Parallel Scientific Computing in C++ and MPI

A Seamless Approach to Parallel Algorithms and Their Implementation

GEORGE EM KARNIADAKIS ROBERT M. KIRBY II

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org

First published 2003

Printed in the United States of America

Typefaces Stone Serif 9/12 pt., Franklin Gothic Cond, and Futura Cond. Obl. *System* LaTeX 2_ε [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging in Publication Data

Karniadakis, George.

Parallel scientific computing in C++ and MPI : a seamless approach to parallel algorithms and their implementation / George Em Karniadakis and Robert M. Kirby II.

p. cm.

Includes bibliographical references and index.

ISBN 0-521-81754-4 – ISBN 0-521-52080-0 (pb.)

1. Parallel processing (Electronic computer) 2. C++ (Computer program language) 3. Data transmission systems. I. Kirby, Robert M., 1975– II. Title.

QA76.58 .K37 2003
004′.35 – dc21 2002034805

ISBN 0 521 81754 4 hardback
ISBN 0 521 52080 0 paperback

Contents

1

Scientific Computing and Simulation Science

1.1 WHAT IS SIMULATION?

Science and engineering have undergone a major transformation at the research level as well as at the development and technology level. The modern scientist and engineer spend more and more time in front of a laptop, a workstation, or a parallel supercomputer and less and less time in the physical laboratory or in the workshop. The virtual wind tunnel and the virtual biology laboratory are not a thing of the future; they are here! The old approach of "cut and try" has been replaced by "simulate and analyze" in several key technological areas such as aerospace applications, synthesis of new materials, design of new drugs, and chip processing and microfabrication. The new discipline of nanotechnology will be based primarily on large-scale computations and numerical experiments. The methods of scientific analysis and engineering design are changing continuously, affecting both our approach to the phenomena that we study as well as the range of applications that we address. Whereas there is an abundance of software available to be used as almost a "black box," working in new application areas requires good knowledge of fundamentals and mastering of effective new tools.

In the classical scientific approach, the physical system is first simplified and set in a form that suggests what type of phenomena and processes may be important and, correspondingly, what experiments are to be conducted. In the absence of any known type of governing equations, dimensional inter dependence between physical parameters can guide laboratory experiments in identifying key parametric studies. The database produced in the laboratory is then used to construct a simplified "engineering" model that, after field-test validation, will be used in other areas of research, product development, and design and possibly lead to new technological applications. This approach has been used almost invariably in every scientific discipline, from engineering and physics to chemistry and biology.

The simulation approach follows a parallel path but with some significant differences. First, the phase of the physical model analysis is more elaborate: The physical system is cast in a form governed by a set of partial differential equations, which represent continuum approximations to microscopic models. Such approximations are not possible for all systems, and sometimes the microscopic model should be used directly. Second, the laboratory experiment is replaced by simulation, that is, by a numerical experiment based on a discrete model. Such a model may represent a discrete approximation of the continuum partial differential equations, or it may simply represent a statistical representation of the microscopic model. Finite difference approximations

on a grid are examples of the first case, and Monte Carlo methods are examples of the second case. In either case, these algorithms have to be converted to software using an appropriate computer language, debugged, and run on a workstation or a parallel supercomputer. The output is usually a large number of files of a few megabytes to hundreds of gigabytes, being especially large for simulations of time-dependent phenomena. To be useful, this numerical database needs to be put into graphical form using various visualization tools, which may not always be suited for the particular application considered. Visualization can be especially useful during simulations where interactivity is required as the grid may be changing or the number of molecules may be increasing.

The simulation approach has already been followed by the majority of researchers across disciplines in the past few decades. The question is whether this is a new science and how one could formally obtain such skills. Moreover, does this constitute fundamental new knowledge or is it a "mechanical procedure," an ordinary skill that a chemist, a biologist, or an engineer will acquire easily as part of "training on the job" without specific formal education? It seems that the time has arrived where we need to reconsider boundaries between disciplines and reformulate the education of the future *simulation scientist*, an interdisciplinary scientist.

