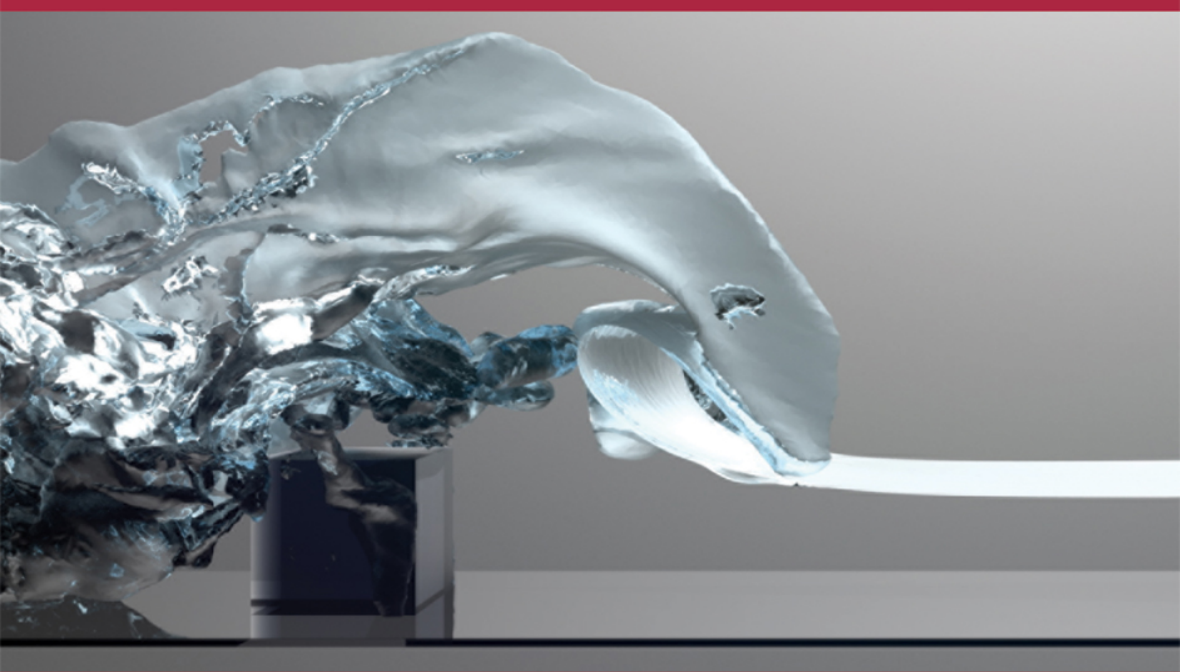
COMPUTER ENGINEERING SERIES

Parallel Scientific Computing

Frédéric Magoulès
François-Xavier Roux
Guillaume Houzeaux



ISTE
WILEY

Parallel Scientific Computing

Series Editor
Serge Petiton

Parallel Scientific Computing

Frédéric Magoulès
François-Xavier Roux
Guillaume Houzeaux

ISTE
WILEY

First published 2016 in Great Britain and the United States by ISTE Ltd and John Wiley & Sons, Inc.

ISTE Ltd
27-37 St George's Road
London SW19 4EU
UK

www.iste.co.uk

John Wiley & Sons, Inc.
111 River Street
Hoboken, NJ 07030
USA

www.wiley.com

Cover photo generated by Guillermo Marin, Barcelona SuperComputing Center (Spain).

Library of Congress Control Number: 2015955799

British Library Cataloguing-in-Publication Data
A CIP record for this book is available from the British Library
ISBN 978-1-84821-581-8

Contents

Preface

Scientific computing has become an invaluable tool in many diverse fields, such as physics, engineering, mechanics, biology, finance and manufacturing. For example, through the use of efficient algorithms adapted to today's computers, we can simulate, without physically testing costly mock-ups, the deformation of a roof under the weight of snow, the acoustics of a concert hall or the airflow around the wings of an aircraft.

