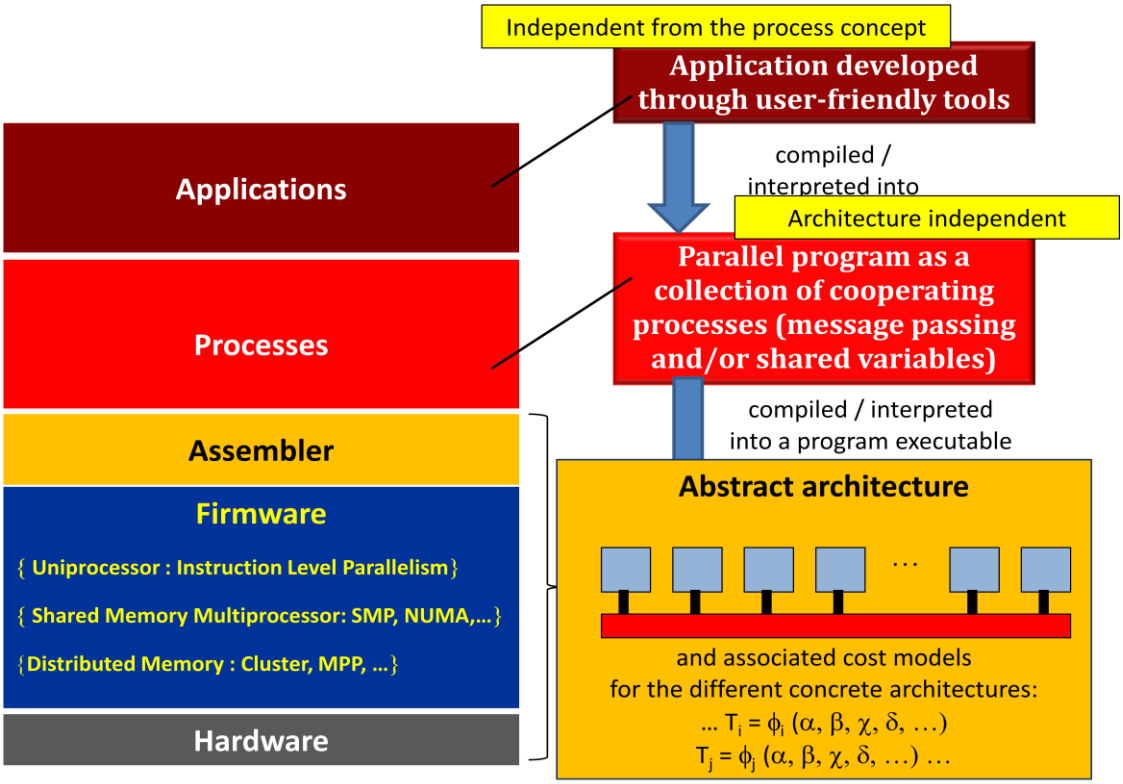
Start-up bibliography on general sources and on Computer Science Department research:

1. D.A. Patterson, J.H. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*. Morgan Kaufman Publishers Inc.
2. D. E. Culler, J.P. Singh, A. Gupta, *Parallel Computer Architecture – A Hardware/Software Approach*. Morgan Kaufmann.
3. B. Wilkinson, M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall.
4. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. In Research Monographs in Parallel and Distributed Computing, 1989.
5. M. Cole, "Bringing skeletons out of the closet: a pragmaticmanifesto for skeletal parallel programming," *Paralle Computing*,vol. 30, no. 3, pp. 389–406, 2004.
6. M. Danelutto, "Dynamic Run Time Support for Skeletons". In *Proc. of the International Conference ParCo99*, 1999, Parallel Computing Fundamentals & Applications, pages 460-467.
7. M. Vanneschi, "The programming model of ASSIST, an environment for parallel and distributed portable applications". *Parallel Computing* vol. 28, n.12, pp. 1709-1732, 2002.
8. D. Gannon, et al., "Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications". *Journal of Cluster Computing*, Vol. 5, No. 3, 2002, pp. 325-336.
9. D.A. Patterson, "Computer Science Education in the 21st Century". *Communication ACM*, March 2006, pp. 27-30.
10. K. Asanovic, et al, "A View of the Parallel Computing Landscape". *Communication ACM*, Oct. 2009.
11. H. Franke, at al, "Introduction to the Wire-Speed Processor". *IBM Journal of Res. & Dev*., vol. 54, n. 1, Jan/Feb 2010.
12. C. Bertolli, R. Fantacci, G. Mencagli, D. Tarchi, M. Vanneschi, Next generation grids and wireless communication networks: towards a novel integrated approach. *Wireless Communications and Mobile Computing*, vol. 8, 2008, Wiley InterScience.

2. Structured parallel computations

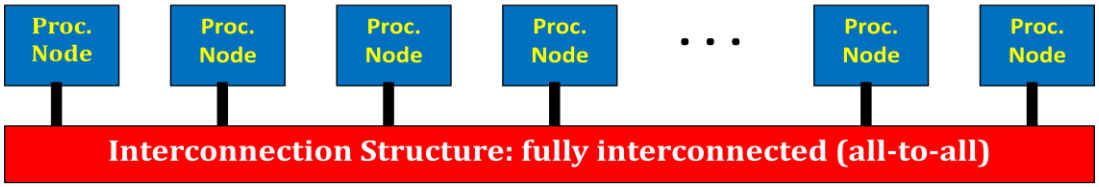
2.1 Level structuring, cost model, and abstract architecture

In Section 1.6 of the Background Part, we have discussed the strong relationship between level structuring and the design methodology for parallel computations, stressing the importance of the *abstract architecture* and of the *cost model* concepts:



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

We remind that, 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*;

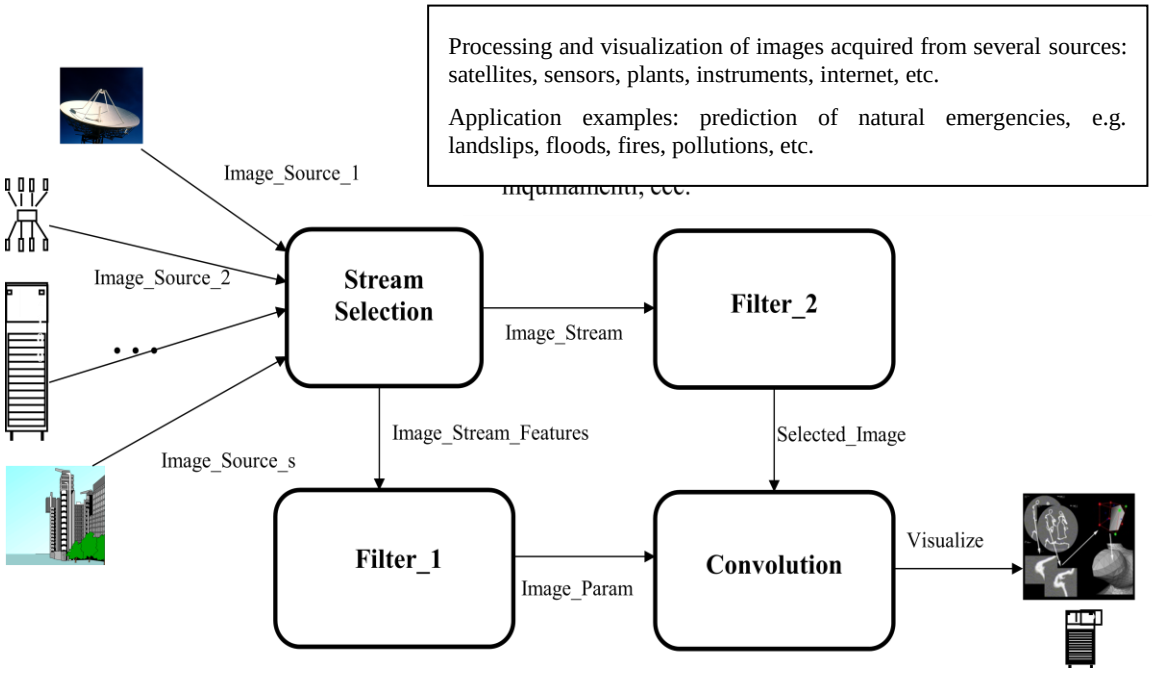
2. every node has the same characteristics of the (abstract) architecture of the processing nodes;
3. nodes interact through a *fully interconnected network*, in which each node is connected by a dedicated link to any other node. An interprocess communication channel is implemented by the physical channel connecting the corresponding processing nodes;
4. if the concrete architecture is able to support *calculation-communication overlapping*, then it is supported by the abstract architecture too;

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.*

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:



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 module into a parallel computation* which is functionally equivalent, i.e. without modifying the semantics and 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 performance metrics such as *service time T* (inverse of *processing bandwidth*, or average throughput) and *completion time* T_c (the mean 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.

The *latency* is defined as the mean time to execute one stream element.

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

This introductory considerations show 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 initial 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 metrics 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 are transformed into tractable problems.

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

We will consider the following parallel paradigms:

A. stream-parallel paradigms

- *Farm*
- *Pipeline*
- *Data-flow*
- *Functional partitioning*

B. data-parallel paradigms

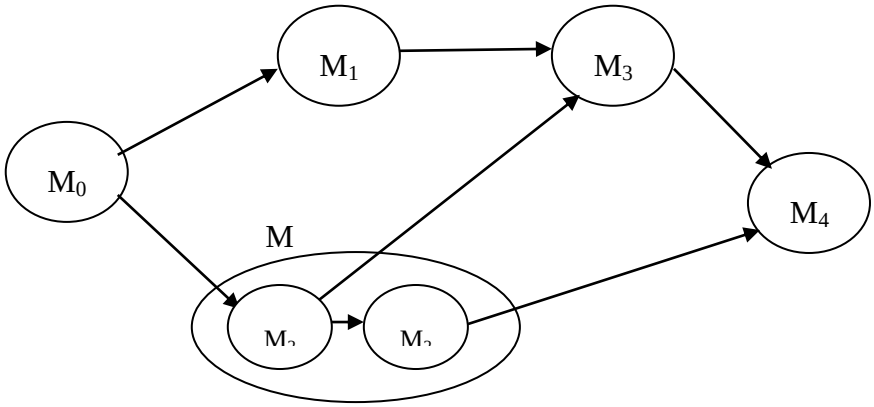
- *Map*
- *Reduce, parallel prefix*
- *Stencils*
- *Divide & Conquer*

Class *A* is defined to operate on data *streams only*, while class *B* is able to operate *on single values and also on streams*. That is, a stream-based computation can be parallelized by paradigm of class A, or paradigms of class B, or any composition of them. Just to have a first idea:

- 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:



Each module can be parallelized according to *its own parallelism paradigm*. Stream parallel and data-parallel paradigms can be composed in complex 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. Notably, 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 the universality of the adopted paradigms has been proved (see the Cole's definition of algorithmic 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. Metrics for performance evaluation

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

As introduced, the various parallel paradigms will be characterized by *cost models* that allow the designer and/or the compiler to evaluate the performance metrics 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, interprocess communication latency).

3.1 Service time and bandwidth, latency, and completion time

Let us first consider *a single sequential module* M *operating on streams*. M implements a set of *k* operations, each one with mean service time T_i and occurrence probability p_i, where

$$\sum_{i=1}^{k} p_i = 1$$

Thus, the ***mean service time*** (***service time***, in the following) is

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

The ***mean 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 as number of operations per second (*ops).*

Observations

1) These definitions (that have been adopted at the firmware and assembler level, see Part 0) 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 instance 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 will 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 *mean 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 and of its utilization factor.

3) Therefore, the above definitions of B and T_c refer to the *ideal bandwidth*, B_{id}, and to the *ideal completion time*, $T_{c\text{-}id}$, 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 mean 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* Σ composed of *n* modules.

The ***parallelism degree*** of Σ 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 will use the notation $X^{(n)}$. For example, the parallelization problem consists in transforming a system or an application $\Sigma^{(1)}$ into a system or an application $\Sigma^{(n)}$ which is functionally equivalent.

The ***latency*** of $\Sigma^{(n)}$, $L^{(n)}$, is defined as the mean 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 $\Sigma^{(n)}$, $B^{(n)}$, o *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 notable situations exist in which meaningful performance metrics 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 and 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 metrics, 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.

3.2 Efficiency and scalability

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

The ***relative efficiency*** is defined as:

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

where the effective and ideal values are considered *for the same parallelism degree*. The relative efficiency is a universal and very meaningful metric, 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:

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

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

During the course we will see that, *while efficiency is applicable to any parallel problem, there are some cases in which scalability is of scarce or null value*.

A large class of problems in which scalability is a meaningful metric is when we actually transform a given sequential computation into a parallel one. 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:

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

The *effective* values of these parameters will be $T^{(n)} \geq T_{id}^{(n)}$, $B^{(n)} \leq B_{id}^{(n)}$.

Notice that, though very important, these relations are meaningful only *when the equivalent sequential computation exists*. For example, it is possible that a problem is specified in parallel form, and that the equivalent sequential version cannot be defined.

When scalability is meaningful, a simple relation between scalability and efficiency can be easily derived:

$$s = \varepsilon \cdot n$$

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 ε 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 ε will monotonically decrease tending to zero.

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

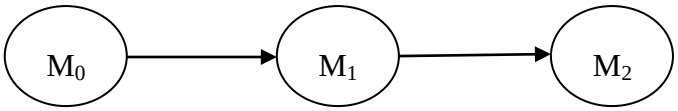
$$\varepsilon = \frac{T_{c-id}^{(n)}}{T_c^{(n)}} \qquad\qquad s = \frac{T_c^{(1)}}{T_c^{(n)}} = \varepsilon \cdot n$$

Finally, some spurious case can sometimes occur: ε 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 configurations related to $B^{(n)}$ 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 and exploitation of reuse). 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:



M_0 generates a stream as a result of the application of function F_0 to a set of *m* local values. M_1 applies a function F_1 to each element received from M_0 and sends the results to M_2. M_2 applies a function F_2 to each element received from M_1 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 latencies on performance is negligible.

The ideal and effective service time of every module is equal to *t*, and it is also equal to 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, in subsequent Sections we will prove this results systematically.

Therefore, the processing bandwidth and completion time are:

$$B^{(3)} = \frac{1}{t} \quad elements\ processed\ per\ second \quad , \quad T_c^{(3)} \sim m\,t$$

In this case, an equivalent sequential module operating on streams *can* be defined and it is characterized as follows:

$$T^{(1)} = L^{(1)} = 3\,t \quad , \quad B^{(1)} = \frac{1}{3\,t} \quad elements\ processed\ per\ second \quad , \quad T_c^{(1)} = 3\,m\,t$$

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* situation 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:

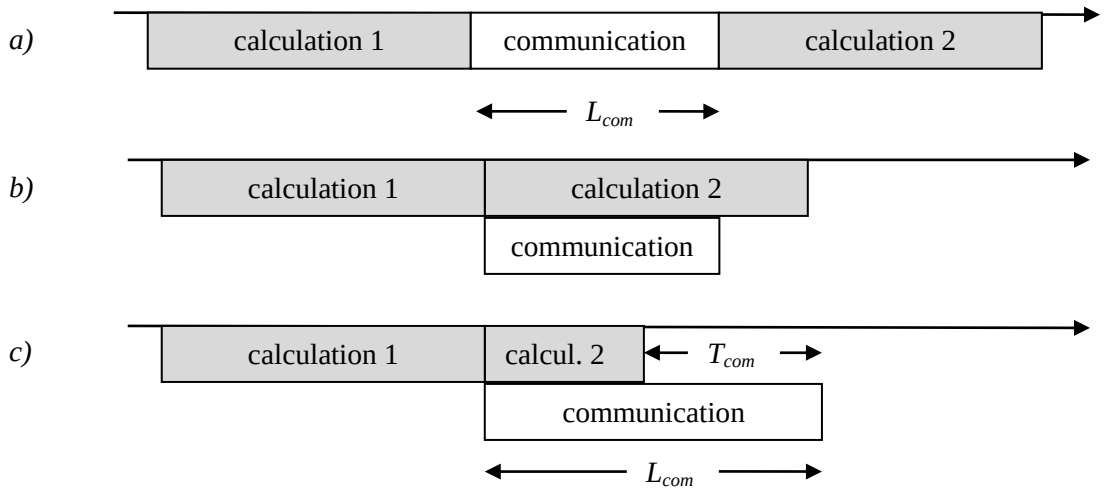
$$T^{(3)} > t \qquad L^{(3)} > 3t$$

3.3 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. In this way, the communication latency L_{com} can be masked, at least in part.

At the process level, L_{com} is evaluated as shown in Part 0, Section 6.3.

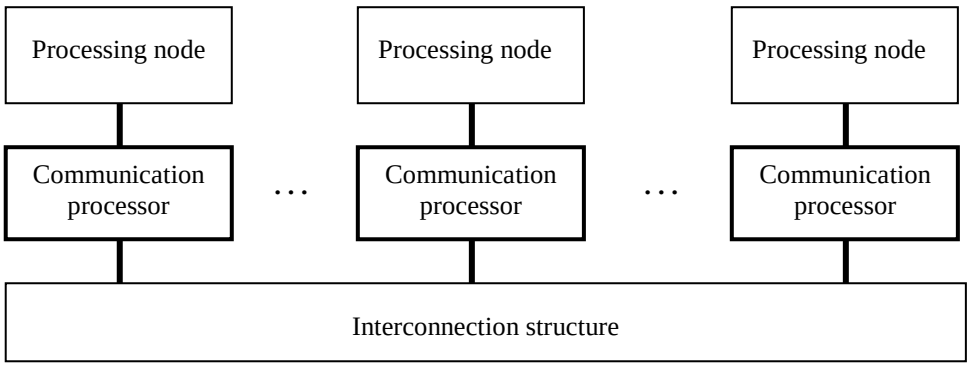
The following figure shows feasible temporal situations of a module behavior:



Assume that a message sent by the module is the result of the phase indicated by "calculation 1":

a) 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 between the phases), or by the absence of proper architectural supports (see case *b*);

b) 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:



This simplified scheme will be detailed in the next section;

c) 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. Calculation is *partially overlapped* to communication, which has a non-null impact on the service time.

3.3.1 Ideal service time in presence of communication

Case *c)* could be considered the most general one. Let us denote by T_{com} the ***mean 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 the calculation and communication phases are sequentially ordered ($T_{com} = L_{com}$).

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

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

where T_{calc} is the average calculation time. We have:

$$T_{com} = \begin{cases} L_{com} - T_{calc} & if\ L_{com} > T_{calc} \\ 0 & otherwise \end{cases}$$

Thus, equivalently we can write:

$$\boldsymbol{T = max\ (T_{calc},\ L_{com})}$$

This important, yet simple, formula characterizes the cost model of the *ideal* service time in presence of communications.

The *latency o*f the module is always affected by the communication latency:

$$L = T_{calc} + L_{com}$$

Example

Let us re-consider the example at the end of Sect. 3.2. Relaxing assumption *b)*:

$$T_{id}^{(3)} = t \qquad T^{(3)} = t + T_{com} \qquad \varepsilon = \frac{t}{t+T_{com}} = \frac{1}{1+\frac{T_{com}}{t}}$$

For example, assume that (Part 0, Section 6):

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 $T_{setup} = 10^{-2}\ t$ and $T_{transm} = 10^{-3}\ t$, and contains a communication processor for every processing node.

Therefore:

$$L_{com} = T_{setup} + L * T_{transm} = 10^{-2}\ t + 10^{-3}\ L\ t \sim 10^{-3}\ L\ t$$

For $L \leq 10^3$, we have $L_{com} \leq T_{calc}$, i.e. $T_{com} = 0$ 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 to $T_{calc} = t$.

The latency of the whole computation is:

$$L^{(3)} = 3\ t + 2\ L_{com}$$

where term $2\,L_{com}$ could be $3\,L_{com}$ or $4\,L_{com}$ if we take into account also possible communications to M_0 and from M_2.

3.3.2 Communication processor and communication thread

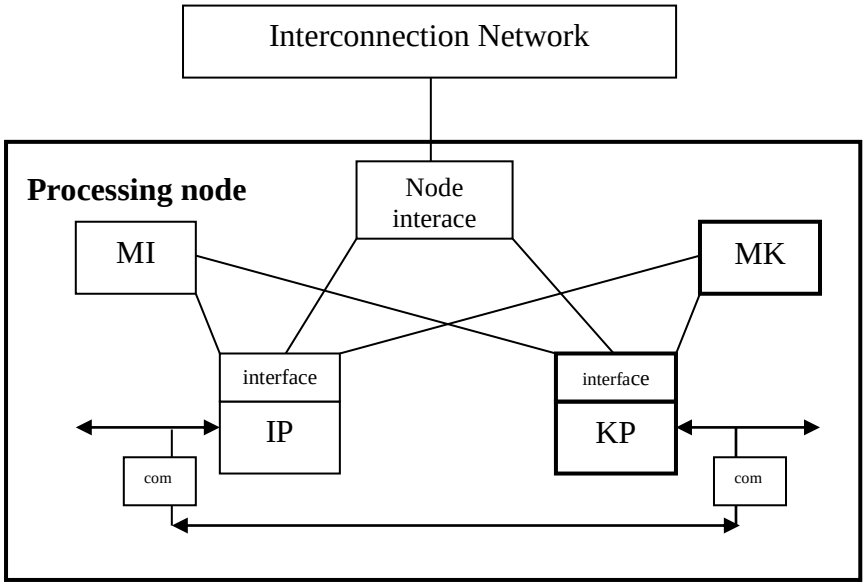
The needed architectural support consists in a *communication processor* (KP) associated to the *main processor* (IP) of the processing node. *KP is dedicated, or specialized, to the execution of the run-time support functionalities*, and in particular the *send* primitive.

In terms of cost, notice that the number of nodes is not doubled if KP is specialized and *integrated in the same chip* of IP, for example KP is a coprocessor in a multicore component: this is a state-of-the-art technological feature.

The principle is the following:

- when IP has to execute a *send* primitive on an *asynchronous* channel, it delegates this task to KP and continues the execution;
- the *receive* primitive is executed by IP entirely.

The architectural scheme of the processing node is based on a shared memory cooperation between IP and KP. This can be achieved by realizing the node as a small multiprocessor with dedicated processors, for example:



where MI and MK are shared local memories or, better, the shared secondary cache, if the node is integrated on a single chip.

As said, KP could be an *input-output coprocessor*, sharing memory with IP through DMA and/or Memory mapped I/O. This kind of implementation is adopted also when the communication processor is provided "inside" the interconnection network box. That is, the externally available links of the interconnection network (i.e. the links to be connected to the processing nodes) are I/O links.

The parameter passing from IP to KP is done *by reference* (by *capability*, see Part 0, Section 4.5) through shared memory, without additional copies. IP prepares a data structure *S* containing the channel identifier and the message reference. The reference to *S* is transmitted to KP via I/O (in the figure; via the pair of specialized I/O units *com-com*).

KP is a "daemon", that, once activated by the interrupt from the associated *com* unit, acquires the parameters into its addressing space, with very low overhead, and starts the *send* execution.

In this example, we have a clear proof of the *capability role* in the shared-pointer problem. The KP process must share any possible message, target variable and PCB of communicating processes. A static allocation of such objects in the KP addressing space is extremely inefficient or practically impossible. The dynamic allocation allowed by the capability addressing solves the problem in an elegant and efficient manner.

Let us know analyze the *send* run-time support in more depth, in order to generalize what has been studied in Part 0, Section 6.2.

The *send* semantics is: copy the message and, if the asynchrony degree becomes saturated (buffer_full), suspend the sender process. Of course, this condition must by verified in presence of KP too.

If IP delegates the *send* execution *entirely* to KP, then some complications are introduced in the *send* implementation because of the management of the waiting state of the sender process, and if we wish to achieve the objective of delegating *more than one* asynchronous communications (on the same channel or on different channels) overlapped to the same calculation section, e.g.

calculation ...; send ...; send ...; send ...; ... calculation ...

A simpler and efficient solution, able to achieve this objective, consists in the following scheme:

- *IP itself verifies the saturation of channel asynchrony*;
- IP delegates to KP the *send* continuation;
- if (*buffer_size* = *k*) IP sets the *wait* boolean variable in the channel descriptor, and suspends the sender process (as usually, if *k* denotes the asynchrony degree, *k + 1* is the number of buffer elements);

In the *zero-copy* communication, IP controls the *validity bit* too, if provided.

The channel descriptor is *locked* by IP and *unlocked* by KP (see Part 2 for locking issues).

This scheme eliminates the complexity of the general solution (full delegation to KP), at the expense of *an initial phase executed by IP itself, thus not overlapped to the internal calculation*. In the interprocess communication cost model, the latency of this phase must be included in the T_{calc} parameter. In practice, the overlapping is applied to the channel descriptor buffer manipulation, to the message copy, and to the low-level scheduling actions on the destination process.

Equivalently, an alternative to the physical communication processor is a ***communication thread***, provided that the node architecture is *multithreaded*. "Hardware multithreading" (simultaneous multithreading) is studied in Section 22, and the KP-thread solution is exemplified in Section 22.4.3.

4. Stream computations studied as queuing systems and queuing networks

The cost models for parallel computations will be expressed in terms of fundamental results in the area of *Queuing 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.

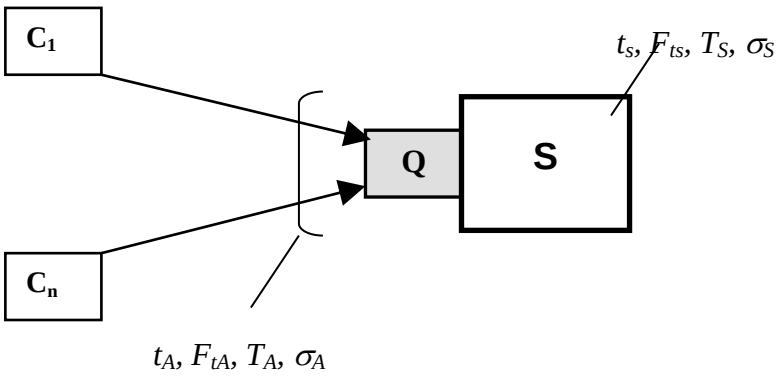
Queuing Theory is a very general approach based on mathematical tools, especially Probability Theory and Stochastic Processes.

> 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 Queuing Theory, called *Queuing Networks*, aiming to solve complex compositions of queuing systems according to a simplified, yet rigorous, mathematical treatment.

4.1 Queuing systems and utilization factor

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



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 same 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 mathematical and stochastic assumptions are validated.

The queuing 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 Queuing Theory have been derived for infinite length queues. Though in a computing system this size is fixed or is limited, the utilization of results for the infinite length queues often represents a first approximation to have a rough idea of the system parameters. Where possible, the analytical treatment of finite length queues will be used.
- The *probability distribution of the random variable* ***service time***, t_S

  $$F_{t_S}(t) = \Pr(t_s \le t)$$

  with mean value T_S e variance σ_S. The *mean rate 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)*

  $$F_{t_A}(t) = \Pr(t_A \le t)$$

  with mean value T_A e variance σ_A. The *mean rate of interarrivals is denoted by* $\lambda = 1/T_A$.

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

$$\rho = \frac{T_S}{T_A} = \frac{\lambda}{\mu}$$

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

When $\rho > 1$, the server represents a *bottleneck* with respect to the clients requests: after a transient period in which responses are returned to the clients in a finite time, during the steady-state behavior, the average number of queued request grows indefinitely and so does the response time. This situation occurs only when the client does not need an explicit request from the server, notably, in a computation described by an ***acyclic graph***: if this condition occurs, *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*, when $\rho > 1$ the client is temporarily blocked each time it tries to send a new request, thus *in the steady-state behavior the client request rate adapts to the server service rate*. For this reason the situation $\rho > 1$ is a ***transient*** one. However, for our purposes, it is meaningful, because it *is the condition we have to verify, in order to discover the possible existence of a bottleneck.*

In other words, our final goal is to evaluate the performance metrics of single modules and of the whole computation in the steady-state situation. However, we are interested in studying both the ***transient behavior*** and the ***steady-state behavior***:

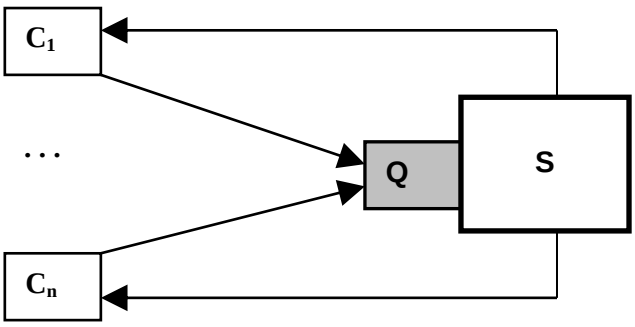
- if $\rho < 1$, the server is not a bottleneck, and the distinction between the transient phase and the steady-state phase has no meaningful impact on the average performance measures;
- if $\rho > 1$, the server is a bottleneck, and a non-null transient phase exists before reaching the steady-state behavior. Once the behavior is stable, the mean interarrival time becomes equal to the mean service time of the server (however,

instantaneous fluctuations around the mean values exists and could be considerable), thus the mean service times of the clients are increased with respect to the initial values, i.e. with respect to the ideal service times considering each client "in isolation".

The condition $\rho = 1$ deserves a special explanation:

- from a pure mathematical point of view, only if *both* the interarrival and the service time distributions are the uniform ones (null variance), then the server is not a bottleneck: server and clients are perfectly synchronized at each time instant. However, for any other distribution of the interarrival time *or* of the service time, for $\rho \to 1$ *the response time tends to infinity;*
- from our purposes of performance evaluation, for *acyclic graph* computations, $\rho = 1$ denotes a "limit situation" in which, on the average, the clients are not delayed, although considerable fluctuations around the mean values exist and, in presence of finite size queues, considerable blocking periods could be introduced in the client behavior. From a mathematical point of view, in a acyclic graph computation, the condition $\rho = 1 - \delta$, with $\delta > 0$ arbitrarily small, is sufficient for a steady-state behavior without bottlenecks.

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*):



We assume that each client performs the request and then *waits* for the result from the server. In this case, it is always $\rho < 1$, because the systems has a *self-stabilizing behavior*, i.e. a temporary increase of the interarrival time has the effect of a decrease of the utilization factor and of 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 ρ (tending to one) cause a greater mean response time, that lowers the client processing bandwidth compared to the ideal value (i.e. the value in isolation).

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 ρ value.

To summarize the previous discussion and to draw a first conclusion:

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 ρ 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 due to fluctuation of service and interarrival times. Moreover, with proper asynchrony degree, we are allowed to exploit the basic Queueing Theory results for infinite length queues: in the majority of cases, an asynchrony degree of few units is sufficient for a good approximation, thus for reducing the blocking probability to acceptable value so that the blocking overhead can be neglected.

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 an 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 Queuing Systems in terms of probability distributions is necessary mainly for request-reply computations. This case will be studied in Section 15.

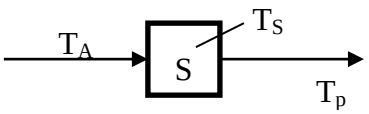
For acyclic graph computations, *Queuing Networks* are a sufficiently powerful methodology. They do not utilize an explicit analytical treatment in terms of probability distributions, instead they are based on some basic results about the *information flows* in a network of queues. For our purposes, the Queueing Networks results make it possible to analyze a graph computation in order to discover bottlenecks and critical paths, thus in order to evaluate the mean values of performance metrics in the steady-state behavior . This issue will be dealt with in the Section 4.2.

4.2 Fundamental properties of computations studied as queueing networks

Our current goal is to study *acyclic* graph computations in order to evaluate the performance metrics of the single modules and of the whole computation (shortly: "to solve the graph"). As said, the case of graphs containing request-reply loops will be studied in Section 15.

4.2.1 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 mean 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:

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

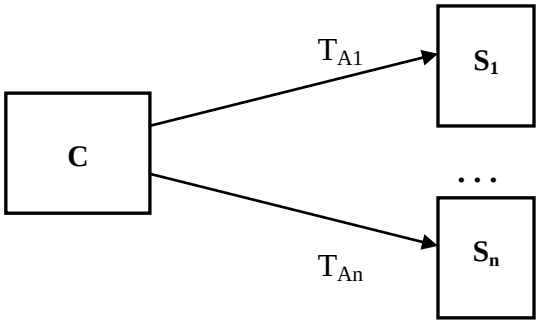
Equivalently:

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

This result is valid according to the point of view of Queuing Networks. From the mathematical point of view of Queuing Theory it is sufficiently approximate if a proper asynchrony degree is provided, as we always assume in our evaluations (see the discussion at the end of Section 4.1).

4.2.2 The "server partitioning" theorem

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



Let T_p the interdeparture time of *C* towards any server, and $p_1, \ldots, p_n$ the probabilities that a generic request of *C* is directed to $S_1, \ldots, S_n$ respectively, where

$$\sum_{i=1}^{n} p_i = 1$$

If all the utilization factors $\rho_1, \ldots, \rho_n$ are less than one, then the interarrival time of the generic server S_i is given by:

$$T_{A_i} = \frac{T_p}{p_i}$$

Proof

Let a module P, with service time T_P, be the client of a server module S with probability *p*, and client of a set of other modules with probability *1 – p*. Because, by hypothesis, S and the other servers have utilization factors less than one, the interarrival times from P do not depend on the servers service time. For this reason, the interarrival time t_{PS} from P to S is a random discrete variable with the following distribution:

t_{PS}	with probability
T_P	p
$2\ T_P$	$p\ (1 - p)$
…	
nT_P	$p\ (1 - p)^{n-1}$
…	

Thus, the mean value of the interarrival time is:

$$T_{PS} = \sum_{n=0}^{\infty} n\, T_P\, p\, (1-p)^{n-1} = \frac{p\, T_P}{1-p} \sum_{n=0}^{\infty} n\, (1-p)^n$$

According to the general property:

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

we have:

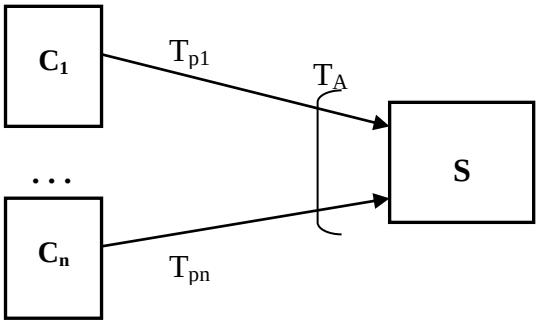
$$T_{PS} = \frac{T_S}{p}$$

□

The hypothesis "*if all the utilization factors ρ_1, ..., ρ_n are less than one"* is very important. As usually, the interarrival times, evaluated with this theorem, are valid in the *transient* behavior of the computation. If at least a server is a bottleneck, in the *steady-state* behavior its mean interarrival time increases (becomes equal to the mean service time), thus the mean interdeparture time of the client increases and must be re-evaluated. As a consequence, the interarrival times to the other servers must be re-evaluated. This problem will be studied in Section 4.2.5.

4.2.3 The "multiple clients" theorem

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



<u>*In the transient behavior*</u>, the *interarrival time to the server* is given by:

$$T_A = \frac{1}{\lambda} = \frac{1}{\sum_{i=1}^{n} \lambda_i} = \frac{1}{\sum_{i=1}^{n} \frac{1}{T_{p_i}}}$$

That is, *the global interarrival rate is equal to the sum of the individual interarrival rates from the clients*:

$$\lambda = \sum_{i=1}^{n} \lambda_i$$

Equivalently: *the global interdeparture rate of a set of clients is equal to the sum of the individual interdeparture rates*.

Proof

The arrival rate from the generic client is given by

$$\lambda_i = \frac{1}{T_{p_i}}.$$

Therefore, in the transient behavior the total number of requests received by *S* in a time unit is the sum of the individual arrival rates from each client:

$$\lambda = \sum_{i=1}^{n} \lambda_i$$

□

Important observations

1) 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 system $\Sigma = \{ \Sigma_1, .., \Sigma_n \}$ is equal to the sum of the individual bandwidths:*

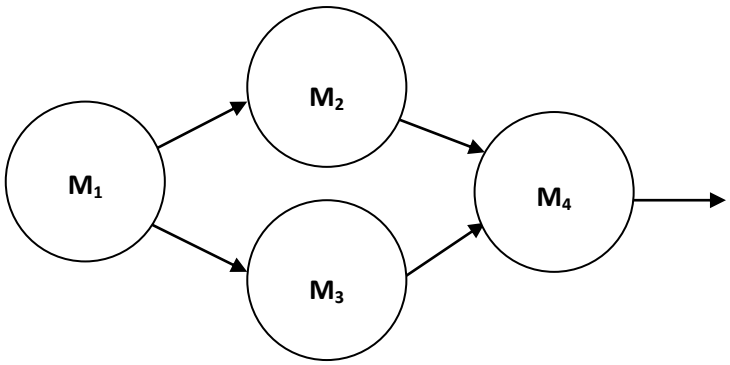
$$B = \sum_{i=1}^{n} B_i$$

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 Σ considered *in isolation*: not necessarily the whole system, in which Σ is possibly included, is able to exploit such bandwidth. Sections 4.3, 4.4 will discuss these important issues in detail.

2) The interarrival time, evaluated with this theorem, is valid in the *transient* behavior of the computation. *If the server is a bottleneck*, in the *steady-state* behavior the mean interarrival time increases (adapting to the mean service time), thus the mean interdeparture time of each client increases and must be re-evaluated. This problem will be studied in the Section 4.2.6.

4.2.4 Evaluation example of a graph computation without bottlenecks

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



Let *t* be a standardized time unit. M_1 generates a stream of N = 10K integer values with interdeparture time equal to *30t*. Each element of the stream is sent to M_2 with probability equal to 3/4, otherwise to M_3. Let the calculation times of M_2, M_3, M_4 be *20t, 100t, 15t* respectively.

Σ is a computation with parallelism degree $n_\Sigma = 4$, whose *processing bandwidth cannot be greater than the stream generation bandwidth* of M_1. Therefore the *ideal service time* of Σ is

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

The interarrival times to M_2 and M_3 respectively are given by:

$$T_{A2} = \frac{T_{p1}}{\frac{3}{4}} = 40t$$

$$T_{A3} = \frac{T_{p1}}{\frac{1}{4}} = 120t$$

The utilization factors of M_2 and M_3 are:

$$\rho_2 = \frac{T_2}{T_{A2}} < 1$$

$$\rho_3 = \frac{T_3}{T_{A3}} < 1$$

M_2 *and* M_3 are not bottlenecks. Thus, their interdeparture times are:

$$T_{p2} = T_{A2} = 40t$$

$$T_{p3} = T_{A3} = 120t$$

The interarrival time to M_4 is:

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

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 interdeparture time is equal the effective service time of the computation.

The relative efficiency and scalability reach the ideal values:

$$\varepsilon_\Sigma = \frac{T_{id}^{(4)}}{T^{(4)}} = 1$$

$$s^{(4)} = \varepsilon_\Sigma * 4 = 4$$

The completion time of Σ is approximately given by:

$$T_c^{(4)} \cong N\, T^{(4)} = 300\, K\, t$$

The computation latency is evaluated as:

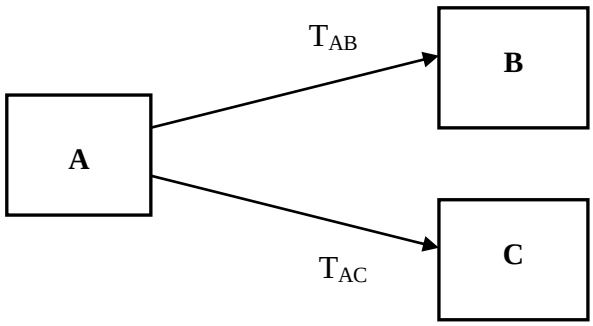
$$L_\Sigma = T_1 + \frac{3}{4}\, T_2 + \frac{1}{4}\, T_3 + T_4 = 85t$$

4.2.5 Multiple-servers with bottlenecks

As discussed in Sections 4.2.2 and 4.2.3, the server partitioning theorem and the multiple clients theorem are valid in the *transient* behavior of the computation. Thus, they are valid also in the steady-state behavior only if there are no bottlenecks. Otherwise (at least a server is a bottleneck in a multiple-server computation, or the server is a bottleneck in a multiple-clients computation), all the interarrival and interdeparture times must be *re-evaluated* in order to have a correct evaluation of the steady-state behavior performances.

In this Section we study the problem of multiple-servers computation, while the multiple-client computation will be studied in the next Section.

