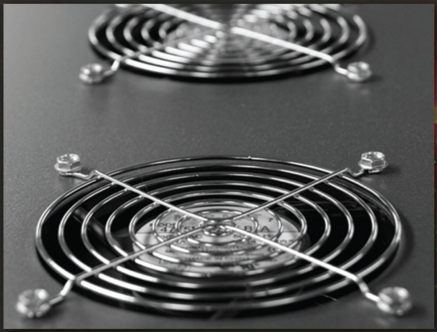
Chapman & Hall/CRC
Computational Science Series

# Introduction to High Performance Computing for Scientists and Engineers



Georg Hager
Gerhard Wellein

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

# Introduction to High Performance Computing for Scientists and Engineers

# Introduction to High Performance Computing for Scientists and Engineers

Georg Hager

Gerhard Wellein

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number: 978-1-4398-1192-4 (Paperback)

**Library of Congress Cataloging-in-Publication Data**

Hager, Georg.
Introduction to high performance computing for scientists and engineers / Georg Hager and Gerhard Wellein.
p. cm. -- (Chapman & Hall/CRC computational science series ; 7)
Includes bibliographical references and index.
ISBN 978-1-4398-1192-4 (alk. paper)
1. High performance computing. I. Wellein, Gerhard. II. Title.

QA76.88.H34 2011
004'.35--dc22 2010009624

Dedicated to Konrad Zuse (1910–1995)

He developed and built the world's first fully automated, freely programmable computer with binary floating-point arithmetic in 1941.

# Contents

**Foreword** **xiii**

**Preface** **xv**

**About the authors** **xxi**

**List of acronyms and abbreviations** **xxiii**

**1 Modern processors** **1**
1.1 Stored-program computer architecture . . . . . 1
1.2 General-purpose cache-based microprocessor architecture . . . . . 2
1.2.1 Performance metrics and benchmarks . . . . . 3
1.2.2 Transistors galore: Moore's Law . . . . . 7
1.2.3 Pipelining . . . . . 9
1.2.4 Superscalarity . . . . . 13
1.2.5 SIMD . . . . . 14
1.3 Memory hierarchies . . . . . 15
1.3.1 Cache . . . . . 15
1.3.2 Cache mapping . . . . . 18
1.3.3 Prefetch . . . . . 20
1.4 Multicore processors . . . . . 23
1.5 Multithreaded processors . . . . . 26
1.6 Vector processors . . . . . 28
1.6.1 Design principles . . . . . 29
1.6.2 Maximum performance estimates . . . . . 31
1.6.3 Programming for vector architectures . . . . . 32

**2 Basic optimization techniques for serial code** **37**
2.1 Scalar profiling . . . . . 37
2.1.1 Function- and line-based runtime profiling . . . . . 38
2.1.2 Hardware performance counters . . . . . 41
2.1.3 Manual instrumentation . . . . . 45
2.2 Common sense optimizations . . . . . 45
2.2.1 Do less work! . . . . . 45
2.2.2 Avoid expensive operations! . . . . . 46
2.2.3 Shrink the working set! . . . . . 47

2.3 Simple measures, large impact . . . . . . . . . . . . . . . . . . . . 47
2.3.1 Elimination of common subexpressions . . . . . . . . . . . 47
2.3.2 Avoiding branches . . . . . . . . . . . . . . . . . . . . . . . 48
2.3.3 Using SIMD instruction sets . . . . . . . . . . . . . . . . . 49
2.4 The role of compilers . . . . . . . . . . . . . . . . . . . . . . . . . 51
2.4.1 General optimization options . . . . . . . . . . . . . . . . . 52
2.4.2 Inlining . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
2.4.3 Aliasing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
2.4.4 Computational accuracy . . . . . . . . . . . . . . . . . . . 54
2.4.5 Register optimizations . . . . . . . . . . . . . . . . . . . . 55
2.4.6 Using compiler logs . . . . . . . . . . . . . . . . . . . . . . 55
2.5 C++ optimizations . . . . . . . . . . . . . . . . . . . . . . . . . . 56
2.5.1 Temporaries . . . . . . . . . . . . . . . . . . . . . . . . . . 56
2.5.2 Dynamic memory management . . . . . . . . . . . . . . . 59
2.5.3 Loop kernels and iterators . . . . . . . . . . . . . . . . . . 60