The goal of this book is to explain and illustrate, using concrete examples, the recent techniques of scientific computing for the numerical simulation of large-size problems, using systems modeled by partial differential equations. The different methods of formation and solving of large linear systems are presented. Recent numerical methods and related algorithms are studied in detail. Implementation and programming techniques are discussed for direct and preconditioned iterative methods, as well as for domain decomposition methods. Programming techniques based on message-passing and loop parallelization are illustrated by using examples that employ MPI and OpenMP.

The main objective of this book is to examine numerical techniques applied to parallel computing for machines with a very large number of processors and distributed memory. Knowledge of numerical analysis, and basic computer science concepts are required for optimal understanding. Though the functioning of scientific computers is described, this book will not go beyond what is useful for writing efficient programs. The underlying

idea is to show, in a reasonably complete manner, recent numerical methods used in scientific computing, with an emphasis on their adaptation to parallel computing. We present a number of examples of parallel algorithms, which are more or less standard in scientific computing. Most of these examples are drawn from problems arising from the implementation of the finite element method.

We follow a didactic approach and gradually introduce mathematical and computing concepts where appropriate, and whenever the need arises to enhance understanding. And we present, as examples, the introduction of new architectural characteristics of computers, and current management issues of parallelism due to the increasing complexity of applications. This book is designed to be an introduction to the issues of parallel computing for users of scientific computers, and is not meant as a reference work on parallel computing in terms of information technology (IT).

This book is intended primarily for Master's students of applied mathematics, as well as of computational mechanics, and more generally to students in all fields of engineering who are concerned with high-performance computing. It may also interest any engineer faced with the numerical simulation of large-scale problems from systems modeled by partial differential equations, as well as more generally, the solving of large linear systems.

Portions of this book have been used, for a number of years by the authors, in lectures on scientific computing at Wuhan University of Science and Technology (China), Université Pierre et Marie Curie (France), Université Henri Poincaré (France), Conservatoire National des Arts et Métiers (France), École Centrale des Arts et Manufactures (France), École Normale Supérieure de Cachan (France), École Supérieure des Sciences et Technologies de l'Ingénieur de Nancy (France), Institut des Sciences de l'Ingénieur de Toulon et du Var (France), University Duisburg-Essen (Germany), Chuo University (Japan), Doshisha University (Japan), Keio University (Japan), University of Electro Communications (Japan) and the Partnership for Advanced

Computing in Europe (PRACE) Training Course at Barcelona Supercomputing Center (Spain).

Frédéric Magoulès
École Centrale des Arts et Manufactures (École Centrale Paris), France
University of Pécs, Hungary

François-Xavier Roux
Université Pierre et Marie Curie, France
ONERA, France

Guillaume Houzeaux
Barcelona Supercomputing Center, Spain
October 2015

Introduction

Recent advances in computer architectures (clock frequency, cache, memory hierarchy, multi-core, etc.) have led to the development of today's scientific computers with millions of cores, which often carry out more than 10^{15} floating-point operations per second (flops). For comparison, this figure would correspond to more operations in 1 second than the world's population could make in 2 days, with our estimation based on one floating-point operation per second per person. Twice a year, the TOP500 establishes the list of the most powerful (declared) supercomputers in the world, in terms of flops. Currently, the first rank supercomputer, Tianhe-2 in China, is composed of more than 3 million cores and has a maximum performance of almost 34 petaflops. Nowadays, the limitation in the increase of computational power is the electrical power needed to run these systems. The power of the aforementioned supercomputer is 17,808 kW, which corresponds to the consumption of an average European city of 80,000 inhabitants. The development of more ecological supercomputers is thus a challenge and is now a high priority area of research. But, it is not only necessary to develop lower power computers... but also to develop efficient algorithms that take full benefit of these architectures.

This book is an introduction to high-performance computing (HPC). Its purpose is to present some numerical methods, using scientific supercomputers, for solving engineering problems that cannot be treated by using classical computers. Current issues of HPC are successively addressed: data parallelism, vectorization, message-passing, parallel formation of matrices, parallelization of the product of matrices, direct and iterative parallel methods for solving large linear systems, etc.