If the assumptions of the server partitioning theorem is 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:



where A, B, C have service times T_A, T_B T_C, and let p the probability that A requests a service to B. Let us assume that B is a bottleneck:

$$\rho_B = \frac{T_B}{T_{AB}} > 1$$

where the transient value of T_{AB} is evaluated according to the server partitioning:

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

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

The *steady-state 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 the A activity is:

$$T'_A - T_A = p\,(T_B - T_{AB})$$

The steady-state interarrival time to C is:

$$T'_{AC} = \frac{T'_A}{1-p} = \frac{p}{1-p}\, T_B$$

Moreover, it can be proved that *there cannot be more than one bottleneck, that is the evaluation must be done by considering the server module having the highest utilization factor.*

The previous example is generalized by a theorem due to *Gabriele Mencagli*.

Proposed exercise: introduce some bottlenecks in the graph of Sect. 4.2.4, and solve it.

4.2.6 Multiple-clients with bottleneck

The problem is to find the steady-state effective service times of all the clients. This problem can be solved by applying general methods of Queueing Theory, once the interarrival and service time distributions are known. This approach will not be studied in this course.

Instead, we are interested in methods based upon Queueing Networks and Information Flows to solve the graph. A method of resolution for acyclic graph computations with bottlenecks, and an algorithm, have been formulated by *Gabriele Mencagli*, taking into account the multiple-server case too. The method is not valid for any graph, but for a very significant and large class of graphs: acyclic computation with a *single source* module.

The method and the algorithm are described in a companion document.

4.3 Ideal and effective bandwidth of modules and graph computations

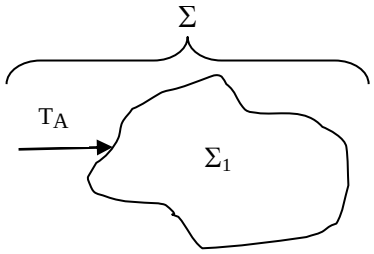
The next step in our methodology is a set of general results about

1) how to evaluate <u>ideal</u> and <u>effective</u> bandwidths of acyclic graph computations,

2) how to evaluate the <u>optimal degree of parallelism</u>.

Let us start with issue 1, while issue 2 will be studied in the next Section.

Let us consider an acyclic computation Σ consisting of a module, or subsystem, Σ_1 having 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 <u>of Σ_1</u>*.

The *ideal* service time of Σ_1 is evaluated by considering Σ_1 <u>*in isolation*</u>, i.e. the ideal service time is related to its internal calculation time. If Σ_1 is a

single module, it is characterized by an average calculation time T_{calc}, thus the ideal service time of Σ_1 is given by

$$T_{\Sigma 1-id} = T_S = T_{calc}$$

If Σ_1 is *n*-parallelization (parallelism degree equal *n)* of a sequential subsystem with average calculation time T_{calc},, the *ideal* service time of Σ_1 is given by

$$T_{\Sigma 1-id} = T_S = \frac{T_{calc}}{n}$$

Otherwise, if Σ_1 is parallel and the equivalent sequential computation is not known, than the service time of Σ_1 in isolation must have been provided.

Because Σ_1 belongs to a system Σ, the *effective* service time of Σ_1 is given by its ***interdeparture time***:

$$T_{\Sigma 1} = T_p = max\ (T_S, T_A)$$

In other words, considering Σ_1 as part of a more complex system, the interarrival and the interdeparture time of Σ_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 Σ_1, considered as belonging to a complex system. By definition:

$$\varepsilon_{\Sigma_1} = \frac{T_{\Sigma_1-id}}{T_{\Sigma_1}} = \frac{T_s}{max(T_s, T_A)}$$

Therefore:

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

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

In other words, *it 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* Σ. We have:

$$T_\Sigma = T_{\Sigma 1} = T_p$$

$$T_{\Sigma-id} = T_A$$

In fact:

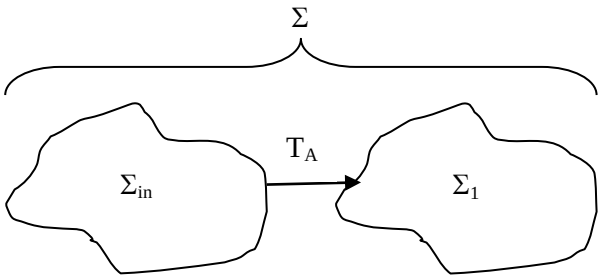
1) *the effective service time is the interdeparture time of the graph* (that coincides with the interdeparture time of Σ_1);

2) no parallel realization of Σ_1 exists able to achieve an effective service time lower than the interarrival time.

The relative efficiency of the whole system Σ 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, Σ_{in} generating the stream is explicitly represented.

By induction, any acyclic system can be derived by expanding Σ_{in}.

4.4 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 Σ_1 have service time T_s and interarrival time T_A, with $T_s > T_A$. Thus, Σ_1 is a bottleneck for the whole system. Let us parallelize Σ_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, provided that a parallelism paradigm exists able to actually exploit this parallelism degree.

- *If this parallelization is possible, then the effective service time of Σ_1 is given by:*

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

  and the relative efficiency:

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

- *If the optimal parallelization is not possible, the parallelized Σ_1 is still a bottleneck, thus*

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

  In this situation, let

  $$n_0 < \bar{n}$$

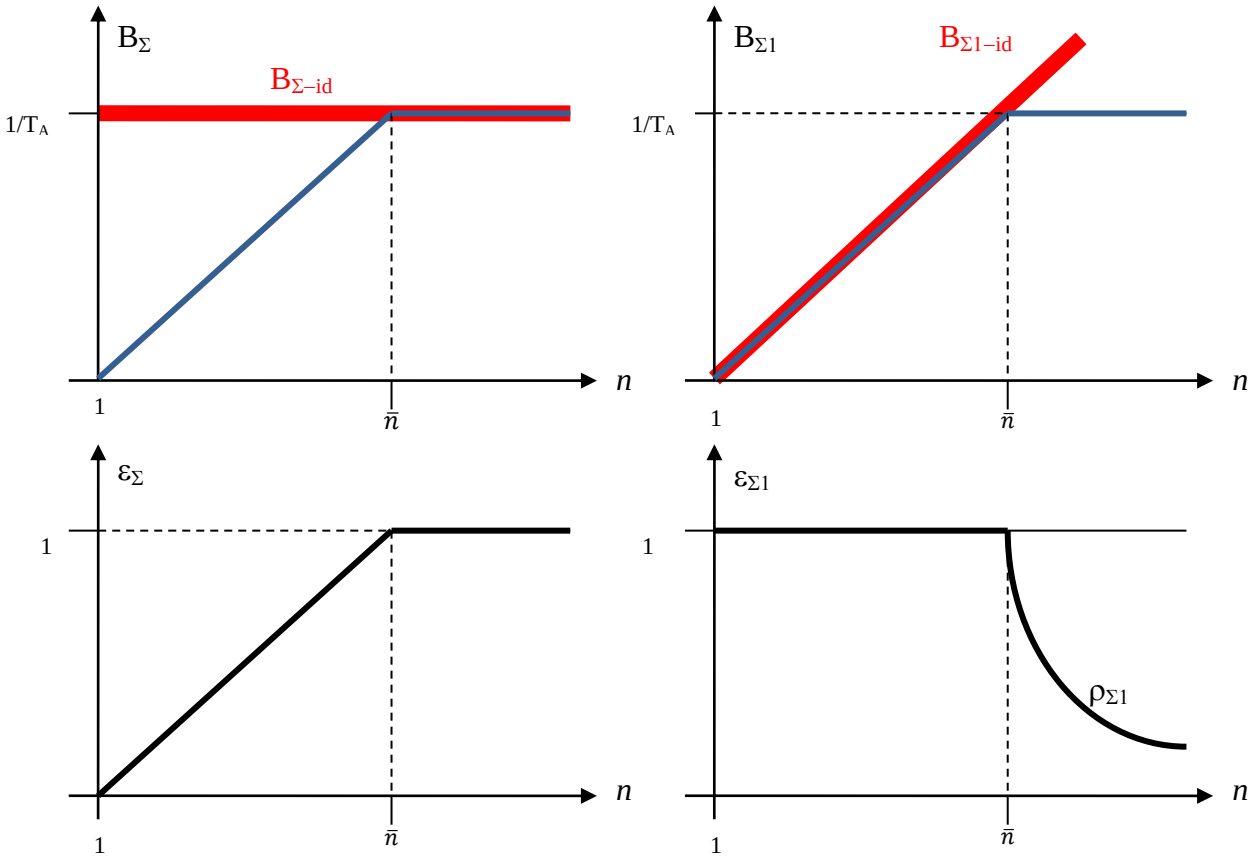
  the best parallelism degree achievable. The effective service time is given by:

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

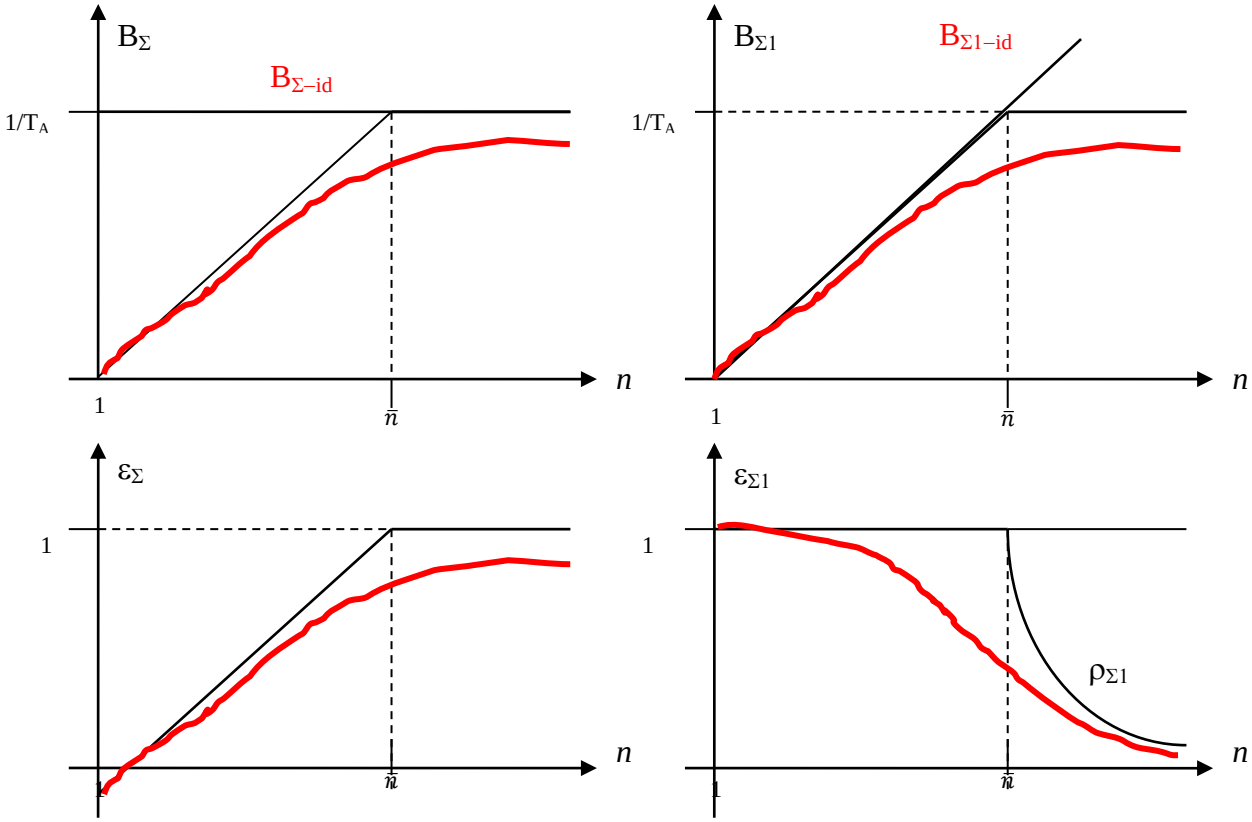
Notice that if the ratio T_s/T_A is not integer and if we choose $n = \bar{n} = \left\lceil \frac{T_s}{T_A} \right\rceil$, the utilization factor (the relative efficiency) is inevitably less than one. However this is the best solution that can be found: some additonal "resources" are needed in order to minimize the service time. On the other hand, if we choose $n = \left\lfloor \frac{T_s}{T_A} \right\rfloor$, the minimum amount of resources are used for achieving the maximum efficiency, but the service time is not optimal.

In subsequent Sections about the various parallel paradigms, 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 Σ consisting of a subsystem Σ_1, as functions of the parallelism degree *n* of Σ_1:

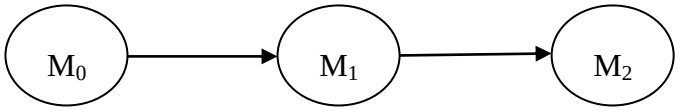


In many real-life applications, these shapes are valid as first approximations. Because of some performance degradation reasons, in general linear segments are replaced by asymptotic curves, for example:



4.5 Detailed example of acyclic graph analysis and optimization

Let us consider a computation Σ described by the following graph:



Modules functionalities are specified in a LC-like formalism (Part 0, Section 6.1):

parallel M_0, M_1, M_2;

M_0 :: *int* A[N][N]; *int* C[N];

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

send C *to* M_1 }

M_1 :: *int* C[N]; *int* X;

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

M_2 :: int D[N]; int X;

∀ i = 0 .. N–1:{ *receive* X *from* M_1; D[i] = F_2 (X) }

where $N = 10^2$.

Let the average processing times of F_0, F_1, F_2 be equal to $10^2\tau$, $10^6\tau$, $10^5\tau$, respectively, where τ = 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 (Part 0, Section 6.2.5). Le the communication cost model (Part 0, Section 6.3) be characterized by $T_{setup} = T_{transm} = 10^3\tau$.

a) Evaluate the ideal and effective bandwidth *B*, and the relative efficiency ε.

b) Parallelize the bottlenecks, and evaluate the ideal and effective bandwidth *B*, and the relative efficiency ε, 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 Σ.

To solve this problem we apply the methods and techniques of the previous Sections.

a) The calculation time of M_0 is evaluated as

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

since the loop control has a negligible impact (see Part 0 performance evaluation at vthe various levels of interest). This is the value of the *ideal* service time of M_0.

The communication latency for the channel M_0- M_1 is

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

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

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

This is the value of M_0 interdeparture time, thus of M_1 interarrival time:

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

This is also the *ideal service time of the whole computation* Σ:

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

Notice that $T_{calc\text{-}0}$ is the ideal service time *of* M_0 (not of Σ), thus the M_0 relative efficiency is given by $\varepsilon_0 = T_{calc\text{-}0}/T_0 = 0{,}1$. However, as far as Σ 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):

$$L_{com\text{-}1} = T_{send}(1) = T_{setup} + T_{transm} = 2 * 10^3\tau$$

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

$$T_1 = T_{p1} = T_{A2} = T_{calc\text{-}1} = 10^6\tau$$

The ideal service time of M_2 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:

$$T_{2\text{-}id} = T_{calc\text{-}2} = 10^5\tau$$

and M_2 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:

$$\varepsilon_\Sigma = \frac{T_{\Sigma-id}}{T_\Sigma} = 0{,}1$$

The processing bandwidth is:

$$B_\Sigma = \frac{1}{T_\Sigma} = \frac{10^9}{10^6} = 10^3 \; processed\; arrays\; per\; second$$

b) A parallel version of the bottleneck module M_1 has the optimal parallelism degree:

$$n_{opt} = \frac{T_1}{T_{A1}} = 10$$

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

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

$$T_\Sigma = T_{A1} = 10^5 \tau = T_{\Sigma\text{-}id}$$

$$B_\Sigma = \frac{1}{T_\Sigma} = \frac{10^9}{10^5} = 10^4 \; processed\; arrays\; per\; second$$

$$\varepsilon_\Sigma = \frac{T_{\Sigma-id}}{T_\Sigma} = 1$$

c) Let us evaluate the performance measures of M_1, considered as a part of Σ, as a function of the M_1 parallelism degree *n*:

$$B_{1-id} = \frac{1}{T_{1-id}} = \frac{1}{\frac{T_{calc}}{n}} = 10^3\, n \quad for\; any\; n$$

$$B_1 = \begin{cases} \frac{1}{T_1} = \frac{1}{\frac{T_{calc}}{n}} = 10^3\, n & for\; n < 10 \\ \frac{1}{T_{A1}} = 10^4 & for\; n \geq 10 \end{cases}$$

$$\varepsilon_1 = \frac{B_1}{B_{1-id}} = \begin{cases} 1 & for\; n < 10 \\ \frac{10}{n} = \rho_1 & for\; n \geq 10 \end{cases}$$

while for the whole computation Σ we have:

$$B_{\Sigma-id} = \frac{1}{T_{\Sigma-id}} = 10^4 \quad for\ any\ n$$

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

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

We recognize the shapes that have been anticipated, in a general and qualitative way, at the end of Sect. 4.4.

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 ($\varepsilon_2 = 0{,}1$).

5. General characteristics of parallel paradigms

Starting from this Section, we apply the methodology of Sections 3, 4 to the study of structured parallelism paradigms and to the design of complex parallel applications.

We can recognize 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 ::    int A[m], B[m]; …;
        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 the 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 or other overhead sources might smooth, or even annul, the difference).

6. 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. It is worth noting that sometimes some application or architectural constraints could force the utilization of this paradigm. However, if such constraints can be removed, often 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 functions* for each element *x* of the input stream:

$$F(x) = F_n (F_{n-1} (\dots F_2 (F_1 (x)) \dots))$$

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 (data-parallel solutions will be used in this case).

6.1 Cost model

As usually, let T_{calc} the average calculation time of the whole function $F(x)$:

$$T_{calc} = \sum_{i=1}^{n} T_{calc_i}$$

Initially we assume that the component functions $F_1, \dots, F_n$ have the same calculation time:

$$t = \frac{T_{calc}}{n}$$

Moreover, assume that all stages have the same T_{com} (≥ 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:

$$T^{(n)} = T_{stage} = \frac{T_{calc}}{n} + T_{com}$$

while the ideal service time is:

$$T_{id}^{(n)} = \frac{T_{calc}}{n}$$

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 (in the transient behavior) for the whole computation, in the steady-state behavior we have:

$$T^{(n)} = max_i \, (T_i) = max_i \, (T_{calc_i} + T_{com_i})$$

$$T_{id}^{(n)} = \frac{T_{calc}}{n} = \frac{\sum_{i=1}^{n} T_{calc_i}}{n}$$

$$\varepsilon = \frac{T_{id}^{(n)}}{T^{(n)}} = \frac{\sum_{i=1}^{n} T_{calc_i}}{n \ max_i \ (T_{calc_i} + \ T_{com_i})}$$

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 Section 4.5).

For the first two stages we have:

$$T_{p_0} = \max(T_A, T_0) = T_{A_1}$$

$$T_{p_1} = \max(T_{A1}, T_1) = \max(\max(T_A, T_0), T_1) = \max(T_A, T_0, T_1)$$

Applying the induction reasoning, if

$$T_{p_i} = \max(T_A, T_0, ..., T_i)$$

then:

$$T_{p_{i+1}} = \max(T_A, T_0, ..., T_{i+1})$$

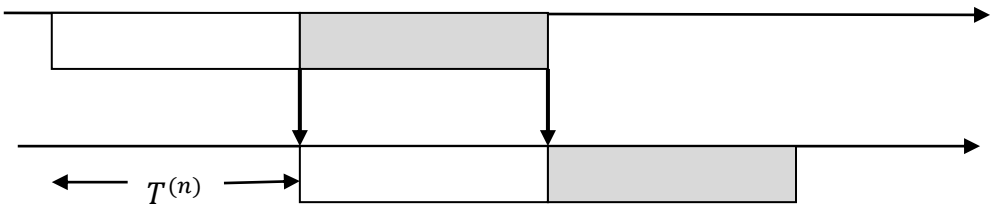
for any *i*.

In conclusion:

$$T_\Sigma = T_{p_{n-1}} = \max(T_A, T_0, ..., T_{n-1}) = \max(T_A, \max_{j=0}^{n-1}(T_i))$$

Completion time

Let *m* be the finite stream length. For our purposes, generality is not lost if we consider a balanced pipeline, or an *equivalent* pipeline in which each stage has effective service time $T^{(n)}$. It is sufficient to observe a pipeline with $n = 2$ and $m = 2$, whose temporal behavior is:



We recognize a *filling transient phase* of duration

$$(n-1)\,T^{(n)}$$

followed by a *steady-state phase* and an *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:

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

that will be applied to all the other computations on streams, 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 $m >> n$.

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:

$$\text{M} :: \quad \forall\,(a, b, c): \{\, y_0 = F_0(a, b);\ y_1 = F_1(a, c);\ y = F(y_1, y_2) \,\}$$

Conceptually, the parallelism degree is equal to three. In Section 9 we will 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. In this case the graph is a 3-stage pipeline, by providing that some unmodified input values are sent onto intermediate streams:

- S_0 :: *receive* (input_stream, (a, b, c)); *send* (stream_01, (F_0(a, b), a, c));
- S_1 :: *receive* (stream_01, (y_0, a, c)); *send* (stream_12, (y_0, F_1(a, c)));
- S_2 :: *receive* (y_0, y_1); *send* (output_stream, F(y_0, y_1)).

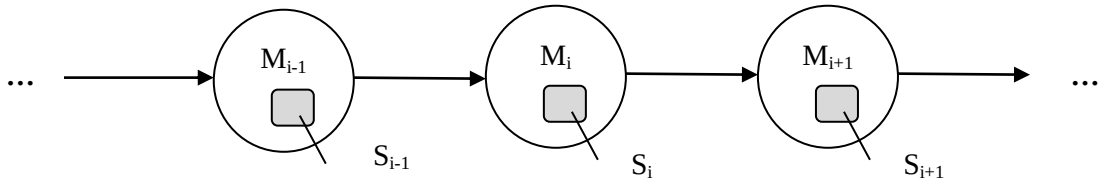
We will see that, with respect to a data-flow solution, the pipeline service time is the same, while the pipeline latency is higher.

6.2 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 *disjoint*. If the whole *n-stage* pipeline computation operates on a data structure *S*, we must be able to obtain:

$$S = (S_0, \ldots, S_i, \ldots, S_{n-1}) \mid S_i \cap S_j = 0 \quad \forall i, j\ ;\ i \neq j$$

where distinct data partitions are encapsulated into distinct stages:



Example

Let us consider the following sequential computation:

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

$$\forall x: \quad \{\, \text{index} = search\,(x, A); \quad y = \sum_{i=1}^{index} B[i] \,\}$$

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).

6.3 Function replication and data partitioning: loop unfolding

In several sequential programs, we can transform an iterative computation into the 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. If the sequential computation has this very powerful property, *a pipeline implementation with function replication and data partitioning* is possible.

For example, the following sequential module operating on streams:

```
M ::   int A[m]; input stream: int s0; output stream: int s;
       { receive s0 from input_stream;
         s = s0;
         for (i = 0; i < m; i++)
                s = F (s, A[i]);
         send s onto output_stream
       }
```

can be transformed by a *k-unfolding*:

```
M ::   int A[m]; input stream: int s0; output stream: int s;
       { receive s0 from input_stream;;
         s = s0;
         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: see Section 12.1).

The *parametric* LC description is the following:

```
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. 5, this a simple, yet meaningful, *data-parallel* implementation of the given problem. In Section 13.9 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 $T_A = L_{com} = 10\,t$, where t is a conventional time unit.

We wish to transform M according to a pipeline solution.

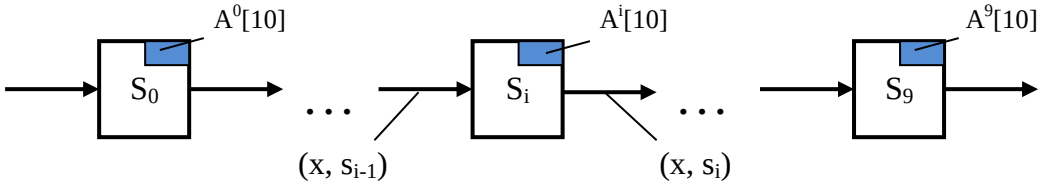
As we know, the optimal value of parallelism degree *n* is given by:

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

Correspondingly, we obtain:

$$T_{\Sigma}^{(10)} = T_{\Sigma-\mathrm{id}}^{(10)} = 10\,t \qquad\qquad \varepsilon_{\Sigma}^{(10)} = 1$$

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

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

The latency is given by:

$$L_{\Sigma}^{(10)} = T_{calc} + 10\, L_{com} = 200\, t$$

7. Stream-equivalent computations

In Sect. 5 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 ::    int A[M], B[M]; …;
        for (i = 0; i < M; i++)
                B[i] = F (A[i])
```

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

```
parallel unpack; compute; pack;

unpack::    int A[M]; channel out input_stream;
            for (i = 0; i < M; i++)
                    send (input_stream, A[i]);

compute::   int x; function F …; channel in input_stream (1); channel out
            output_stream;
            for (i = 0; i < M; i++)
                    { receive (input_stream, x); send (output_stream, F(x)) };

pack::      int B[M]; channel in output_stream (1);
            for (i = 0; i < m; i++)
                    receive (output_stream, B[i])
```

The *compute* module can be further parallelized according to one or more parallel paradigms.

Consider the *unpack* module: the calculation phase has a very fine grain (*for* loop control and read A[i]) compared to the communication latency. Thus, its service time is:

$$T_{unpack} \sim T_{arrival\text{-}compute} = L_{com}$$

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

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

$$n = \frac{T_F}{L_{com}}$$

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

Consequently, if a parallelization is feasible, it is $T_F > L_{com}$, thus $T_{com\text{-}compute} = 0$.

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*.

8. Optimal communication grain

The formulation of Sect. 7 assumes that "unpacking" is applied to just one element of the original array. However, we could also think about "*L*-unpacking", with $L \geq 1$ values 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 is the completion time:

$$T_c \sim m\,T$$

In fact, the communication cost model

$$L_{com}\,(L) = T_{setup} + L\,T_{transm}$$

suggests to build messages whose length is able "to write off" T_{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:

$$T_A = L_{com}\,(L) = T_{setup} + L\,T_{transm}$$

Since

$$T_{calc} = L\,T_F$$

the optimal parallelism degree of *compute* becomes:

$$n = \frac{L\,T_F}{T_{setup} + L\,T_{transm}}$$

Rigorously, to optimize T_c we should refer to a specific parallel paradigm with its own cost model: find the value of *L* which minimizes the function $T_c(L)$, if a minimum exists, e.g. such that $dT_c/dL = 0$.

Alternatively, in order to have an easy and approximate, yet general, technique to find the optimal *L*, we can solve the equation:

$$T_c \sim \frac{M}{L}T = \frac{M}{L}L_{com}(L)$$

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:

$$\frac{M}{L} = a\,\frac{L\,T_F}{T_{setup} + L\,T_{transm}}$$

from which the second order equation in *L*:

$$a T_F L^2 - M T_{transm} L - M T_{setup} = 0$$

The approximation is also due to the fact that the last message has to be "padded" with some fictitious vales for integer rounding reasons. However, it is acceptable.

For example, let us consider a computation with $M = 10^3$, $T_F = 10^3\ \tau$, $T_{setup} = 10^3\ \tau$, $T_{transm} = 10^2\ \tau$. For $L = 1$, no parallel version can be found because *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

$$L \sim 16 \qquad n \sim 6 \qquad T_c \sim 1{,}6 * 10^5\ \tau$$

with a stream length of $m \sim 60$ and a service time $T = L_{com} = 2{,}6 * 10^3\ \tau$.

9. 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 programs and on parallel architectures (Sections 19, 20, 21, 22 on Instruction Level Parallelism, and Part 2).

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

$$R_1 = F_1 (D_1) \; ; \; R_2 = F_2 (D_2)$$

where ";"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*:

$$R_1 = F_1 (D_1) \; || \; R_2 = F_2 (D_2)$$

if all the following conditions hold:

$$\begin{cases} R_1 \cap D_2 = \varnothing \\ R_1 \cap R_2 = \varnothing \\ D_1 \cap R_2 = \varnothing \end{cases}$$

(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).

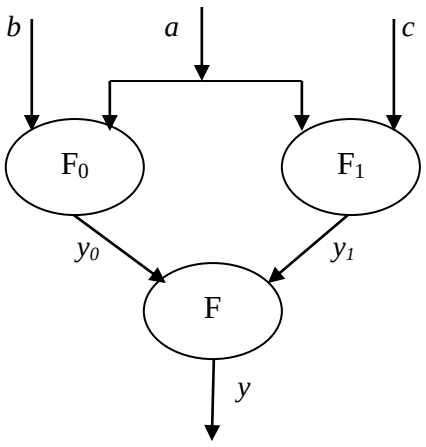
Given a sequential computation, a *partial ordering* of operations is built through the computation-wide application of Bernstein conditions (quadratic complexity procedure). *The precedence relations express the strictly needed data dependences*. The partial ordering is represented by a ***data-flow graph***, which represents the computation graph in this model.

Example

Consider again the module operating on streams (already implemented in pipeline, Sect. 6.1):

$$M :: \quad \forall (a, b, c): \{ y_0 = F_0(a, b); y_1 = F_1(a, c); y = F(y_1, y_2) \}$$

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 $F_0(a, b)$ e $F(y_1, y_2)$.

The cost model is analogous to the pipeline case:

$$T_\Sigma = \max(T_A, \max_{j=0}^{n-1}(T_i))$$

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 streams and on *single data values* too.

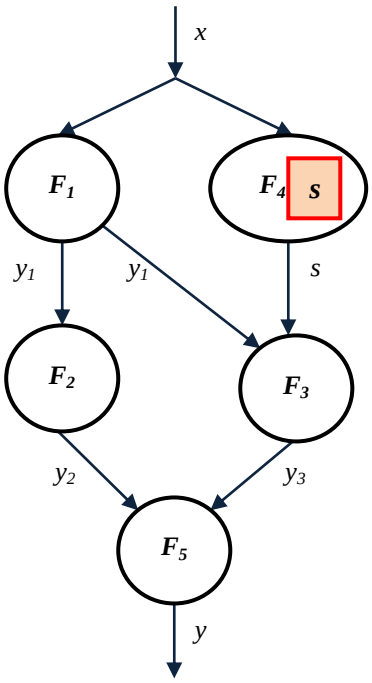
As in the pipeline case, data are partitioned, and computations with state are compatible with this paradigm.

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 :

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

Let all integer functions have mean calculation time equal to *t*. Let $L_{com} = t/10$, and assume that communication processors are provided. Let the interarrival time be $T_A = t/2$.



We have:

$$T_{\text{calc}} = 5\,t$$

$$T_{\Sigma-id} = T_A = \frac{t}{2}$$

The optimal parallelism degree is:

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

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 e 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_3).

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 metrics are:

$$T_{\Sigma}^{(n)} = \begin{cases} \dfrac{5t}{n} & for\ n < 5 \\ t & for\ n \geq 5 \end{cases}$$

$$\varepsilon_{\Sigma}^{(n)} = \begin{cases} \dfrac{n}{10} & for\ n < 5 \\ \dfrac{1}{2} & per\ n \geq 5 \end{cases}$$

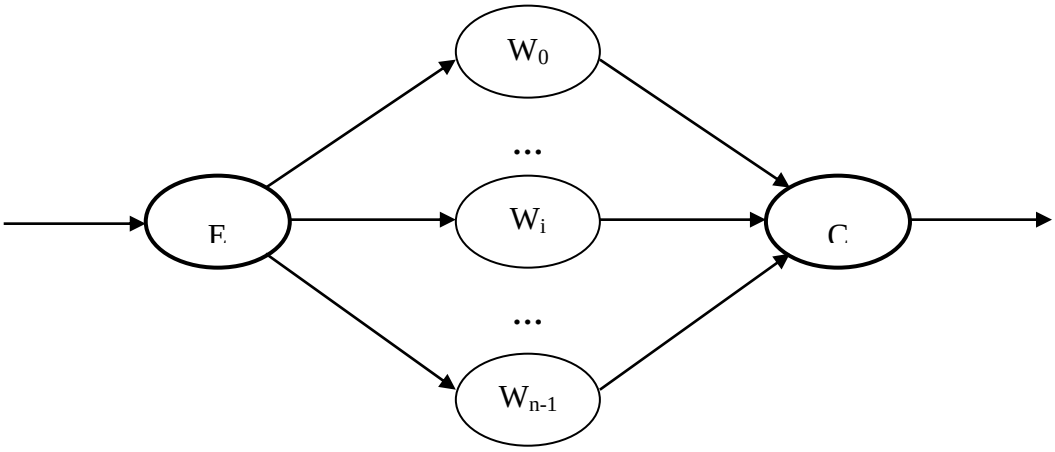
$\varepsilon_{\mathrm{M}}^{(n)} = 1$ for $n \leq 5$, since M remains a *bottleneck* (however, the case $n > 5$ is not significant for this data-flow computation, since the data-flow graph cannot contain more than five nodes).

10. 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 , …, W_n,
- *emitter* (E): a scheduling functionality of input stream values to be distributed to the workers,
- *collector* (C): a collection functionality of worker 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.

In this Section 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 guarantee 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 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 values. 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 ("the user"). 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:

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

where

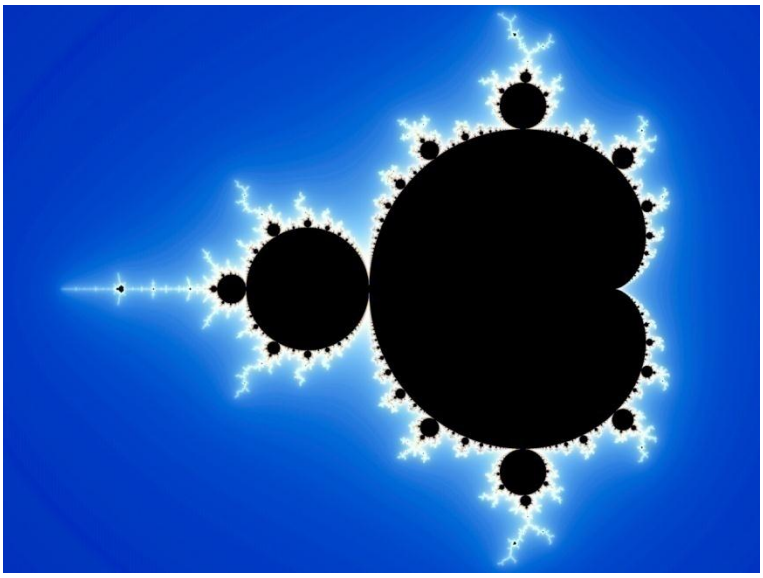
$$M_o (c) = 0$$

$$M_{k+1} (c) = M_k (c)^2 + c$$

The following property holds :

$$\exists k : | M_k(c) | > 2 \Rightarrow c \notin \mathrm{M}$$

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.

10.1 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

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

If the underlying architecture supports communication overlapping, then

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

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:

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

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 Σ.

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

The optimal number of workers is given by the general theory (Sect. 4.4):

$$n_{opt} = \frac{T_{calc}}{T_A}$$

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

$$T_A \geq L_{com}$$

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:

$$T_A < L_{com}$$

Though very unusual, this case cannot be excluded in principle, e.g. the input stream is generated by multiple clients, all having interdeparture time equal to L_{com}. However, in this case the emitter is a bottleneck. In conclusion, the emitter interdeparture time is:

$$T_{p_E} = \max(T_A, L_{com})$$

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:

$$T_{A_i} = n \ \max(T_A, L_{com}) \quad \forall i = 1 \ldots n$$

$$T_{p_i} = \max(T_w, n \ \max(T_A, L_{com})) \quad \forall i = 1..n$$

The collector interarrival rate is given by:

$$\frac{1}{T_{AC}} = \sum_{i=1}^{n} \frac{1}{T_{p_i}} = \min(\frac{n}{T_w}, \frac{n}{n \max(T_A, L_{com})}) = \min(\frac{n}{T_w}, \frac{1}{\max(T_A, L_{com})})$$

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

$$B_\Sigma = \min(\frac{n}{T_w}, \frac{1}{\max(T_A, L_{com})})$$

Thus, the farm service time is:

$$T_\Sigma = \max(\frac{T_w}{n}, \max(T_A, L_{com}))$$

The best number of workers as possible is the *n* value that maximizes the bandwidth:

$$\bar{n} = \frac{Tw}{\max(T_A, L_{com})}$$

We have verified the general theorem (Sect. 4.4) 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:

$$n_{opt} = \frac{T_{calc}}{T_A}$$

where $T_w = T_{calc}$ since, *with the optimal parallelism degree, the worker communications are fully overlapped to the internal calculation time.*

Thus

$$T^{(n_{opt})} = T_A$$

If the ratio T_{calc}/T_A is not integer, we evaluate n_{opt} as

$$n_{opt} = \left\lceil \frac{T_{calc}}{T_A} \right\rceil$$

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

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

if the goal is to maximize the efficiency.

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

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

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:

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

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:

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

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: $T_{setup} = 10^3\,\tau$ and $T_{transm} = 10^2\,\tau$, 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:

$$n = \frac{T_{calc}}{T_A} = \frac{N\,T_F}{T_A} = \frac{5 * 10^6}{5 *\ 10^5} = 10$$

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:

$$T_E = L_{com}(2N) = T_{send}(2N) = T_{setup} + 2N\,T_{trasm} = 10^3\tau + 2\,10^3\,10^2\tau \sim 2\,10^5\ \tau$$

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:

$$n_\Sigma = n + 2 = 12$$

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

We are able to actually achieve the maximum bandwidth and efficiency:

$$T_\Sigma^{(10)} = T_{\Sigma-id}^{(10)} = T_A = 5 \times 10^5\,\tau$$

$$\varepsilon_\Sigma^{(10)} = 1$$

The completion time is:

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

The latency:

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

Example 2

Let us parallelize the following sequential program:

```
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 values (even better: each stream element contains multiple values, see Sect. 8).

Express this parallel program in LC, and evaluate it.

10.2 Farm *vs* pipeline and data-flow

Given a function to be parallelized, let us compare a farm implementation to a pipeline and to 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 $\varepsilon_\Sigma = 0{,}25$.

A *farm* solution with parallelism degree

$$n_W = \frac{T_{calc}}{T_A} = 16 \qquad n = n_W + 2 = 18$$

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 the pipeline structure has higher latency.

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

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 parallelism degree is lower or equal, and the same service time is achieved.

10.3 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: $S = F_1(x, S)$,
- a pure function: $y = F_2(x, S)$.

The latter is implemented according to the usual farm structure.

The former must be executed by *all* workers, in order to maintain all 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 $(op = F_1)$.

Though the sequential module service time is equal to $pT_{F1} + (1 - p)\, T_{F2}$, the service time of the parallel farm version is *not*:

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

The correct formula of the farm service time is:

$$T_S = p\, T_{F_1} + (1-p)\, \frac{T_{F_2}}{n}$$

because parallelism is exploited with *1- p* probability only.

The optimal degree of parallelism is found by imposing $T_S = T_A$:

$$n(p) = \left\lceil \frac{(1-p)\, T_{F_2}}{T_A - p\, T_{F_1}} \right\rceil$$

In order to have a meaningful farm, it must be

$$p < \frac{T_A}{T_{F_1}}$$

As usually, the optimal degree of parallelism can be exploited provided that emitter is not a bottleneck: in this case, the *multicast service time* might be critical. This issue will be discussed in Section 12.1, where the multicast communication will be delat with.

For $p = 0$ we are in the usual stateless farm situation:

$$n(0) = \left\lceil \frac{T_{F_2}}{T_A} \right\rceil$$

11. Functional partitioning with independent workers

Consider a sequential module operating on streams defined as follows:

```
M ::    channel in input_stream (…); channel out output_stream; int op; …
        while (true) do
                { receive (input_stream, (op, x);
                 y = case op of
                        1 : F1(x)
                        …
                        k :  Fk (x):
                 send (output_stream, y)
                }
```

Functions $F_1, \ldots, F_k$ have processing times $T_1, \ldots, T_k$ and occurrence probabilities $p_1, .., p_k$. 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. 4.2.2). Remember that pipeline and data-flow are functional partitioning paradigms too, however they are based on interacting 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*. If T_A is the interarrival time, then

$$T_{A_i} = \frac{T_A}{p_i}$$

Emitter and collector have ideal service time equal to L_{com}, thus they are not bottlenecks in all the computations of interest. If the following condition holds:

$$\forall i : \rho_i \leq 1 \qquad \Rightarrow \qquad \forall i : p_i \cdot T_i \leq T_A$$

then no bottleneck exists, and 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 a bottleneck exists (Sect. 4.2.5), then the service time 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. 10.2): 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 scheme.

12. Collective communications

Besides symmetric and input-asymmetric, deterministic and non-deterministic communications (Part 0, Section 6), 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 6.3 and 10.3 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 distinct partitions of A 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 (*cyclic* distribution), analogous to an interleaved memory (Part 0, Section 2.5);

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, as it happens in some limited degree interconnection networks or in some standard protocols (IPv6).