Let us reexamine some of the requirements following the various steps in the simulation approach. The first task is to select the right representation of the physical system by making consistent assumptions to derive the governing equations and the associated boundary conditions. The conservation laws should be satisfied; the entropy condition should not be violated; the uncertainty principle should be honored. The second task is to develop the right algorithmic procedure to discretize the continuum model or represent the dynamics of the atomistic model. The choices are many, but which algorithm is the most accurate one, or the simplest one, or the most efficient one? These algorithms do not belong to a discipline! Finite elements, first developed by the famous mathematician Richard Courant and rediscovered by civil engineers, have found their way into every engineering discipline as well as into physics, geology, and other fields. Molecular dynamics simulations are practiced by chemists, biologists, material scientists, and others. The third task is to compute efficiently in the ever-changing world of supercomputing. How efficient the computation is translates to how realistic of a problem is solved and therefore how useful the results can be to applications. The fourth task is to assess the accuracy of the results in cases where no direct confirmation from physical experiments is possible, such as in nanotechnology or in biosystems or in astrophysics. Reliability of the predicted numerical answer is an important issue in the simulation approach because some of the answers may lead to new physics or false physics contained in the discrete model or induced by the algorithm but not derived from the physical problem. Finally, visualizing the simulated phenomenon, in most cases in three-dimensional space and in time, by employing proper computer graphics (a separate specialty on its own) completes the full simulation cycle. The rest of the steps followed are similar to those of the classical scientific approach.

In classical science we are dealing with matter and therefore *atoms*, but in simulation we are dealing with information and therefore *bits*; so it is atoms versus bits. We should, therefore, recognize the simulation scientist as a separate scientist, the same way we recognized just a few decades ago the computer scientist as different than the

electrical engineer or the applied mathematician. The new scientist is certainly not a computer scientist, although he or she should be computer literate in both software and hardware. The simulation scientist, is not a physicist, although a sound physics background is needed. Nor is he or she an applied mathematician, although expertise in mathematical analysis and approximation theory is needed.

With the rapid and simultaneous advances in software and computer technology, especially commodity computing and the so-called soupercomputing, every scientist and engineer will have on his or her desk an advanced simulation kit of tools consisting of a software library and multiprocessor computers that will make analysis, product development, and design more optimal and cost effective. But what the future scientists and engineers will need, first and foremost, is a solid interdisciplinary education.

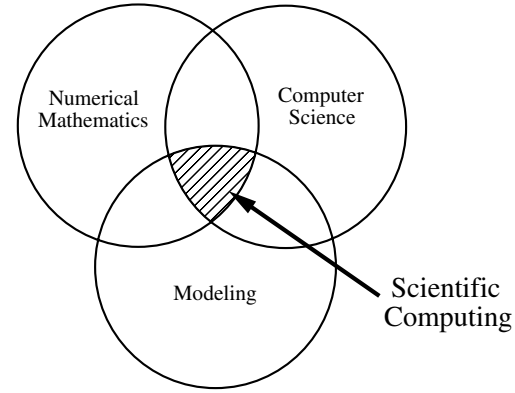


Figure 1.1: Definition of scientific computing as the intersection of numerical mathematics, computer science, and modeling.

Scientific computing is the heart of simulation science, and this is the subject of this book. The emphasis is on a balance between classical and modern elements of numerical mathematics and of computer science, but we have selected the topics based on broad modeling concepts encountered in physico-chemical and biological sciences, or even economics (see Figure 1.1).

1.2 A SEAMLESS APPROACH PATH

Our aim in writing this book has been to provide the student, the future simulation scientist, with a seamless approach to numerical algorithms, modern programming techniques, and parallel computing. Often times such concepts and tools are taught serially across different courses and different textbooks, and hence the interconnection between them is not immediately apparent. The necessity of integrating concepts and tools usually comes after such courses are concluded, for example, during a first job or a thesis project, thus forcing the student to synthesize what is perceived to be three independent subfields into one to produce a solution. Although this process is undoubtly valuable, it is time consuming and in many cases it may not lead to an effective combination of concepts and tools. Moreover, from the pedagogical point of view, the integrated seamless approach can stimulate the student simultaneously through the eyes of multiple disciplines, thus leading to enhanced understanding of subjects in scientific computing.