The presentation of these methods is brought to life by the systematic use of the programming environments of MPI and OpenMP, for which the main commands are gradually introduced. All algorithms presented here are in the form of pseudo-codes. This allows the readers to quickly visualize the properties of these algorithms, in particular, the sequence of operations, dependencies among data, etc. The resolution of various problems, often drawn from concrete applications, is the subject of numerous examples and problem-solving exercises. At the end of this book, an appendix presents more advanced concepts and provides bibliographic data, which enables readers to deepen their acquired knowledge.

For this purpose, the book can be divided into four parts.

The first part, introduced in Chapter 1, discusses the architecture of scientific computers, different types of parallelism and the memory architecture of these computers. Chapter 2 presents programming models, performance criteria and data parallelism. Then in Chapter 3, we provide a concrete example of the product of matrices to illustrate the parallelization process, temporal and spatial locality of data.

The second part provides a concise complementary numerical matrix analysis. Chapter 4 recalls some basic notions of linear algebra and the properties of matrices, and also explains the notation used later in this book. Chapter 5 focuses particularly on sparse matrices in the context of the finite element, finite difference and finite volume methods, and more specifically on their origins and parallel formation. Chapter 6 outlines the main methods of solving linear systems. The implementations of these methods are detailed in the sections which follow.

The third part examines methods for solving large linear systems. Chapter 7 presents the principles of direct methods (LU, Cholesky, Gauss–Jordan and Crout's factorization), which leads to Chapters 8 and 9 that focus, respectively, on the parallelization of LU methods for dense matrices, and then sparse matrices.

The fourth part treats iterative methods for solving large linear systems by using Krylov methods. A quick review of Krylov subspaces and the construction of Arnoldi algorithms are detailed in Chapter 10. Chapter 11 presents the Krylov methods with complete orthogonalization for symmetric

positive definite matrices. Chapter 12 examines exact orthogonalization methods for general matrices, followed by Chapter 13 that considers biorthogonalization methods for non-symmetric matrices. The parallelization techniques of the Krylov methods are discussed and detailed in Chapter 14. Preconditioning techniques and hybrid methods, such as those used in domain decomposition, are briefly described in Chapter 15.

1

Computer Architectures

This chapter does not claim to be a course in computer programming. Only those architectural features which are not obvious to the user, i.e. those that imperatively need to be taken into account for coding which achieves the optimal performance of scientific computers, are presented here. Therefore, we will not be going into the details of hardware and software technologies of computer systems, but will only explain those principles and notions that are indispensable to learn.

1.1. Different types of parallelism

1.1.1. *Overlap, concurrency and parallelism*

The objective of numerical simulation is to approximate, as closely as possible, physical reality, through the use of discrete models. The richer the model, and the more parameters it takes into account, the greater the amount of computational power. The function of supercomputers is to permit the execution of a large number of calculations in a sufficiently short time, so that the simulation tool can be exploited as part of a design process, or in forecasting.

The natural criterion of performance for a supercomputer is based on the speed of calculations, or the number of arithmetic operations achievable per second. These arithmetic operations – addition, subtraction, multiplication or division – involve data, either real or complex, which are represented by floating point numbers. A floating point number is a real number that is

Parallel Scientific Computing, First Edition. Frédéric Magoulès, François-Xavier Roux and Guillaume Houzeaux.

represented by two integers, a mantissa and an exponent. Since computers work in base 2, the value of a real number, represented in floating point representation, is equal to the mantissa or signific and multiplied by 2 times the power of the exponent. The precision of a real number is then limited by the length of the mantissa. The unit used to measure calculation speeds is the "flops" (floating point operations per second). As the frequencies of current microprocessors have increased, the following terms are commonly employed: Mflops, or a million operations per second (Mega = 10^6); Gflops, or a billion operations per second (Giga = 10^9); Tflops, which is a trillion operations per second (Tera = 10^{12}); and even Pflops, or a quadrillion operations per second (Peta = 10^{15}). Speeds are dependent upon the technologies used for components and depend on the frequencies of the microprocessors. Up until the early 2000s, there were enormous improvements in the integration of semi-conductor circuits due to manufacturing processes and novel engraving techniques. These technological advances have permitted the frequency of microprocessors to double, on average, every 18 months. This observation is known as Moore's law. In the past few years, after that amazing acceleration in speeds, the frequencies are now blocked at a few GHz. Increasing frequencies beyond these levels has provoked serious problems of overheating that lead to excessive power consumption, and technical constraints have also been raised when trying to evacuate the heat.