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.

12.1 Multicast

The best implementation has service time and latency:

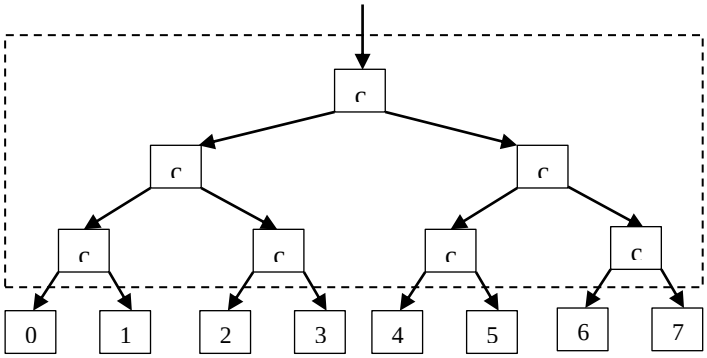
$$T_{multicast} = L_{multicast} = T_{send}(k)$$

on condition that the run-time support exploits n one-to-one communications *in parallel*. This is possible only if the underlying architecture supports this facility.

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

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

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 computation with *n* workers (identified by *0, ..., n-1*): a binary tree of additional (*n – 1)* communication modules (c) implements the multicast communication, where each module executes two point-to-point communications sequentially:



The latency is given by:

$$L_{multicast} = 2 \lceil lg_2(n) \rceil T_{send}(k)$$

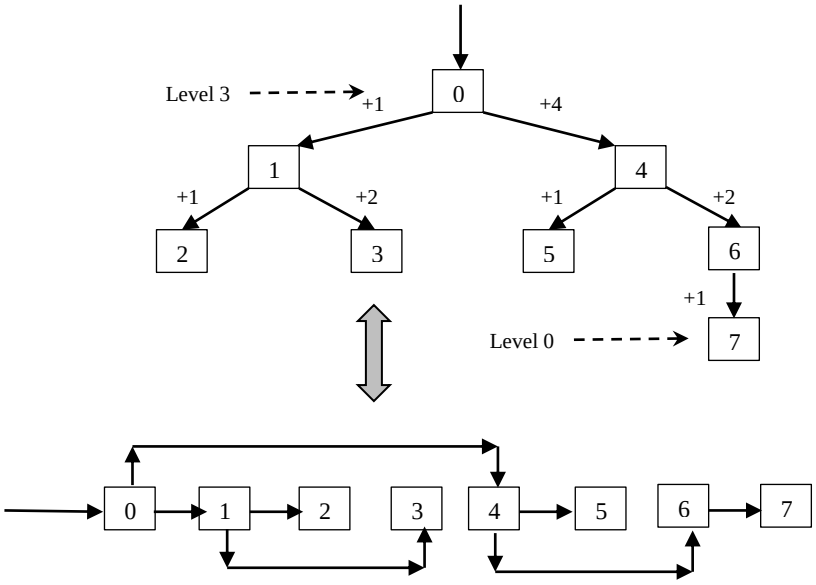
while the service time is just:

$$T_{multicast} = 2\, T_{send}(k)$$

owing to the *pipeline and data-flow effect* through the tree structure. As in similar cases, a logarithmic latency is the best result when a constant latency is not possible or too complex to achieve. The communication banwidth is just two times greater than the minimum one.

The counterpart of this solution is that the number of modules, thus of processing nodes, doubles approximately, and the tree nodes are merely used as communicators.

However, a logarithmic latency multicast can also be implemented *without additional modules/nodes*. In this case, the tree structure is implemented directly by the set of workers themselves according to a distributed strategy. A *tree can always be mapped onto a cube*, and in particular *onto a linear array*, as shown in the following figure for a binary tree:



For the linear array of modules, the rule for deriving the topology of communication channels applies the *depth-first strategy*, and is the following:

- The tree to be mapped onto the linear array of *n* nodes *0, ..., n-1* is composed of *1+$lg_2 n$* levels *0, ..., $lg_2 n$*. Let node 0 be the root at level *$lg_2 n$*. Each node *j* at level *i* ($0 < i \leq lg_2 n$) sends the received message to nodes *j + 1* and *$j + 2^{i-1}$* (for *i = 1* only one message is sent).

Equivalent solutions apply different tree visit strategies.

This multicast implementation saves half the nodes (*n* instead of *2n-1*). However, we have to evaluate the impact on the processing bandwidth of the computation implemented by the set of workers, as shown in the following where the three multicast solutions are evaluated and compared.

An elegant solution is to exploit the ***communication processor***, if present (Section 3.3.2). A worker delagates the sequential execution of two send primitives to the associated KP. *From a conceptual point of view, this solution uses* n *additional modules/nodes. However, from a technological point of view*, as discussed in Section 3.3.2, *the additional cost is not paid if an on-chip KPco-processor is provided.*

Impact of multicast communication in stream-based computations

Let us evaluate the possible impact of the multicast communication on the processing bandwidth of a stream-based computation with *n* workers and interarrival time T_A.

a) With ***linear-latency*** solution implemented in a **centralized** distributor module D, the computation achieves the optimal bandwidth $1/T_A$ if D is not a bottleneck, that is

$$T_D = T_{multicast} = n\,T_{send} < T_A$$

with

$$n = n_{opt} = \frac{T_{calc}}{T_A}$$

In other words, if D is not a bottleneck, the computation bandwidth is not affected by the multicast centralized communication. The possible disadvantage of this solution is the latency, when this parameter is of interest (notably, in a client-server computation, see Sections 15.2, 15:3)

b) In a ***tree-structured implementation***, where D is a binary tree implemented by ***additional n-1 modules***, the optimal bandwidth $1/T_A$ is achieved if:

$$T_D = T_{multicast} = 2\,T_{send} < T_A$$

with

$$n = n_{opt} = \frac{T_{calc}}{T_A}$$

In general, a logarithmic latency is satisfactory.

c) In a ***tree-structured implementation mapped onto the workers*** (D is fully distributed on the set of workers), with the same latency of solution *b)*, the *communication overhead* in the worker activities is a source of degradation.

If the ***communication processor*** is provided, the overhead is eliminated, because the multicast communication is fully independent from the calculation, and the communication latency *2 T_{send}* is fully masked. Thus, the same performance of solution *b)* is achieved in the most technologically advanced way.

For completeness, let us study the case in which *no communication processor* is provided. Now, each worker, before being able to start the calculation, must spend $T_{multicast}$ as the contribution to the distributed multicast implementation. If

$$2\,T_{send} < T_A$$

and

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

the optimal bandwidth is not achieved, because the service time becomes:

$$T_s = T_A + T_{multicast} = T_A + 2\,T_{send}$$

This degradation is acceptable only if

$$T_A \gg 2\,T_{send}$$

a typical situation in which the communication processor is not necessary.

In general, the optimal bandwidth *1/T_A* can be achieved provided that the *parallelism degree is properly increased*:

$$n'_{opt} = \frac{T_{calc}}{T_A - 2\,T_{send}}$$

Let us compare solutions *b)* and *c) without KP*. Mapping the tree structure onto the workers actually saves processing nodes, without a bandwidth degradation, if:

$$n'_{opt} < 2\,n = 2\,\frac{T_{calc}}{T_A}$$

thus if

$$T_A > 4\,T_{send}$$

In conclusion, if

$$2\,T_{send} < T_A < 4\,T_{send}$$

solution *b)* gives the best results. Otherwise solution *c)* is to be preferred with n'_{opt} workers, or, in practice, with n_{opt} workers only if $T_A \gg 2\,T_{send}$.

Example

Let us consider a *farm with internal state* (Section 10.3) with the following characteristics:

- input stream elements are integer arrays *X[M]*, with $M = 10^3$,
- service time of functions F_1 (state modification), F_2 (output generation) are given by $T_{F1} = 7\ 10^6\tau$ and $T_{F2} = 10^7\tau$,
- probability of executing F_1: $p = 0.1$,
- $T_{setup} = 10^3\tau$, $T_{transm} = 10^2\tau$,
- interarrival time $T_A = 10^6\tau$.

The necessary condition $p < T_A / T_{F1}$ is satisfied. The optimal degree of parallelism

$$n = \left\lceil \frac{(1-p)\, T_{F_2}}{T_A - p\, T_{F_1}} \right\rceil = 30$$

can be actually exploited provided that emitter is not a bottleneck. The emitter service time is given by:

$$T_E = p\, T_{multicast}\,(M,n) + (1-p)\, T_{send}\,(M)$$

where:

$$T_{send}\,(M) = T_{setup} + M\, T_{transm} \sim 10^5 \tau$$

With a *linear centralized multicast*:

$$T_{multicast}\,(n,M) = n\, T_{send}\,(M) \sim 3\ 10^6 \tau$$

$$T_E = 3.9\ 10^5 \tau < T_A$$

Therefore, no bottleneck exists: provided that are sufficient processing nodes, the optimal parallelism degree can be exploited, and the effective service time of the farm is equal to T_A.

Let us now assume that $T_{F2} = 10^6 \tau$, thus $n = 120$. With a linear multicast ($T_{multicast} = 1.2\ 10^6$) emitter is a bottleneck: $T_E \sim 1.3\ 10^6$. A *tree-structured* multicast eliminates the emitter bottleneck:

$$T_{multicast}\,(n,M) = 2\, T_{send}\,(M) \sim 2\ 10^5 \tau$$

$$T_E = 1.1\ 10^5 \tau < T_A$$

The best solution is *c)* with the communication processor.

If the communication processor is not provided, since $T_A > 4\, T_{send}$ solution *c)* utilizes fewer nodes than solution *b)*. Because function F_1 *i*s executed with low probability, in practice the optimal bandwidth is achieved without increasing the parallelism degree.

12.2 Scatter

In the simplest case the implementation is sequential, i.e. *n* sequential point-to-point communications to distinct modules, with message size $g = k/n$, executed by a *centralized* distributor. The latency and the service time of the distributor are *linear* in *n*:

$$T_{scatter} = L_{scatter} = n\, T_{send}(g) = n\, T_{setup} + k\, Tt_{rasm}$$

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

$$L_{scatter} = 2 \sum_{i=1}^{\lceil \lg_2 n \rceil} T_{send}\left(\frac{k}{2^i}\right)$$

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

$$T_{scatter} = 2\, T_{send}\,(k/2) = 2\, T_{setup} + k\, Tt_{rasm}$$

which is a *modest improvement* compared to the centralized implementation. In practice, the tree-structured solution is adopted only when the latency has a strong impact on the global performance.

Analogously to the multicast implementation, the tree structure (if used to reduce the latency) can be a separate tree or mapped onto the workers themselves. However now, if the communication processor is not provided, the latter has a more serious impact on the service time degradation of a stream-based computation.

12.3 Gather

At first sight, the gather implementation could be independent of *n*, since the source nodes (workers) send the respective partitions in parallel. However, *in general* the structure re-composition in the destination module (node) could represent a *bottleneck*, at the process or at the firmware level. In other words, even with zero-copy point-to-point communications, the *receive* latency could have a strong impact on the gather cost model.

In such cases, a sufficiently approximated evaluation of the gather latency is:

$$L_{gather} \sim L_{scatter}$$

An important optimization is possible about the service time in some paradigms, notably in *stream-based* computations or in *scatter-compute-gather* data-parallel computations (see Section 13.2), in which a *pipeline effect* can be exploited: the *send* primitives executed by the workers, except the last, are overlapped. In this case:

$$T_{scatter} \sim T_{send}(k)$$

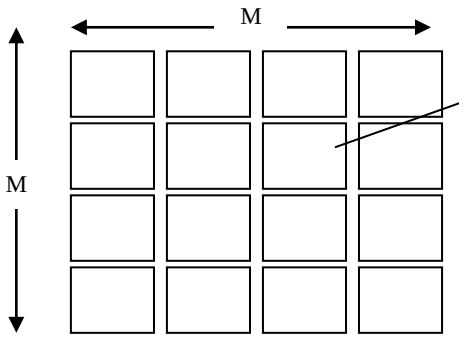
13. Data parallel paradigms

A data parallel computation, on streams and/or on single data values, is characterized by partitioning of (large) data structures and by 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 pure stream-parallel paradigms, though a potential load unbalance could exist.

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

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



Square **block** partition of $g = M^2/n$ elements encapsulated in a worker module. Other possible distributions are **by rows**, **by columns** or **cyclic** (see collective communications).

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 (see Section 13.5, 13.6).

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 with the other workers 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, the stencil may be *fixed* throughout all the computation steps, 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 Section 13.8.

13.1 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 collects 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 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 values. However:

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

b) when a fully automatic solution 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 the sequential computation, we 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 processor version is (much) higher than the actually optimal one. However, the goal of the virtual processor 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 minimum one, but this is not necessarily true in all cases (however, in general the virtual processor grain is (much) smaller than the actual one);

2) we 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 a computation that is suitable for the *map* paradigm. Consider the following sequential loop computation:

int A[M], B[M];

$\forall$ *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:

B[0] = F(A[0);

B[1] = F(A[1]);

…

B[M-1] = F(A[M-1])

or, more concisely:

B[i] = F(A[i]);

B[j] = F(A[j])

for any *i, j* with $i \neq j$.

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:

```
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 processor 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 *maximum parallelism* (*M)* implies the *minimum 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 processor 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];
∀ i = 0 .. M – 1:
        { C[i] = 0;
          ∀ j = 0 .. M–1:
                C[i] = C[i] + A[i, j] ∗ B[j] }
```

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:

```
{ C[0] = 0;
  ∀ j = 0 .. M–1:   C[0] = C[0] + A[0, j] ∗ B[j] }
{ C[1] = 0;
  ∀ j = 0 .. M–1:   C[1] = C[1] + A[1, j] ∗ B[j] }
…
{ C[M-1] = 0;
  ∀ j = 0 .. M–1:   C[M-1] = C[M-1] + A[M-1, 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, <u>*provided that VP[i]*</u> contains

- the *single element C[i]*,
- *all M elements of the i-th row of A*: *A[i, ∗]*
- *all M elements of B*: *B[∗]*.

That is, a map computation is defined by *replicating B* among all VPs:

```
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++)
                C[i] = C[i] + A[i][j] ∗ B[j]
```

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

We have expressed the *maximum parallelism* of the computation (*M*). The *data grain*, though *not the minimum 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, a *VP[M][M]* version has a completion time not better 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) is part of the Virtual Processors 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 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).

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

∀ *j = 0 .. M–1: C[1] = C[1] + A[1, j] ∗ B[j]* }

…

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

∀ *j = 0 .. M–1: C[M-1] = C[M-1] + A[M-1, j] ∗ B[j]* }

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 *executable in parallel*, the *M* virtual processors are not fully independent. That is, the generic *VP[i]* performs an execution sequence like

…

{ *C[i] = 0;*

C[i] = C[i] + A[i, 0] ∗ B[0];

C[i] = C[i] + A[i, 1] ∗ B[1];

…

C[i] = C[i] + A[i, j] ∗ B[j];

…

C[i] = C[i] + A[i, M-1] ∗ B[M-1];

}

…

At the beginning of the *j-th* iteration, each VP needs the value of *B[j]*, which is *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 *static and variable* stencil:

```
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] ∗ x }
```

where *multicast* has to be expanded in the actual version.

To transform the virtual processor 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), of C (*n* linear blocks of *g* elements), and of B (*n* linear blocks of *g* elements), and the *gather* collection of C (*n* linear blocks of *g* elements).

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

- the *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 *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 partial result. Though all VPs operate on different B elements at each step, this algorithm works correctly owing to the *associative* property of addition.

The virtual processors description is:

```
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]);

                          C[i] = C[i] + A[i][j] * B[i] }
```

The actual parametric version is left as an exercize.

Owner computes rule

The previous example allows us to introduce the so-called "*owner computes rule*", which is usually applied in *stencil-based* computations.

The owner computes rule is based on the concept of *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 define 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]*, *all 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 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.

13.2 Map cost model

Each worker has a calculation time

$$T_{map\text{-}calc} = g\ T_F$$

which is the calculation time of the whole workers set. The effective service time must include the impact of data distribution and collection. We are assuming to operate on *single data structures*.

Initially, let us study the worst-case analysis, in which no parallelism is exploited among the scatter-compute-gather phases: The *completion time* is given by:

$$T_c = T_{scatter}(n, g) + T_{map\text{-}calc} + T_{gather}(n, g) \sim 2\ T_{scatter}(n, g) + T_{map\text{-}calc}$$

Let us consider a map without replicated data, and let *k* the size of the partitioned data structure:

$$T_c = 2n(T_{setup} + gT_{trasm}) + gT_F = 2nT_{setup} + 2kT_{trasm} + T_F\frac{k}{n}$$

The optimal parallelism degree can be determined by imposing the first-degree derivative equal to zero:

$$\frac{dT_c}{dn} = 2T_{setup} - T_F\frac{k}{n^2} = 0$$

We can verify that the minimum exists. Rounding up the result:

$$n_{opt} = \left\lceil \sqrt{\frac{k\,T_F}{2T_{setup}}} \right\rceil$$

As said in Section 12.3, 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:

$$T_c = T_{scatter}(n, g) + g\ T_F + T_{send}(g) = (n + 1)\ T_{send}(g) + g\ T_F$$

from which:

$$n_{opt} = \left\lceil \sqrt{\frac{k\,(T_F + T_{trasm})}{T_{setup}}} \right\rceil$$

For coarse grain calculations and/or data, n_{opt} might assume quite large values, often much larger than the number of processing nodes. 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 in a more cost-effective way.

Example

Consider again Example 1 of previous Section:

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

where M = 256K, T_F = 1.000 τ, and $T_{send}\,(L) = T_{setup} + L\ T_{trasm}$ = 1000 τ + 100 L τ.

Applying the second formula for n_{opt}:

$$n_{opt} = 387 \qquad T_{c\text{-}min} = 28{,}3x10^6\ \tau$$

For n in the range (200 – 1000), T_c varies very slowly around the minimum (28,8x10^6 τ - 28,4x10^6 τ). 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(Section 7). 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 will see that often the data parallel implementation might require a higher parallelism degree due to the implementation of scatter and gather.

13.3 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:

```
int A[M][M], old_ A[M][M]; init old_A = A;
repeat
    ∀ i = 0 .. M – 1:
        ∀ 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 Section 13.5).

The virtual processors parallel version consists of a bidimensional array of M^2 VPs, *VP[M, M]*, where the generic *VP[i, j]* encapsulates *A[i, j]* and *old_A[i, j]*.

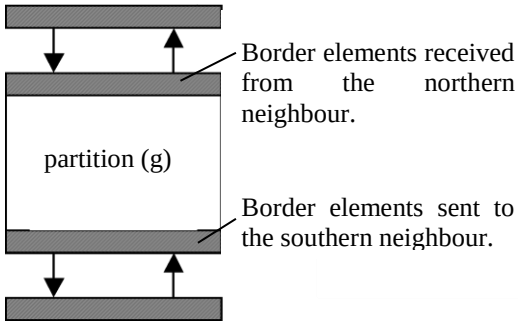
The application of the *owner computes rule* implies that *VP[i, j]* is the only VP authorized to modify *A[i, j]* and *old_A[i, j]*. To implement the 4-neighbours stencil, at each step *VP[i, j]* must receive *old_A[i – 1, j]* , *old_A[i + 1, j]*, *old_A[i, j – 1]* , and *old_A[i, j + 1]* from *VP[i – 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 neighbous 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 guarantee deadlock avoidance.



Assuming zero-copy communications, the completion time is given by:

$$T_c = s\ [\ 4\ T_{send}(\sqrt{g}) + g\ T_F + T_{reduce}(n)\ ]$$

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:

$$T_c = s\ [\ 2\ T_{send}(M) + g\ T_F + T_{reduce}(n)\ ]$$

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)

consists of 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:

$$T_c = s\ [\ 2\ T_{send}(1) + g\ T_F + T_{reduce}(n)\ ]$$

The detailed implementation of the various versions of this benchmark is left as an exercise.

13.4 A benchmark for static, variable stencils

Let us consider the following sequential computation:

int A[M, M]];

$\forall\ i = 0 .. M - 1:$

$\forall\ j = 0 .. M–1:$

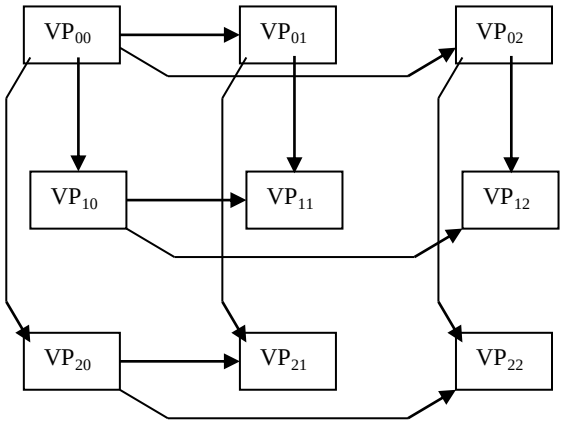
$\forall\ h = 0 .. M–1:$

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 exemplifies 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:

$$T_c = T_{scatter}(n, g) + M\,(\,T_{multicast}(g) + g\,T_F\,) + T_{gather}(n, g)$$

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} parameters (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.

13.5 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 ⊗, returns the following scalar value

$$x = reduce\ (A, \otimes) = A[0] \otimes A[1] \otimes \ldots \otimes A[M\text{-}1]$$

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:

int A[M]; …

while (∃ i ∈ 0 .. M – 1 : G (A[i]) < 0) do …

Let a boolean array *B[M]* defined in this way:

$$\forall\ i = 0\ ..\ M-1 : B[i] = if\ G(A[i]) \geq 0$$

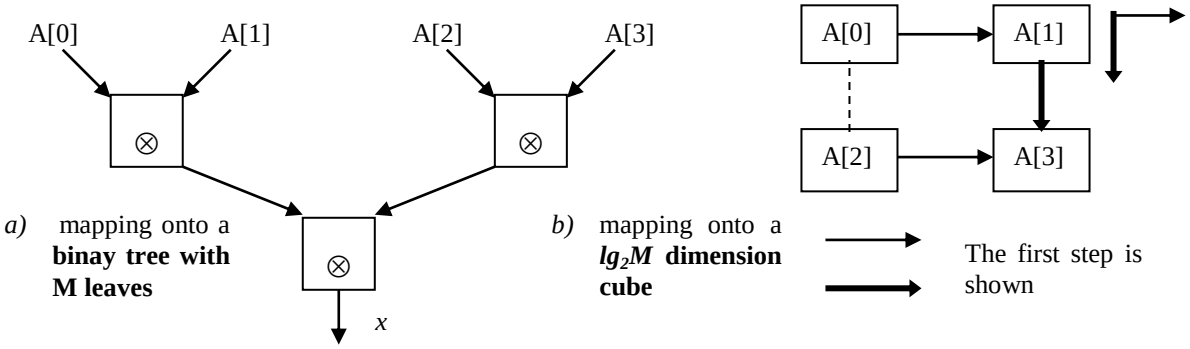
The computation can be expressed as:

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(M)*. The data parallel form is *static, variable stencil*.

Owing to the associativity of operator ⊗, 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_2M 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_2M dimension *binary cube*. This property is exploited in order to implement the virtual processors computation with *M* VPs only, as shown in figure b):

- at the *1st* step, VP[1] computes A[1] = A[0] ⊗ A[1] on its own state A[1] and on A[0] which is received along the *1st dimension* from VP[0]; in parallel, VP[3] computes A[3] = A[2] ⊗ A[3] on its own state A[3] and on A[2] which is received along the *1st dimension* from VP[2];
- at the *2nd* step, VP[3] computes A[3] = A[1] ⊗ A[3] on its own state A[3] and on A[1] which is received along the *2nd dimension* from VP[1].

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):

$\forall j = 1\ \ldots\ lg_2\ M$:

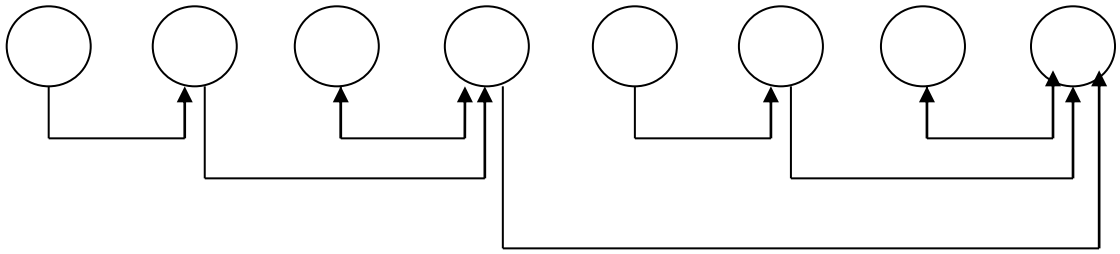
$\forall i = 1\ \ldots\ M$ ***in parallel***:

$if\ ((i+1)\ mod\ 2^j = 0)\ then\ A[i] = A[i - 2^{j-1}] \otimes A[i]$

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 owned 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):



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*ed to all VPs (all workers), notably according to a tree structure mapped onto the worker modules (Section 12.1).

13.6 Map-reduce computations

Interesting examples of data-parallel computations are structured as the composition of *map* and *reduce* paradigms. A notable example is represented by the Google Search Engine: in a query, an input object is searched in a (very large) distributed knowledge base according to the *map* paradigm, then a ranking and ordering algorithm is executed according to the *reduce* paradigm; the properly encoded result is the web page presented to the user.

Let us evaluate a map-reduce computation in which (as in the Search Engine example) an array *A[M]* is already partitioned, and an input value *x* is multicasted. The result of the computation is the value:

$$y = reduce\ (map\ (A, x, F), G)$$

where F is the search function of *x* in *A*, and G is the ranking and ordering function operating on the F results.

The student is invited

a) to solve this problem, finding the optimal parallelism degree,

b) to extend this example to the case in which the input parameter is a single value *(A, x)*,

c) to extend this example to the case in which there is a stream of values *x* or a stream of values (*A, x).* See Section 14.8 for stream data-parallel computations.

Another example of map-reduce computation

Let us consider the following sequential computation, where *F* and *G* are integer functions:

int A[M];

while ($\exists i \in 0 .. M-1 : G(A[i]) < 0$) *do*

$\forall j = 0 .. M-1:$

$A[j] = F(A[j])$

The data parallel solution is a *map + reduce composition*.

As seen, let a boolean array *B[M]* defined in this way:

$\forall i = 0 .. M-1 : B[i] = if\ G(A[i]) \geq 0$

The computation can be expressed as:

while not ***reduce (B, and)*** *do* ***A = map (A, F)***

In the virtual processor version, the generic VP[i] encapsulates A[i]. A scheme of VP[i] behavior is:

VP[i] ::

compute $B[i] = if\ G(A[i]) \geq 0$;

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 = true* then terminate, otherwise compute *A[i] = F (Ai])* 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 student is invited to express the completion time T_c and to find the optimal degree of parallelism by imposing:

$$\frac{dT_c}{dn} = 0$$

13.7 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 ⊗:

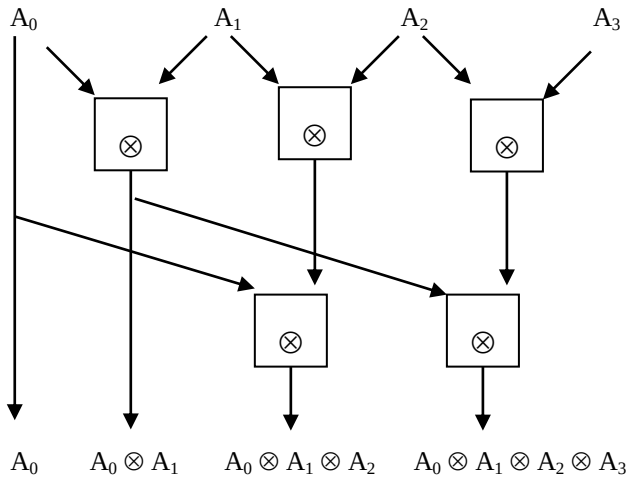
$A_0 = A_0$

$A_1 = A_0 \otimes A_1$

…

$A_{M--1} = A_0 \otimes A_1 \otimes A_2 \otimes \ldots \otimes A_{M-1}$

In parallel, this computation requires $lg_2 M$ steps, as shown in the following figure:



We can apply many considerations of the *reduce* implementation. The virtual processor tree-structured computation can be mapped onto a $lg_2 M$ dimension cube, although not all communications occur between neighbors.

The VP abstract computation is:

$\forall j = 1 \ldots lg_2 M$:

$\forall i = 1 \ldots M$ ***in parallel***:

if $(i \geq 2^j)$ *then* $A[i] = A[i - 2^{j-1}] \otimes A[i]$

The *parallel prefix* completion time is equal to the *parallel reduce* one:

$$T_c = T_{scatter}(n, g) + T_{par\text{-}prefix}(n, g)$$

where:

$$T_{par\text{-}prefix}(n, g) = (g - 1) T_{\otimes} + lg_2 n (T_{send}(1) + T_{\otimes})$$

The *parallel prefix* computation has some interesting features. Unlike *reduce*, at each step some *apparently redundant* operation are executed (for example, $A_1 \otimes A_2$ 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:

int A[M], B[M];

B[0] = A[0];

$\forall$ *i = 0 .. M - 1:*

B[i] = A[i] + B[i – 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:

$$T_c = T_{scatter}\left(n, \frac{M}{n}\right) + T_{parallelprefix}\left(\frac{M}{n}\right) + T_{gather}\left(n, \frac{M}{n}\right)$$

The student is invited to complete this analysis, to find the optimal degree of parallelism, and to evaluate the completion time for a certain parallel architecture.

13.8 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), compute, and collection (gather) or reduce.*

If T_A is the interarrival time, the optimal parallelism degree

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

of the compute 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 compute 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 τ, T_{setup} = 1.000 τ , T_{transm} = 100 τ, T_A = 150.000 τ.

Both a *farm* and a *map* parallelization are potentially feasible solutions, with optimal parallelism degree:

$$n = \left\lceil \frac{M\,T_F}{T_A} \right\rceil = 7$$

In the *farm* solution, the emitter E is not a bottleneck, because:

$$T_A > T_{send}(M) = T_{setup} + M\,T_{transm} = T_E$$

For the *map* solution, we verify that the *scatter* functionality is not a bottleneck:

$$T_{scatter} = n\,T_{setup} + M\,T_{transm} = 109.226\,\tau < T_A$$

just with a linear 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:

$$scatter\,(n_0) = \frac{M\,T_F}{n_0}$$

That is:

$$n_0\,T_{setup} + M\,T_{transm} = \frac{M\,T_F}{n_0}$$

from which we obtain a second degree equation, whose acceptable solution is:

$$n_0 = 63$$

Example 3

Let us consider Example 1 of Sect.10.1:

> 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:
>
> $$\forall\, i = 0\,..\,N-1 : C[i] = F(A[i], B)$$
>
> 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: $T_{setup} = 10^3\,\tau$ and $T_{transm} = 10^2\,\tau$, 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 $500*10^3\ \tau$.

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:

$$T_{scatter} = n\, T_{setup} + N\, T_{trasm} \sim 10^5\ \tau < T_A$$

thus it is not a bottleneck, and so is the *gather* functionality.

A linear multicast has a service time of about 10^6 τ, thus it represent a bottleneck. A *tree-structured multicast* is not a bottleneck, since it has service time:

$$T_{multicast} = 2\, T_{send}(N) \sim 2\ 10^5\ \tau\ <\ T_A$$

In order 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:

$$n_{\Sigma} = n\ + 16 = 26$$

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*. (to be evaluated).

13.9 Pipelining and other structures for data parallel implementations

In Sect. 6.3 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. 6.3:

> 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.

14. 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:

i) parallel paradigms are *parametric* in the parallelism degree;

ii) 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:

$$\sum_i n_i = N - ps$$

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. 4.4, 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 ± 1 in the parallelism degree of some modules.

Example

Let a computation Σ 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 $4*10^5\tau$ and overlapped communications;
- S_1: *farm* with $T_{calc} = 4*10^6\tau$, thus with 10 workers, plus centralized emitter and collector. Consequently, its interdeparture time is equal to $4*10^5\tau$;
- S_2: *map* with $T_{calc} = 6*10^7\tau$, thus with 150 workers. Assume that scatter and gather functionalities are not bottlenecks.

The whole service time is equal to $4*10^5\tau$, 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 $1{,}1*10^6\tau$.
- The number of workers of S_1 is decreased from 10 to 4; its interdeparture time is increased by $1/\alpha$, thus becomes about $1{,}1*10^6\tau$.
- The number of workers of S_2 is decreased from 150 to 52; its interdeparture time is increased by $1/\alpha$, thus becomes about $1{,}1*10^6\tau$.

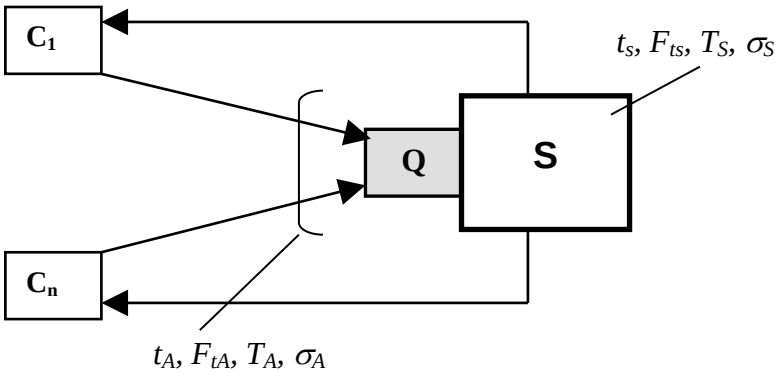
In conclusion, the Σ service time increases from $4*10^5\tau$ to $1{,}1*10^6\tau$ and the relative efficiency is still ~1, because also the ideal bandwidth has been reduced by α.

15. Queuing systems and client-server computations with request-reply behavior

15.1 Analytical treatment of Queuing Systems

In Section 4.1 we have introduced some basic characteristics of Queuing Systems. We saw that for acyclic graphs *Queuing 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 Queuing Systems, in terms of probability distributions of interarrival and service times, is necessary mainly for cyclic graph computations with a request-reply behavior:



This Section is dedicated to this issue. Only the minimal basic results of Queuing 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 metrics of interest in evaluating a queuing system are:

a. **mean length of queue**, L_Q: average number of client requests in the waiting queue;

b. **mean number of requests in the system**, N_Q: with respect to L_Q, it includes the currently served request(s);

c. **mean waiting time in queue**, W_Q: average time spent by a request in the waiting queue;

d. **mean waiting time in the system**, R_Q: with respect to W_Q, it includes the time spent on the currently served request(s). R_Q 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 the interarrival rate.

Moreover, the following relations holds for queueing systems with *a sequential server* and $\rho < 1$:

$$N_Q = L_Q + \rho$$

$$R_Q = W_Q + T_S$$

The former holds since *the average number of requests currently in the service phase is equal to the utilization factor* ρ, because in this kind of computations it is always ρ < 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 *mean latency* of the server. This formula is more general and includes the sequential case too. Notice that the impact of the server service time is confined in the first addend (through ρ), while the impact of the server latency is on the second one. However, only in a limited number of cases the analytical treatment has been extended to parallel servers in order to evaluate W_Q. In conclusion, the above formula has a universal conceptual and qualitative value, it can always be applied in all known cases of sequential servers, and when the parallel server has been studied.

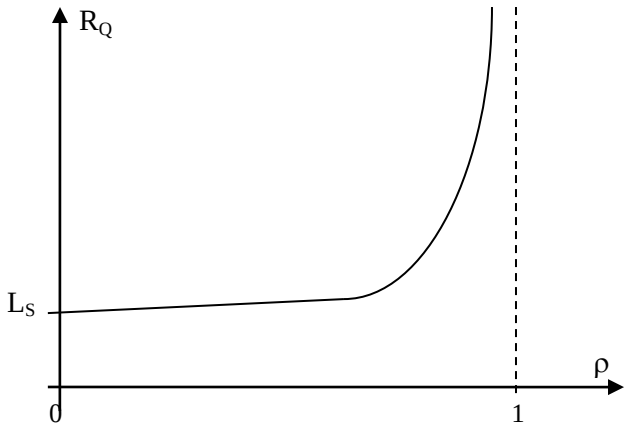
A fundamental concept, to understand queueing systems behavior qualitatively, is the following: *all parameters are monotonically increasing with* ρ and, for ρ < 1 and ρ → 1, they tend asymptotically to infinity, for example:

$$\lim_{\rho \to 0} L_Q = 0$$

$$\lim_{\rho \to 1} L_Q = \infty$$

$$\lim_{\rho \to 0} R_Q = L_S$$

$$\lim_{\rho \to 1} R_Q = \infty$$



That is, *in order to have prompt reply from a server, this should be underutilized. Trying to increasing the utilization degree of a server has a negative effect on the client performance.*

Some notable cases of queuing 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 Queuing Theory.

15.1.1 M/M/1 queue

The basic case is an infinite FIFO queue with negative exponential distributions of interarrival and services:

$$F_{t_A}(t) = \Pr(t_A \le t) = 1 - e^{-\lambda t}$$

$$F_{t_S}(t) = \Pr(t_s \le t) = 1 - e^{-\mu t}$$

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.

15.1.2 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-Khinchnine formula, is valid for *exponential interarrival time distribution* and *general (i.e., any) service time distribution*, M/G/l:

$$N_Q = \left(\frac{\rho}{1-\rho}\right)\left[1 - \frac{\rho}{2}\left(1 - \mu^2\sigma_S\right)\right]$$

Formulas for all the other parameters are derived as seen in the previous Sections:

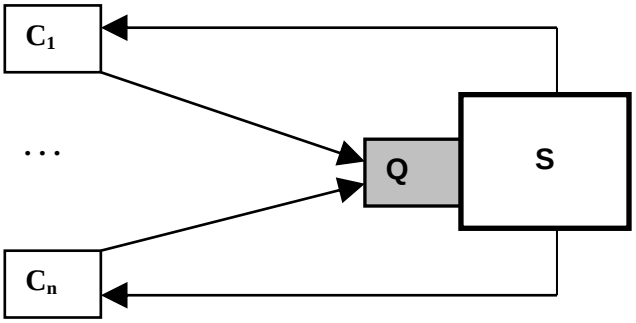
Notice that the N_Q expression $\rho/(1-\rho)$ for the M/M/1 queue is multiplied by a corrective factor taking into account the effective service distribution through its mean $(T_S = 1/\mu)$ and variance σ_S.

A particular case, of special significance for our purposes, is the *M/D/1* queue, with constant service time distribution, for which $\sigma_S = 0$:

$$N_Q = \left(\frac{\rho}{1-\rho}\right)\left(1 - \frac{\rho}{2}\right)$$

15.2 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:

C_i:: inizialize *x* …	S:: /possible initialization of state/;
while (true) **do** {	**while (**true) **do** {
< send *x* to S >;	< receive *x* from C ∈{C_1, …, C_n} >;
< receive *y* from S >;	*y = F (x, ...) ;*
x = G (y, ...)	< send *y* to C >;
}	/possible modification of state/ }

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:

$$\begin{cases} T_{cl} = T_G + R_Q \\ R_Q = W_Q(\rho, T_S, \sigma_S) + Ls \\ \rho = \dfrac{T_S}{T_A} \\ T_A = \dfrac{T_{cl}}{n} \end{cases}$$

whose solution is subject to constraint $\rho < 1$, as discussed in Section 4.1.

By these equations it is easily derived that $N_Q < n$ always, as it is expected.

For the proper W_Q expression, a second, or higher, order equation in ρ is derived by substitution, which admits one and only one real, positive solution satisfying $\rho < 1$.

In the following, we report the functions 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.