As discussed in the previous section, in the scientific simulation approach there are several successive stages that lead from

1. the real-world problem to its mathematical formulation,
2. the mathematical description to the computer implementation and solution, and
3. the numerical solution to visualization and analysis.

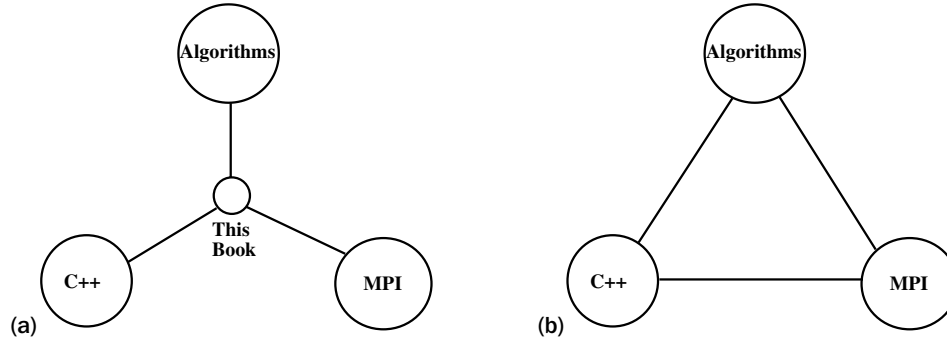


Figure 1.2: (a) Simultaneous integration of concepts in contrast with (b) the classical serial integration.

In this book, we concentrate on stage 2, which includes not only the mathematics of numerical linear algebra and discretization but also the implementation of these concepts in C++ and MPI.

There are currently several excellent textbooks and monographs on these topics, but these lack the type of integration that we propose. For example, the book by Golub and Ortega [45] introduces pedagogically all the basic parallel concepts, but a gap remains between the parallel formulation and its implementation. Similarly, the books by Trefethen and Bau [88] and Demmel [26] provide rigor and great insight into numerical linear algebra algorithms, but they do not provide sufficient material on discretization and implementation. Popular books in C++ (e.g., by Stroustrup [86]) and MPI (e.g., by Pacheco [73]) are references that teach programming using disconnected algorithmic examples, which is useful for acquiring general programming skills but not for parallel scientific computing. Our book treats numerics, parallelism, and programming equally and simultaneously by placing the reader at a vantage point between the three areas, as shown in the schematic of Figure 1.2a, and in contrast with the classical approach of connecting the three subjects serially, as illustrated in Figure 1.2b.

1.3 THE CONCEPT OF PROGRAMMING LANGUAGE

In studying computer languages, we want to study a new way of interacting with the computer. Most people are familiar with the use of software purchased online or from your local computer store; such software ranges from word processors and spreadsheets to interactive games. But have you ever wondered how these things are created? How do you actually "write" software? Throughout this book we will be teaching through both lecture and example how to create computer software that solves scientific problems. Our purpose is not to teach you how to write computer games and the like, but the knowledge gained here can be used to devise your own software endeavors.

It has been stated by some that the computer is a pretty dumb device, in that it only understands two things – *on* and *off*. Like sending Morse code over a telegraph wire with signals of dots and dashes, the computer uses sequences of zeros and ones as its language. The zeros and ones may seem inefficient, but it is not just the data used,

but the rules applied to the data that make the computer powerful. This concept, in theory, is no different than human language. If we were to set before you a collection of symbols, say a, b, c, d, . . . z, and indicate to you that you can use these to express even the most complex thoughts and emotions of the human mind and heart, you would think we were crazy. Just twenty-six little symbols? How can this be? We know, however, that it is not merely the symbols that are important but the rules used to combine the symbols. If you adhere to the rules defined by the English language, then books like this can be written using merely combinations of the twenty-six characters! How is this similar to the computer? The computer is a complex device for executing instructions. These instructions are articulated by using our *two-base* characters, 0 and 1, along with a collection of rules for combining them together. This brings us to our first axiom:

AXIOM I: *Computers are machines that execute instructions. If someone is not telling the computer what to do, it does nothing.*

Most people have had some experience with computers, and immediately they will read this statement and say: "Hey, I have had my computer do all kinds of things that I didn't want!" Ah, but read the axiom carefully. The key to this axiom is the use of the term *someone*. The one thing to keep in mind is that some human, or collection of humans, developed software to tell the computer what to do. At a relatively low level, this would be the people who wrote the operating system used by the computer. At a higher level, this would be the people who developed the word processor or game that you were using. In both cases, however, someone determined how the computer would act and react to your input. We want you, after reading this book and understanding the concepts herein, to be able to be in the driver's seat. This leads us to our second axiom:

AXIOM II: *Computer programming languages allow humans a simplified means of giving the computer instructions.*

We tell you that we want you to be in the driver's seat, and you tell us "I don't want to learn how to communicate in zeros and ones. Learning English was hard enough!" You can imagine how slowly the computer age would have progressed if every programming class consisted of the following lecture scheme. Imagine the first day of class. On the first day, the instructor tells you that you will be learning the two basic components of the computer language today: 0 and 1. The instructor may force you to say zero and one a few times and then to write zero and one many times on a piece of paper for practice, but then, what else would there be to learn concerning your character set? Class dismissed. Then, for the rest of the semester, you would spend your time learning how to combine zeros and ones to get the computer to do what you want. Your first assignment might be to add two numbers a and b and to store the result in c (i.e., c = a + b). You end up with something that looks like the following:

```
01011001010001000111010101000100
01011001011100100010100101000011
00111010101000100111010100100101
01011101010101010101010000111101
```

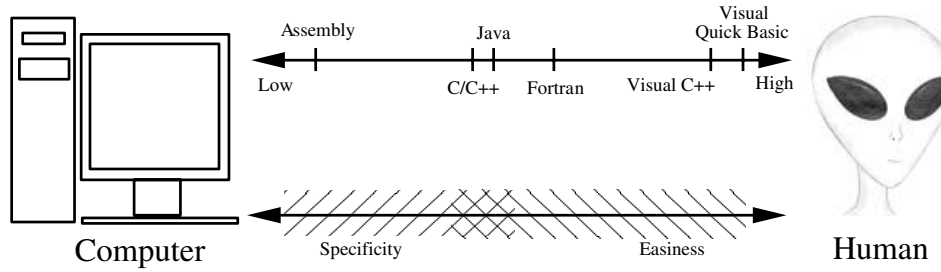


Figure 1.3: Programming languages provide us a means of bridging the gap between the computer and the human.

This seems like a longwinded way of saying

$$c = a + b,$$

but this is what the computer understands, so this is how we must communicate with it. However, humans do not communicate in this fashion. Human language and thought use a higher abstraction than this. How can we make a bridge for this gap? We bridge this gap via programming languages (see Figure 1.3).

The first programming language we will mention is *assembly*. The unique property of assembly is that for each instruction, there is a one-to-one correspondence between a command in assembly and a computer-understandable command (in zeros and ones). For instance, instead of writing

```
01011001010001000111010101000100
```

as a command, you could write "load a $1." This tells the computer to load the contents of the memory location denoted by "a" into register $1 in the computer's CPU (central processing unit). This is much better than before. Obviously, this sequence of commands is much easier for the human to understand. This was a good start, but assembly is still considered a "low-level language." By low level we mean that one instruction in assembly is equal to one computer instruction. But as we said earlier, we want to be able to think on a higher level. Hence, "higher level" languages were introduced. Higher level languages are those in which one instruction in the higher level language equals one or more computer-level instructions. We want a computer language where we can say "c = a + b"; this would be equivalent to saying

```
load a $1
load b $2
add $1 $2 $3
save $3 c
```

One high-level instruction was equivalent to four lower level instructions (here written in pseudo-assembly so that you can follow what is going on). This is preferable for many reasons. First, we as humans would like to spend our time thinking about how to solve the problem, not just trying to remember (and write) all the assembly code! Second, by writing in a higher level language, we can write code that can work on multiple computers, because the translation of the higher level code can be done by a compiler into the assembly code of the processor on which we are running.