At the beginning of the 1980s, the fastest scientific computers clocked in around 100 MHz and the maximum speeds were roughly 100 Mflops. A little more than 20 years later, the frequencies are a few GHz, and the maximum speeds are on the order of a few Tflops. To put this into perspective, the speeds due to the evolution of the basic electronic components have been increased by a factor in the order of tens, yet computing power has increased by a factor bordering on hundreds of thousands. In his book *Physics Of The Future*, Michio Kaku observes that: "Today, your cell phone has more computer power than all of NASA back in 1969, when it placed two astronauts on the moon." How is this possible? The explanation lies in the evolution of computer architectures, and more precisely in the use of parallelization methods. The most natural way to overcome the speed limits linked to the frequencies of processors is to duplicate the arithmetic logic units: the speed is twice as fast if two adders are used, rather than one, if the functional units can be made to work simultaneously. The ongoing

improvements in semi-conductor technology no longer lead to increases in frequencies. However, recent advances allow for greater integration, which in turn permits us to add a larger number of functional units on the same chip, which can even go as far as to completely duplicate the core of the processor. Pushing this logic further, it is also possible to multiply the number of processors in the same machine. Computer architecture with functional units, or where multiple processors are capable of functioning simultaneously to execute an application, is referred to as "parallel". The term "parallel computer" generally refers to a machine that has multiple processors. It is this type of system that the following sections of this work will primarily focus on.

But even before developing parallel architectures, manufacturers have always been concerned about making the best use of computing power, and in particular trying to avoid idle states as much as possible. This entails the recovery of execution times used for the various coding instructions. To more rapidly perform a set of operations successively using separate components, such as the memory, data bus, arithmetic logic units (ALUs), it is possible to begin the execution of a complex instruction before the previous instruction has been completed. This is called instruction "overlap".

More generally, it is sometimes possible to perform distinct operations simultaneously, accessing the main or secondary memory on the one hand, while carrying out arithmetical operations in the processor, on the other hand. This is referred to as "concurrency" . This type of technique has been used for a long time in all systems that are able to process several tasks at the same time using timesharing. The global output of the system is optimized, without necessarily accelerating the execution time of each separate task.

When the question is of accelerating the execution of a single program, the subject of this book, things become more complicated. We have to produce instruction packets that are susceptible to benefit from concurrency. This requires not only tailoring the hardware, but also adapting the software. So, parallelization is a type of concurrent operations in cases where certain parts of the processor, or even the complete machine, have been duplicated so that instructions, or instruction packets, often very similar, can be simultaneously executed.

1.1.2. *Temporal and spatial parallelism for arithmetic logic units*

The parallelism introduced in the preceding section is also referred to as spatial parallelism. To increase processing output, we can duplicate the work; for example, with three units we can triple the output.