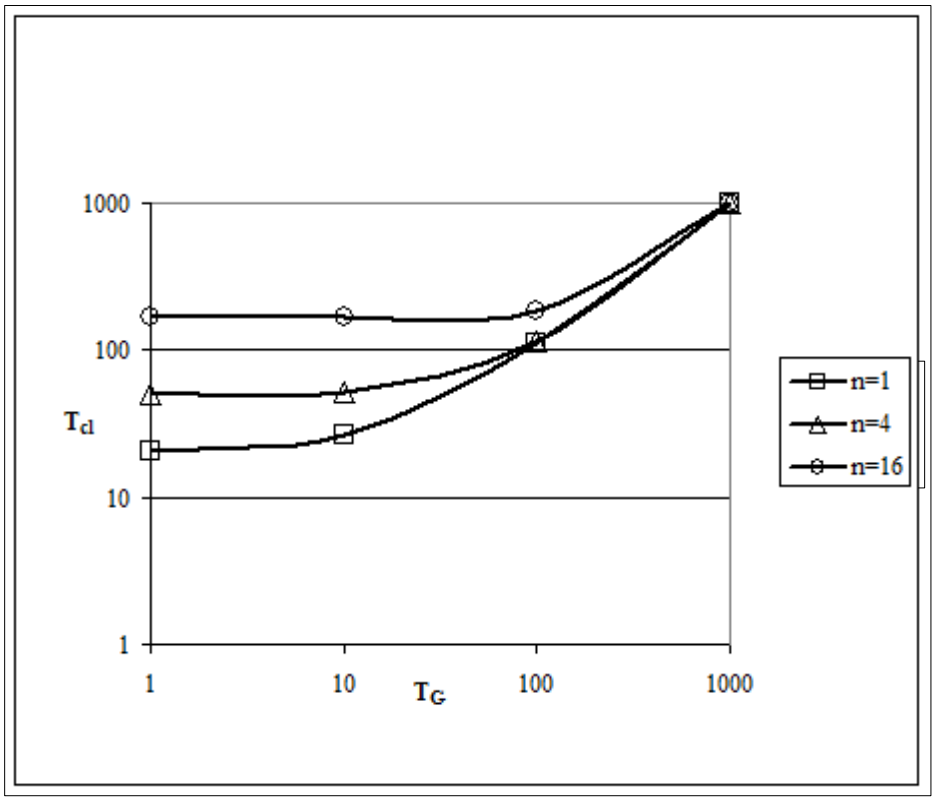


Figure A

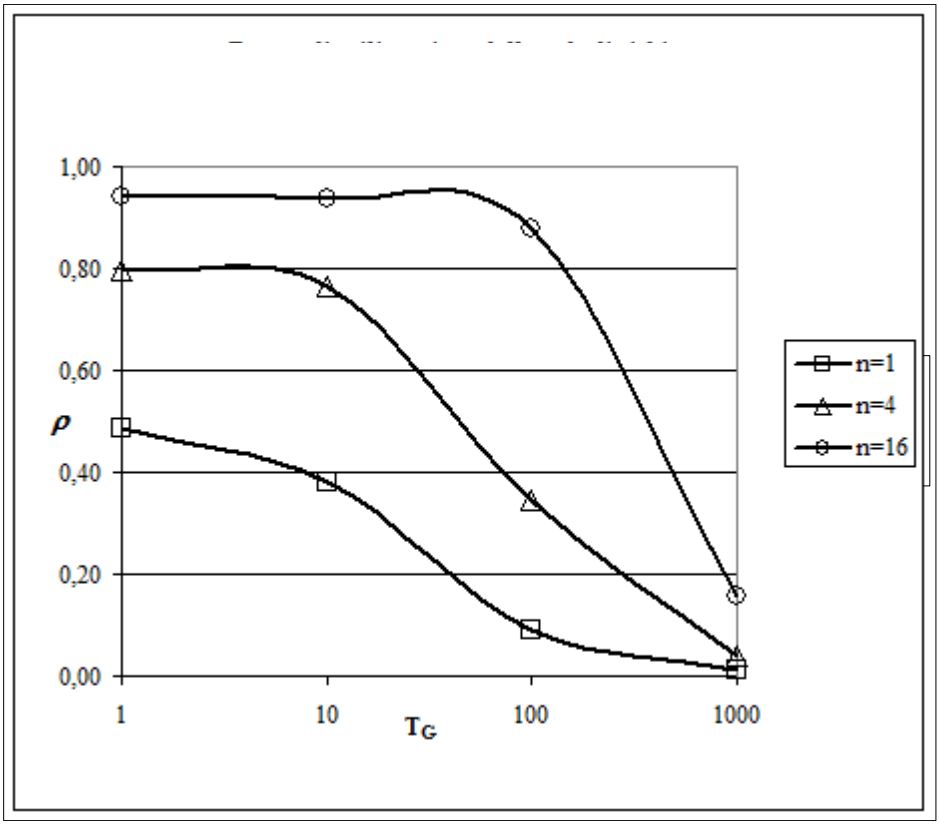


Figure B

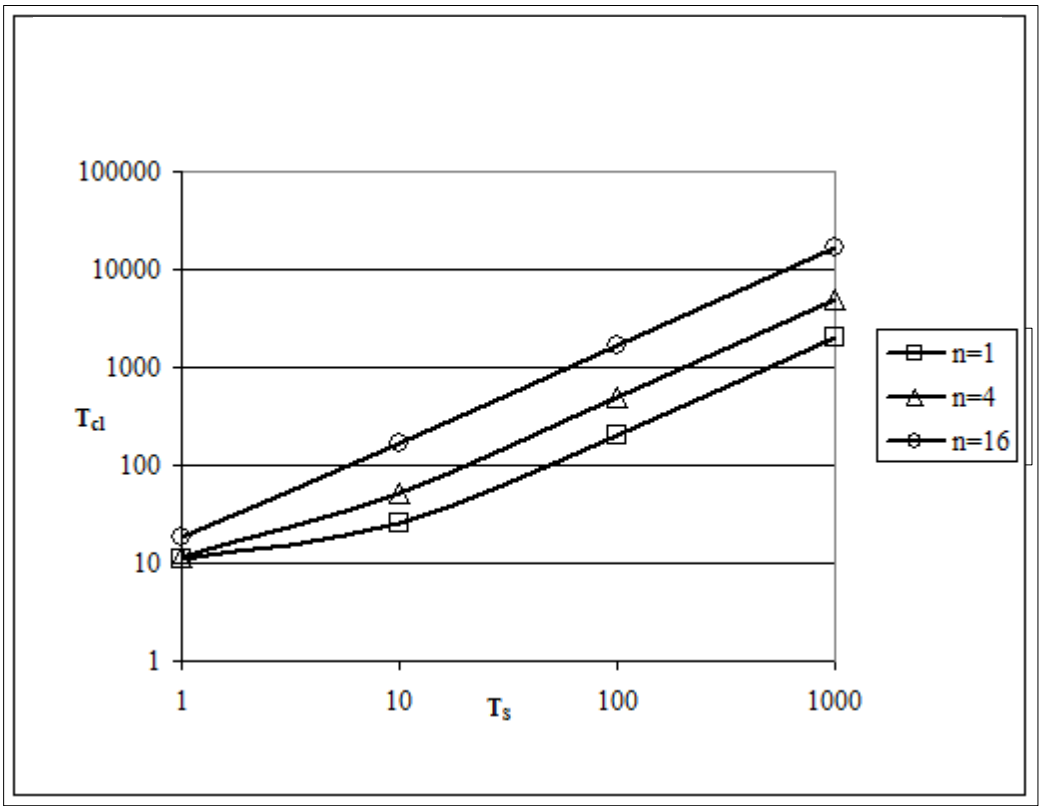


Figure C

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 ρ decreases, and R_Q decreases. If T_S increases, ρ increases, thus R_Q increases. If n increases, then T_A decreases, ρ 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, ρ increases, R_Q increases, thus $R_Q/T_G \to \infty$, so $\varepsilon \to 0$; if the client ideal service time T_G increases, ρ 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)

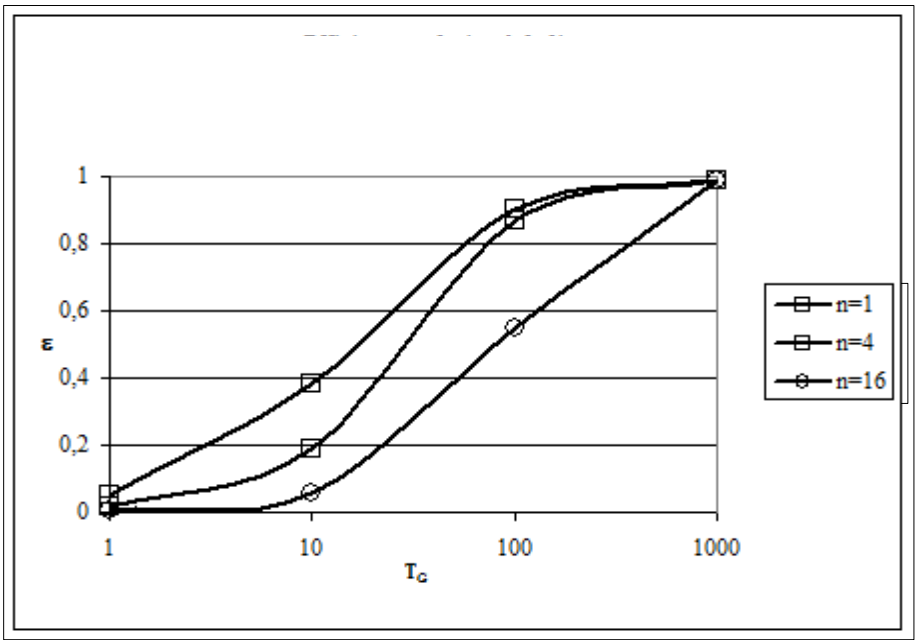


Figure D

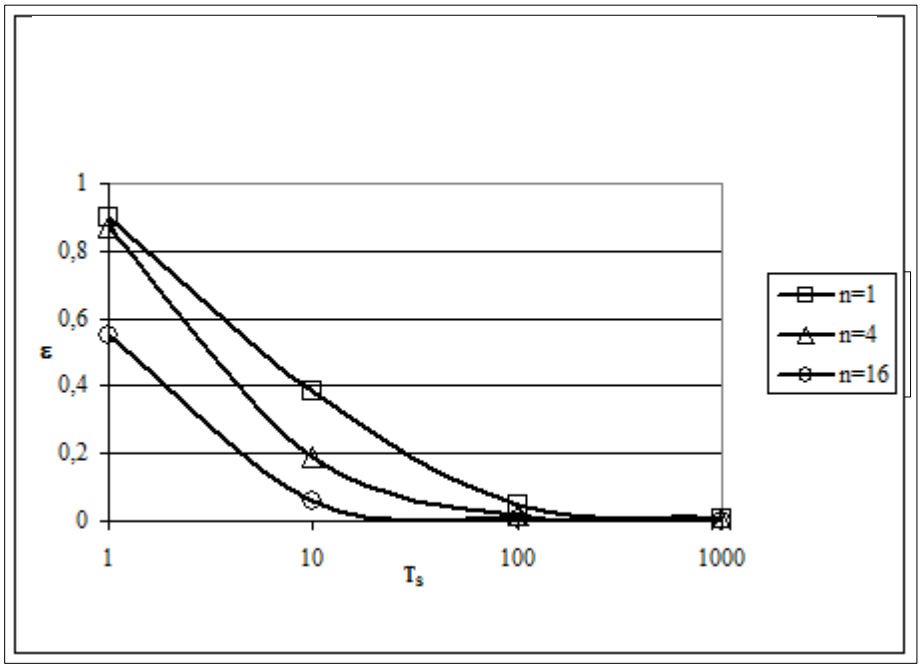


Figure E

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 *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.

The optimal value of *m* is, in general, less than *n*. Finding this value as the client-server system solution in closed form is not always simple, or even impossible, while a numeric solution is more tractable.

If *m* increases, then T_S decreases, thus ρ decreases, so W_Q decreases: T_{cl} decreases, and ε_{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:

$$\begin{cases} T_{cl} = T_G + R_Q \\ R_Q = W_Q(\rho, T_s, T_A) + L_s \\ \rho = \dfrac{T_s}{T_A} \\ T_A = \dfrac{T_{cl}}{n} \\ \rho < 1 \end{cases}$$

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

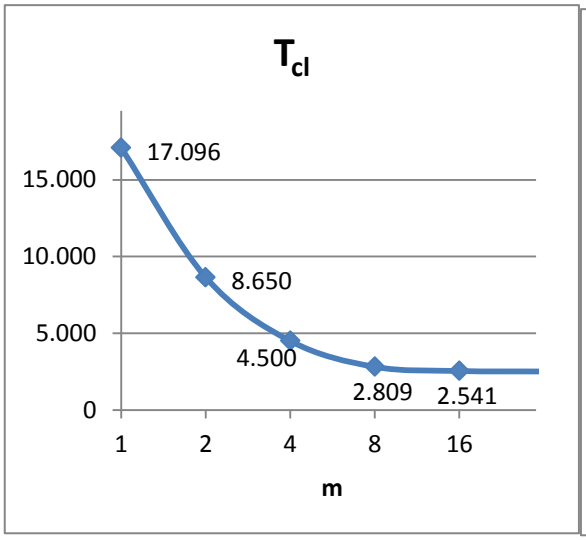
The latency has been evaluated, respectively in the constant and logarithmic case, as:

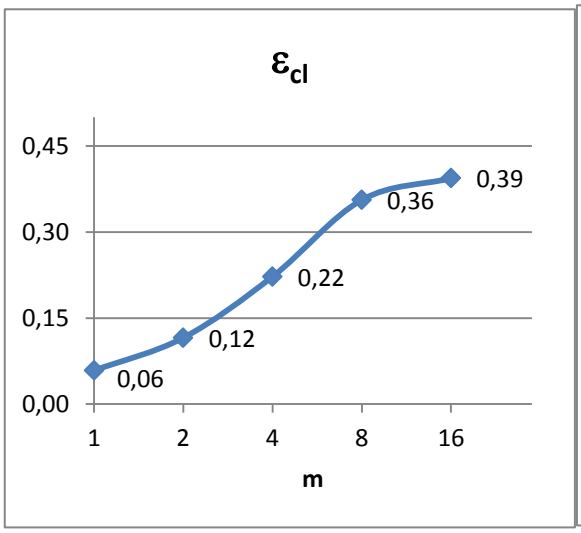
$$L_s = T_{calc} + L_0 \qquad L_s = T_{calc} + L_0 (1 + lg_2 (m))$$

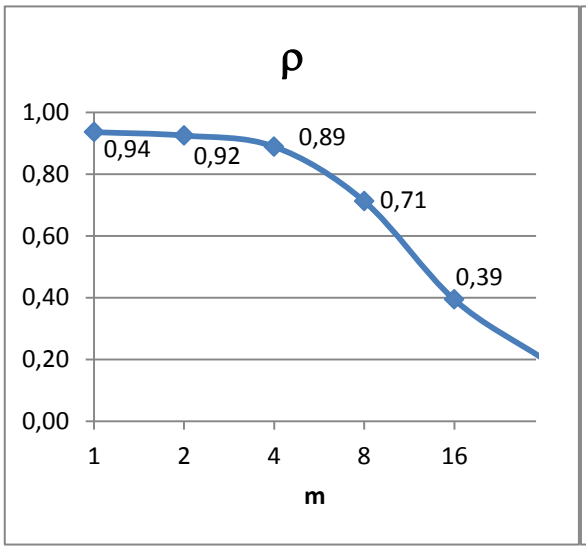
with $L_0 = 500$ t.

The results are reported in the following curves:

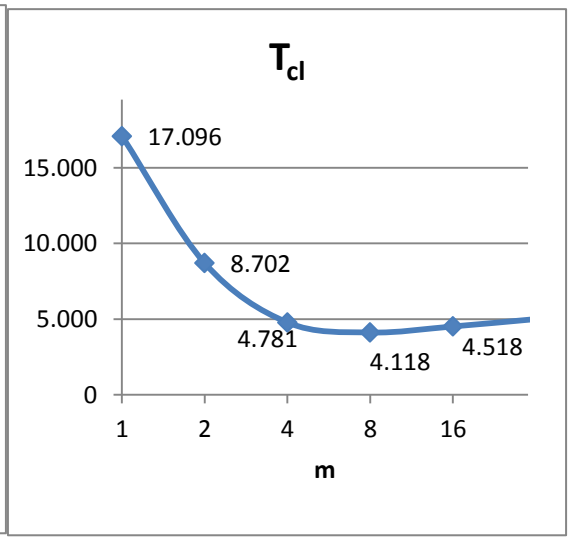
Constant latency

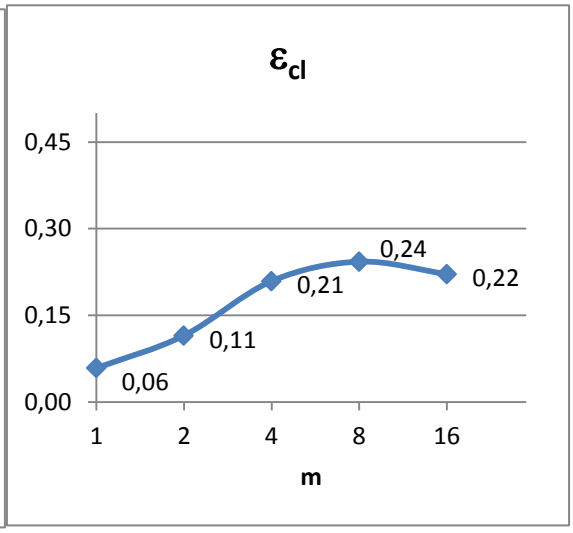


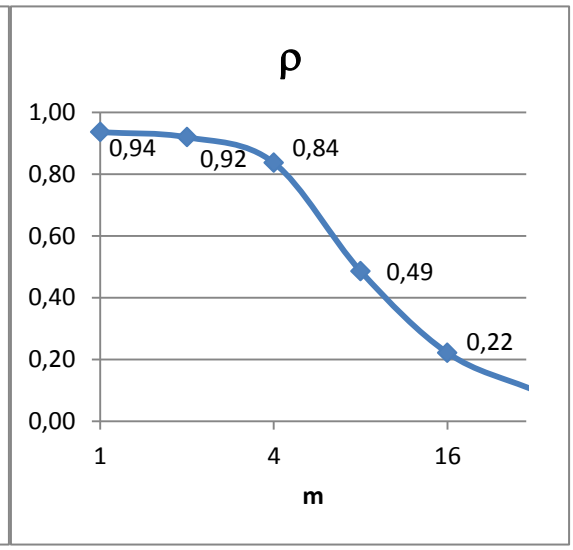




Logarithmic latency



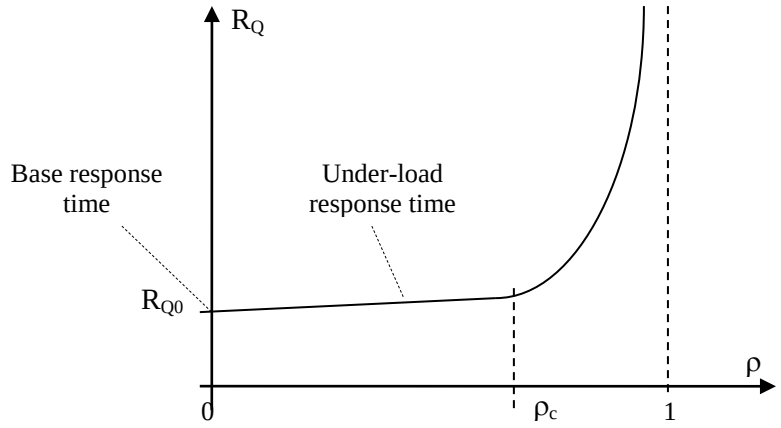




15.3 Client-server implementation issues

15.3.1 Critical value of utilization factor

In the typical shape of R_Q as a function of ρ (as reported in the following figure), we can recognize an interval of ρ values with a relatively slow slope of R_Q, until a critical value ρ_c, after which the slope is very fast:



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.

15.3.2 Farm and data parallel structures for server implementation

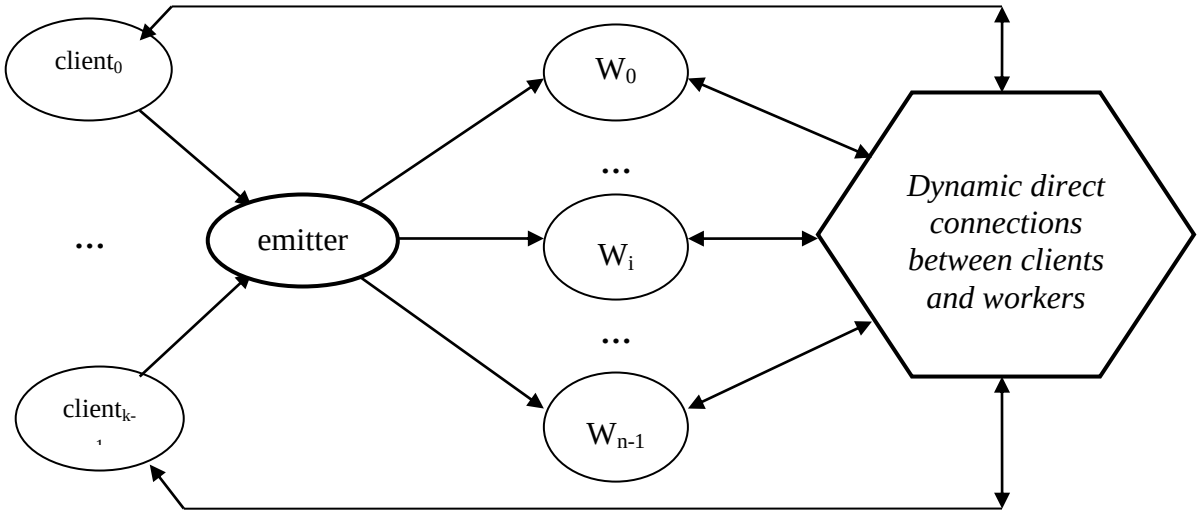
The parallelization of a server, in a client-server computation, can be done by farm or data parallel paradigms, with interesting implications.

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.

15.3.3 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.

Once a worker W_j is scheduled to serve the accepted client C_i, and possibly the internal state of W_j is initialized, W_j interact with C_i through a dynamic **direct** connection: the hexagonal block shown in the figure has to be interpreted as a set of all-to-all channels used to connect each client with the scheduled worker.

We can think that the scheduling connection phase is implemented according to a farm paradigm. However, in general the most important aspect of the parallelization is the *internal structure of the workers*: in turn, each worker has a parallel structure. Because the transaction evolution consists in a sequence of request and replies, *the interaction client-worker is not stream-based*. Therefore, if the transaction steps operate on large data structures, a data-parallel implementation of workers is adopted (otherwise, a data-flow scheme).

In large applications (e.g. web servers), and especially when individual transactions are different from each other, the processing nodes used for worker parallelization could belong to a common large pool and could be dynamically allocated according to the specific transaction requirements.

The *close_transaction* request is communicated to the emitter, which provides to disconnect the client from the worker, whose nodes become available again.

Proposed exercise

A computation Σ has a client-server structure, with *n* identical clients, operating on transactions.

Each transaction is defined in the following way:

- initially the client starts the transaction by sending an integer value *c*;
- once accepted, the client executes a sequence of steps. During each step it executes a local function G operating on a integer array *B[M]*, sends *B* and the G result which is an integer arrays *A[M]*, and then receives the pair *(B, end_transaction)*, where *B* is the new input argument of G and *end_transaction* is a boolean value. This behavior is repeated until *end_transaction* is true;

- once received the value *c*, the server performs the actions needed to start the transaction, and then executes the sequence of steps by interacting with the accepted client: for each step, the server receives the new value of *A and B*, and executes the following computation:

  do

  $$\forall\ i = 0\ ..\ M - 1 : B[i] = F\ (c, A[i], B[i])$$

  while $\exists\ j$ *: absolute_value* $(B[i] - A[i]) < r$

  After the step execution, the pair *(B, end_transaction)* is sent to the client, where *end_transaction* is true if and when the *do .. while* loop terminates.

All the parameters are known: n, M, T_F, T_G, T_{setup}, T_{transm}.

a) Design the server with the optimal parallelism degree m, taking into account that there are n clients to be served concurrently and each transaction has to be executed in parallel.

b) Evaluate the completion time of a generic transaction consisting of s steps.

c) Evaluate the relative efficiency of the server and of the generic client.

16. Solved exercises

This Section contains further exercises and questions on the various Sections of Part 1, the most part used for the lectures and exams of previous years. Solutions and extensions are discussed.

In Part 2, some of these exercises will be used to study the interrelations between the parallelization methodology and the parallel architectures.

16.1 Exercise 1

The following sequential computation works on a single value (A, B, C):

int A[M], B[M][M], C[M];

$\forall$ *i = 0 .. M – 1*:

$\forall$ *j = 0 .. M – 1*:

A[j] = F (A[j], B[*][j], C[i])

a) Apply the virtual processors model to define at least one equivalent data parallel computation.

b) Define an equivalent farm-based computation, and its optimal parallelism degree, assuming that $M = 10^3$, $T_F = 64*10^2\tau$, $T_{setup} = 10^3\tau$, $T_{transm} = 10^2\tau$.

c) Consider the farm-based parallel computation of point *b):*

c1. Explain how it is executed by a parallel machine with *N* processing nodes, where $N < n$, where *n* is the optimal parallelism degree.

c2. Explain under which conditions the emitter ideal service time can be evaluated as $T_{send}(k)$, where *k* is number of words of a stream element.

a) Data parallel VP version. According to the owner computes rule, there is potential "parallelism on *j"*, i.e. the innermost loop iterations can be executed in parallel, *provided that* this parallel transformation is consistent with the program semantics on indexes *i* and *j*. We realize that the innermost and the outermost loops can be exchanged each other, thus:

int A[M], B[M][M], C[M];

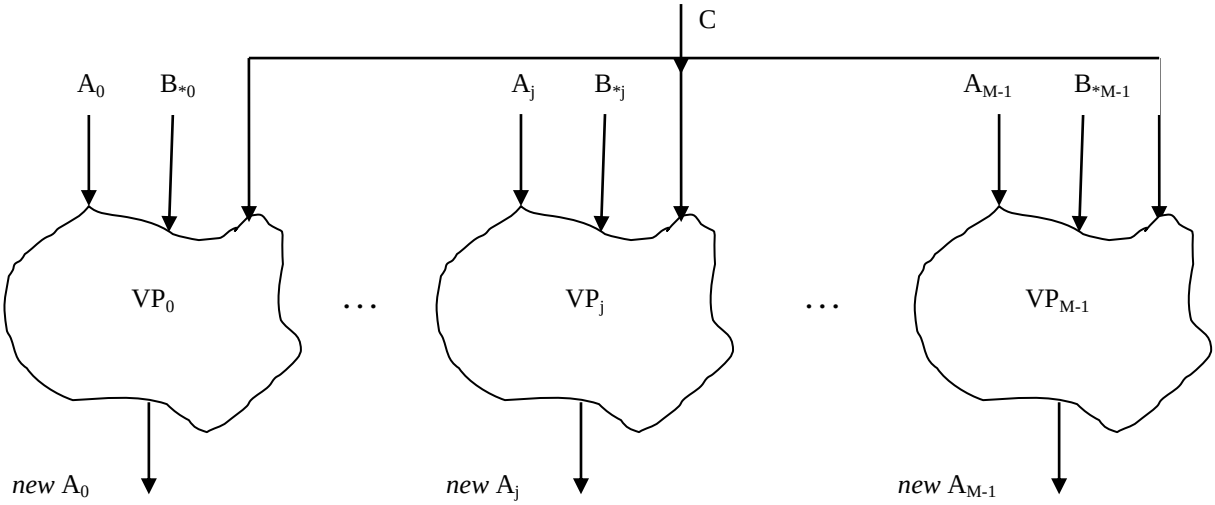
for all ***j*** *= 0 .. M – 1* **in parallel**:

$\forall$ *i = 0 .. M – 1*:

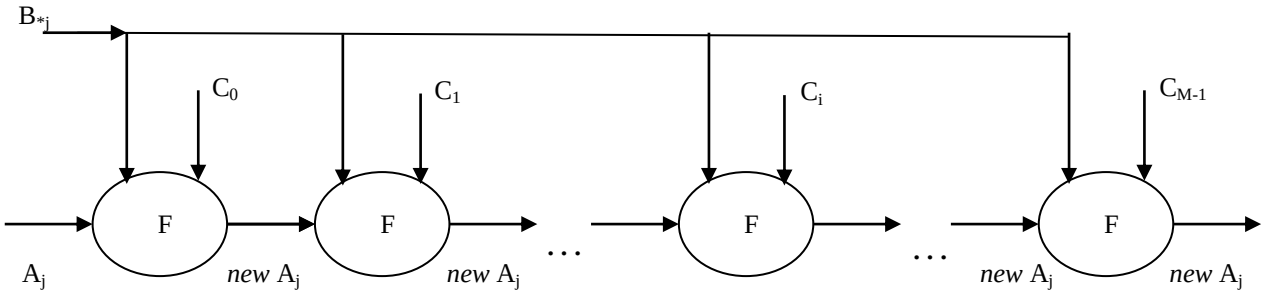
A[j] = F (A[j], B[*][j], C[i])

The data parallel computation is defined by a linear array of virtual processors, *VP[M]*.

To determine this parallel structure *formally*, we apply the Bernstein conditions to the loop-unrolled computation in order to find the data dependences. The following data-dependence graph (data-flow graph) is obtained:



where the *j*-th subgraph (*j*-th virtual processor) is expanded as follows:



According to the distribution strategy of *C* elements, two data parallel solutions can be defined:

- *Map*: *C* is replicated; *VP[j]* encapsulates *A[j], B[*][j],* and *C[*];*
- *Stencil*: *C* is partitioned; *VP[j]* encapsulates *A[j], B[*][j],* and *C[j]*. The stencil is so defined: at the sequential step *i*, the *C[i]* value is rendered available to all the other VPs (in a message-passing system: a multicast communication executed by *VP[i]*). This requires M^2 communication channels among VPs (or other optimized solutions to be evaluated).

b) Farm solution. Because the sequential computation works on a *single* value (A, B, C), a farm solution is feasible on condition that a proper stream is generated inside the parallel computation itself (the condition that the computation has a purely functional semantics is satisfied). A module UNPACK, encapsulating A, B and C, is in charge of supplying the farm with the proper stream elements (*stream-equivalent computation*).

The analysis of point *a)* allows us to formally *recognize the stream* to be generated: since the parallelism is "on *j*", the *j*-th stream element is composed by *A[j], B[*][j],* and *C[*],* i.e. the data distribution adopted in the map solution. This result can be of some utility and can be generalized:

- given a sequential *function* operating on a *single value*, if a *map* data parallel transformation is feasible, than a stream-equivalent *farm* solution is feasible too,

where the generated stream elements coincide with the data distribution in the map virtual processors version;

- in other words: the virtual processors approach is applied in the farm paradigm too, in order to define the stream structure in a stream-equivalent computation.

As far as the cost model is concerned, the interarrival time to the farm is given by:

$$T_A = T_{send}\,(2M + 1) = T_{setup} + (2M + 1)\,T_{transm} \sim 2M\,T_{transm} = 2 * 10^5\ \tau$$

which is equal to the ideal service time of the farm emitter too, under the condition that *zero-copy* communications are adopted.

Having defined the stream, the farm computation is the parallelization of the sequential computation:

$$\forall\ i = 0\ ..\ M - 1:$$
$$A[j] = F\,(A[j], B[*][j], C[i])$$

whose calculation time is equal to $M\,T_F$. Therefore, the optimal parallelism degree is given by:

$$n = \left\lceil \frac{T_{calc}}{T_A} \right\rceil = \frac{M\,T_F}{T_A} = 32$$

With this parallelism degree, the farm service time, thus the service time of the farm-based computation (unpack-farm-pack), is equal to T_A.

If the number of processing nodes is $N < n$, the bandwidth decreases.

If we adopt the approach "one process per processing node", the scalability lowers accordingly to the ratio N/n, i.e. the service time becomes:

$$T_s = \frac{n}{N}\,T_A$$

More precisely (Sect. 15), since 4 processing nodes must be used used for the unpack, emitter, collector and pack processes ("service processes", whose number is not "parametric", as the workers number is), the number of worker processes becomes $N - 4$, thus

$$T_s = \frac{n}{N-4}\,T_A$$

Extension

It is proposed to further develop the three parallel solutions of Question 1:

1. data parallel map
2. data parallel stencil
3. farm-based

passing to the *actual implementation of the data parallel versions too*, with the optimal parallelism degree. In any version, an IN process encapsulates A, B, C (unpack in the farm-based computation), and an OUT process the new A value (farm-based pack).

The study includes the issues of interest when dealing with optimizations of parallel applications:

- in both data parallel versions: *collective communications*,
- in the stencil data parallel versions: *alternative solutions to the stencil implementation*,
- in the farm-based version: *optimal communication grain*.

Consider also constraints on the properties of function *F*, notably the case in which *F* is *associative*.

Compare the completion time and the memory size of the three versions.

16.2 Exercise 2

a) Consider a computation Σ structured as an acyclic graph of modules $M_0, \ldots, M_{n-1}$. Explain how the relative efficiencies of each M_i and of the whole Σ are evaluated.

b) Consider a client-server request-reply computation Σ composed by identical client modules $C_0, \ldots, C_{n-1}$ and a server module *S*. Explain how the relative efficiencies of the generic C_i, of *S*, and of the whole Σ are evaluated.

c) Referring to cases *a)* and *b)*, show at least one situation in which the relative efficiency and the bandwidth, both evaluated as functions of the parallelism degree, have similar qualitative shapes (*e.g.* both are increasing monotonic functions), and at least one situation in which the relative efficiency and the bandwidth have quite different qualitative shapes (*e.g.* one of the two is an increasing monotonic function, the other is a decreasing monotonic function).

a) According to the course theory (in particular, Sect. 5), the relative efficiency of the generic module M_i is evaluated as follows:

$$\varepsilon_{M_i} = \begin{cases} \rho_{M_i} \;\; if \;\; \rho_{M_i} < 1 \\ 1 \quad\;\; if \;\; \rho_{M_i} \geq 1 \end{cases}$$

where the utilization factor ρ_{M_i} is equal to the ratio of the module ideal service time (evaluated *in isolation*), and its interarrival time, which is evaluated by solving the graph.

For the whole computation:

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

where T_A is the inverse of the stream generation bandwidth, and T_p is the interdeparture time of the whole graph. Both parameters are determined by solving the graph (it is possible that some fictitious streams and modules must be modeled).

b) According to the course theory (in particular, Sect. 5 and Sect. 16), the relative efficiency of the generic client C_i is evaluated as follows:

$$\varepsilon_{cl} = \frac{T_{cl-id}}{T_{cl}} = \frac{T_G}{T_G + R_Q} = \frac{1}{1 + \frac{R_Q}{T_G}}$$

where T_G is the client ideal service time, and R_Q the server response time, which is evaluated by solving the general system of equations:

$$\begin{cases} T_{cl} = T_G + R_Q \\ R_Q = W_Q(\rho, T_S, \sigma_S) + Ls \\ \rho = \dfrac{T_S}{T_A} \\ T_A = \dfrac{T_{cl}}{n} \end{cases}$$

with the constraint $\rho < 1$. The expression of the queue waiting time, W_Q, depends on the assumed probability distributions of service time and of interarrival time random variables.

For the server:

$$\varepsilon_S = \rho = \frac{T_S}{T_A}$$

which is the solution of the above system of equations.

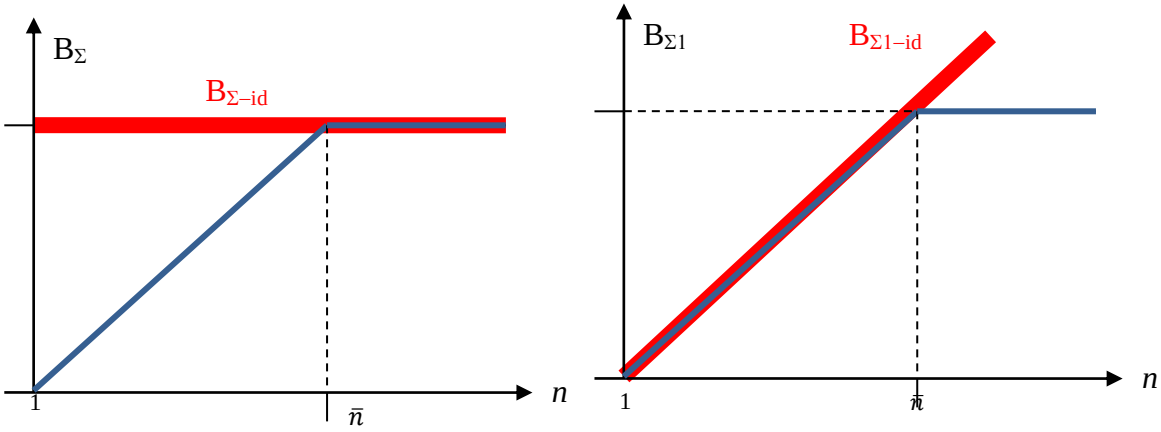
The result of the whole computation can be considered as the result of the clients set, i.e. the result stream is the server requests stream. Therefore, for any n:

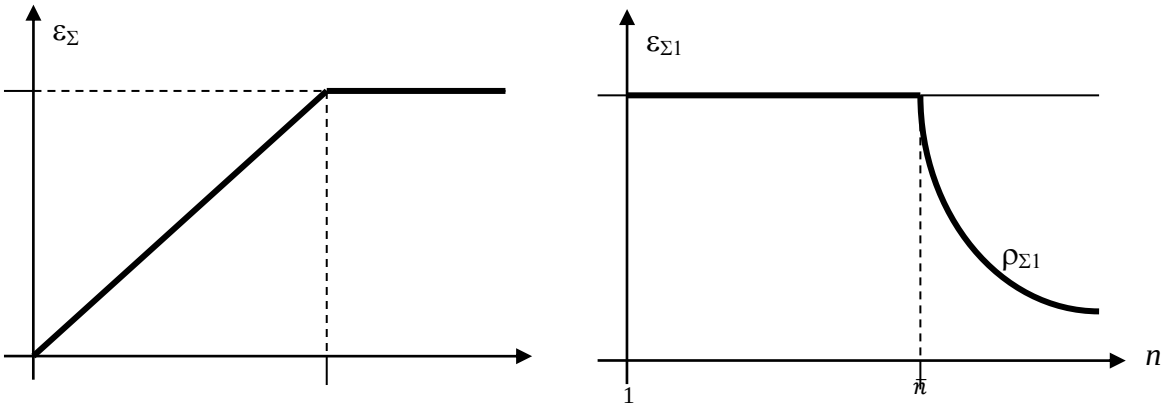
$$\varepsilon_\Sigma = \varepsilon_{cl}$$

(of course, the number of clients has an outstanding impact on the various performance metrics, including the efficiencies themselves).

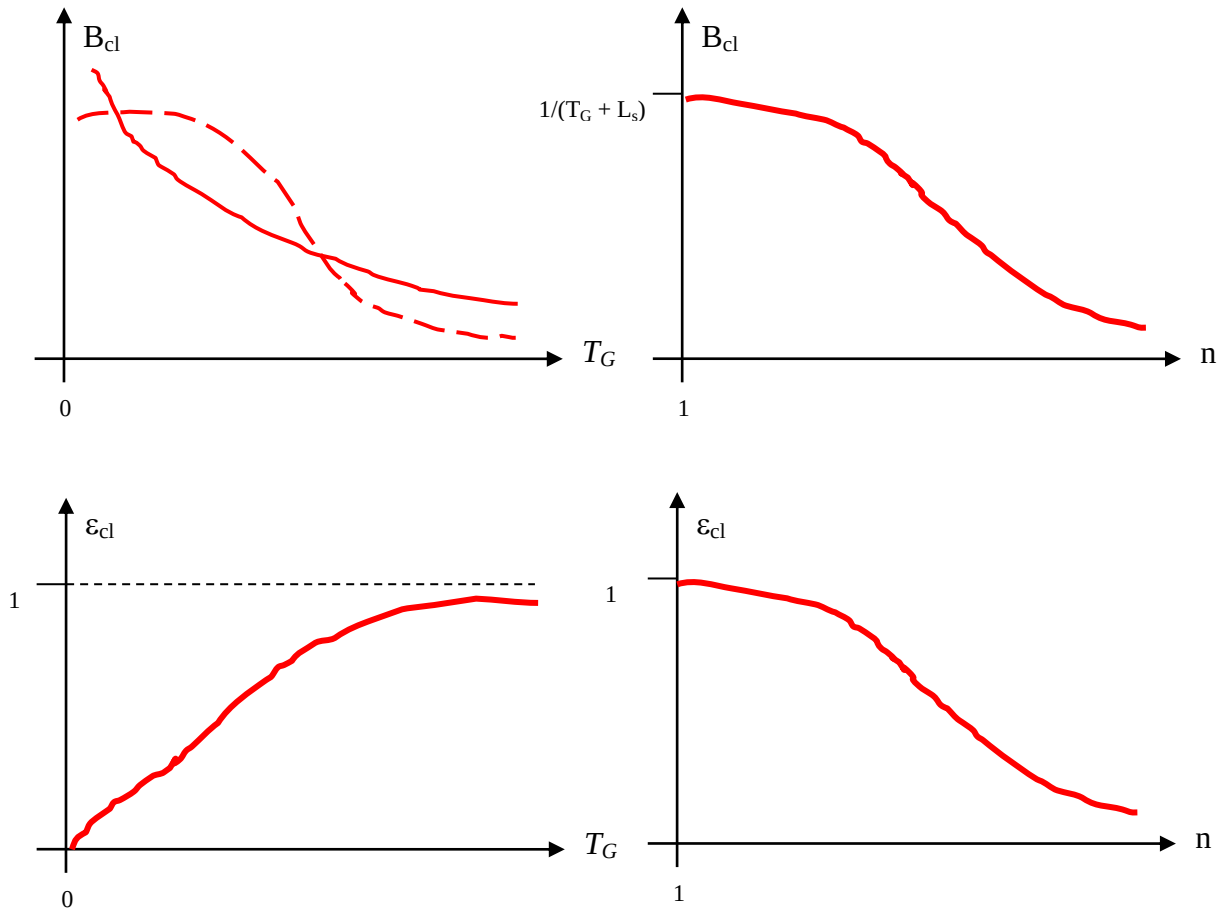
c) For an acyclic graph computation, we consider the bandwidth and the relative efficiency as functions of the *parallelism degree of a parallelized bottleneck module*, assuming it is the only one bottleneck. These functions are determined by applying the general theory of Sect. 5.