**3 Data access optimization 63**
3.1 Balance analysis and lightspeed estimates . . . . . . . . . . . . . . 63
3.1.1 Bandwidth-based performance modeling . . . . . . . . . . 63
3.1.2 The STREAM benchmarks . . . . . . . . . . . . . . . . . . 67
3.2 Storage order . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
3.3 Case study: The Jacobi algorithm . . . . . . . . . . . . . . . . . . 71
3.4 Case study: Dense matrix transpose . . . . . . . . . . . . . . . . . 74
3.5 Algorithm classification and access optimizations . . . . . . . . . . 79
3.5.1 $O(N)/O(N)$ . . . . . . . . . . . . . . . . . . . . . . . . . 79
3.5.2 $O(N^2)/O(N^2)$ . . . . . . . . . . . . . . . . . . . . . . . 79
3.5.3 $O(N^3)/O(N^2)$ . . . . . . . . . . . . . . . . . . . . . . . 84
3.6 Case study: Sparse matrix-vector multiply . . . . . . . . . . . . . . 86
3.6.1 Sparse matrix storage schemes . . . . . . . . . . . . . . . . 86
3.6.2 Optimizing JDS sparse MVM . . . . . . . . . . . . . . . . 89

**4 Parallel computers 95**
4.1 Taxonomy of parallel computing paradigms . . . . . . . . . . . . . 96
4.2 Shared-memory computers . . . . . . . . . . . . . . . . . . . . . . 97
4.2.1 Cache coherence . . . . . . . . . . . . . . . . . . . . . . . 97
4.2.2 UMA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
4.2.3 ccNUMA . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
4.3 Distributed-memory computers . . . . . . . . . . . . . . . . . . . . 102
4.4 Hierarchical (hybrid) systems . . . . . . . . . . . . . . . . . . . . . 103
4.5 Networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
4.5.1 Basic performance characteristics of networks . . . . . . . . 104
4.5.2 Buses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
4.5.3 Switched and fat-tree networks . . . . . . . . . . . . . . . . 110
4.5.4 Mesh networks . . . . . . . . . . . . . . . . . . . . . . . . 112
4.5.5 Hybrids . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

**5 Basics of parallelization 115**
5.1 Why parallelize? . . . 115
5.2 Parallelism . . . 116
5.2.1 Data parallelism . . . 116
5.2.2 Functional parallelism . . . 119
5.3 Parallel scalability . . . 120
5.3.1 Factors that limit parallel execution . . . 120
5.3.2 Scalability metrics . . . 122
5.3.3 Simple scalability laws . . . 123
5.3.4 Parallel efficiency . . . 125
5.3.5 Serial performance versus strong scalability . . . 126
5.3.6 Refined performance models . . . 128
5.3.7 Choosing the right scaling baseline . . . 130
5.3.8 Case study: Can slower processors compute faster? . . . 131
5.3.9 Load imbalance . . . 137

**6 Shared-memory parallel programming with OpenMP 143**
6.1 Short introduction to OpenMP . . . 143
6.1.1 Parallel execution . . . 144
6.1.2 Data scoping . . . 146
6.1.3 OpenMP worksharing for loops . . . 147
6.1.4 Synchronization . . . 149
6.1.5 Reductions . . . 150
6.1.6 Loop scheduling . . . 151
6.1.7 Tasking . . . 153
6.1.8 Miscellaneous . . . 154
6.2 Case study: OpenMP-parallel Jacobi algorithm . . . 156
6.3 Advanced OpenMP: Wavefront parallelization . . . 158

**7 Efficient OpenMP programming 165**
7.1 Profiling OpenMP programs . . . 165
7.2 Performance pitfalls . . . 166
7.2.1 Ameliorating the impact of OpenMP worksharing constructs 168
7.2.2 Determining OpenMP overhead for short loops . . . 175
7.2.3 Serialization . . . 177
7.2.4 False sharing . . . 179
7.3 Case study: Parallel sparse matrix-vector multiply . . . 181

**8 Locality optimizations on ccNUMA architectures 185**
8.1 Locality of access on ccNUMA . . . 185
8.1.1 Page placement by first touch . . . 186
8.1.2 Access locality by other means . . . 190
8.2 Case study: ccNUMA optimization of sparse MVM . . . 190
8.3 Placement pitfalls . . . 192
8.3.1 NUMA-unfriendly OpenMP scheduling . . . 192

8.3.2 File system cache . . . . . . . . . . . . . . . . . . . 194
8.4 ccNUMA issues with C++ . . . . . . . . . . . . . . . . . . 197
8.4.1 Arrays of objects . . . . . . . . . . . . . . . . . . . 197
8.4.2 Standard Template Library . . . . . . . . . . . . . . . 199