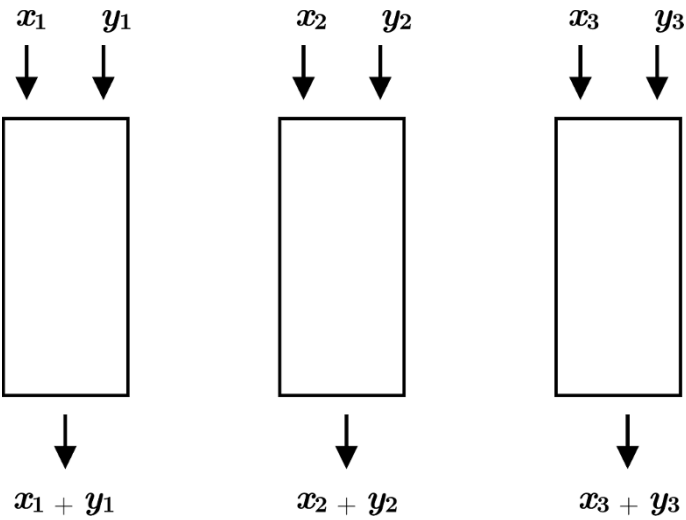


Figure 1.1. *Spatial parallelism: multiplication of units*

There is also what is called temporal parallelism, which relies on the overlap of synchronized successive similar instructions. The model for this is the assembly line. The principle consists of dividing up assembly tasks into a series of successive operations, with a similar duration. If the chain has three levels, when the operations of the first level are completed, the object being assembled is then passed onto the second level where immediately the operations of the first level for a new object are begun, and so forth. Thus, if the total time of fabrication of each object consists of three cycles, a new finished object is completed every three cycles of the chain. Schematically, this is as fast as having three full workshops running simultaneously. This way of functioning allows us, on the one hand, to avoid duplication of all the tools, and on the other hand, also assures a more continuous flow at the procurement level of the assembly line. This type of processing for the functional units of a computer is referred to as “pipelining”. This term comes

from the fact that a pipeline is in fact a form of transportation chain, unlike independent forms of transportation using trucks, trains or boats.

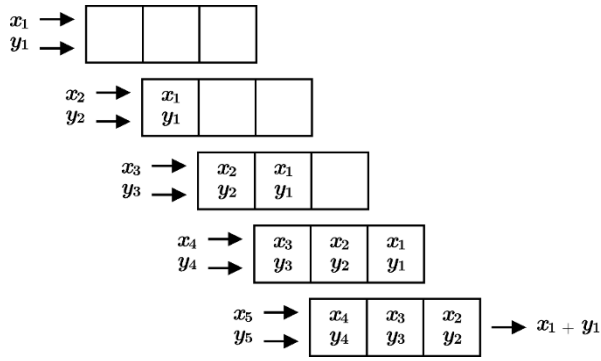


Figure 1.2. *Temporal parallelism: pipeline of additions*

To illustrate how pipeline processing works, let us consider the example of the addition of floating point numbers. This operation takes place in three stages. The first step consists of comparing the exponents, so that the mantissas can be aligned; the second step involves adding up the mantissas; and third to normalize the result by truncating or shifting the mantissa. More specifically, we will take an example written in base 10, just to show how this works. In our example, let us assume the mantissa has four digits. To add 1234×10^{-4} and -6543×10^{-5}, we notice that $-4 - (-5) = 1$. Thus, we have to shift the second mantissa one cell to the right. This is exactly how we treat two operands that we would like to add, when we write one over the other and we align the position of the decimal points. Returning to our example, we then calculate the addition of the mantissas: $1234 + (-0654) = 0580$. And finally, the normalization of the result consists of again shifting the mantissa, but this time back one cell to the left, and reducing the exponent by 1, which gives the final result of 5800×10^{-5}. As an aside, we can note that, in the same way that the precision of the decimal representation of a real number is limited by the size of the mantissa, operations are performed by using an approximation due to this truncation, even if we momentarily expand the size of the mantissa to limit round-off errors. In fact, what is the right extension to five digits of 1234×10^{1}: 12340, 12345 or 12350?

The potential improvements in performance obtained by using pipeline architectures are limited by the size of elementary operations that we can execute in just one clock cycle, like the addition of two signed integers. It would not serve any purpose to split tasks up more than that. We will find both temporal and spatial parallelism used simultaneously in scientific computers that employ pipeline architectures, which is to say that they use multiple pipelined units.

1.1.3. *Parallelism and memory*

A vision of performance based solely on the speed of the execution of arithmetic operations leaves out one important element – the data.