For the whole computation Σ and for the parallelized module $\Sigma 1$, respectively, we have:





For the client-server computation, we consider the client bandwidth and the client relative efficiency as functions of the number n of clients, and as functions of the client ideal service time T_G. These functions are determined by applying the general theory of Sect. 5 and 6:



16.3 Exercise 3

A 3-stage pipeline Σ is composed of processes PROD, P, CONS.

PROD encapsulates a square matrix *D[M][M]* and produces a stream whose elements are the rows of *D*.

P encapsulates the square matrix *B[M][M]* and the integer *c* statically, and is defined as follows:

```
int A[M]; int B[M][M]; int c;
    ∀ i = 0 .. M – 1:
        { receive (input_stream, A);
          ∀ j = 0 .. M – 1:
              A[j] = F (A[j], B[j][*], c);
          send (output_stream, A)
        }
```

CONS builds a square matrix *E[M][M]*, respecting the same order of the corresponding rows of *D*.

The parameter values are $M = 10^3$, $T_F = 64*10^2\tau$, $T_{setup} = 10^3\tau$, $T_{transm} = 10^2\tau$.

The abstract architecture has 32 nodes, each one with communication processor.

Define a farm version and a data-parallel version for P. For each of them evaluate the service time and the relative efficiency of Σ and of P.

a) The interparture time of PROD is T_{send} (M) ~ $10^5\tau$. This is the ideal service time for Σ: $T_{\Sigma id} = 10^5\tau$, and the interarrival time T_A to P.

Each stream element includes the row index value, to be propagated to CONS (it has a negligible impact on the evaluation).

For P $T_{calc} = 64\ 10^5\tau$. Thus, the optimal parallelism degree is n = 64: it cannot be exploited, since the number of available nodes is N = 32. However, before applying the parallelism reduction, the parallelization solution must be studied.

Farm version: B and *c* are replicated in the workers. The emitter is not a bottleneck: for zero-copy communications, $T_E = T_{send}(M) = T_A$, and the worker communications are masked by the calculation time. Thus, in principle, the service time of P and of Σ is equal to the ideal service time with n = 64 workers and $n_\Sigma = 68$. However, N = 32, of which $n_{act} = 28$ can be actually used for the workers. Thus P remains a bottleneck, and

$$T_{Pid} = T_{calc} / 28 = 2{,}3\ 10^5\tau,\ T_P = T_{Pid},\ \varepsilon_P = 1$$

$$T_{\Sigma id} = T_A = 10^5\tau,\ T_\Sigma = T_P = 2{,}3\ 10^5\tau,\ \varepsilon_\Sigma = 1/2{,}3 = 0{,}44$$

Data parallel version: studying the computation form

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

the virtual processors structure is VP[M], where the generic VP[j] encapsulates the partition *A[j]*, the partition *B[j][*]*, and *c*. As *c* is scalar, it can only be replicated. Thus, the solution is a *map*, where *B* is statically partitioned by rows, and *c* is statically replicated. The input array *A* must be scattered, and the result array *new-A* must be gathered.

We must evaluate whether the scatter functionality is a bottleneck. It can be shown that it is a bottleneck for n = 64 workers by applying the linear scattering. However, also applying the tree-structured scattering the utilization factor is slightly >1, on the other hand, the limited number of nodes prevent the use of process tree structures, thus, in order to save nodes, only the linear scattering is valid. By imposing that the scatter service time is equal to the service time of the workers set, we find a more suitable value for *n*. For a linear scattering, we obtain n = 50 (a second degree equation has to be solved).

In any case, because N = 32, P remains a bottleneck, and 28 workers are available with a linear scattering, with partition size = M/28 (properly rounded and padded).

The performance measures for P and Σ are the same of the farm version. However, in this problem this is due to the limited number of nodes: with N ≥ 68, the farm is able to achieve the ideal bandwidth, while the map is not.

Other optional considerations might concern the farm-map comparison.

16.4 Exercise 4

A computation Σ is composed of a process Q with an input stream.

Q encapsulates an integer *c* and an integer array *A[M]*, with $M = 10^5$, and receives an integer array *B[M]*. For each *B*, Q modifies the variable *c* in the following way:

$$c = \sum_{i=0}^{M-1} F(A[i], B[i], c)$$

The input stream is generated by a process with average interdeparture time equal to $32*10^6\tau$.

Function *F* has a constant processing time equal to $10^4\tau$.

We wish to parallelize Q for an architecture with 64 processing nodes, each with communication processor, zero-copy interprocess communications, $T_{setup} = 10^3\tau$ and $T_{transm} = 10^2\tau$.

a) Define and evaluate a parallel version of Q. Determine its parallelism degree, the ideal and effective service time and the relative efficiency of Σ and Q.

b) Show and explain the function of the ideal and effective bandwidth, and the function of the relative efficiency, of both Σ and Q, varying the parallelism degree of Q between one and infinity.

a) A possible pseudo-code of sequential Q is:

int A[M], B[M], X[M]]; int c = initial_state;

∀ *received B*:

{ ∀ *i = 0 .. M – 1*:

X[i] = F (A[i], B[i], c);

c = reduce (X, +)

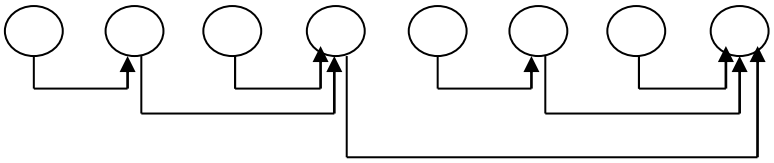
}

The variable *c* is the internal state of the computation, modified for each stream element. Thus, a farm parallelization is not feasible.

A data-parallel, according to the composition of *map* and *reduce* paradigms, is the solution to be adopted. In terms of virtual processors, the parallel version of Q consists of the *VP[M]* array, where the generic *VP[i]* contains *A[i]* statically, *B[i]* received for each stream element, *X[i]*, and *c*. The variable *c* is *replicated* in all VPs, initialized at the same value, and *modified consistently* in all VPs at the end of the *reduce* computation.

Each VP can start a new computation on a new *B only after* having updated its own copy of *c*.

The parallel *reduce* in logarithmic time is provided (in the actual version, it will be verified against other implementations). This is realized by emulating a lg_2M dimension cube by means of the same linear array of VPs with *M-1* communication channels of type *int*:



At the end of the *reduce* computation, *VP[M-1]* contains the updated value of *c* to be multicasted to all the other VPs. A *logarithmic multicast* employs additional *M-1* integer communication channels, which are dual with respect to the *reduce* channels:

In a high-level pseudo-code:

VP[i]::

while true *do*

{ *receive new B;*

X[i] = F (A[i], B[i], c);

take part to reduce (X, +);

if (i = M) then multicast (c) else receive new c

}

The actual implementation has a parallelism degree *n* and a partition size *g = M/n* (for *A, B, X*).

The *B* distribution is implemented by a process executing a Scatter: as usually, this is executed in pipeline with the workers. With zero-copy communications, each time a worker starts a new computation, it finds the relative *B* partition ready.

Provided that the Scatter functionality is not a bottleneck, the service time of Q is given by:

$$T_s = \frac{T_{calc}}{n} + L_{reduce}(n,1) + L_{multicast}(n,1)$$

where L_{reduce} and $L_{multicast}$ are the *latencies* of *reduce* and *multicast* operations. Of course, no overlapping of *reduce* and *multicast* communications is possible for semantic reasons.

Any implementation of *reduce* starts with a *local reduce* in each worker (in parallel) applied sequentially to each local partition *X[g]*. In fact, an addition is executed after each application of function *F*, so the cost of g addition instructions is quite negligible compared with the other latencies to be evaluated.

The *n* integer results of the local *reduce*s are then used as inputs to the second phase of *reduce* to produce the updated value of *c*. A decentralized logarithmic *reduce* has a cost

$$L_{reduce}(n,1) = lg_2\,(n)\,T_{send}(1)$$

thus, it is more convenient than an implementation centralized in a separate process:

$$L_{centralized-reduce}(n,1) \sim gather\,(n,n) \sim n\,T_{send}(1)$$

The same is true for the *multicast*, which will be implemented according to the decentralized logarithmic strategy:

$$L_{multicast}(n,1) = lg_2\,(n)\,T_{send}(1)$$

Thus,

$$T_s = \frac{T_{calc}}{n} + 2\ lg_2\,(n)\,T_{send}(1)$$

Formally, the optimal parallelism degree can be determined by solving (numerically) the equation:

$$\frac{T_{calc}}{n} + 2\ lg_2\,(n)\,T_{send}(1) = T_A$$

In practice, it is sufficient to adopt a simpler, approximate procedure: provided that Scatter is not a bottleneck, we evaluate

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

Then, L_{reduce} and $L_{multicast}$ are calculated using this value of *n*, and the value of T_s is corrected. Due to the low latencies L_{reduce} and $L_{multicast}$, the correction is very limited. In the worst case (i.e. if the corrected T_S is > T_A), the optimal parallelism degree could be increased to *n+1*.

In our case:

$T_{calc} = 10^9 * \tau$

$n = 32$ $n_\Sigma = n + 2 = 34$ (including the Scatter and the Producer processes) < *N*

$g = 3{,}125 * 10^3$

$T_{scatter} \sim 10 * 10^6\ \tau < T_A$

$T_{s\text{-}map} = \frac{T_{calc}}{n} = 31{,}25 * 10^6\ \tau$

$L_{reduce} = 5{,}5 * 10^3\ \tau$

$L_{multicast} = 5{,}5 * 10^3\ \tau$

We obtain:

$T_s = T_{s\text{-}map} + L_{reduce} + L_{multicast} = 31{,}26 * 10^6\ \tau$

which is very close to $T_{s\text{-}map}$, and still < T_A. Thus, no additional worker is needed to achieve the best service time as possible.

The performance metrics for Q are:

$$T_{\mathrm{Q}-id} = T_S$$

$$T_{\mathrm{Q}} = T_A$$

$$\varepsilon_Q = \rho_Q = \frac{T_S}{T_A} = 0{,}98$$

For Σ (modeled with a fictitious output stream composed of the generated *c* values):

$$T_{\Sigma-id} = T_A$$

$$T_{\Sigma} = T_{\mathrm{Q}} = T_A$$

$$\varepsilon_{\Sigma} = 1$$

As in general, when there is no bottleneck in a computation Σ, the ideal bandwidth of Σ is actually achieved, at the expense of a slight underutilization of some component modules (Q in this case).

b) See also Sect. 5.4. Q performance functions:

- neglecting the impact of *reduce* and *multicast*, the ideal bandwidth is linear in *n* (n/T_{calc}). Taking into account of the impact of *reduce* and *multicast*, the curve is slightly lower than the straight line;
- the effective bandwidth coincides with the ideal one for $n < 32$, while for $n \geq 32$ it is constant and equal to $1/T_A$;
- the efficiency is equal to one for $n < 32$, while for $n \geq 32$ it tends to zero (~ $T_{calc}/n\ T_{A)}$.

Σ performance functions:

- the ideal bandwidth is constant and equal to $1/T_A$;
- the effective bandwidth is about linear in *n* (~ n/T_{calc}) for $n < 32$ (same consideration as before), while for $n \geq 32$ it is constant and equal to $1/T_A$;
- the efficiency is about linear in *n* for $n < 32$, while for $n \geq 32$ it is constant and equal to one.

16.5 Exercise 5

A computation Σ is composed of a process P with an input stream.

P encapsulates an integer array $V[M_1]$, with $M_1 = 10^4$, and receives an integer array $W[M_2]$, with $M_2 = 10^5$.

For each W, P executes the following computation:

{ *b = reduce (W, G);*

∀ *i = 0 .. M₁ – 1:*

∀ *j = 0 .. M₂ – 1:*

V[i] = F (b, V[i], W[j])

}

where *G* is an associative function.

The input stream is generated by a process with average interdeparture time equal to $10^8\tau$. Functions *F* and *G* have constant processing time equal to 10τ.

We wish to parallelize P for an architecture with $N = 64$ processing nodes, each with communication processor, zero-copy interprocess communications, $T_{setup} = 10^3\tau$ and $T_{transm} = 10^2\tau$.

Define and evaluate *two* distinct parallel versions of P. For each version determine the parallelism degree *n*, and the corresponding ideal and effective service time and relative efficiency of Σ and P.

The sequential version of P is characterized as follows:

- initially *b* is evaluated on the received W: $T_{reduce} = M_2\ T_G = 10^6\tau$;
- then the double loop is executed, with a cost: $T_c = M_1\ M_2\ T_F = 10^{10}\tau$;
- thus Tcalc ~ Tc = $10^{10}\tau$.

The ideal parallelism of a parallel solution is

$$n_{id} = \frac{Tcalc}{TA} = 10^2$$

which cannot be exploited because $N = 64$. In the best case, considering two nodes allocated to the produced process and to an input interface process, the exploitable parallelism is

$$n = 62$$

P remains a bottleneck and the service time is

$$T_S = T_{calc}/n = 1{,}6\ 10^8\tau$$

provided that the data distribution has a service time lower than Ts.

The computation has an *internal state* V, modified for each W, thus the farm paradigm cannot be applied.

There are two *data-parallel* solutions: *map* and *stencil-based*. Both work on the result of data-parallel reduce, which must be implemented using the same *n* workers directly in order to save processing nodes.

According to the virtual processors approach, there are M_1 VPs, VP[M_1], where the generic VP[i] encapsulates VP[i] and *b* replicated.

In the map version W is *replicated* in all VPs.

In the stencil version W is *partitioned*, with each VP containing M_2/M_1 elements. At each step j, VP[j] multicasts the W[j] value to all other VPs.

The map version could also be implemented according to a pipeline structure, at the expense of a linear latency and linear multicast of W. In the following, we will not apply this solution, since, in this case, a classical map with logarithmic multicast has better service time and latency.

In both actual versions, the reduce is implemented according to three phases:

- local reduce operating on $g = M_2/n$ elements, with cost $16\ 10^3\tau$;
- logarithmic reduce, according to a tree structure mapped onto the same workers, with cost

$$T_{send}(1)\ lg_2 n = 6\ 10^3\tau;$$

- multicast the final result *b* from W_{M-1}, according to a tree structure mapped onto the same workers, with cost

$$T_{send}(1)\ lg_2 n = 6\ 10^3\tau.$$

Thus, the reduce service time is about $28\ 10^3\tau$, which is negligible compared to the estimated service time T_s and T_A.

In the actual map version, the W multicast has a service time greater than T_s if implemented according to a linear approach, thus it is implemented by a tree structure mapped onto the same workers, with service time:

$$T_{multicast} = T_{send}(M2)\ lg_2 n = 6\ 10^7\tau < T_s$$

In the actual stencil version, a linear scatter of W has a service time

$$T_{scatter} \sim T_{send}(M_2) \sim 10^7\tau < Ts$$

The stencil communication pattern is a multicast of one element, or of *g* elements, thus is has a negligible effect on the service time T_s.

Therefore, the estimated service time is really achieved in both versions.

Performance metrics for P with parallelism degree *n* : ideal and effective service time = $T_S = 1{,}6\ 10^8\tau$, $\varepsilon_P = 1$.

Performance metrics for Σ with parallelism degree *n* : ideal service time $T_A = 10^8\tau$, effective service time $T_S = 1{,}6\ 10^8\tau$, $\varepsilon_\Sigma = 1/1{,}6 = 0{,}62$.

16.6 Exercise 6

A computation is composed of a process Q operating on streams. Q encapsulates an integer array *C[M]* statically, with a given initial value. Each element of the input stream is a couple *(A[M], op),* where *A* is an integer array and *op* is a boolean. Each element of the output stream is an integer array *B[M]*.

Q is defined as follows:

```
while (true) do
    { receive (input_stream, (A, op));
      if op
          then   ∀ i = 0 .. M–1:
                        C[i] = F1 (A[i], C[i])
          else   { ∀ i = 0 .. M–1:
                        { B[i] = 0;
                          ∀ j = 0 .. M–1:
                              B[i] = F2 (A[i], B[i], C[j])
                        }
                   send (output_stream, B)
                 }
    }
```

The probability p that $op = true$ is equal to 10^{-2}. The array size M is equal to 10^4. Let $t = 10^3 \tau$ (τ = clock cycle of processing nodes): $T_{F1} = 10^{-1} t$, $T_{F2} = 10^{-2} t$, $T_{setup} = t$, $T_{transm} = 10^{-1} t$, interarrival time = $10^4 t$.

The parallel architecture has $N = 200$ processing nodes, each one with communication processor. Interprocess communication is implemented according to the zero-copy technique.

Define, explain and evaluate two parallel versions of Q, based on the farm and on the data-parallel paradigm respectively.

Evaluate how the memory hierarchy of nodes is exploited according to the property of the computation.

The computation has internal state C, which is modified only if $op = true$ with (low) probability $p = 10^{-2}$.

A first view of parallelizations

A *farm with internal state* solution is potentially feasible (the necessary condition on p has to be verified), with C replicated and kept consistent through the execution of the *first operation* ($op = true$) in all the workers (A multicasted). Parallelism is exploited only in the *second operation* ($op = false$), with A distributed on-demand as usually in a farm structure.

A *data-parallel* solution is feasible: we have to decide whether C is statically partitioned or statically replicated. For $op = true$, partitioning is sufficient to execute a map computation, for $op = false$ partitioning implies a stencil computation while a map computation is obtained through replication. These considerations will be validated by the virtual processor method. Replication implies a simpler implementation of the second operation at the expense of wasting memory for the first operation. Thus, the solution with *partitioned C* will be chosen provided that the stencil overhead is acceptable: this issue has to be evaluated.

The calculation times are:

$$T_{calc1} = M\ T_{F1} = \ 10^3\ t$$
$$T_{calc2} = M^2\ T_{F2} = \ 10^6\ t$$

On the average:

$$T_{calc} = p\ T_{calc1} + (1-p)\ T_{calc2} \sim T_{calc2}$$

The output communication is fully overlapped, since:

$$T_{send}\ (M) = \ T_{setup} + M\ T_{trasm} \sim 10^3\ t$$

The optimal parallelism degree is:

$$n = \frac{T_{calc}}{T_A} = 10^2$$

N = 200 processing nodes are largely sufficient, also by taking into account the allocation of few service processes.

The data memory size (arrays *A, B, C*) is *3M* ~ 30K words in the uniprocessor version. The computation has the property of locality on *A* and *B*, and locality + *reuse* on *C*. Thus, the best exploitation of the cache hierarchy is achieved by imposing that *C*, after the first utilization, is maintained in the primary cache permanently (~ 10K words). With a "good" current-technology for the primary data cache, this can be actually implemented in the sequential solution.

Farm with internal state

The parallelism degree is given by (see Course Notes, Part 1, Section 11.3; this result is easily obtained by imposing that the service time is equal to the interarrival time):

$$n_{farm-state} = \frac{(1-p)\ T_{calc2}}{T_A - p\ T_{calc1}}$$

In practice, because of the low value of *p*:

$$n_{farm-state} \sim \ n$$

The necessary condition on the probability value (see the formula above):

$$p < \frac{T_A}{T_{calc1}}$$

is always satisfied.

On the average, all the communications from the workers to the collector are overlapped to the internal calculation.

The emitter module is in charge of testing *op*: this calculation has a quite negligible impact. If *op = true* A is multicasted, otherwise A is distributed on-demand. A linear multicast has latency

$$T_{E-multicast} = n\ T_{send}\ (M) = \ 10^5\ t$$

On the average, the emitter service time is:

$$T_E = p\ T_{E-multicast}\ + (1-p)\ T_{send}\ (M) \sim 2\ \ 10^3\ t\ <\ \ T_A$$

Thus, *even with a simple linear multicast*, the farm solution is feasible with the optimal parallelism degree, in practice achieving the ideal service time and the maximum efficiency:

$$T_\Sigma = T_{\Sigma-\mathrm{id}} = T_A$$

$$\varepsilon_\Sigma = 1$$

(rigorously, service time and efficiency are slightly lower).

The data memory size per node (arrays *A*, *B*, *C*) is the same of the uniprocessor version: *3M* ~ 30K words. A far as the cache hierarchy exploitation, the same considerations done for the sequential computation apply to the farm solution too.

Data-parallel

Applying the owner computes rule, the data-parallel computation is modeled as a linear array of virtual processors, *VP[M]*, where the generic *VP[i]* operates on *A[i]*, *op*, *B[i]* and *C[i]*. In particular, as said, a statically partitioned allocation of C is the most convenient.

Each virtual processor provides to test the value of *op*, which is replicated.

If *op = true*, a *map* computation is executed (in the alternative allocation of replicated C, memory could be unnecessarily wasted), otherwise a *static and variable stencil* computation is executed, where the stencil at sequential step *j* consists in a multicast from *VP[j]*. For *op = false*, each virtual processor executes the loop on *j* sequentially (*M* steps).

Passing to the actual implementation with parallelism degree $n = 10^2$, the partitions of *A*, *B*, *C* have size equal to

$$g = \frac{M}{n} = 10^2$$

A is scattered by an input module, *B* is gathered by an output module, *C* is statically partitioned, and *op* is multicasted by the input module.

The input module executes the scattering of *A* and the multicast of *op*. Obviously, the *op* multicast has a negligible impact on the input module service time. With a linear implementation of scatter and gather:

$$T_{input} = T_{output} \sim T_{scatter}(n, g) = n\,T_{setup} + M\,T_{transm} \sim 10^3\,t < T_A$$

Thus, *even with a simple linear scatter and gather*, no bottleneck is introduced by the data distribution and collection functionalities: the optimal parallelism *n* can be actually exploited.

In the first operation (executed with negligible probability) each worker has service time equal to T_A.

In the second operation (executed with probability close to one) each worker executes the loop on *j* sequentially for *g* times ($g\,M = M^2/n$ steps). The multicast communication (with message size equal to *g*) for implementing the stencil could be overlapped to the worker calculation. This is not realistic with a linear multicast implementation (though the latency of about $10^3 t$ is lower than the service time). However, communication-calculation overlapping can be actually achieved with a tree-structured multicast (implemented by all the workers themselves, onto which a binary cube is mapped), since only two *send(g)* primitives (latency of about *20 t*) are needed in each worker. Rigorously, we have only to take into account some initialization, transient delays (*20 t*).

Thus, *with a tree-structured multicast*, in practice the effective service time is equal to the ideal one (T_A) and the efficiency is equal to one (rigorously, as for the farm version, service time and efficiency are slightly lower).

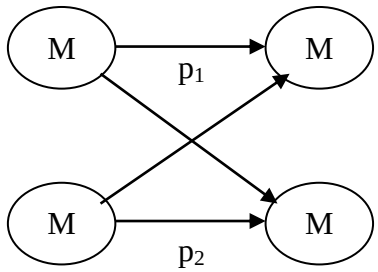
In conclusion, we can affirm that the choice of partitioned *C* is the best because the memory occupancy is minimized and, at the same time, the impact on partitioning on the service time (for the second operation, executed with probability close to one) is quite negligible.

The data memory size per node (partitions of *A, B, C*) is only *3g* = 300 words. Of course, any data cache is able to actually exploit the reuse on array *C*.

16.7 Exercise 7

The OR-graph computation, shown in the figure, operates on streams generated by M_{01} and M_{02}.

Modules M_{01}, M_{02}, M_1, M_2 have ideal service times T_{01}, T_{02}, T_1, T_2, respectively. All the communications are overlapped to internal calculations. Let p_1 (p_2) the probability that the result of an execution of M_{01} (M_{02}) is forwarded to M_1 (M_2).



a) Determine the condition on the set of parameters (T_{01}, T_{02}, T_1, T_2, p_1 p_2) in order that the best bandwidth as possible is achieved.

b) Evaluate the ideal and the effective bandwidth, with condition *a)* verified.

c) Evaluate the relative efficiency of the whole computation and of the single modules, with condition *a)* verified.

The given computation has ideal bandwidth:

$$B_{\Sigma-id} = \frac{1}{T_{01}} + \frac{1}{T_{02}}$$

since

- this is the aggregate bandwidth of the sub-computation represented by modules M_{01} and M_{02}, which generate the streams and operate in parallel, and
- the effective service times of M_{01} and M_{02} are equal to the ideal ones (the communications are fully overlapped).

The ideal bandwidth can be actually achieved provided that both M_1 and M_2 are not bottlenecks, i.e.

$$\rho_1 \leq 1 \textit{ and } \rho_2 \leq 1.$$

Let us apply, for the transient behavior, the server partitioning theorem and the multiple client theorem to evaluate the interarrival times to M_1 and M_2. The no-bottleneck condition is expressed as:

$$\begin{cases} \dfrac{1}{T_{A1}} = \dfrac{p_1}{T_{01}} + \dfrac{1-p_2}{T_{02}} \leq \dfrac{1}{T_1} \\ \dfrac{1}{T_{A2}} = \dfrac{1-p_1}{T_{01}} + \dfrac{p_2}{T_{02}} \leq \dfrac{1}{T_2} \end{cases}$$

Under this condition:

$$B_\Sigma = B_{\Sigma - id}$$

$$\varepsilon_\Sigma = 1$$

Since the effective service times of M_{01} and M_{02} are equal to the ideal ones:

$$\varepsilon_{01} = 1$$

$$\varepsilon_{02} = 1$$

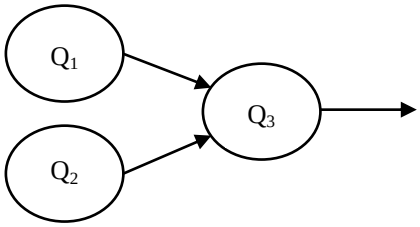
Moreover, since M_1 and M_2 are not bottlenecks:

$$\varepsilon_1 = \rho_1 = T_1 \left(\frac{p_1}{T_{01}} + \frac{1-p_2}{T_{02}} \right)$$

$$\varepsilon_2 = \rho_2 = T_2 \left(\frac{1-p_1}{T_{01}} + \frac{p_2}{T_{02}} \right)$$

16.8 Exercise 8

Consider the following parallel computation Σ having an OR-graph structure:



Module Q_1 (Q_2) encapsulates an integer array $A_1[M]$ (an integer array $A_2[M]$), with $M = 10^4$, and generates a stream, of length M, whose elements are the elements of A_1 (of A_2).

For each stream element, Q_3 executes an integer function F with processing time equal to $50t$, where t is a conventional time unit whose value is much greater that the instruction service time of the processing nodes.

a) Determine a parallel version of Q_3 such that Σ achieves the best processing bandwidth as possible on a parallel architecture having N = 128 nodes with communication processor, $T_{setup} = 0.9t$ and $T_{transm} = 0.1t$.

b) Evaluate the service time and the relative efficiency of Σ, of Q_1, of Q_2 and of Q_3, and the completion time of Σ.

c) Verify whether a different, sequential implementation of Q_1 and of Q_2 exists able to improve the completion time of Σ, and, if so, evaluate such completion time.

a) According to the assumption about t, a very good approximation of the interdeparture time of Q_1 and of Q_2 is $T_{send}(1) = t$ (*add explanations*).Thus, according to the multiple clients theorem, the interarrival time to Q_3 is

$$T_A = \frac{t}{2}$$

Q_3 can be parallelized as a *farm* (the only possible solution according to the specifications), whose optimal parallelism degree would be:

$$n_{id} = \frac{Tcalc}{T_A} = 100$$

(communications of workers are fully overlapped to calculation).

However, this degree of parallelism is not feasible, because the emitter is a bottleneck

$$T_E = T_{send}(1) = t > T_A$$

Q_3 remains as bottleneck. Thus, the best service time of Q_3, and of Σ, that we can achieve is equal to $T_E = t$, with an effective degree of parallelism :

$$n = \frac{Tcalc}{T_E} = 50$$

which is compatible with the given number *N* of nodes.

b) The ideal service time of Σ is *t/2*. The effective service time of Q_1 and Q_2 becomes *2t*. The stream length is equal to *2M*. Thus, according to Part 1, Sect. 5.3, 5.4, the required performance metrics are (*add explanations*):

$$T_\Sigma = t \qquad T_{Q1} = 2t \qquad T_{Q2} = 2t \qquad T_{Q3} = t$$

$$\varepsilon_\Sigma = \frac{1}{2} \qquad \varepsilon_{Q1} = \frac{1}{2} \qquad \varepsilon_{Q2} = \frac{1}{2} \qquad \varepsilon_{Q3} = 1$$

$$T_{c\Sigma} \sim 2\,M\,t$$

c) A coarser grain of the messages is potentially able to improve the completion time. Let *L* be the number of array elements belonging to the same stream element generated by Q_1 and Q_2. We can apply the method of Part 1, Sect. 9 (*add proper explanations*) with $T_E = T_{send}(L) = T_{setup} + L\,T_{transm}$, and $T_{calc} = 50\,L\,t$. The result is

$$L = 4 \qquad \text{corresponding to} \qquad n = 153$$

However, since *N* = 128, we must reduce the degree of parallelism (number of workers) to

$$n = N - 4 = 124$$

corresponding to

$$L = 3$$

$$T_{c\Sigma} \sim \frac{2M}{L} T_{send}(L) \sim 0{,}8\,M\,t$$

17. Further 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) *int A[M]; //G is an associative function//*

```
{ ∀ i = 0 .. M – 1
        if ( reduce (A, G) > 0 ) then
                { x = A[i] ;
                  ∀ j = 0 .. M – 1
                        A[j] = F (A[j] , x)
                };
}
```

2) Module operating on input stream R; S is local.

```
int R[M], S[M]; int result;
∀ i = 0 .. M–1
        ∀ j = 0 .. M–1
                S[i] = F (R[j, S[i]);
result = reduce (S, +)
```

3) *int A[M][M], B[M][M], C[M][M];*

$$\forall\, i,j = 0\,..\,M-1\colon C_{ij} = \sum_{k=0}^{M-1} F(A_{ik}\,, B_{kj})$$

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.

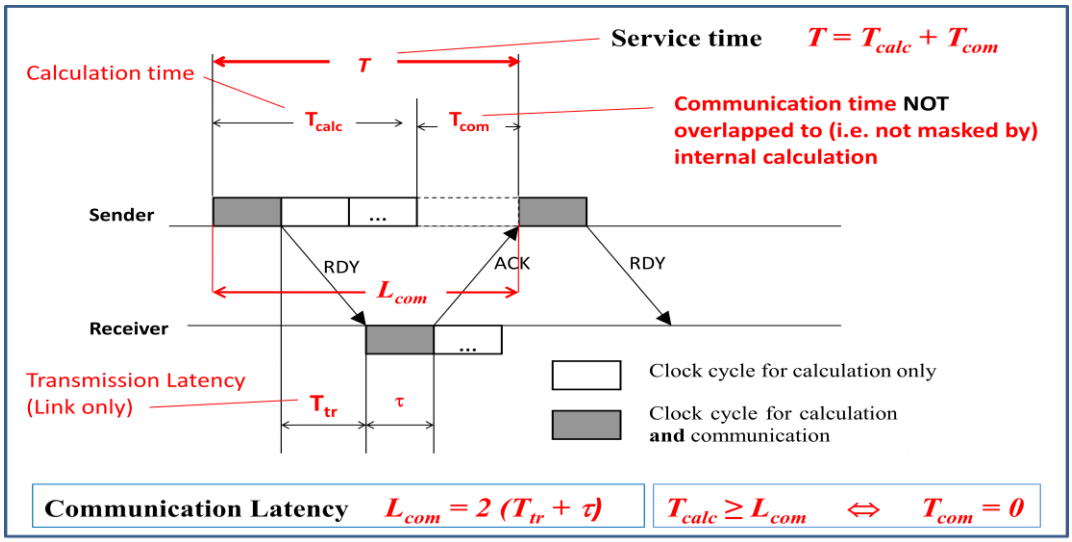
18. Parallel systems at the firmware level

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 inherent communication overlapping.

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 (Part 0, Section 2.4). 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. 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 3.3.

The **service time** of the communicating module is increased by the communication time not overlapped to calculation, T_{com}:

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

$$T_{com} = \begin{cases} L_{com} - T_{calc} & if\ L_{com} > T_{calc} \\ 0 & otherwise \end{cases}$$

Equivalently:

$$T = T_{calc} + T_{com} = max\ (T_{calc},\ L_{com})$$

The specificity at the firmware level is given by the **communication latency** formula:

$$L_{com} = 2\,(\tau + T_{tr})$$

For example, if $T_{tr} = 5\tau$ (typical value for medium-length *inter-chip* links), a calculation with $T_{calc} \geq 12\tau$ (relatively fine grain) is fully overlapped to communication, i.e the processing bandwidth cannot be greater than $1/12\tau$.

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, we'll see that, in a scalar pipeline CPU (Section 20), the ideal performance can be estimated as $1/2\tau$ instructions per second, for a *n*-way superscalar CPU (Section 21) as $n/2\tau$ instructions per second.

18.2 Double buffering

The communication bandwidth can be improved, with the same latency, by increasing the asynchrony degree of the communication. For $k = 2$, an elegant solution exists which does not require a separate buffer unit (Part 0, Section 2.4.2), instead modifies the interface in the communicating units.

Consider again a communication from U1 to U2. Now we use two interfaces (RDY0, MSG0, ACK0), (RDY1, MSG1, ACK1). U1 and U2 agree in alternating a communication through the first interface and a communication through the second one, i.e. this is the way in which their microprograms are written.

The net effect is that the service time becomes

$$T_{s\text{-}com} = \tau + T_{tr}$$

that is, one half of the normal solution, because two consecutive communications can be partially overlapped.

18.3 An example of firmware-level parallel computation

Let us consider again the Example of Sect. 6.3:

> 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 1τ, and *y* values are consumed by unit U_2 with calculation time 1τ.

Let the transmission latency of any link be equal to 4τ, and the stream length *m* 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:

$$L_{com} = 2\,(\tau + T_{tr}) = 10\,\tau$$

Therefore, the effective service time of U_1 and U_2 is equal to L_{com}, and $\varepsilon_1 = \varepsilon_2 = 0.1$.

The interarrival time to U, thus the system ideal service time, is equal to L_{com}:

$$T_{\Sigma-id} = T_A = 10\,\tau$$

The microprogram of U (Part 0, Section 2) 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:

$$T_{calc} = 100\,\tau$$

The optimal parallelism degree is:

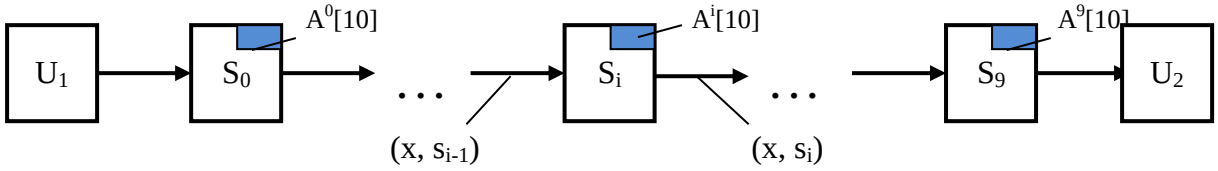
$$\bar{n} = \frac{T_{calc}}{T_A} = 10$$

For $n = \bar{n}$ we obtain:

$$T_\Sigma^{(10)} = T_{\Sigma-\mathrm{id}}^{(10)} = 10\,\tau \qquad \varepsilon_\Sigma^{(10)} = 1 \qquad T_c^{(10)} \sim m\,T_\Sigma^{(10)} = 10.000\,\tau$$

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. 6.3)

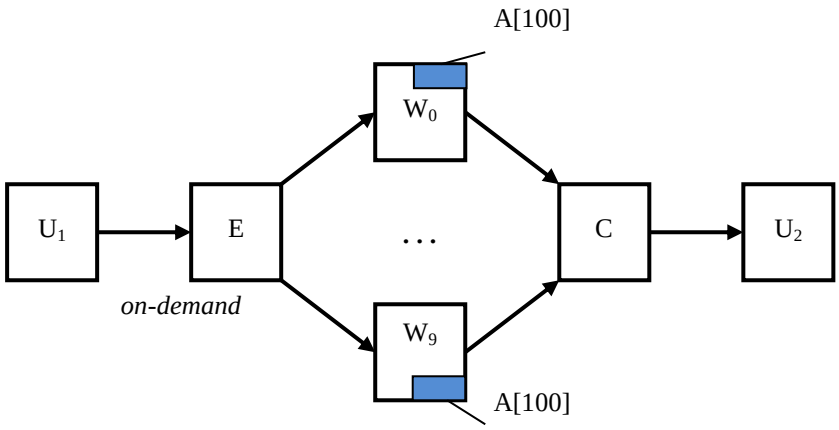


The result is:

- $n_\Sigma = \bar{n} + 2 = 12$, 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



The emitter unit is not a bottleneck, since it service time is equal to L_{com} ($\varepsilon_E = 1$). *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:

$$n_\Sigma = \bar{n} + 4 = 14$$

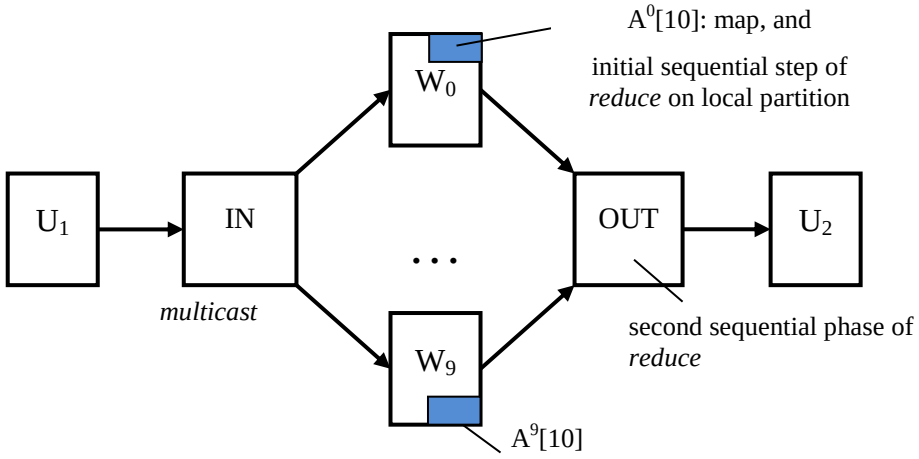
The memory capacity is 10 times larger than the sequential realization.

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:

$$L_\Sigma^{(10)} = (\tau + T_{tr}) + (\tau + T_{tr}) + (100\,\tau + T_{tr}) + (\tau + T_{tr}) + \tau = 104\,\tau + 4\,T_{tr} = 120\,\tau$$

c) Data-parallel: map + reduce with statically partitioned A and multicasted x



IN (service time L_{com}) and OUT (service time: 10τ for second phase of reduce) are not bottlenecks. Thus, the service time is equal to the ideal one, with:

$$n_\Sigma = \bar{n} + 4 = 14$$

In this computation load balancing is achieved, due to the zero-variance computation.

The whole memory capacity is equal to the sequential realization.