As you read through this book and do the exercises found herein, always be mindful that our goal is to utilize the computer for accomplishing scientific tasks encountered in simulation science. At a high level, there is a science or engineering problem to solve, and we want to use the computer as a tool for solving the problem. The means by which we will use the computer is through the writing and execution of programs written using the computing language C++ and the parallel message passing libraries of MPI.

1.4 WHY USE C++ AND WHAT IS MPI?

The algorithms we present in the book can certainly be implemented in other languages (e.g., FORTRAN or Java) as well as using other communication libraries, such as PVM (parallel virtual machine). However, we commit to a specific language and parallel library to provide the student with the immediate ability to experiment with the concepts presented. To this end, we have chosen C++ as our programming language for a multitude of reasons. First, it provides an object-oriented infrastructure that accommodates a natural breakdown of the problem into a collection of data structures and operations on those structures. Second, the use of C++ transcends many disciplines beyond engineering, where traditionally FORTRAN has been the prevailing language. Third, C++ is a language naturally compatible with the basic algorithmic concepts of

- partitioning,
- recursive function calling,
- dynamic memory allocation, and
- encapsulation.

Similarly, we commit to MPI (message passing interface) as a message passing library because it accommodates a natural and simple partitioning of the problem, it provides portability and efficiency, and it has received wide acceptance by academia and industry.

The simultaneous integration we propose in this book will be accomplished by carefully presenting related concepts from all three subareas. Moving from one chapter to the next requires different dosages of new material in algorithms and tools. This is explained graphically in Figure 1.4, which shows that although new algorithms are introduced at an approximately constant rate, the introduction of new C++ and MPI material vary inversely. We begin with an emphasis on the basics of the language, which allows the student to immediately work on the simple algorithms introduced initially; as the book progresses and the computational complexity of algorithms increases the use of parallel constructs and libraries is emphasized.

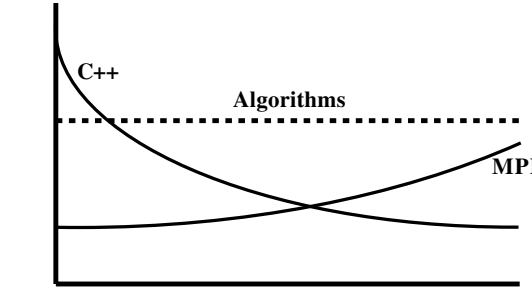
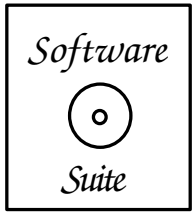


Figure 1.4: Progression of new material throughout the book in the three areas shown in Figure 1.2.

More specifically, to help facilitate the student's immersion into object-oriented thinking, we provide a library of *classes* and *functions* for use throughout the book. The classes contained in this library are used from

the very beginning of the book as a natural, user-defined extension of C++. As the book progresses, the underlying logic and programming implementation of these classes are explained, bringing the student to a deeper understanding of the development of C++ classes. We will denote all classes used within the book and not inherent to C++ with the letters SC, such as the classes *SCVector* and *SCMatrix*.



The SC notation is used to clearly distinguish between C++ defined and user-defined data types and also to accentuate the utility of user-defined types within the C++ programming language. As students become more familiar and confident in their ability to devise and use data types, we encourage them to use these facilities provided by the language for more effective programming and problem solving. All the codes of this book and many more examples are included in the *software suite*, which is distributed with this book.