In a scientific code, in general, the most important part of calculation resides in the resolution phase of discretized problems, which necessitate algebraic operations on arrays in one or more dimensions. An emblematic operation of this sort of calculation is the linear combination of vectors:

for $i = 1$ **to** n
  $y(i) = y(i) + \alpha \times x(i)$
end for

At each iteration of the loop, it is necessary to carry out an addition and a multiplication, to recuperate the data $x(i)$ and then the data $y(i)$, from the memory, and then finally update the result $y(i)$. The data α which are the same for each iteration can be conserved in the internal buffer memory cells of the processor, the registers, for the duration of the execution of the loop. Finally, the memory needs to be accessed 3 times, twice in read mode and once in write mode, for these two arithmetical operations. Therefore, nothing is gained by multiplying the number of arithmetic units, nor processors, if the "bandwidth" of the memory is not also increased. If we want to increase the speed of calculation, it is to be able to treat models with more parameters. So, the memory should be of a sufficiently large size to contain all the data, and simultaneously be fast enough to supply all the available arithmetic logic units. To carry this out, the memory would need to function at a faster rate than the processors, which is obviously unrealistic because both the processors and the memory use the same semi-conductor technology.

Therefore, what is most important for the realization of high-performance scientific computers is, in reality, the architecture of the memory.

1.2. Memory architecture

As we just saw, computing power is in large part dependent on the duplication of functional units, which treat the data stored in the memory. Naturally, this raises the question about how to best supply data to the logic units, so as to obtain optimal performance.

1.2.1. *Interleaved multi-bank memory*

To simultaneously increase the size and bandwidth of the memory, the obvious solution consists of duplicating memory units. Doing this evidently increases the size of the memory, however, access time for a particular piece of data always remains the same. So that the global access times can be increased, as in the case of accessing a series of data from an array, it is necessary for the different memory units to function simultaneously in parallel. The different elements of the array, which occupy successive memory addresses, must be allocated to the different memory units, or banks. The memory is thus referred to as interleaved.

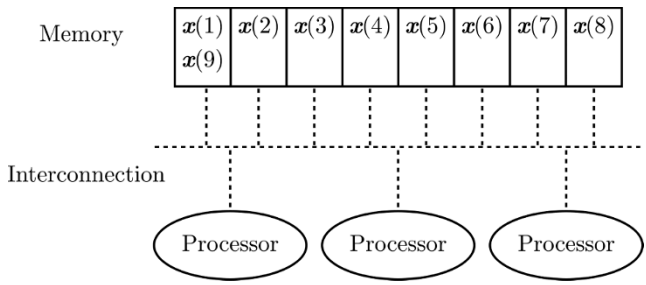


Figure 1.3. *Interleaved multi-bank memory*

Let us assume that an array x is allocated to eight memory banks, as is illustrated in Figure 1.3. Access time to one bank would normally be eight clock cycles, yet after an initialization phase, the interleaved multi-bank memory is capable of providing a new value $x(i)$ with each cycle, because each bank is called upon only once during the eight cycles.

In reality, the access time to a bank is of the order of a few tens of cycles. To best supply in a satisfactory manner a few tens of processors, each with its own

arithmetic unit, a few hundred or even a few thousand banks would be required. The memory controller and the interconnection network between the banks and all the processors would then become very complex and consequently very costly to implement.

This solution is only found in specialized scientific supercomputers known as "vector supercomputers". The term "vector" indicates that these computers, in order to facilitate the administration of the system and to improve performance, feature a set of instructions which do not cover just one datum, but a full series of data in an array, known as a vector. The processors generally have vector registers, which are capable of temporarily storing these vectors.

This solution is not "scalable", in the sense that system performance does not increase in a linear manner, according to the number of processors. In fact, to effectively put into place an increase in the number of processors, it is necessary to raise the memory bandwidth, which means increasing the number of banks. If not, the processors will not be correctly supplied and the total computing speed will not increase just because more processors have been added. Moreover, if the number of processors and the number of banks are multiplied by a factor p, the complexity of interconnection between the memory and processors will increase by a factor of p^2. At a certain point, the state of the technology imposes a barrier on the maximal size of systems that are feasible.