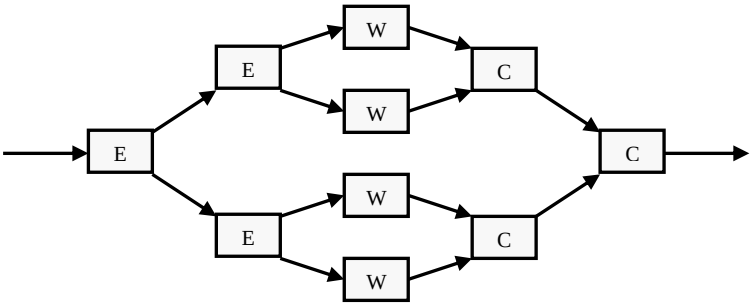
The latency is the minimum for all possible realizations:

$$L_{\Sigma}^{(10)} = (\tau + T_{tr}) + (\tau + T_{tr}) + (10\,\tau + T_{tr}) + (10\,\tau + T_{tr}) + \tau = 23\,\tau + 4\,T_{tr} = 39\,\tau$$

18.4 Interconnection structures for farm e data-parallel systems

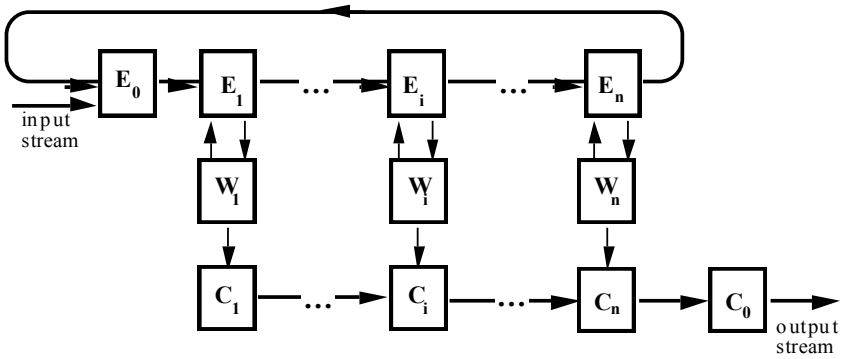
In the firmware realization of farm and data parallel systems, some functionalities are critical from the *pin-count* problem viewpoint, if implemented by single units: emitter, collector, scatter, gather, multicast. On the other hand (Part 0, Section 2.4.5), interconnection structures for *highly parallel* 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 interesting 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 tasks 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.

19. Instruction Level Parallelism architectures

A general-purpose computer with a sequential organization, i.e. able to execute only one instruction at the time, is characterized by low performance and low efficiency (see the elementary processor of Part 0, Sections 3.4, 3.5). Cache memory on chip (primary and possibly secondary cache: Part 0, Section 5.2) is a very important technique to reduce instruction *latency* and *service time*, 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* (ILP) 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:

- *Pipeline*: for the basic parallelization of the firmware interpreter,
- *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.

Due to the nature of this parallelization problem, computations are ***stream-based***, i.e. one or more instruction streams is generated and executed.

19.1 Basic Pipeline CPU

This is the basic starting point for ILP architectures: each element of the instruction stream is a *single* assembler instruction. Figure 1 gives a first idea of the pipeline organization of a CPU.

The principle is the ***parallelization of the firmware interpreter***, instead of having a sequential interpreter as in the elementary CPU of Part0, Sections 3.4, 3.5. The parallelization is based on the pipeline paradigm, although we'll see that the resulting architecture is not a pure pipeline, instead applies also other parallelization paradigms and the client-server interaction.

For simplicity, all the treatment of Instruction Level Parallelism will be done assuming that no double buffering is employed in the firmware communications. Thus

$$L_{com} = 2\tau$$

on chip (Section 18.1), and the *ideal instruction service time* is

$$T_{id} = 2\tau$$

The ideal Performance (instruction bandwidth) is:

$$\wp_{id} = \frac{1}{2\tau} = \frac{clock\ frequency}{2} value \quad instructions\ per\ second$$

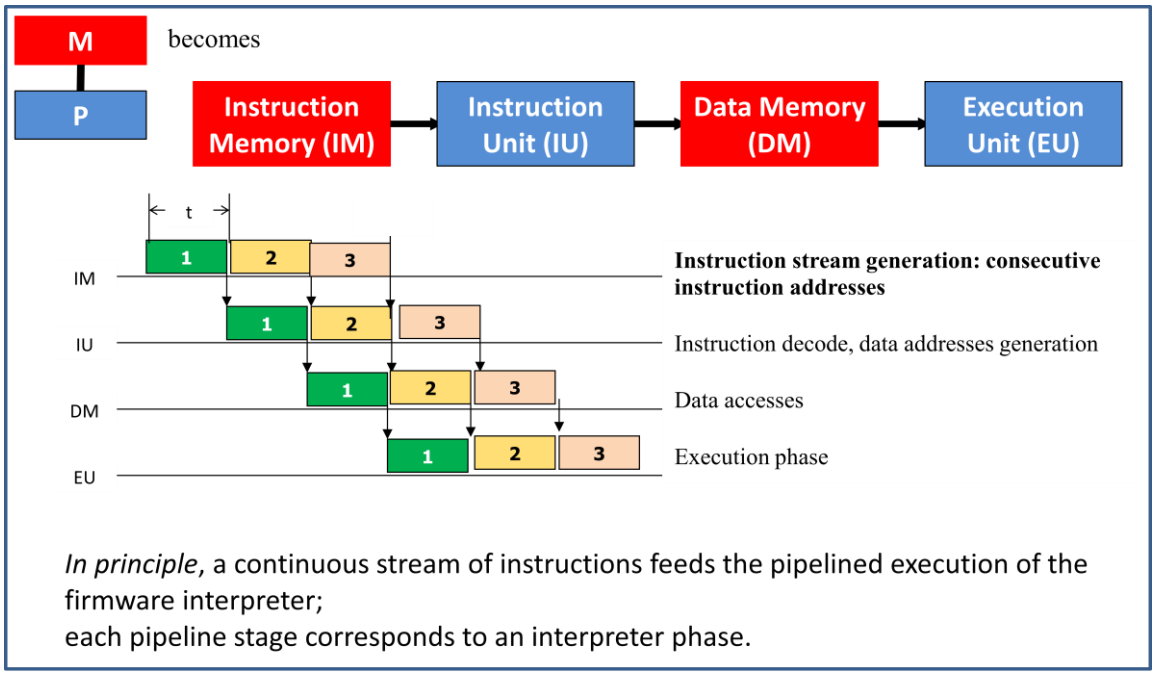


Figure 1

A more detailed pipeline CPU organization is shown in Figure 2.

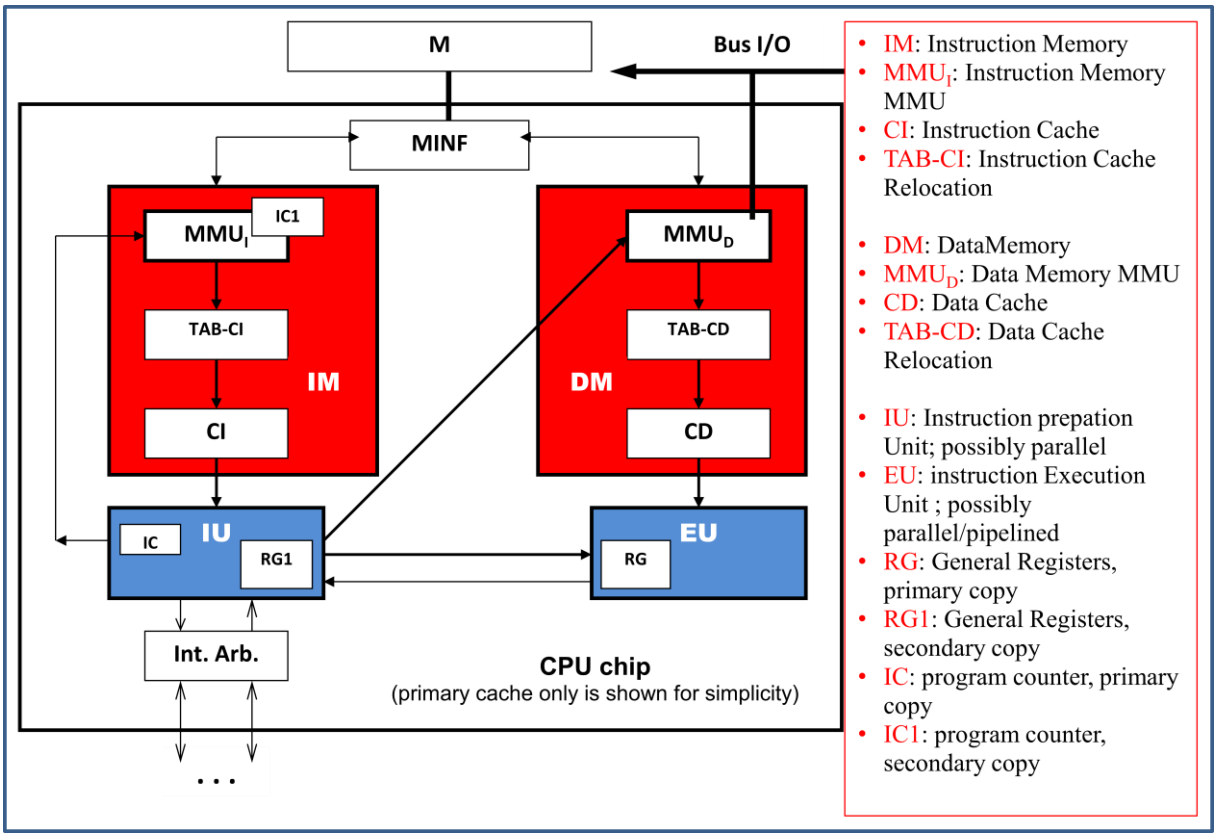


Figure 2

The ***Instruction Memory*** (IM), which is basically an instruction cache, generates the instruction stream, with an interdeparture time 2τ, *at consecutive addresses*. The ***Instruction Unit*** (IU) is able to decode and to prepare any instruction with a service time of 2τ. The ***Data Memory*** (DM), which is basically a data cache, is able to read/write a word with a service time 2τ. The ***Execution Unit*** (EU) is able to execute any arithmetic logic instruction with a service time equal to 2τ, *though its latency may be greater according to the kind of operation*.

> If double buffering is used, all the treatment is valid by merely replacing $T_{id} = t = 2\tau$ with $T_{id} = t = \tau$, thus doubling the ideal bandwidth. The assumption $T_{id} = t = \tau$ is frequent in the industrial products description.

An ***abstract architecture*** is defined in Figure 3 for a Risc machine:

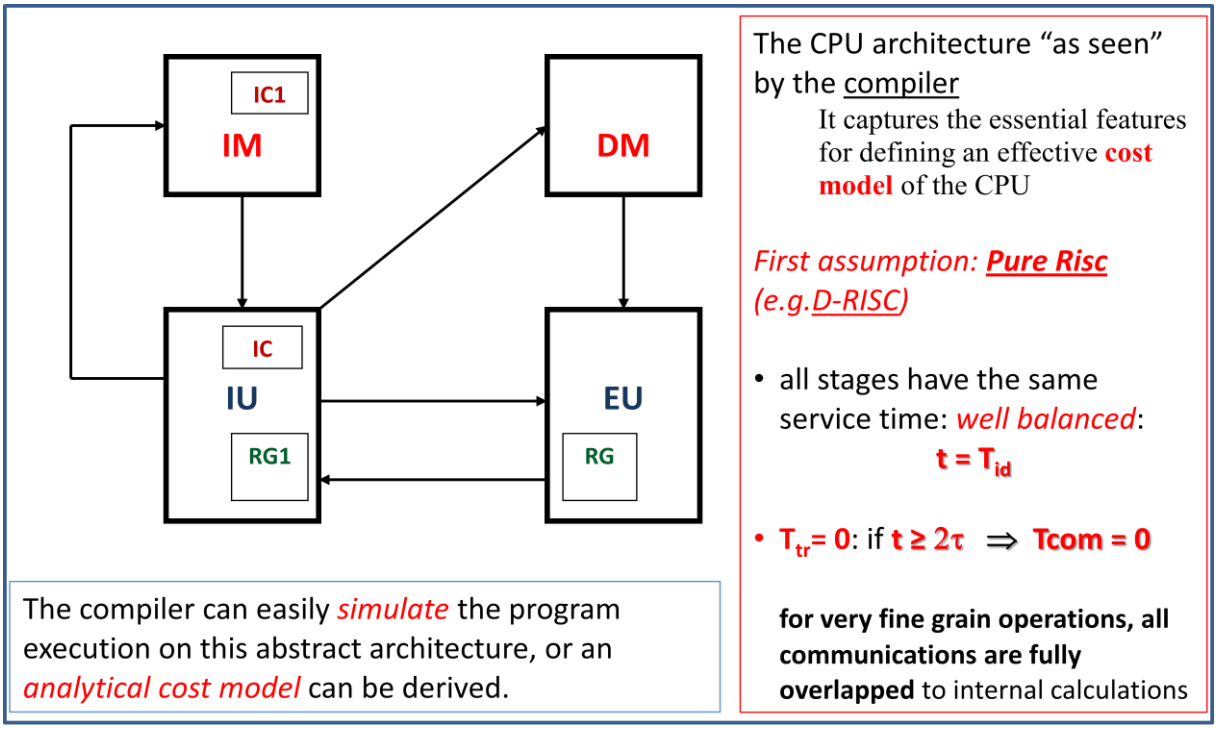


Figure 3

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).

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 the instruction service time increase. The two main performance degradations correspond to the existence of the two feedbacks in the graph:

a) ***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.

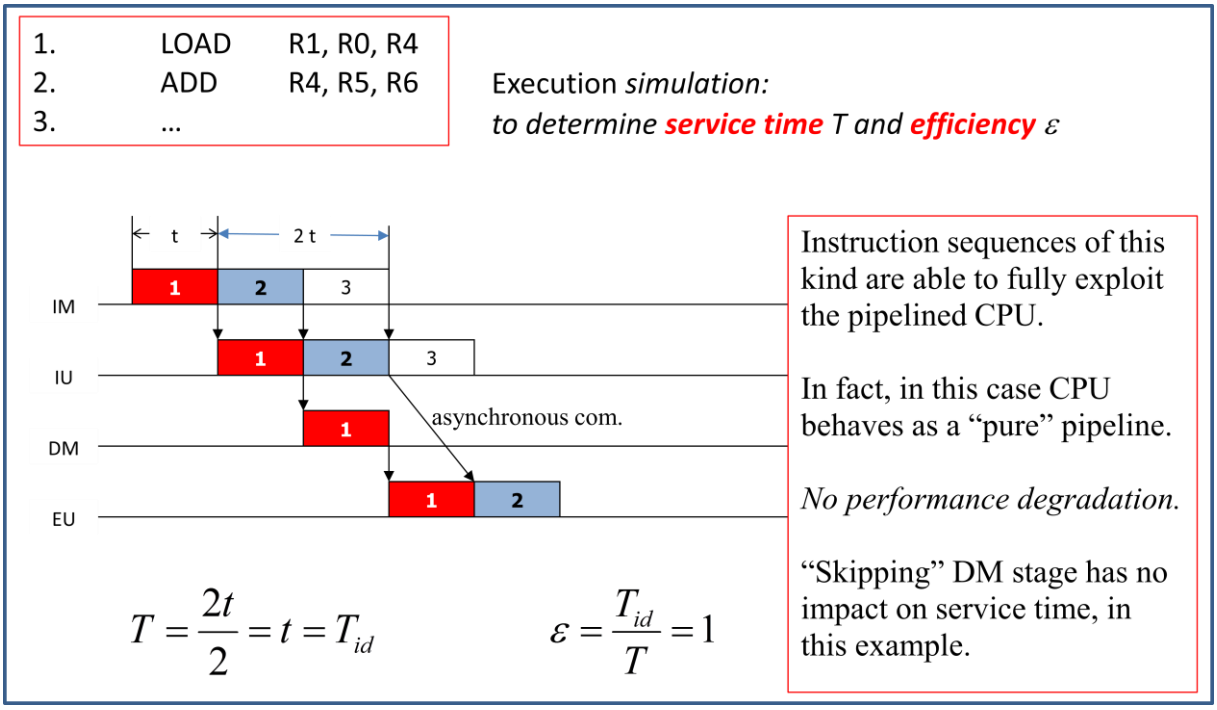


Figure 4

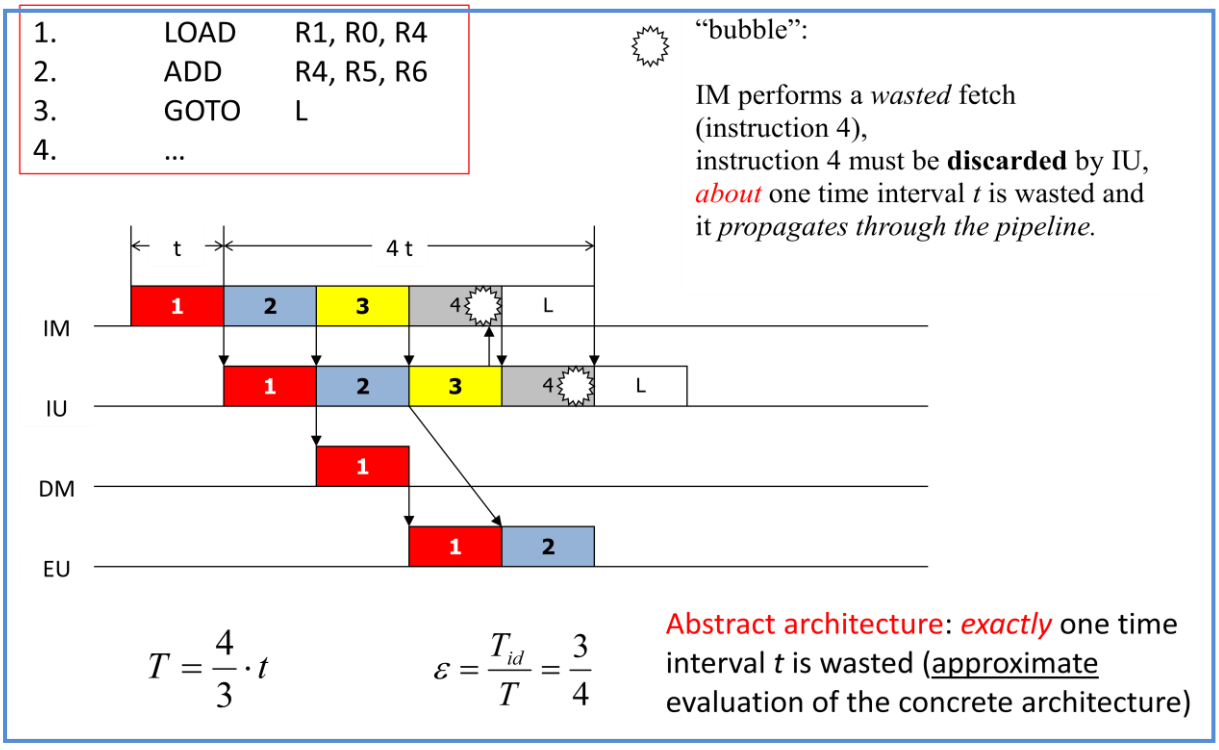


Figure 5

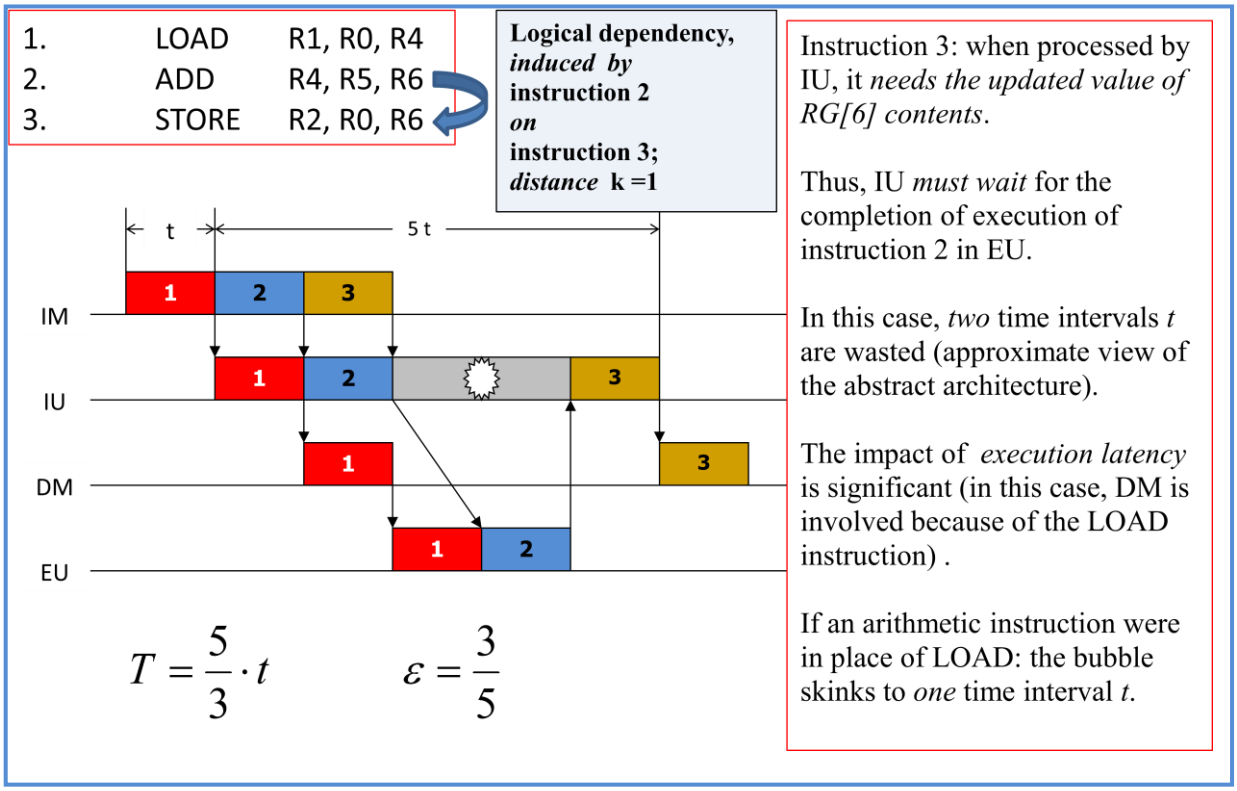


Figure 6

The general problem of performance degradations can be studied according to the general cost model for *client-server computations with request-reply behavior* (Sect. 15). 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 ILP machines. 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.

19.2 Parallel-pipelined Execution Unit

The queuing model for evaluating Pipeline CPU logical dependencies introduces an interesting and critical 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 (t) through parallelization techniques, while the latency can be reduced by optimized hardware technologies and proper parallelization. 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 (Part 0, Section 2), 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:

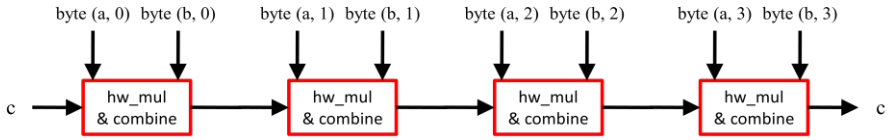
Multiply *a*, *b*, result *c*

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:



service time = 2τ, latency = 8τ

Figure 7

Thus, the service time has been minimized (2τ), ***while the latency has been only increased logarithmically*** (8τ 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.

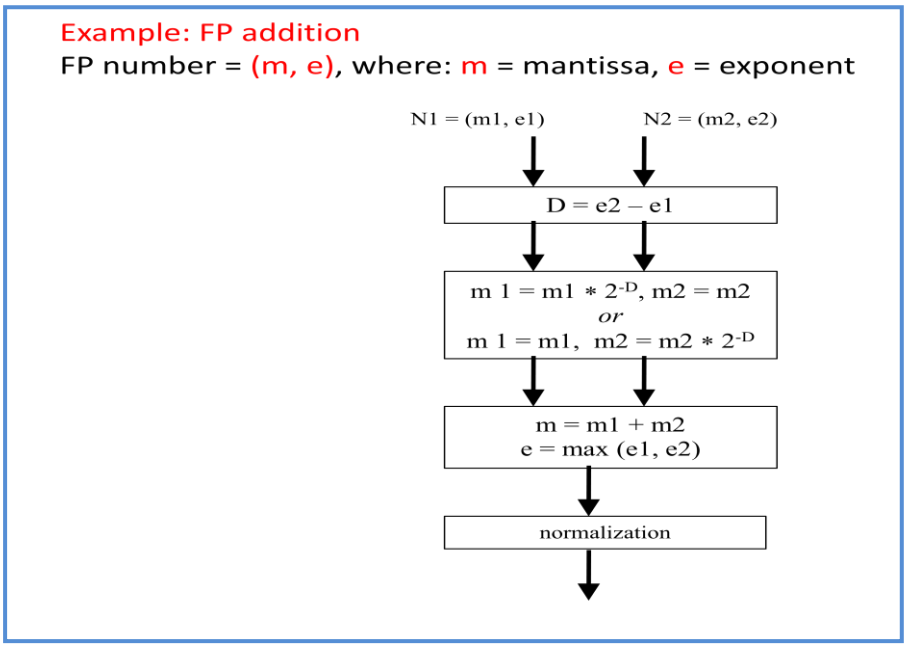


Figure 8

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 (2τ). A cost-effective solution is the *functional partitioning with independent workers* (Section 11), as shown in Figure 9.

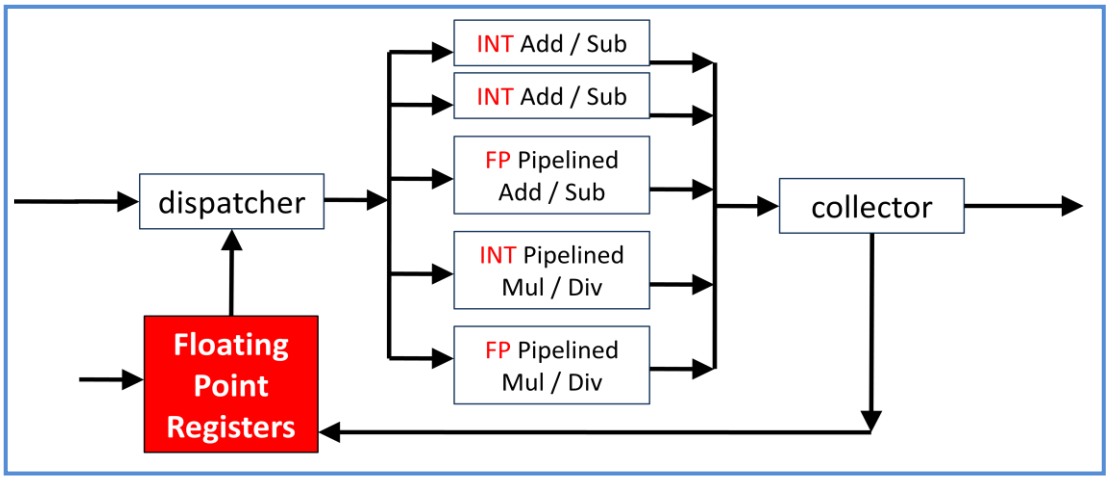


Figure 9

In Sect. 11 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 2τ. A *farm* realization (i.e. general-purpose workers, Section 10) 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 potential increase in the degradation due to logical dependencies, which now exist also ***inside EU*** (besides *between IU and EU* as discussed till now).

20. Scalar pipelined CPU architectures

In this section we review the basic issues of Section 19 in more depth, and introduce some compile time optimizations.

Before starting, let us describe an important architectural issue: synchronization mechanisms for machine state consistency.

20.1 Synchronization mechanisms IU-EU, IU-IM

Two main synchronization problems exist for ensuring consistency of machine state:

1) consistent usage of *general register copies* (main copy RG[64] in EU, secondary copy RG1[64] in IU, assuming 64 general register as in D-RISC), and

2) consistent usage of *program counter copies* (main copy IC in IU, secondary copy IC1 in IM, or more precisely in MMU_I).

The solutions are the following:

1) an array of non-negative *semaphore* registers SEM[64], initialized to zero, is associated to the array of general registers copy RG1[64] in IU. When IU processes an arithmetic or Load instruction with destination register RG[i], SEM[i] is incremented. When EU modifies RG[i], the same value is also sent to IU, where it is written into RG1[i] and SEM[i] is decremented. When IU wants to read RG[i] (e.g. to compute an address in Load/Store instructions, to test a predicate in conditional branch instructions), it waits until SEM[i] = 0. Of course, IU has a nondeterministic behavior (Part 0, Sections 2.4, 3.6.2, 6.1.4): each communication from EU is listened during any clock cycle in parallel with the "normal" behavior (analogously to interrupts);

2) when a branch is actually executed, IU modifies IC and the same value is also sent to IM (MMU_I). As soon as MMU_I, having a nondeterministic behavior, receives a new value of IC, this value is written into IC1, and a new instruction stream is initialized starting from this address. When IU receives an instruction from IM, the instruction is accepted only if it belongs to the *valid* instruction stream, otherwise it is discarded. In order to distinguish the instruction validity, a *unique identifier* is associated to each stream element, whose value has been communicated by IU when a branch target address is sent to IM. IU accepts the received instruction if the associated identifier is equal to the last identifier IU sent to IM. To implement such an identifier, a simple solution is that the unique identifier coincides with IC value itself.

20.2 Performance evaluation and optimizations

Let us consider the following program for integer array addition:

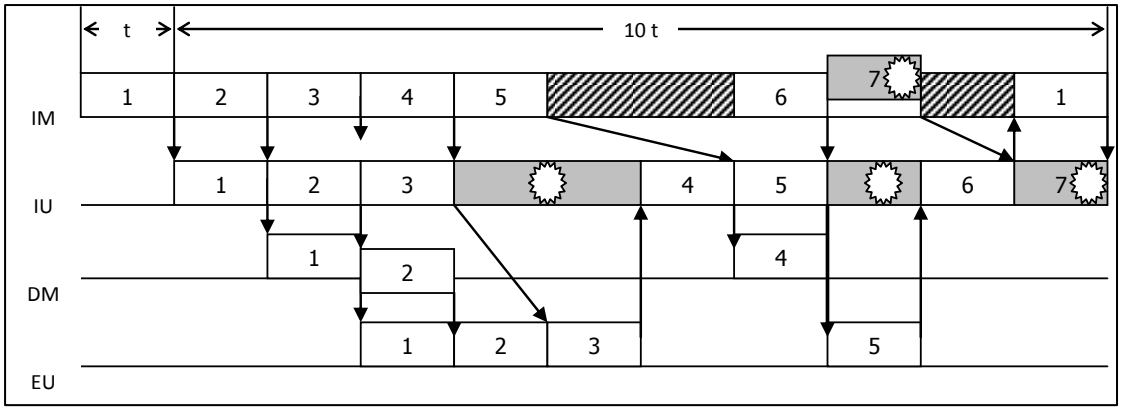
int A[N], B[N], C[N];

∀i = 0 .. N–1: C[i] = A[i] + B[i]

compiled in D-RISC as follows:

```
1.      L :     LOAD    RA, Ri, Ra
2.              LOAD    RB, Ri, Rb
3.              ADD     Ra, Rb, Rc
4.              STORE   RC, Ri, Rc
5.              INCR    Ri
6.              IF <    Ri, RN, L
7.              END
```

The execution simulation of a generic loop iteration is shown in the following figure:



Notice that, in this example, all the arithmetic operations have *unit-latency* in the execution phase (EU). In the next section the performance will be evaluated for examples using a parallel-pipelined EU.

There are degradations due to logical dependencies between instructions (instruction 3 *induces a logical dependency of distance 1* on instruction 4, instruction 5 induces a logical dependency of distance 1 on instruction 6) and due to the branch caused by instruction 6 (with probability ~ 1 for large *N*). The effective service time is (6 instructions are processed in a time interval of size *10t*):

$$T = \frac{10\,t}{6} = \frac{5}{3}\,t$$

Thus the relative efficiency of CPU:

$$\varepsilon = \frac{t}{T} = \frac{3}{5}$$

This program can be optimized at compile time. Any optimization is based on a proper *reordering of instructions* ("code movements"), in such a way that the performance degradation causes are alleviated (possibly eliminated), while preserving the program semantics.

Logical dependency degradations can be reduced by trying *to increase the distance* between the instruction inducing the logical dependency (for example, 5) and the instruction affected by the logical dependency (for example, 6).

In our example, INCR can be anticipated in order to increase the distance between INCR and IF:

```
1.      L :     LOAD    RA, Ri, Ra
2.              LOAD    RB, Ri, Rb
3.              INCR    Ri
4.              ADD     Ra, Rb, Rc
5.              STORE   RC, Ri, Rc
6.              IF <    Ri, RN, L
7.              END
```

on condition that the base address of C is initialized to the real value minus one.

In so doing, we have eliminated the effect of the dependency induced by INCR on IF, while the dependency of ADD on STORE remains. It can be verified that (see the execution simulation):

$$T = \frac{3}{2}\,t \qquad \varepsilon = \frac{2}{3}$$

Branch degradations can be alleviated in several ways, for example by *loop unfolding* or *in-line procedure expansion* (in order to eliminate some branch instructions), or by code movement. In the last case, an interesting technique, called ***delayed branch***, consists in instruction reordering in such a way that the "bubble" caused by the branch is actually filled with useful work (of course without altering the program semantics). An instruction *annotation* ("delayed_branch") must be provided at the assembler level machine, according to which (at the firmware level) the IU interpreter does not discard the instruction received after the branch instruction (as it happens when the delayed branch technique is not applied).

In our example, the STORE instruction can be moved after the IF instruction, so that STORE is usefully fetched during the time slot after IF (thus, no bubble actually occurs, without modifying the program semantics):

```
1.      L :     LOAD    RA, Ri, Ra
2.              LOAD    RB, Ri, Rb
3.              INCR    Ri
4.              ADD     Ra, Rb, Rc
5.              IF <    Ri, RN, L, delayed_branch
6.              STORE   RC, Ri, Rc
7.              END
```

In this way, only the logical dependency of distance 2 of INCR on IF remains, with no branch degradation. The dependency of ADD on STORE has no effect, because it is "interleaved" with the dependency of INCR on IF: the delay introduced in IU in the IF instruction delays the reading of STORE too, so that STORE finds the operands ready. By the execution simulation, we can verify that:

$$T = \frac{4}{3}\,t \qquad \varepsilon = \frac{3}{4}$$

with a notable improvement. For example, the completion time passes from the non-optimized version

$$T_c = 6\,N\,T = 6\,N\,\frac{10}{6}\,t = 10\,N\,t = 20\,N\,\tau$$

to the final optimized version:

$$T_c = 6\,N\,T = 6\,N\,\frac{8}{6}\,t = 8\,N\,t = 16\,N\,\tau$$

For example, with $N = 10^9$ and $\tau = 0{,}4$ nsec, T_c passes from 8 sec to 6,4 sec.

20.3 Data dependences

Formally, the feasibility of optimizations is based on the application of *Bernstein conditions*, to discover ***data dependences*** between instructions. As studied in Section 9, in general a sequential computation

$$F_1 (D_1) \rightarrow R_1 \; ; \; F_2 (D_2) \rightarrow R_2$$

can be transformed into a computation in which F_1 and F_2 are executed simultaneously, *or* in the reverse order, if

$$\begin{cases} R_1 \cap D_2 = \varnothing \\ R_1 \cap R_2 = \varnothing \\ R_2 \cap D_1 = \varnothing \end{cases}$$

Conversely, the conditions for the existence of data dependences are, in general:

$$\begin{cases} R_1 \cap D_2 \neq \varnothing & \textit{true dependence (Read-after-Write)} \\ R_1 \cap R_2 \neq \varnothing & \textit{output dependence (Write-after-Write)} \\ R_2 \cap D_1 \neq \varnothing & \textit{antidependence (Write-after-Read)} \end{cases}$$

The terminology associated to the kinds of data dependences reveals important aspects of their meaning.

The *true dependences* are the fundamental ones to ensure computation consistency at any level and for any computational model and architecture.

Output dependences are fundamental too. However, in the ILP case, this kind of dependence is not so critical to be avoided if a sufficiently large number of general registers is available (as in Risc machines) and they are allocated properly. When only few registers exist (as in many Cisc machines), register reuse is forced and the output dependences become quite critical. In these cases, since the compiler is not able to avoid all possible output dependences, the so-called *Register Renaming* technique is applied. It consists in a *dynamic allocation* of assembler-visible registers into a much larger number of additional firmware-visible registers. In our architectural model, this technique can be implemented in EU Master using suitable firmware resources, such a mapping table.

In the ILP case, *antidependences* deserve a special attention. As it happens in parallel micro-operations (Part 0, Section 2), in ILP architecture this kind of dependence is *a source of useful parallelism*, instead of being an impediment to parallelism exploitation. For example, the following sequence

j. LOAD RX, Ri, Rx

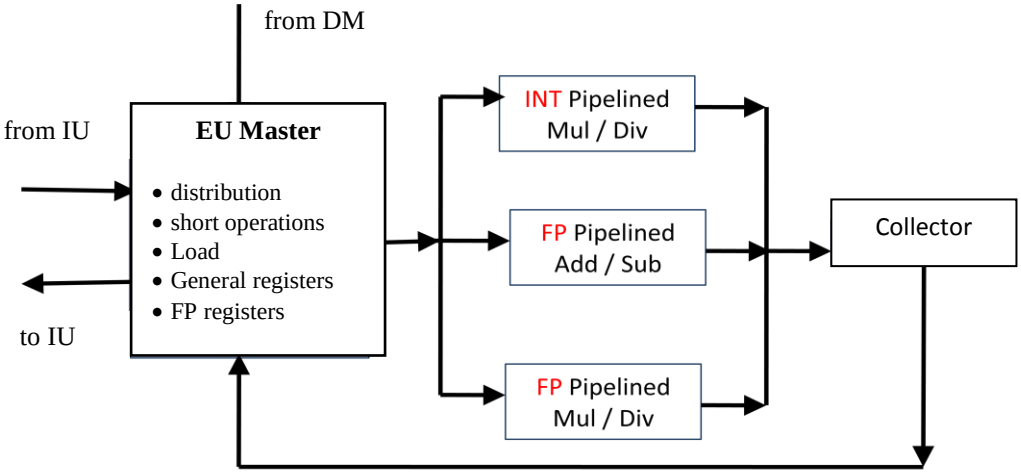
j-1. INCR Ri

is affected by an antidependence on register RG[Ri]. However, in an ILP machine instructions *j* and *j+1* are not dependent each other: as soon as instruction *j* is initiated by IU, the value of RG[Ri] is saved or, better, it is used to compute the address RG[RX] + RG[Ri]. Thus, when EU will modify RG[Ri] in executing instruction *j+1*, this modification has no effect on the correct IU state. In conclusion, the compiler can usefully introduce antidependences IU-EU in the optimized code.

20.4 Impact of parallel-pipelined EU on performance and optimizations

Let us study the performance impact of a parallel-pipelined EU to execute *fixed-point multiplication/division* operations. Let us assume (as it is realistic) that EU is has a parallel-pipeline structure, where the fixed-point multiplication/division operations are implemented with 4 pipeline stages.

The *dispatcher unit* (Section 19. 2) is in charge of directly executing "short" arithmetic instructions, like addition/subtraction, and the final part of Load instructions. The dispatcher unit will also be called ***EU Master*** unit:



Fixed and floating point General Registers are contained in EU Master, that dispatches the proper operand values to the arithmetic pipeline when needed, updates registers according to the received results, and sends such results to the Instruction Unit for consistent local update.

Let us consider the following program:

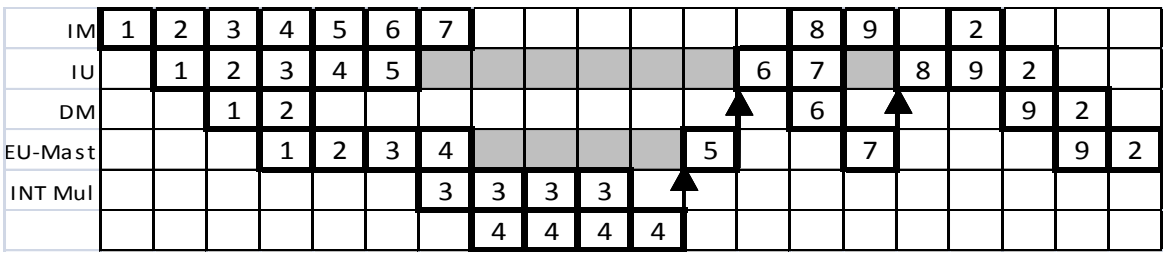
$$int\ A[N], B[N], C[N];$$

$$\forall i = 0\ ..\ N–1: C[i] = A[i] * B[i] + A[i] / B[i]$$

compiled in D-RISC as follows, by using only the delayed branch optimization:

```
1.          LOAD   RA, Ri, Ra
2. LOOP:    LOAD   RB, Ri, Rb
3.          MUL    Ra, Rb, Rx
4.          DIV    Ra, Rb, Ry
5.          ADD    Rx, Ry, Rc
6.          STORE  RC, Ri, Rc
7.          INCR   Ri
8.          IF <   Ri, RN, LOOP, delayed_branch
9.          LOAD   RA, Ri, Ra
```

The execution simulation of a generic loop iteration is the following:



A *single* pipelined fixed-point unit (*INT Mul* in the figure) is sufficient to execute *in parallel* the multiplication (3) and the division (4), because they are fully *independent* (the reader can verify such independence through the application of Bernstein conditions).