1.5 WHAT ABOUT OpenMP?

Because of the recent proliferation of distributed shared-memory (DSM) machines in the scientific computing community, there is much interest in how best to appropriately utilize both the distributed and the shared-memory partitioning of these systems. MPI provides an efficient means of parallel communication among a *distributed* collection of machines; however, not all MPI implementations take advantage of *shared memory* when it is available between processors (the basic premise being that two processors, which share common memory, can communicate with each other faster through the use of the shared medium than through other communication means).

OpenMP (open multi processing) was introduced to provide a means of implementing shared-memory parallelism in FORTRAN and C/C++ programs. Specifically, OpenMP specifies a set of environment variables, compiler directives, and library routines to be used for shared-memory parallelization. OpenMP was specifically designed to exploit certain characteristics of shared-memory architectures such as the ability to directly access memory throughout the system with low latency and very fast shared-memory locks. To learn more about OpenMP, visit **`www.openmp.org`**.

A new parallel programming paradigm is emerging in which both the MPI and OpenMP are used for parallelization. In a distributed shared-memory architecture, OpenMP would be used for intranode communication (i.e., between a collection of processors that share the same memory subsystem) and MPI would be used for internode communication (i.e., between distinct distributed collections of processors). The combination of these two parallelization methodologies may provide the most effective means of fully exploiting modern DSM systems.

1.6 ALGORITHMS AND TOP TEN LIST

The Greeks and Romans invented many scientific and engineering algorithms, but it is believed that the term "algorithm" stems from the name of the ninth-century Arab mathematician *al-Khwarizmi*, who wrote the book *al-jabr wa'l muqabalach*, which

eventually evolved into today's high school algebra textbooks. He was perhaps the first to stress systematic procedures for solving mathematical problems. Since then, some truly ingenious algorithms have been invented, but the algorithms that have formed the foundations of scientific computing as a separate discipline were developed in the second part of the twentieth century. Dongarra and Sullivan put together a list of the *top ten algorithms* of the twentieth century [33]. According to these authors, the following algorithms (listed in chronological order) had the greatest influence on science and engineering in the past:

1. **1946:** The Monte Carlo method for modeling probabilistic phenomena.
2. **1947:** The Simplex method for linear optimization problems.
3. **1950:** The Krylov subspace iteration method for fast linear solvers and eigensolvers.
4. **1951:** The Householder matrix decomposition to express a matrix as a product of simpler matrices.
5. **1957:** The FORTRAN compiler that liberated scientists and engineers from programming in assembly.
6. **1959–1961:** The QR algorithm to compute many eigenvalues.
7. **1962:** The Quicksort algorithm to put things in numerical or alphabetical order fast.
8. **1965:** The fast Fourier transform to reduce operation count in Fourier series representation.
9. **1977:** The integer relation detection algorithm, which is useful for bifurcations and in quantum field theory.
10. **1987:** The fast multipole algorithm for N-body problems.

Although there is some debate as to the relative importance of these algorithms or the absence of other important methods in the list (e.g., finite differences and finite elements), this selection by Dongarra and Sullivan reflects some of the thrusts in scientific computing in the past. The appearance of the FORTRAN compiler, for example, represents the historic transition from assembly language to higher level languages, as discussed earlier. In fact, the first FORTRAN compiler was written in 23,500 assembly language instructions! FORTRAN has been used extensively in the past, especially in the engineering community, but most of the recent scientific computing software has been rewritten in C++ (e.g., the Numerical Recipes [75]).

In this book we will cover in detail the algorithms 3, 4, 6, and 8 from the aforementioned mentioned list, including many more recent versions, which provide more robustness with respect to round-off errors and efficiency in the context of parallel computing. We will also present discretizations of ordinary and partial differential equations using several finite difference formulations.

Many new algorithms will probably be invented in the twenty-first century – hopefully some of them from the readers of this book! As Dongarra and Sullivan noted, "This century will not be very restful for us, but is not going to be dull either!"