1.2.2. *Memory hierarchy*

It is altogether possible to produce memory units with short access times, but which have a reduced capacity. Specifically, increasing the number and density of circuits allows us to add memory on the same chip as the processor. This memory can have an access time of one cycle, however, its size is limited. Nevertheless, the time unit to access large capacity memory can be reduced by putting access procedures into place, which address contiguous blocks of data.

Between a memory of a large size and the processor, there is "cache", which is rapid, and serves the purpose of temporary storage for data used by the processor. In order to optimize the bandwidth between the memory and cache, the transfers are carried out using small blocks of data called "cache lines".

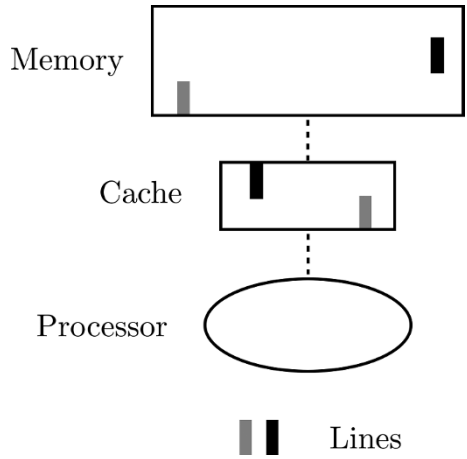


Figure 1.4. *Cache memory*

The lines transferred to the cache are only temporary copies of memory lines. The system manages the list of lines that are present in the cache. When the processor needs data, there are two possibilities:

– Either the line of data is already in the cache, in which case that is favorable, because the access time to the data would be that of the cache.

– Or, the line is not in the cache and therefore it must be sent from the memory. But before doing this, space in the cache needs to be freed up, so a former cache-line is sent back to the memory first. The cache line being sent back is preferably the one that had the longest inactive period, and is thus considered to be the least useful. Figure 1.5 illustrates this mechanism.

With this type of operation, a new potential problem crops up – the access times to the memory become non-uniform. The organization of the data structures in the code and their use patterns will strongly influence the smooth functioning of the memory system, which reduces both efficiency and performance levels.

First, the system relies on transfer mechanisms, by block or line, of data stored in the memory toward the cache, which is supposed to enhance the data

flow rate. Each time data are accessed in memory, the entire line that contains the data is recopied to the cache, before the data are finally transmitted to the processor. If only these data are subsequently used, it is evident that the transfer procedure is more costly than unitary access directly to the memory. However, if other data that are part of the memory line are used by the processor, either in the same instruction or in instructions that immediately follow, then the mechanism proves to be beneficial. Therefore, the use of "spatial locality of data" would be encouraged to access contiguous data in memory.

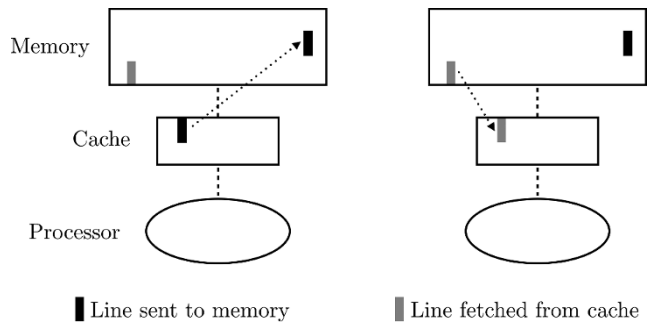


Figure 1.5. *Management of the cache memory*

Yet, if the processor is using data from the same line repeatedly in a short period, the data will stay in the cache line and provide for quick access. Therefore, we should try to group, in a short time frame, successive access to the same datum, which favors the use of "temporal locality of data".

In effect, it is the cache memory of the processor that loads it. To assure a supply to separate processors, separate caches are also needed.

As we can see, both caches share the same central memory. If both caches need to simultaneously access it, access times will be longer. Moreover, this can also produce the phenomenon of a conflict of access to the memory lines. In fact, if a line is stored in the cache, the processor that uses it could modify the data values. If another processor wants to access the same line, the updated content needs to be written to the central memory first. At any given point in time, a line can only be used in write mode by a single processor. So, at each instant, the main memory needs access to a list of the lines stored in the