In addition to logical dependencies induced by EU on IU, called ***IU-EU logical dependency*** (in the example, 5 on 6 and 7 on 8), with a parallel EU we have also to deal with ***EU-EU logical dependencies***. In our example, MUL and DIV induce (actually, DIV induces) an EU-EU dependency on ADD (5).

In general, *EU-EU dependencies have effect on IU-EU dependencies*, according to the ordering of some instructions: in fact, the *latency* of the pipelined operations introduce a further increase of the EU *response time*, with respect to the ideal case in which such operation has unit latency. In general, the net effect is a greater delay affecting the IU-EU dependencies. In our example, the delayed execution of 5 (waiting in EU Master for the result of 4, because of the EU-EU dependency induced by 4 on 5) sensibly increases the bubble in IU caused by the dependency of 5 on 6.

On the contrary, the IU-EU dependency induced by 7 on 8 is not affected by the parallel-pipelined EU structure, since instruction 7 is "short" (executed by EU Master with unit latency) and there is no residual effect of "long" operations (which have been completed, because 7 is executed after 5).

The performance metrics are:

$$T = \frac{15}{8}\,t \qquad \varepsilon = \frac{8}{15}$$

The reader is invited

- to compare these metrics with the corresponding ones in the ideal case of a CPU with unit-latency EU;
- to introduce additional optimizations against logical dependencies too, and to evaluate their effect.

20.5 Cost model for Risc scalar CPUs

The cost model determines the effective instruction service time *T* of a program compiled for the abstract architecture. It avoids the need of a graphical simulation, yet achieving the same approximation. For a Risc machine, like D-RISC o similar, the cost model is of low complexity and simple to be applied.

First, let us evaluate the impact of *branches.*

In the abstract architecture approximation, a (non delayed) branch causes a bubble of latency *t*. If λ denotes the *branch probability*, the service time in presence of branches is:

$$T = (1-\lambda)\cdot t + \lambda \cdot 2t = (1+\lambda)\cdot t$$

The most important contribution of the cost model is the analytical evaluation of the *logical dependencies* degradation. Let Δ be the average delay in IU processing due to logical dependencies. Thus, the service time general formula is

$$T = (1+\lambda)\cdot t + \Delta$$

The CPU relative efficiency is:

$$\varepsilon = \frac{T_{id}}{T} = \frac{t}{(1+\lambda)\cdot t + \Delta} = \frac{1}{1+\lambda+\frac{\Delta}{t}}$$

From Section 19.1 we know that conceptually:

$$\Delta = d\, R_Q$$

where d is the logical dependency probability and R_Q the EU response time to instructions affected by logical dependencies:

$$R_Q = W_Q(\rho) + L_S$$

An analysis of R_Q based on general Queueing Theory is too complex for practical utilization. For example, interarrival and service time distributions are not known, and/or very difficult to predict. However, at least for ***Risc*** machines it is possible to derive a simple, yet powerful, cost model without the direct application of Queueing Theory formulas. The cost model is applied to any program.

Without losing generality, we derive the cost model for D-RISC machines.

In order to understand the principle according to which the cost model is determined, let us start with the basic case of machines with unitary latency EU. After that, we will generalize the model to machine with parallel-pipelined EU.

Unitary latency EU

In the formula

$$R_Q = W_Q(\rho) + L_S$$

the server (EU) latency is just:

$$L_S = t$$

The waiting time of instructions in the IU-EU (logical) queue Q can be evaluated as:

$$W_Q = (N_Q - k)\, t$$

where

- k is the logical dependency distance,
- N_Q is the average number of instructions waiting in the {Q, EU} subsystems.

In fact, if $k = 1$ a logical dependency is "solved" (i.e., IU is unblocked) when *all* the instructions currently in {Q, EU} have been executed, because the instruction inducing the logical dependency is the last instruction currently in the queue. If $k > 1$, IU has to wait for the execution of the $(N_Q - k)$-th instruction in the queue.

By considering all the logical dependencies in a program, the evaluation of Δ is given by:

$$\Delta = t \cdot \sum_{k=1}^{\bar{k}} d_k \cdot \left(N_{Q_k} + 1 - k\right)$$

where d_k is the probability of a logical dependency of distance k, N_{Qk} the respective N_Q, and $\bar{k}$ is the maximum value of k for which $d_k > 0$ and $N_{Q_k} + 1 - k \geq 0$.

Now, we need a method for evaluating N_Q. For a D-RISC machine with unitary latency EU, there are very few distinct cases to be evaluated, thus we derive N_Q simply by enumeration, using the graphical simulation to evaluate each of them.

For this purpose, let us define the ***critical sequence*** of an instruction as follows:

- let the instruction I_{j-k} induce a logical dependency on I_j with distance *k*;
- the critical sequence of I_j, $CS(I_j)$, is the sequence of instructions ending with I_{j-k} and consisting exclusively in instructions executed by EU (arithmetic-logic, LOAD).

The distinct cases in a D-RISC program are the following. The reader is invited to verify the assertions through the graphical simulation:

- *k = 1 and $CS(I_j)$ does not contain LOAD instructions:*

$$\Delta/dk = t \quad \Rightarrow \quad N_{Qk} = 1$$

- *k = 1 and $CS(I_j)$ contains at least LOAD instruction (i.e. it is sufficient to consider CS starting with a LOAD):*

$$\Delta/dk = 2\,t \quad \Rightarrow \quad N_{Qk} = 2$$

  With respect to the previous case, NQ is increased by one because of the additional latency introduced by DM;

- *k = 2 and $CS(I_j)$ does not contain LOAD instructions:*

$$\Delta/dk = 0 \quad \Rightarrow \quad N_{Qk} = 1$$

- *k = 2 and $CS(I_j)$ contains at least a LOAD instructions:*

$$\Delta/dk = t \quad \Rightarrow \quad N_{Qk} = 2$$

- *k > 2 :*

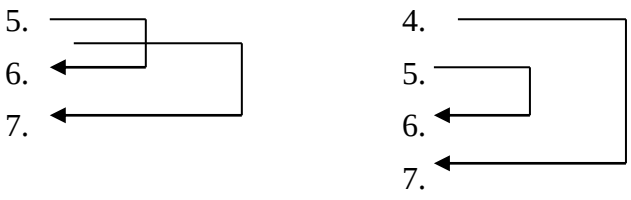
$$\Delta/dk = 0$$

In other word, the operative rule for applying the cost model is:

- *for each logical dependency, the associated $N_Q = 2$ if the critical sequence contains a LOAD, otherwise $N_Q = 1$.*

At this point, the formula for Δ can be applied.

Moreover, the cost model is completed by the evaluation of ***nested and interleaved logical dependencies***. Consider the following cases:



In the first case, 5 on 7 dependency has no effect, since the values needed by 7 in IU have been already waited in 6. In the second case, 4 on 7 dependency has no effect, since the values needed by 7 in IU have been already produced by EU before the values that unblocked 6.

Example

Let us verify the results obtained in Section 20.2 for the array addition. In the non-optimized version we have:

- a logical dependency of distance $k = 1$, with probability 1/6 and $N_Q = 2$ induced by ADD on STORE,
- a logical dependency of distance $k = 1$, with probability 1/6 and $N_Q = 1$ induced by INCR on IF.

Applying the cost model, $\Delta = 3$ t/6. Being $\lambda = 1/6$, we have $T = 10$ t/6.

In the first optimized version, without delayed branch:

- a logical dependency of distance $k = 1$, with probability 1/6 and $N_Q = 2$ induced by ADD on STORE,

Thus $\Delta = 2$ t/6. Being $\lambda = 1/6$, we have $T = 9$ t/6.

In the final optimization with delayed branch:

- a logical dependency of distance $k = 2$, with probability 1/6 and $N_Q = 2$ induced by INCR on IF;
- a logical dependency induced by ADD on STORE has no effect because it is interleaved with, and after, the dependency of INCR on IF;
- in the cost model, the application of delayed branch is evaluated as $\lambda = 0$.

Applying the cost model, $\Delta = 2$ t/6. Being $\lambda = 0$, we have $T = 8$ t/6.

Parallel-pipelined EU

The increased latency of EU has a double effect on the service time, an on Δ in particular:

1. when the instruction inducing the logical dependency IU-EU is a "long-latency" one,
2. when the instruction (long or short) inducing the logical dependency IU-EU is affected by a logical dependency EU-EU induced by a long-latency instruction. In this case, the EU-EU dependency delay is added to the IU-EU dependency delay. Moreover, there could be a chain of EU-EU dependencies whose effects are added.

The cost model is a generalization of the unitary-latency EU model, in that we separate the two effects described above (IU-EU, EU-EU). In the following

Distances and probabilities of IU-EU dependencies and of EU-EU dependencies will be denoted by symbols k and d_k and by h and d_h, respectively.

The effect of branches and the determination of N_Q remain the same of the unitary-latency EU model,

The cost model is so characterized:

1. the delay Δ is evaluated as

$$\Delta = \Delta_1 + \Delta_2$$

2. Δ_1 is part of the delay in IU caused by an IU-EU dependency, evaluated without taking into account possible EU-EU dependencies in the critical sequence, and by taking into account the latency of the instruction inducing the dependency:

$$\Delta_1 = t \sum_{k=1}^{\bar{k}} d_k \ (N_{Q_k} + L_{pipe-k} + 1 - k)$$

where $L_{pipe\text{-}k}$ is the relative latency of the instruction inducing the deoendency, measured in number of pipeline stages (e.g., 4 stages for an integer multiply/divide). For instructions executed by EU_Master directy:

$$L_{pipe\text{-}k} = 0$$

3. Δ_2 is the delay in EU_Master caused by dependencies EU-EU which
   a. belong to the critical sequence,
   b. contribute to induce the IU-EU dependency. If more than one in this situation are independent, we must evaluate the latency of their parallel data-flow execution.

   Δ_2 is given by:

$$\Delta_2 = t \sum_{h=1}^{\bar{h}} d_h \ (L_{pipe-h} + 1 - h)$$

where $L_{pipe\text{-}h}$ is defined analogously to $L_{pipe\text{-}k}$.

For *nested and interleaved dependencies*, now we must evaluate both the *external* and the *internal* one, and

$$\Delta = max\ (\Delta_{esterna}, \Delta_{interna})$$

Example

Let us verify the service time of the program of Section 20.4. The logical dependency of INCR on IF introduces a delay t/8. The logical dependency of ADD on STORE (distance k = 1, probability 1/8, N_Q = 2, $L_{pipe\text{-}k}$ = 0) has MUL and DIV in the critical sequence inducing an EU-EU dependency on ADD (distance h = 1, probability 1/8, $L_{pipe\text{-}h}$ = 4): MUL and DIV are independent and executed in parallel (in the same or in distinct functional units), thus only DIV has impact in Δ_2. We have:

Δ_1 = 2 t/ 8 Δ_2 = 4 t/8 T = 15 t/8

Example

Let us consider the following program:

int A[N], B[N], C[N];

∀i = 0 .. N–1: C[i] = (A[i] ∗ B[i] - 1) / (A[i] / B[i])

compiled in D-RISC as follows, by using only the delayed branch optimization:

```
1.          LOAD   RA, Ri, Ra
2. LOOP:    LOAD   RB, Ri, Rb
3.          MUL    Ra, Rb, Rx
4.          DECR   Rx
5.          DIV    Ra, Rb, Ry
```

```
6.          INCR   Ry
7.          DIV    Rx, Ry, Rc
8.          STORE  RC, Ri, Rc
9.          INCR   Ri
10.         IF <   Ri, RN, LOOP, delayed_branch
11.         LOAD   RA, Ri, Ra
```

The logical dependency of DIV on STORE is evaluated as follows:

Δ_1: k = 1, d_k = 1/10, N_{Qk} = 2, $_{Lpipe\text{-}k}$ = 4; thus Δ_1 = 6 t/10;

Δ_2: h = 1, d_h = 1/10, $L_{pipe\text{-}h}$ = 4; the data-flow graph of the loop body is such that (A_i*B_i - 1) and (A_i/B_i + 1) are independent and executed in parallel, thus Δ_2 = 4 t/10;

20.6 In-order *vs* out-of-order behavior: static *vs* dynamic optimizations

In all the previous examples the CPU units behave ***in-order***, that is they process instructions in the same order in which they are read from IM. For example, observe the in-order behavior of both IU and EU in the previous execution simulation.

In general, the CPU architecture contains hardware queues in front of every unit, notably a QI queue from IM to IU subsystems, a QR queue from IU to DM, a QIE queue from IU to EU, and a QD queue from DM to EU. If these queues have a FIFO discipline, in an in-order architecture a new element from a queue is not extracted until the previous one has not been fully processed by the unit.

In-order architectures rely on *static (i.e. at compile time) optimizations* to exploit the CPU resources in the most efficient way as possible.

Out-of-order *behavior* is possible: the goal is to *introduce dynamic (i.e. at run time) optimizations*. In principle, the same instruction reorderings, that are performed statically by an optimizing compiler, can be performed dynamically.

In an in-order architecture the role of data-dependence discovery is played by the compiler, in an out-of-order architecture by the firmware interpreter of the various CPU units.

The simplest form of out-of-ordering can be exemplified as follows. Let us consider the previous example with a parallel-pipelined EU. Instruction 7 could have been processed in IU before instruction 6 without altering the program semantics. That is, while IU is waiting for the result of instruction 5 as input value of instruction 6, IU observes the next instruction at the head of the QI queue (instruction 7), and, by the application of the Bernstein conditions, verifies that it is independent from instruction 6. So, instruction 7 is prepared by IU, sent to EU, and executed by EU, as soon as possible: in this way, the resolution of the logical dependency of 7 on 8 is anticipated, with a notable improvement of the service time.

The example is meaningful also because of the useful application of antidependences in

```
6.          STORE  RC, Ri, Rc
7.          INCR   Ri
```

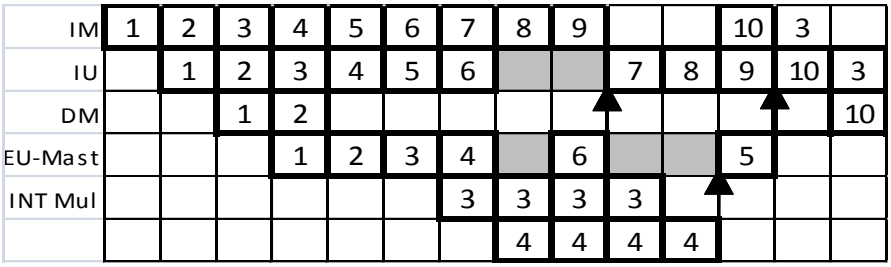
to exploit useful parallelism.

The reader is invited to show the execution simulation by applying this form of out-of-ordering.

It is interesting to know that *the same, or comparable, service time can be achieved by applying static optimizations in an in-order architecture*. In the example, we apply optimizations to logical dependencies by replicating the two LOAD instructions after the INCR, and by applying delayed branch by moving the STORE after the IF:

```
1.          LOAD   RA, Ri, Ra
2.          LOAD   RB, Ri, Rb
3. LOOP:    MUL    Ra, Rb, Rx
4.          DIV    Ra, Rb, Ry
5.          ADD    Rx, Ry, Rc
6.          INCR   Ri
7.          LOAD   RA, Ri, Ra
8.          LOAD   RB, Ri, Rb
9.          IF <   Ri, RN, LOOP, delayed_branch
10.         STORE  RC, Ri, Rc
```

The graphical simulation is similar (though not identical) to the out-of-order execution:



This "equivalence" result (between performance of statically and dynamically optimized versions aof the same program) cannot be generalized rigorously, since it is affected by too many parameters charactering the firmware-assembler architecture and the compiler optimizations. However, the example is meaningful to show the essence of the problem.

Out-of-ordering can be applied to optimize the behavior of EU too, as it can be verified on the examples.

The out-of-ordering form, which has been exemplified, consists in *preserving the FIFO queuing discipline*. It is the most simple, yet effective, form of out-of-ordering.

More aggressive forms of out-of-ordering exist in many machines: they are based on an *associative search* in the input queues, whose discipline is no more FIFO. In principle, these forms introduce more powerful optimizations. The main problem of out-of-ordering is the ***architecture complexity***, which grows very rapidly with the parallelism degree that is allowed in presence of dynamic reorderings. The firmware architecture must adopt many sophisticated, yet complex, techniques to control and to ensure the consistency of the machine and program state (registers and data cache contents), such as

- unique identifiers associated to each stream element,
- replication of registers and/or register check-pointing,
- register renaming,
- branch prediction,

as well as a set of associated firmware algorithms. Though efficiently implemented (in some cases with elegant solutions), these techniques greatly contribute to increase the CPU chip complexity, thus its size and, most important, its *power consumption*.

Coupled with the superscalar and multithreading techniques (studied in the next sections), this fact has caused the phenomenon of the *"Moore's law stop":* not because the physical implementation of regularly increasing clock frequencies is no more feasible, instead because the cost and the power consumption of CPU chips has become intolerable, and no relevant increase of performance corresponded to this cost and power consumption increase.

On the contrary, the simple *FIFO-preserving* approach to out-of-ordering does not increase the architecture complexity appreciably and often gives interesting performances, especially if used in connection with optimizing compilers. According to this approach, the compiler has a further degree of freedom in choosing the optimizations according to the program characteristics: in some cases static optimizations are effective, in other cases they are of uncertain application (e.g. in presence of many branches and data dependent loop durations). In the latter situations, the compiler can rely on the predictable behavior of FIFO-preserving dynamic optimizations too.

Last but not least: *why* complex out-of-ordering techniques and dynamic optimizations? The main motivation has a strictly market-driven nature: the requirement to utilize portable programs for which only the executable *binary* code is available (binary *legacy* code) for a specific assembler machine (e.g. x-86 or mips). Passing from a computer model to the next (with the same assembler machine), the computer player tries to exploit the technological and architectural evolutions at best, also for programs that cannot be recompiled.

Only when a technological revolution actually occurs, this trend is abandoned in favor of "new" application programs for which the source version is available. At this point, a good trade-off is represented by simpler (in-order or "controlled" out-of-order) architectures and optimizing compilers.

As a matter of fact, we are just in the middle of such a revolution, caused by the multicore approach to computer architecture and programming.

21. Superscalar CPU architectures

The superscalar CPU is an extension of the pipeline architecture in which each stream element contains *n* instructions, with *n* > 1 (*n-way superscalar*). Ideally, every time slot $t = 2\tau$, *n* instructions are fetched by IM and prepared by IU, and at most *n* instructions will be processed also by the other subsystems DM and EU. That is,

$$T_{id} = \frac{2\tau}{n} \qquad \wp_{id} = \frac{n}{2\tau}$$

Typical values of *n* are in the range *n = 2* (base case and current trend in highly parallel multicores) to *n = 4*. CPU for more powerful servers, built in the last decade, had *n = 8* or even more.

The base architecture (*2-way superscalar*) does not pose realization problems, since in two clock cycles a sequential IM is able to read two instructions, a sequential IU to prepare two instructions, DM to start reading/writing of at most two data, EU to start or to complete at most two instructions.

Higher values of *n* implies a parallel realization of instruction and data caches (interleaving, or long word).

The parallel-pipeline EU structure, with a suitable number of functional units, is able to offer the requested average bandwidth. A typical realization provides from *n/2* to *n* long fixed-point and floating point functional units.

In principle, a single-unit IU can process any number of instructions per second, since the requested functions could be implemented by combinatorial circuits. However, the complexity of such an IU increases rapidly with *n*. For high values of *n*, a parallel realization of IU is mandatory.

All the architecture complexity problems are much more tractable for Risc machines. For this reason, even Cisc machines adopt the technique to interpret a primitive Cisc instruction through a sequence of Risc-like instructions. All the more reason, out-of-ordering techniques greatly contributes to increase the architectural complexity.

With respect to the scalar architecture, performance degradations are relatively greater, i.e. performance gradually increases and efficiency sensibly decreases with *n*. This is also due to the fact that sequential programs lack parallelism even at instruction level, and the utilization factor, in the client-server queueing model of CPU, increases with *n*.

21.1 VLIW superscalar architecture

Grouping *n* instruction in a stream element could be done respecting the order of the original program code, independently of their semantics and contents. This solution is often a scarcely efficient solution. It is sufficient to think about logical dependencies, IU-EU and EU-EU, between instructions of the same stream element. Moreover, if a branch instruction is in position *m* < *n* of a stream element, then all the successive *n–m* instructions could be wasted. In fact, the branch prediction technique has been introduced just to alleviate this problem in superscalar machines.

A clean and elegant solution has been offered by the so- called ***Very Long Instruction Word (VLIW)*** model, which makes intensive use of compile-time optimizations. In VLIW terminology, a *long instruction is* the implementation of a stream element. The main feature of a VLIW long instruction is that it is composed of *n independent* instructions, of

which at most one may be a branch instruction provided that the target instruction is the first of a distinct long instruction.

In order to respect these constraints, when it is not possible to fill the long instruction with useful independent instructions, the compiler inserts a proper number of *NOP* instructions. In a sense, *NOP* are "static bubbles", i.e. bubble that cannot be eliminated and, for this reason, are recognized at compile time in order to have a predictable abstract machine and to sharply reduce the architectural complexity of IU and all the other units.

21.2 Examples

The following examples are referred to programs executed by a *2-way* superscalar D-RISC CPU.

Example 1

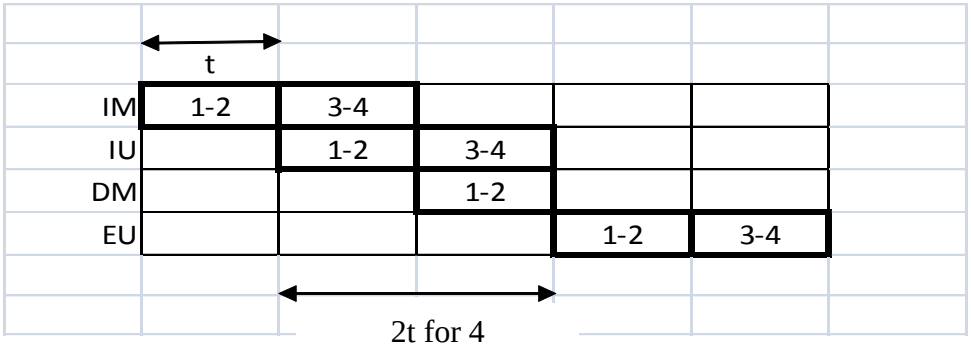
1. LOAD RA, Ri, Ra
2. LOAD RB, Ri, Rb
3. ADD Ra, Rb, Rx
4. INCR Rb

In the scalar architecture we have: $T = t = T_{id}$, and $\varepsilon = 1$.

In the 2-way superscalar architecture, let us adopt the following syntax for long instructions:

1-2. LOAD RA, Ri, Ra | LOAD RB, Ri, Rb
3-4. ADD Ra, Rb, Rx | INCR Rb

Observe that the constraint of independent instructions per long instruction is satisfied. In this example, no NPO is necessary. The execution simulation is the following:



The performance metrics are

$$T = \frac{2\,t}{4} = \frac{t}{2} = T_{id} \qquad \varepsilon = 1$$

The relative gain with respect to the scalar architecture is:

$$s = \frac{T_{scalar}}{T_{superscalar}} = 2$$

That is, we are in an ideal case in which a 2-way superscalar architecture actually doubles the performance of a scalar architecture. This is due to the absence of any performance degradation: branches, logical dependencies, *and* NOPs.

Example 2

Let us consider a program with one logical dependency IU-EU:

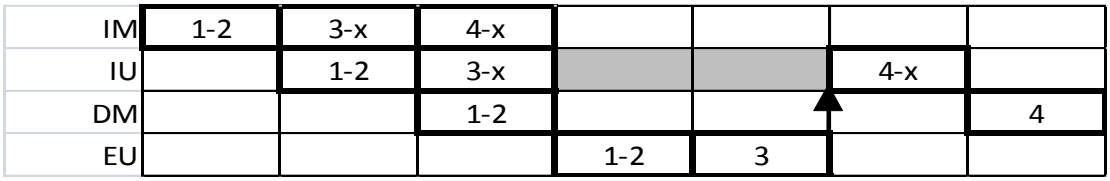
```
1.  LOAD   RA, Ri, Ra
2.  LOAD   RB, Ri, Rb
3.  ADD    Ra, Rb, Rx
4.  STORE  RC, Ri, Rx
```

In the scalar case: *T = 6 t/4, ε = 0,66.*

In the superscalar case, the following long instructions are recognized:

```
1-2.  LOAD   RA, Ri, Ra  |  LOAD   RB, Ri, Rb
3-x.  ADD    Ra, Rb, Rx  |  NOP
4-x.  STORE  RC, Ri, Rx  |  NOP
```

As shown in the graphical simulation,



the performance metrics are:

T = 5 t/4 *ε = 0,4* *s = 6/5 = 1,2*

Because of degradations (one logical dependency IU-EU *and* two NOPs), the service time, though better than the scalar case, is far from one half (*s* = 1,2), thus the efficiency is lower than the scalar case.

Example 3

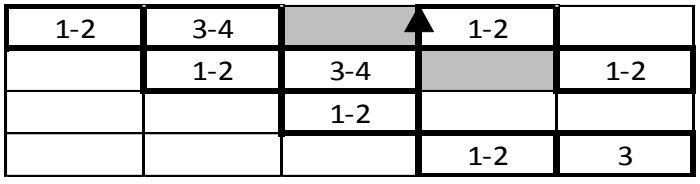
This simple benchmark contains a branch (a jump):

```
1. LOOP:  LOAD   RA, Ri, Ra
2.        LOAD   RB, Ri, Rb
3.        ADD    Ra, Rb, Rx
4.        GOTO   LOOP
```

In the scalar case: *T = 5 t/4, ε = 0,8.*

In the superscalar case:

```
LOOP:  1-2.  LOAD   RA, Ri, Ra  |  LOAD   RB, Ri, Rb
       3-4.  ADD    Ra, Rb, Rx  |  GOTO   LOOP
```



T = 3 t/4 *ε = 0,66* *s = 5/3 = 1,66*

Example 4

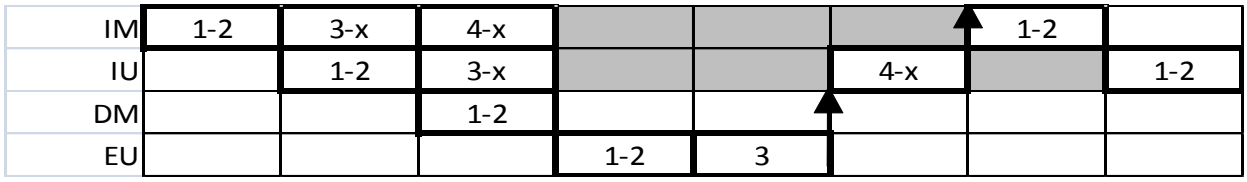
This example contains an IU-EU logical dependency and a branch:

```
1. LOOP:   LOAD   RA, Ri, Ra
2.         LOAD   RB, Ri, Rb
3.         ADD    Ra, Rb, Rx
4.         IF ≠ 0   Rx, LOOP
```

In the scalar case: *T = 7 t/4*, *ε = 0,57*.

In the superscalar case:

```
LOOP:   1-2.  LOAD   RA, Ri, Ra   |   LOAD   RB, Ri, Rb
        3-x.  ADD    Ra, Rb, Rx   |   NOP
        4-x.  IF ≠ 0   Rx, LOOP   |   NOP
```



T = 6 t/4 *ε = 0,33* *s = 7/6 = 1,17*

Example 5: vector addition

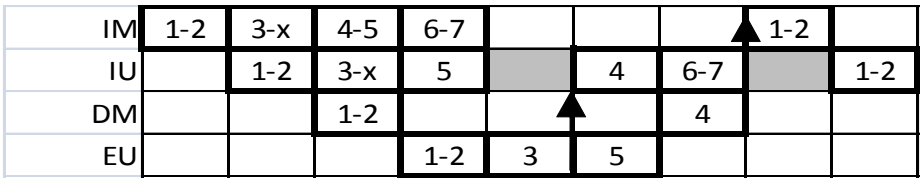
Non-optimized version:

```
LOOP:   1. LOAD   RA, Ri, Ra
        2. LOAD   RA, Ri, Rb
        3. ADD    Ra, Rb, Rx
        4. STORE  RC, Ri, Rx
        5. INCR   Ri
        6. IF <   Ri, RN, LOOP
```

In the scalar case: *T = 10 t/6*, *ε = 0,6*

In the superscalar case:

```
LOOP:   1-2.  LOAD   RA, Ri, Ra…  |   LOAD   RA, Ri, Rb
        3-x.  ADD    Ra, Rb, Rx   |   NOP
        4-5.  STORE  RC, Ri, Rx   |   INCR   Ri
        6-7.  IF <   Ri, RN, LOOP |   NOP
```



T = 7 t/6 *ε = 0,43* *s = 10/7 = 1,43*

Notice the characteristic of in-order behavior for superscalar architecture: in the long instruction 4-5, instruction 4 is blocked in IU by a logical dependency, however instruction 5 is enabled and can be forwarded to EU where it is executed (4 and 5 are independent, since an antidependence exists between them).

Optimized version:

```
LOOP:   1. LOAD   RA, Ri, Ra
        2. LOAD   RA, Ri, Rb
        3. INCR   Ri
        4. ADD    Ra, Rb, Rx
        5. IF <   Ri, RN, LOOP, delayed_branch
        6. STORE  RC, Ri, Rx
```

In the scalar case: *T = 7 t/6, ε = 0,86.*

In the superscalar case:

```
        1-2. LOAD   RA, Ri, Ra  |  LOAD   RA, Ri, Rb
LOOP:   3-4. INCR   Ri  |  ADD    Ra, Rb, Rx
        5-6. IF <   Ri, RN, LOOP, delayed_branch  |  STORE  RC, Ri, Rx
        7-8. LOAD   RA, Ri, Ra  |  LOAD   RA, Ri, Rb
```

IM	1-2	3-4	5-6	7-8			3-4
IU		1-2	3-4			5-6	7-8
DM			1-2				5
EU				1-2	3-4		

T = 5 t/6 *ε = 0,6* *s = 7/5 = 1,4*

Example 6 with parallel EU

Internal loop of matrix-vector product, with static optimizations:

```
1.          LOAD   RA, Rj, Ra
2. LOOPj:   LOAD   RB, Rj, Rb
3.          INCR   Rj
4.          MUL    Ra, Rb, Rx
5.          ADD    Rc, Rx, Rc
6.          IF <   Rj, RM, LOOPj, delayed_branch
7.          LOAD   RA, Rj, Ra
```

In the scalar case: *T = t, ε = 1.*

In the superscalar case:

```
        1-2. LOAD   RA, Rj, Ra  |  LOAD   RB, Rj, Rb
LOOPj:  3-4. INCR   Rj  |  MUL    Ra, Rb, Rx
        5-6. ADD    Rc, Rx, Rc  |  IF <   Rj, RM, LOOPj, delayed_branch
        7-8. LOAD   RA, Rj, Ra  |  LOAD   RB, Rj, Rb
```

IM	1-2	3-4	5-6	7-8			3-4			
IU		1-2	3-4	5		6	7-8			
DM			1-2							
EU-Mas				1-2	3-4					5
INT Mul						4	4	4	4	

T = 5 t/6 *ε = 0,6* *s = 6/5 = 1,2*

21.3 Cost model forRisc-VLIW superscalar CPUs

The cost model for the Risc scalar CPUs (Section 20.5) is extended to the Risc-VLIW superscalar architecture in the following way:

- $T_{id} = t/n$,
- the degradation due to static bubbles (NOP) is added.

By denoting with θ the *probability of NOPs evaluated with respect to the number of instructions in the scalar code*, the cost model becomes

$$T = \frac{t}{n}(1 + \theta) + \lambda t + \Delta$$

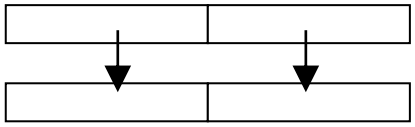
where Δ is evaluated as in Section 20.5 with the following specific rules:

1. logical dependency *distances* must be evaluated with respect to the superscalar code;

2. the logical dependency *probabilities* must be evaluated with respect to the number of instructions in the scalar code;

3. a LOAD instruction, belonging to the same long instruction of an instruction affected by logical dependency, does not belong to the critical sequence;

4. if two long instructions are bound by two or more logical dependency, they are considered one logical dependency only:

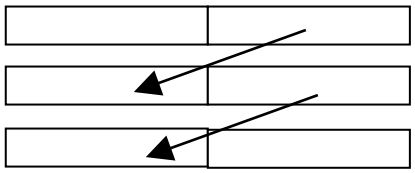


For example:

1-2. ADD Ra, Rb, Rx | INCR Ry
3-4. STORE RC, Ri, Rx | IF < 0 Ry, LABEL

k = 1, d_1 = 1/…

5 *chained* logical dependencies are evaluated as interleaved ones:



For example:

1-2. ADD Ra, Rb, Rx | NOP
3-4. STORE RC, Ri, Rx | INCR Ry
5-6. IF < 0 Ry, LABEL | …

k = 1, d_1 = 1/… dominates

Moreover:

- *antidependencies* can be contained in long instructions, since they increase the parallelism degree,
- *the delayed-branch application could require to replicate long instructions.*

Examples for 2-way superscalar machines

Let us evaluate the examples of Section 21.2 using the cost model.

Example 1

```
1-2.  LOAD   RA, Ri, Ra   |  LOAD   RB, Ri, Rb
3-4.  ADD    Ra, Rb, Rx   |  INCR   Rb
```

$\theta = 0$, $\lambda = 0$, $\Delta = 0$: T = t/2, s = 2.

Example 2

```
1-2.  LOAD   RA, Ri, Ra   |  LOAD   RB, Ri, Rb
3-x.  ADD    Ra, Rb, Rx   |  NOP
4-x.  STORE  RC, Ri, Rx   |  NOP
```

$\theta = 2/4$, $\lambda = 0$, $\Delta = 2t/4$: T = 5t/4, s = 1.2.

Example 3

```
LOOP:   1-2.  LOAD   RA, Ri, Ra   |  LOAD   RB, Ri, Rb
        3-4.  ADD    Ra, Rb, Rx   |  GOTO   LOOP
```

$\theta = 0$, $\lambda = 1/4$, $\Delta = 0$: T = 3t/4, s = 1.66.

Example 4

```
LOOP:   1-2.  LOAD   RA, Ri, Ra   |  LOAD   RB, Ri, Rb
        3-x.  ADD    Ra, Rb, Rx   |  NOP
        4-x.  IF ≠ 0 Rx, LOOP     |  NOP
```

$\theta = 2/4$, $\lambda = 1/4$, $\Delta = 2t/4$: T = 6t/4, s = 1.17.

Example 5: vector addition, optimized version

```
                1-2.  LOAD   RA, Ri, Ra   |  LOAD   RA, Ri, Rb
LOOP:           3-4.  INCR   Ri           |  ADD    Ra, Rb, Rx
                5-6.  IF <   Ri, RN, LOOP, delayed_branch  |  STORE  RC, Ri, Rx
                7-8.  LOAD   RA, Ri, Ra   |  LOAD   RA, Ri, Rb
```

$\theta = 0$, $\lambda = 0$, $\Delta = 2t/6$: T = 5t/6, s = 1.4.

Example 6 with parallel EU: internal loop of matrix-vector product, with static optimizations

```
                1-2.  LOAD   RA, Rj, Ra   |  LOAD   RB, Rj, Rb
LOOPj:          3-4.  INCR   Rj           |  MUL    Ra, Rb, Rx
                5-6.  ADD    Rc, Rx, Rc   |  IF <   Rj, RM, LOOPj, delayed_branch
                7-8.  LOAD   RA, Rj, Ra   |  LOAD   RB, Rj, Rb
```

$\theta = 0$, $\lambda = 0$, $\Delta = 2t/6$: T = 5t/6, s = 1.2.

21.4 Parallel subsystems in *n-way* superscalar architectures

With high values of *n*, in order to reduce the complexity of the various units, it is convenient to adopt parallel structures, instead of centralized ones.

The scheme in the following figure, with *n = 8*, contains four 2-way superscalar identical IUs, each one operating on two instructions of the long instruction. In this version, the program counter and the general register array, with associated synchronization semaphore registers, are centralized in a single copy (RU unit).

It can be shown that this centralization point can be eliminated, obtaining a *fully replicated version*, with a *high-bandwidth EU*.

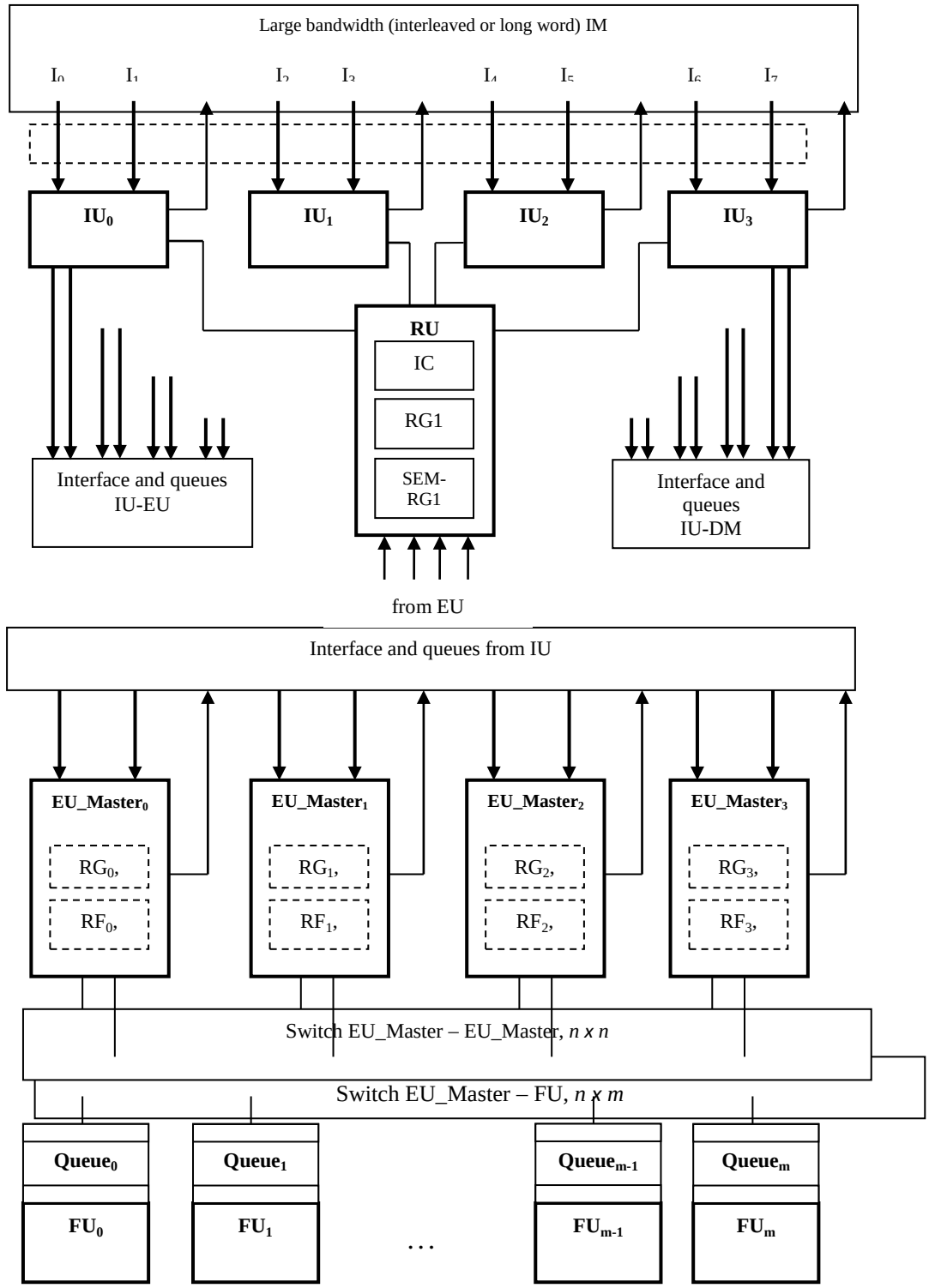
The *high-bandwidth data cache* is implemented by an interleaved structure. It can be proved that, taking into account the conflicts of distinct request to the same module, the evaluation of the data bandwidth is given by:

$$B_M = \sum_{k=1}^{n} \frac{k^2 \, (n-1)!}{n^k \, (n-k)!}$$

that can be approximated to:

$$B_M = n^{0,56}$$

with an error ≤ 4% for $n \leq 45$.


Large bandwidth (interleaved or long word) IM
I0
I1
I2
I3
I4
I5
I6
I7
IU0
IU1
IU2
IU3
RU
IC
RG1
SEM-RG1
Interface and queues IU-EU
Interface and queues IU-DM
from EU
Interface and queues from IU
EU_Master0
EU_Master1
EU_Master2
EU_Master3
RG0,
RF0,
RG1,
RF1,
RG2,
RF2,
RG3,
RF3,
Switch EU_Master – EU_Master, n x n
Switch EU_Master – FU, n x m
Queue0
Queue1
Queuem-1
Queuem
FU0
FU1
FUm-1
FUm
. . .

22. Multithreading

The transition from scalar to superscalar CPU architecture is not accompanied by proportional improvements of performance, mainly because of the limited degree of parallelism between instructions of a *single* sequential program. Degradations are caused by frequent data dependencies, and optimizations aim to mask latencies causing such dependencies. For these reasons, it has been introduced the idea of exploiting *parallelism* among instructions of *distinct* sequential programs running on the *same ILP CPU*. Provided that proper architectural supports to program concurrency are available, this idea implies that, *during the same time slot (t)* or a limited number of time slots, the CPU units are simultaneously active in processing instructions belonging to distinct programs. In other words: using instruction level parallelism to achieve program level parallelism too.

This idea, called *multithreading (MT)* for reasons to be explained successively, can be interpreted in two ways:

1. as a *technological evolution of ILP architectures*, in order to exploit the CPU resources more efficiently,
2. as an *alternative to multiprocessor architectures*, in particular multicore, or *chip-multiprocessor (CMP)*, architectures.

Point 2 deserves a first comment:

a) both MT and CMP aim to exploit the silicon chip area for the simultaneous execution of more than one sequential programs;

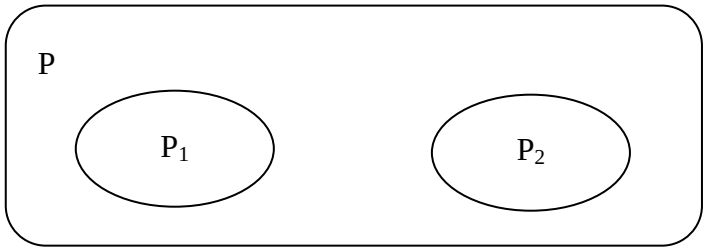
b) there is no doubt that the multiprocessor, and in particular the CMP line, is characterized by much higher degrees of parallelism, notably from 10^1 to 10^3 and over in the current trend, while a MT CPU can support only very few concurrent programs (2 – 4 in typical cases);

c) *the two lines are not in contrast*: the current trend in HPC is to have highly parallel machines, with a high number of CMP-based processing nodes, each node consisting of one or more processors, and each processor providing a limited MT capability.

In order to understand the nature of MT architecture, we just start by informally discussing the following (hard) problem: is a MT CPU, capable of executing *m* simultaneous threads, equivalent to a multiprocessor with *m* processors?

22.1 A simple example of multiprocessor *vs* multithreading

Let us consider the following simple parallel computation P, composed of two independent modules:



P_1::

1. MUL Ra1, Rb1, Rx1
2. DIV Rc1, Rd1, Ry1
3. ADD Rx1, Ry1, Rz1
4. STORE RC1, Ri1, Rz1

P_2::

a. LOAD RA2, Rj2, Rp2
b. LOAD RB2, Rj2, Rq2
c. SUB Rp2, Rq2, Rr2
d. INCR Rr2
e. STORE RD2, Rj2, Rr2

Let us execute P on two distinct architectures, realized with compatible technologies (single chip, same clock cycle, same cache, etc).

1) The first architecture is a ***multicore*** CMP with *n = 2* identical, ***scalar*** CPUs. Assuming that P_1 and P_2 are loaded in the primary caches of CPU_1 and CPU_2, the service and completion times are given by (see graphical simulation):

$$T_1 = 9\ t/4 = 2{,}25\ t \qquad T_{c1} \sim 9\ t$$

$$T_2 = 7\ t/5 = 1{,}4\ t \qquad T_{c2} \sim 7\ t$$

For P:

$$T_c = max\ (T_{c1}, T_{c2}) \sim 9\ t$$

Total number of instruction executed on the multiprocessor: N = 9

$$T \sim T_c / N = t$$

That is, CMP is equivalent to a single CPU able to execute one instruction every *t* seconds.

2) The other architecture is a single *n = 2-way* ***superscalar*** CPU with ***multithreading***. This architecture is able to execute, in the same long instruction, two instructions both belonging to P_1, or both to P_2, or one to P_1 (P_2) and the other to P_2 (P_1). For example, the generated MT code for P may be:

P::

1-a. MUL Ra1, Rb1, Rx1 | LOAD RA2, Rj2, Rp2
2-b. DIV Rc1, Rd1, Ry1 | LOAD RB2, Rj2, Rq2
3-c. ADD Rx1, Ry1, Rz1 | SUB Rp2, Rq2, Rr2
4-d. STORE RC1, Ri1, Rz1 | INCR Rr2
x-e. NOP | STORE RD2, Rj2, Rr2

with the following service and completion time:

$$T = 10\ t/9 \qquad T_c \sim 10\ t$$

which are comparable to the performance metrics of CMP. They can even be equal to the CMP metrics with suitable static optimizations, for example:

1-2. MUL Ra1, Rb1, Rx1 | DIV Rc1, Rd1, Ry1
a-b. LOAD RB2, Rj2, Rq2 | LOAD RA2, Rj2, Rp2
3-c. ADD Rx1, Ry1, Rz1 | SUB Rp2, Rq2, Rr2
4-d. STORE RC1, Ri1, Rz1 | INCR Rr2
x-e. NOP | STORE RD2, Rj2, Rr2

obtaining:

$$T = t \qquad T_c \sim 9\ t$$

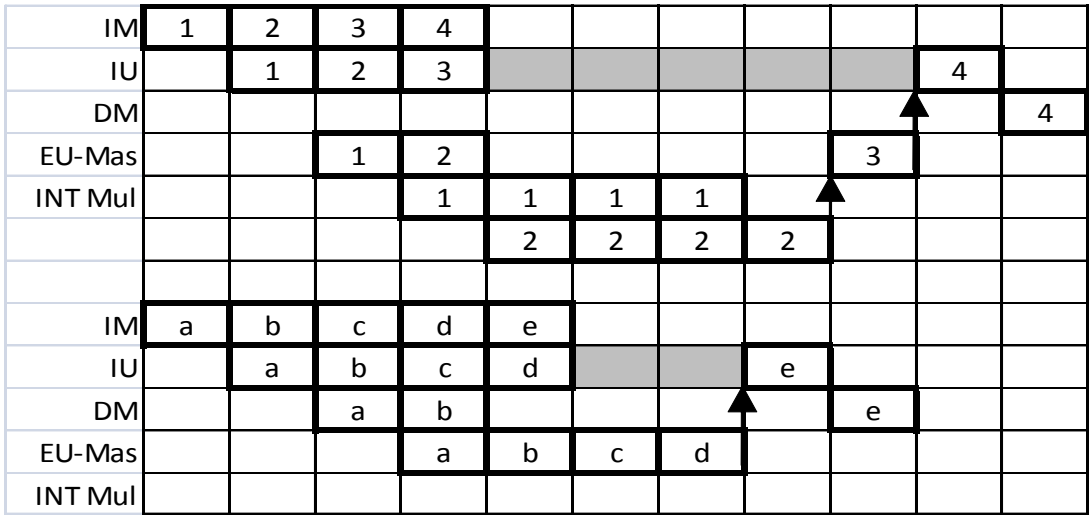
In conclusion, the two architectures exploit the parallelism of P according to different approaches, but the performance results are comparable or even equal.

CMP executes processes P_1 and P_2 on the distinct scalar CPUs, each one with its own degradations (bubbles), as shown in the following figure. We observe that there are some time slots (*t*) during which both IUs are active, or both are blocked, or one active and the other blocked.

This simple observation is the intimate motivation for the MT architecture:

a) if the single IU is able to process both instructions of a long instruction, then the maximum parallelism between processes is exploited,

b) even more meaningful: if a bubble occurs in a process, it is possible that the other process is able to fill the bubble itself.



Points *a)* and *b)* exemplify the two "souls" of MT parallelism: *a)* finding *independent* instructions to be executed simultaneously, *b) masking latencies*.

This can be shown by comparing the execution simulations of CMP and MT in the next fugure (the non-optimized version is shown, since it is the most explanatory case):

In general it is interesting to compare the following architectures:

1) multiprocessor with *p* superscalar CPUs, each with *n/p* ways,
2) multithreaded architecture on a single superscalar *n-way* CPU,

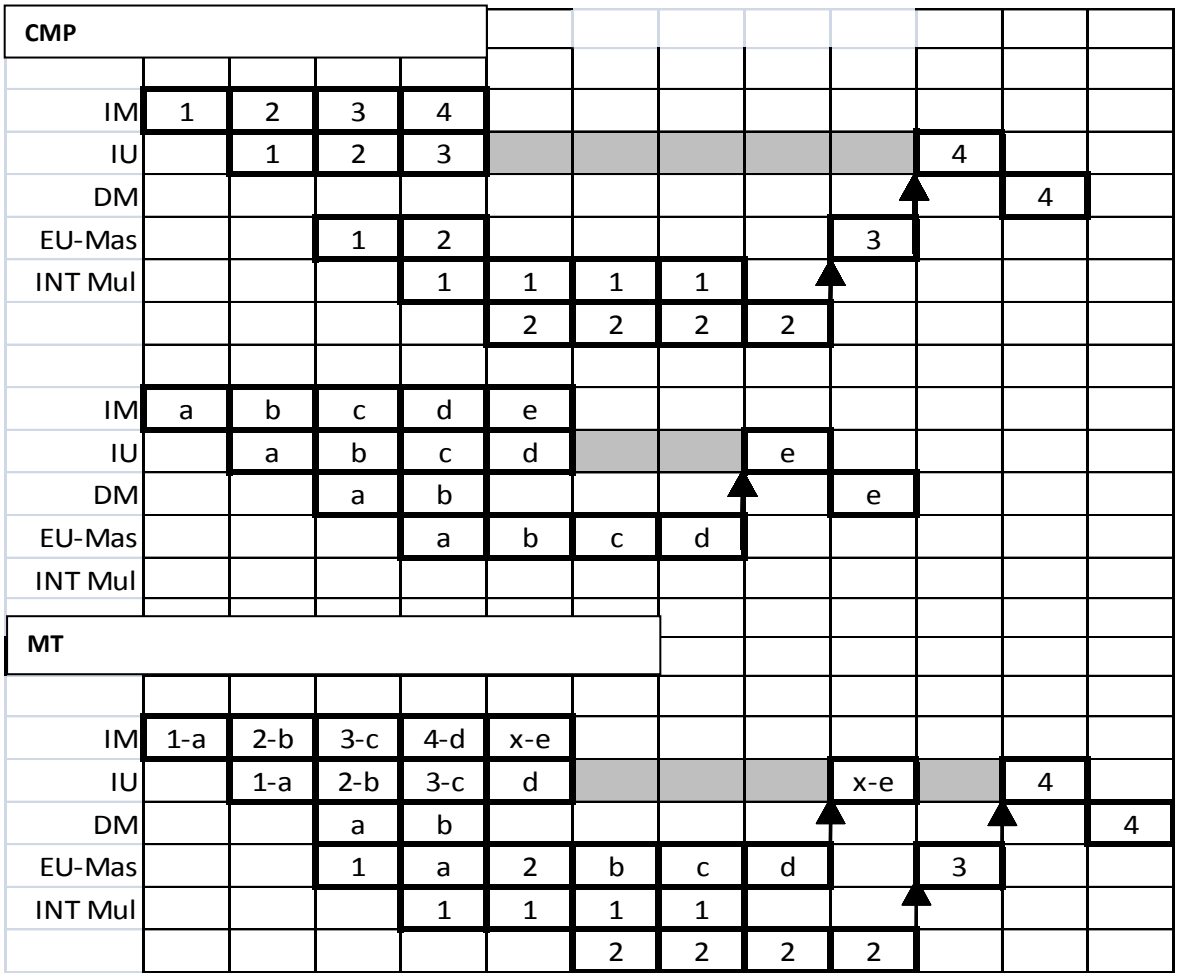
or a variety of mixed cases.

Of course, lines 1 and 2 are *not* comparable for any parallelism degree *n*:

1) the multiprocessor architecture is *scalable* by nature, even with very high values of *n*, provided that "good" parallel programs are designed;
2) the single-0CPU multithreaded architecture is meaningful only for *low values* of *n*.

Currently, several CMP products have each core supporting a limited number of threads (2 – 4). The clear trend is to provide many cores with simple (scalar or superscalar with very few ways) architecture, notably with *in-order* behavior.

The efficient exploitation of the MT feature in parallel programs, for high parallelism multiprocessors, is still an open research issue. The following section contains a first basic characterization to understand the problem.



22.2 Scalability of multithreading *vs* single thread

Let us consider again the example:

P_1::

1. MUL Ra1, Rb1, Rx1
2. DIV Rc1, Rd1, Ry1
3. ADD Rx1, Ry1, Rz1
4. STORE RC1, Ri1, Rz1

P_2::

a. LOAD RA2, Rj2, Rp2
b. LOAD RB2, Rj2, Rq2
c. SUB Rp2, Rq2, Rr2
d. INCR Rr2
e. STORE RD2, Rj2, Rr2

As seen, the MT 2-way superscalar CPU has performance metrics:

$$T = t \qquad T_c \sim 9\,t$$

Let us execute the same computation on a 2-way superscalar architecture with *a single thread*:

1-2. MUL Ra1, Rb1, Rx1 | DIV Rc1, Rd1, Ry1
3-x. ADD Rx1, Ry1, Rz1 | NOP
4-x. STORE RC1, Ri1, Rz1 | NOP
a-b. LOAD RA2, Rj2, Rp2 / LOAD RB2, Rj2, Rq2
c-x. SUB Rp2, Rq2, Rr2 | NOP
d-x. INCR Rr2 | NOP
e-x. STORE RD2, Rj2, Rr2 | NOP

obtaining:

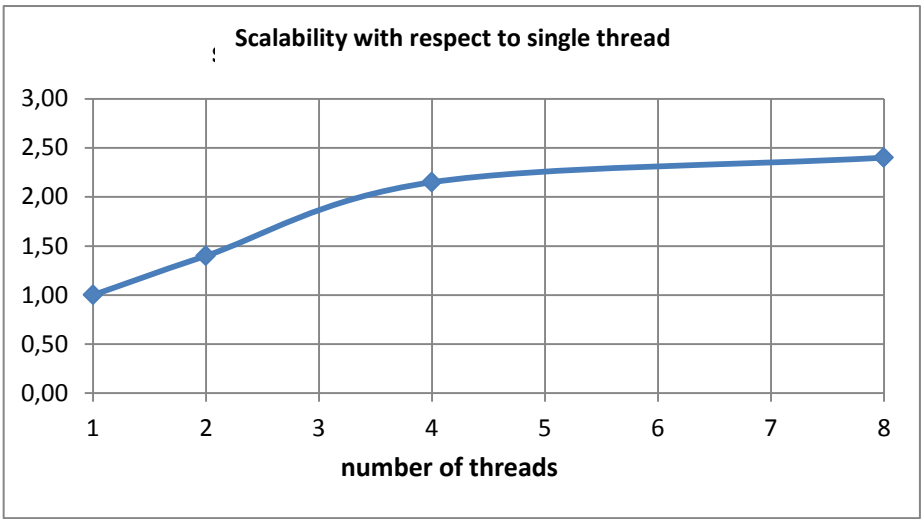
$$T = 14\ t/9 \qquad T_c \sim 14\ t$$

The *scalability* of the 2-thread version with respect to the 1-thread version is:

$$s_{MT} = 14/9 = 1{,}6$$

That is, the superscalar architecture is better exploited in MT modality. However, the performance gain is sensibly lower that 100%, in our example it is 60%. In fact, some initial benchmarks for Intel Xeon with Hyperthreading declared an improvement of 65% with respect to the Xeon architecture of the previous generation.

More in general, the following figure show very meaningful performance evaluation measures on the *standard benchmark suite Spec95 and Splash2* in multiprogramming and parallel processing version. The scalability ranges is in the interval 1.4 – 2.4 with a number of threads in the interval 2 – 8.



This evaluation has been done assuming that each long instruction contains instructions belonging to distinct threads.

22.3 Multithreaded architectures

The parallel activities executed by a MT CPU, on a time slot basis, are referred to using the term ***thread***, with a more specific meaning with respect to the traditional notion of OS-supported multiprogramming and/or of a set of concurrent activities defined in the same addressing space.

The meaning is a concurrent computational activity which is ***supported directly at the firmware level*** (to reinforce this idea, sometimes the term “hardware thread” is used). For this purpose the firmware architecture of a MT CPU, able to execute at most *m* threads simultaneously, has to provide:

1. *m independent contexts*, where, as usually, a context is represented by the program counter and the registers (general registers, floating point registers) visible at the assembler level;
2. a *tagging* mechanism, to distinguish instructions in execution, belonging to distinct threads, using shared resources of the pipeline structure (notably, EU Master and arithmetic functional units, or data cache);
3. a *thread switching* mechanism to activate context switching at the thread level.

Independent contexts could be emulated on a single register set. However, often they are implemented by *multiple register sets* (*m* physically distinct copies of program counter, general registers, and floating point registers).

The key to understand MT is the existence of the specific *run-time support of threads implemented directly at the firmware level* (characteristics 1, 2, 3). From the programmer viewpoint, this may be not visible: a thread can be declared in the usual/traditional way, for example

- a user or a system thread (e.g. Posix thread)
- an applicative program or process.

However, the *compiler* of a MT machine is different from a traditional compiler: it produces proper code to link the *firmware-level run-time support* for threads.

Moreover, a suitable compiler makes it possible also to implement as threads some activities which are not explicitly defined by the programmer or by the system services, for example:

- a concurrent activity recognized automatically by the compiler in a sequential process (e.g. the so-called subordinated threads, microthreads, nanothreads),
- a concurrent activity which is generated directly at run-time.

To further characterize the MT architectures, let us introduce the term *instruction issuing*, or *instruction emission*, to denote the generation phase of a stream element by IM, *in a single time slot*, and its consequent processing in IU. The following class of MT architecture can be distinguished:

I. ***single-issue*** architectures: during a time slot only instructions belonging to a *single thread* are issued. Instruction belonging to distinct threads are issued in distinct time slots. This class can be implemented either on a *scalar* or on a *superscalar* architecture;

II. ***multiple-issue*** architectures: during a time slot instructions belonging to *more than one thread* can be issued. This class, which can be implemented only on *superscalar* architectures, is also called ***simultaneous multithreading*** (SMT). All the examples presented till now are relative to a SMT CPU.

In class I a further distinction consists in ***interleaving*** (IMT) *vs* ***blocking*** (BMT) architectures.

In IMT a fixed ordering is established among concurrent threads, e.g. if *m = 2*, instructions of thread_1 and of thread_2 alternate during consecutive time slots (thread switching occurs every time slot). In BMT there is no fixed ordering, and the thread switching occurs when an event that blocks the execution of an instruction of the running thread occurs (e.g. a logical dependency).

IMT and BMT models are not studied in these notes (the interested reader is referred to the paper “A survey of processors with explicit multithreading”, ACM Computing Surveys, Vol. 35, Issue 1, March 2003).

In the following, we will focus on the SMT class, commercially introduced by Intel with the *Hyperthreading* model. Currently and in perspective, SMT represents the most relevant application of the MT architecture concept.

In the following, we will assume that the threads belonging to the same concurrent computation *share the same addressing space*. This is a necessary prerequisite to exploit

the pipeline behavior, on a time slot basis, in presence of multiple simultaneous threads: their individual instructions and data must be allocated, or allocatable, in the primary instruction and data cache.

22.4 Simultaneous Multithreading

In a SMT *n-way* superscalar architecture, the *n* instructions of the same long instruction (or, in general, simultaneously issued) can belong to *m* threads, where

$$1 \leq m \leq n$$

In general *m* is variable, i.e. every long instruction has its own value of *m*.

The presence of instructions, belonging to distinct threads, in the same stream element makes much easier to respect the independence constraint of a VLIW superscalar architecture.

Let us consider some simple examples.

In the first example we have $m = n = 2$ in all long instructions:

Thread 1:

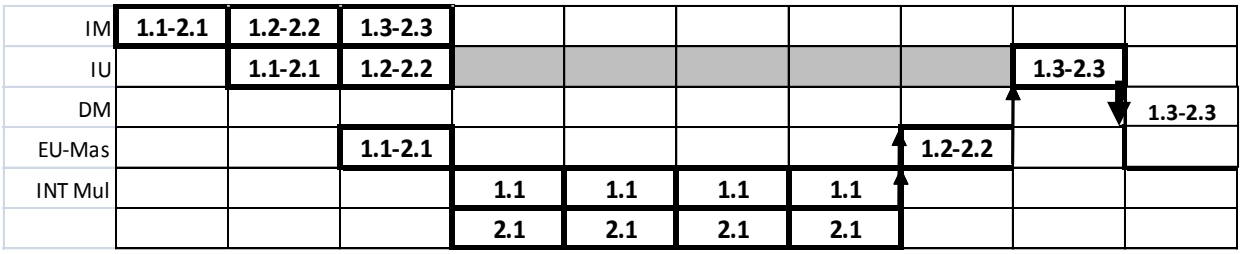
1.1 MUL RA1, Ri1, Rx1
1.2 INCR Rx1
1.3 STORE RB1, Ri1, Rx1

Thread 2:

2.1 MUL RA2, Ri2, Rx2
2.2 INCR Rx2
2.3 STORE RB2, Ri2, Rx2

encoded and executed as:

1.1-2.1 MUL RA1, Ri1, Rx1 | MUL RA2, Ri2, Rx2
1.2-2.2 INCR Rx1 | INCR Rx2
1.3-2.3 STORE RB1, Ri1, Rx1 | STORE RB2, Ri2, Rx2



The two threads evolve with perfect synchronous parallelism at the clock cycle level. The service time of the whole parallel computation is one half with respect to the sequential execution in a non-MT architecture.

This example serves to introduce the SMT model, however it shows only a part of its potential power (maximum parallelism exploitation).

For example, *latency masking* is allowed too:

Thread 1:

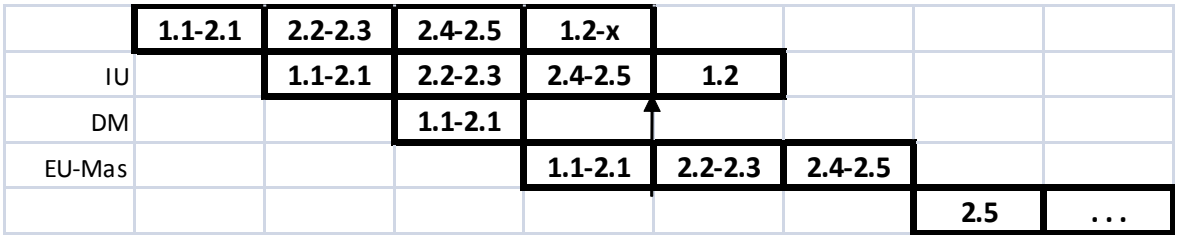
1.1 LOAD RA1, Ri1, Rc1
1.2 STORE RC1, Ri1, Rc1

Thread 2:

2.1 LOAD RX2, Ri2, Rx2
2.2 ADD Rx2, Ry2, Rz2
2.3 INCR Ri2
2.4 INCR Rv2
2.5 MUL Rz2, …, …

Possible encoding and execution:

```
1.1-2.1   LOAD   RA1, Ri1, Rc1   |   LOAD   RX2, Ri2, Rx2
2.2-2.3   ADD    Rx2, Ry2, Rz2   |   INCR   Ri2
2.4-2.5   INCR   Rz2   |   STORE   …
1.2-x.    STORE  RC1, Ri1, Rc1   |   NOP
```



The bubble in Thread 1 has been filled by other long instructions (in this example, belonging to the other thread).

22.4.1 Instruction composition and scheduling

The key of the SMT model is just the *proper composition of long instructions*, in order to meet the double goal: exploiting thread-level parallelism on a time slot basis, and masking latencies.

Instructions to be issued in the same time slot can be recognized *statically* (long instruction in the true meaning of the term) or *dynamically*.

In the dynamic case, the firmware interpreter of IM-IU is in charge of choosing instructions to be issued simultaneously and of composing them in the same steam element at run-time. This is done according to a *dynamic scheduling strategy*, that can also depend on non-predictable events, e.g. logical dependencies induced by variable execution time instructions, or cache misses.

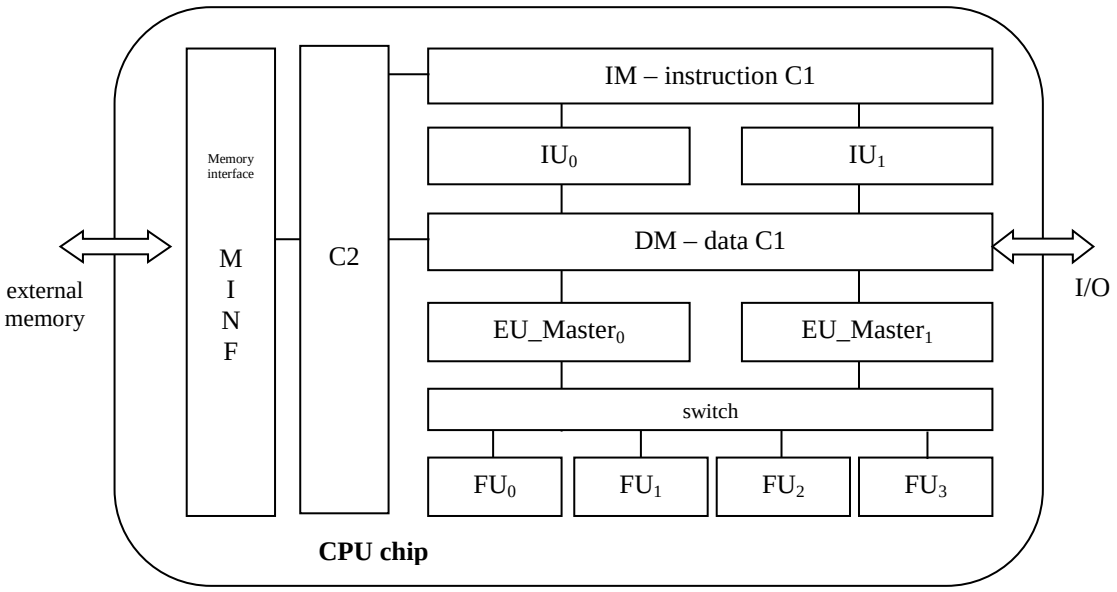
Actually, the most challenging problem of SMT architectures is just *instruction composition* and *thread scheduling*. As usually, a way of reducing the complexity of a design problem is to exploit *constraints* in the form of computation patterns that are common to many applications: notably, exploit the properties of stream-parallel and data-parallel structured paradigms. For example, if two *map* workers, W_{i1} and W_{i2}, are implemented as threads allocated onto the same MT processing node, the best composition strategy is merely to insert in the same long word the same instructions of W_{i1} and W_{i2}. Analogous examples can be done in other structured parallelism paradigms.

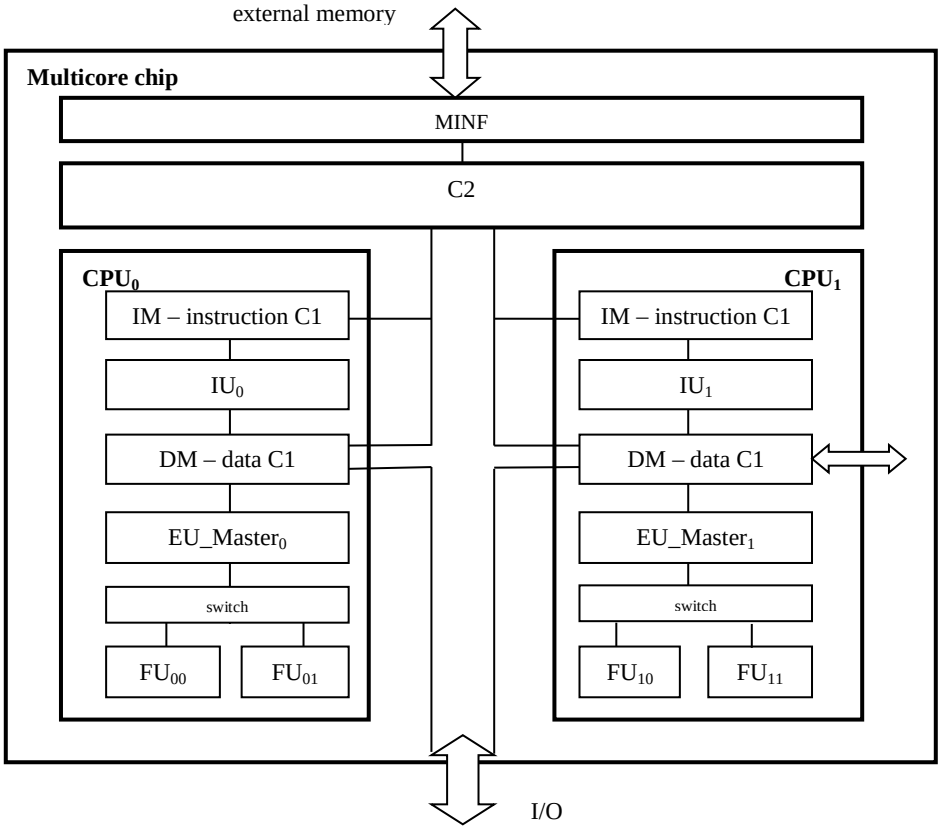
22.4.2 From multithreading to multicore

The two following figures show, respectively,

1. a possible schema of MT CPU chip with replication of IU and EU Master,
2. the structure of a dual-core CMP chip,

realized with the same silicon technology.





The architecture shown in the first figure can support the SMT model with $m = 2$ threads, where each thread utilizes a private subsystem (IU, EU Master), while the other resources (IM, DM, EU functional units, secondary cache) are shared by the threads in execution.

The second figure shows a shared-memory multiprocessor architecture, in which the two single-threaded CPUs occupy about the same area of the chip in the first figure, also taking into account that the number of EU functional units is one half for each CPU. Similarly to the MT architecture, the secondary cache is shared.

More in general, it is interesting to compare the CMP and SMT approaches from several viewpoints. The reader is invited to investigate this issue, along with reflections about the

possibilities offered by the, currently popular, *mixed* approach: multiprocessor with SMT CPUs.

22.4.3 Two examples of multithreading exploitation in multiprocessor architectures

Example 1: data parallelism

Let us consider the vector addition computation with static optimizations. In absence of cache misses, the completion time per iteration is

$$T_{iter,scalar} = 7\,t \qquad\qquad T_{iter,superscalar,2} = 5\,t$$

The same computation can be implemented, at the process level, as a data-parallel *map*, with data partition size $g = N/p$. If each core is *2-way 1-thread* superscalar, the completion time per worker is:

$$T_{c,superscalar,2} = g\,T_{iter,superscalar,2} = 5\,g\,t$$

If each core is *2-way 2-thread* superscalar, each worker can be implemented as two parallel, independent, identical threads, each one operating on *g/2* integers (remember that they share the same addressing space). As said, the simplest, yet most efficient, way to define such threads is to insert correspondent instructions in the same long word: in this case, both thread parallelism exploitation and latency masking are achieved. Each thread has a completion time per iteration:

$$T_{iter,SMT,2} = 7\,t$$

The completion time of the whole 2-thread computation is:

$$T_{c,SMT,2} = g\;T_{iter,SMT,2} = 7\,\frac{g}{2}t = 3.5\,g\,t$$

In conclusion, with a 2-thread implementation of each worker, we achieve a performance gain equal to the scalability of 2-thread SMT with respect to 1-thread superscalar:

$$s_{MT,2} = \frac{T_{c,superscalar,2}}{T_{c,SMT,2}} = 1.4$$

In other words, with *p* CPUs, each one with 2-thread SMT architecture, we actually achieve the effect of a parallelism increase of the architecture: the *equivalent parallelism degree* is not *2 p*, instead in general it is lower, in this example *1.4 p*:

$$p * s_{MT,2} = 1.4\ p\ cores$$

Assuming that the optimal degree of parallelism of the parallel program (see Part 1) is greater or equal, the multiprocessor version with SMT nodes has clear advantages, though not proportional to the number of threads per core. The reader is invited to extend this multithreading application to other structured parallelism paradigms studied in Part 1.

Example 2: communication thread

By a multithreaded architecture of processing nodes, it is possible to emulate the solution with communication processor (KP): a thread is dedicated to the KP functions. When a *send* primitive is invoked by the calculation thread IP, the IP instructions are composed (dynamically) with the KP instructions. Parameter passing is easy since the two threads share the same addressing space.

Similar consideration apply to the overlapping of CPU and I/O activities.

23. Exercises

23.1 Exercise 1

1) A sequential program is defined by the following algorithm, with $M = 10^4$:

int A[M], B[M]; int x;

x = 0;

$\forall$ *i = 0 .. M–1:*

$\forall$ *j = 0 .. M–1:*

*x = x + A[i]*B[j]*

Evaluate the completion time for a D-RISC scalar pipelined CPU with 4-stage multiplier functional unit, primary data cache of 32K words and blocks of 8 words, and secondary cache with access time of two clock cycles.

2) The program in *1)* defines a module Q operating on streams: *A* is received from the input stream, *x* is sent onto the output stream, and *B* is encapsulated statically with a given initial value.

The parallel architecture has $N = 32$ processing nodes, each one with communication processor and zero-copy interprocess communication. $T_{setup} = 10^3\ \tau$, $T_{transm} = 10^2\ \tau$, and the interarrival time = $10^7\ \tau$, where τ is the CPU clock cycle.

i. Define, explain and evaluate two different parallel versions of Q.
ii. Evaluate the relative efficiency of the whole computation and of each module contained in the two parallel versions.

1) The non-optimized compiled code is (general registers Ri and Rx are initialized at zero, RA, RB and RX at the base addresses of A, B and X, RM at the M value):

```
LOOPi:    LOAD   RA, Ri, Ra
          CLEAR  Rj
LOOPj:    LOAD   RB, Rj, Rb
          MUL    Ra, Rb, Ra
          ADD    Rx, Ra, Rx
          INCR   Rj
          IF < Rj, RM, LOOPj
          INCR   Ri
          IF <   Ri, RM, LOOPi
          STORE RX, 0, Rx
```

The program performance depends on the innermost loop (executed M^2 times), thus we will focus on its optimization only. In the shown version, there is a logical dependence IU-EU induced by INCR Rj on IF < Rj (distance k = 1, $N_Q = 2$): the critical sequence contain a long latency instruction, which however is independent of INCR. Moreover, the IF instruction introduces an additional bubble. An optimized code is:

…

```
                LOAD   RB, Rj, Rb, not_deallocate
LOOPj:          INCR   Rj
                MUL    Ra, Rb, Ra
                ADD    Rx, Ra, Rx
                IF < Rj, RM, LOOPj, delayed branch
                LOAD   RB, Rj, Rb, not_deallocate
                …
```

in which both the mentioned degradations are eliminated. Notice that instructions MUL and ADD do not affect the logical dependence delay: despite the long latency of MUL, they are executed in parallel to LOAD, INCR and IF, as it can be verified with the graphical simulation. Thus, the instruction service time is the ideal one:

$$T = t$$

The completion time of the innermost loop is:

$$T_{c\text{-}inner} = 5\ M\ T = 5\ M\ t = 10\ M\ \tau$$

With very good approximation, the ideal completion time (with perfect cache) is:

$$T_{c\text{-}id} \sim M\ T_{c\text{-}inner} = 10\ M^2\ \tau$$

The effective service time must take into account the delay introduced by cache faults:

$$T_c = T_{c-id} + N_{fault} * T_{fault}$$

The program data are characterized by locality of A and B and by reuse of all the elements of B. Since M is less than the cache capacity, B can be maintained in data cache once loaded for the first time, provided that instruction *LOAD ... Rb* contains the annotation "not_deallocate", as shown in the optimized code. Thus

$$N_{fault} * T_{fault} = 2\frac{M}{\sigma} * 2\,\sigma\,\tau = 4\,M\,\tau$$

which is negligible compared to $T_{c\text{-}id}$, owing to the reuse optimization. Notice that A and B are entirely contained in the secondary cache when they are referred. In conclusion,

$$T_c = 10\ M^2\ \tau = 10^9\ \tau$$

2) The completion time evaluated for the sequential program is the ideal service time of module Q operating on streams:

$$T_{calc} = 10^9\ \tau$$

According to the problem specifications, the *receive (..., A)* and the *send (..., x)* primitives do not affect the service time.

The optimal parallelism degree is given by:

$$n = \frac{T_{calc}}{T_A} = 100$$

which is greater than the number N of available nodes. Thus, it must be reduced to $N - n_p$, where n_p is the number of service modules.

Parallelization: farm version

Since Q is a pure function, it can be parallelized by the *farm* paradigm with statically replicated B. For the ideal version with $n = 100$, the Emitter module is not a bottleneck:

$$T_{E-id} = T_{send}(M) \sim 10^6 \tau < T_A$$

thus the farm solution could be able to achieve the optimal service time T_A. Because of the limited number of nodes N and $n_p = 2$, the number of workers is reduced to:

$$n_w = 30$$

so the effective service time is equal to *3,3* T_A.

The relative efficiency of the whole farm computation and of the workers are

$$\epsilon_{farm} = 1$$

$$\epsilon_{worker} = 1$$

both for the ideal and for the effective parallelism degree.

For *n = 100*:

$$\epsilon_{emitter} = \rho_{emitter} = 10^{-1}$$

$$\epsilon_{collector} = \rho_{collector} = 10^{-4}$$

For *n = 30*:

$$\epsilon_{emitter} = \rho_{emitter} = \frac{10^{-1}}{3{,}3}$$

$$\epsilon_{collector} = \rho_{collector} = \frac{10^{-4}}{3{,}3}$$

Parallelization: data-parallel version

A *data-parallel solution* consist in a *map* followed by a *reduce*. With a linear array of *M* virtual processors VP[M], the generic VP[i] encapsulates A[i], the whole B (statically replicated), and a local copy of *x*: let us call it *X[i]*. At the end of the map phase, a global *reduce (X[M], +)* is applied.

In the ideal case, with *n = 100* workers, each one encapsulates *g = M/n = 100* elements of B statically, *g* elements of scattered *A*, and a local *X*. The scatter is not a bottleneck even if implemented by a single module and with *n = 100*:

$$T_{scatter} = n\, T_{send}(g) \sim 10^6 \tau < T_A$$

The reduce is implemented by a tree structure mapped onto the linear array of workers, thus with logarithmic latency. The *(n-1)-th* worker is in charge of sending the final result onto the output stream

However, due to limited N and $n_p = 1$, the number of workers is reduced to

$$n_w = 31$$

so the effective service time is *3,2* T_A (a slightly better bandwidth, compared to the farm version, provided that the load is balanced).

The reduce latency

$$T_{reduce} = lg_2(n_W)\,(T_{send}(1) + T_+) \sim 10^4 \tau \ll T_A$$

which is negligible.

The relative efficiencies are analogous to the farm evaluation:

$$\epsilon_{dp} = 1$$

$$\epsilon_{worker} = 1$$

$$\epsilon_{scatter} = \begin{cases} 10^{-1} \; for \; n = 100 \\ \dfrac{10^{-1}}{3{,}2} \; for \; n = 31 \end{cases}$$

23.2 Exercise 2

Consider the following sequential module, with $M = 10^5$:

```
int A[M], B[M];
    { ∀ i = 1 .. M–2:
          B[i] = A[i] + A[i-1] ∗ A[i+1];
      B[0] = A[0];
      B[M-1] = A[M-1]
    }
```

The module operates on streams, where each input elements is an A array and each output element is a B array.

The interarrival time is equal to 10^5 τ.

We wish to parallelize the module for an all-cache SMP architecture with the following characteristics:

- $N = 128$ processing nodes with clock cycle τ;
- processing nodes are D-RISC scalar pipelined CPUs with 4-stage integer multiplier functional unit. The primary data cache has block size σ = 8 words;
- *2*-ary *n*-fly wormhole interconnection network, with links and flits 1-word wide, and link transmission latency equal to τ;
- interleaved main memory macromodules, with $m = 8$ modules, and clock cycle $\tau_M = 20$ τ.

a) Define the possible parallel versions of Q.

b) Evaluate the interprocess communication latency.

c) Select a parallel version able to optimize the completion time, the latency and the memory capacity, and evaluate its completion time, under the assumption that the under-load memory access latency is equal to the base one.