**9 Distributed-memory parallel programming with MPI 203**
9.1 Message passing . . . . . . . . . . . . . . . . . . . . . 203
9.2 A short introduction to MPI . . . . . . . . . . . . . . . . 205
9.2.1 A simple example . . . . . . . . . . . . . . . . . . . 205
9.2.2 Messages and point-to-point communication . . . . . . . . 207
9.2.3 Collective communication . . . . . . . . . . . . . . . . 213
9.2.4 Nonblocking point-to-point communication . . . . . . . . . 216
9.2.5 Virtual topologies . . . . . . . . . . . . . . . . . . . 220
9.3 Example: MPI parallelization of a Jacobi solver . . . . . . . . 224
9.3.1 MPI implementation . . . . . . . . . . . . . . . . . . 224
9.3.2 Performance properties . . . . . . . . . . . . . . . . . 230

**10 Efficient MPI programming 235**
10.1 MPI performance tools . . . . . . . . . . . . . . . . . . 235
10.2 Communication parameters . . . . . . . . . . . . . . . . 239
10.3 Synchronization, serialization, contention . . . . . . . . . . 240
10.3.1 Implicit serialization and synchronization . . . . . . . . . 240
10.3.2 Contention . . . . . . . . . . . . . . . . . . . . . . 243
10.4 Reducing communication overhead . . . . . . . . . . . . . 244
10.4.1 Optimal domain decomposition . . . . . . . . . . . . . 244
10.4.2 Aggregating messages . . . . . . . . . . . . . . . . . 248
10.4.3 Nonblocking vs. asynchronous communication . . . . . . . 250
10.4.4 Collective communication . . . . . . . . . . . . . . . . 253
10.5 Understanding intranode point-to-point communication . . . . . . 253

**11 Hybrid parallelization with MPI and OpenMP 263**
11.1 Basic MPI/OpenMP programming models . . . . . . . . . . . 264
11.1.1 Vector mode implementation . . . . . . . . . . . . . . . 264
11.1.2 Task mode implementation . . . . . . . . . . . . . . . . 265
11.1.3 Case study: Hybrid Jacobi solver . . . . . . . . . . . . . 267
11.2 MPI taxonomy of thread interoperability . . . . . . . . . . . 268
11.3 Hybrid decomposition and mapping . . . . . . . . . . . . . 270
11.4 Potential benefits and drawbacks of hybrid programming . . . . . 273

**A Topology and affinity in multicore environments 277**
A.1 Topology . . . . . . . . . . . . . . . . . . . . . . . . 279
A.2 Thread and process placement . . . . . . . . . . . . . . . . 280
A.2.1 External affinity control . . . . . . . . . . . . . . . . . 280
A.2.2 Affinity under program control . . . . . . . . . . . . . . 283
A.3 Page placement beyond first touch . . . . . . . . . . . . . . 284

**B Solutions to the problems** **287**

**Bibliography** **309**

**Index** **323**

# *Foreword*

Georg Hager and Gerhard Wellein have developed a very approachable introduction to high performance computing for scientists and engineers. Their style and descriptions are easy to read and follow.

The idea that computational modeling and simulation represent a new branch of scientific methodology, alongside theory and experimentation, was introduced about two decades ago. It has since come to symbolize the enthusiasm and sense of importance that people in our community feel for the work they are doing. Many of us today want to hasten that growth and believe that the most progressive steps in that direction require much more understanding of the vital core of computational science: *software and the mathematical models and algorithms it encodes*. Of course, the general and widespread obsession with hardware is understandable, especially given exponential increases in processor performance, the constant evolution of processor architectures and supercomputer designs, and the natural fascination that people have for big, fast machines. But when it comes to advancing the cause of computational modeling and simulation as a new part of the scientific method there is no doubt that the complex software "ecosystem" it requires must take its place on the center stage.

At the application level science has to be captured in mathematical models, which in turn are expressed algorithmically and ultimately encoded as software. Accordingly, on typical projects the majority of the funding goes to support this translation process that starts with scientific ideas and ends with executable software, and which over its course requires intimate collaboration among domain scientists, computer scientists, and applied mathematicians. This process also relies on a large infrastructure of mathematical libraries, protocols, and system software that has taken years to build up and that must be maintained, ported, and enhanced for many years to come if the value of the application codes that depend on it are to be preserved and extended. The software that encapsulates all this time, energy, and thought routinely outlasts (usually by years, sometimes by decades) the hardware it was originally designed to run on, as well as the individuals who designed and developed it.

This book covers the basics of modern processor architecture and serial optimization techniques that can effectively exploit the architectural features for scientific computing. The authors provide a discussion of the critical issues in data movement and illustrate this with examples. A number of central issues in high performance computing are discussed at a level that is easily understandable. The use of parallel processing in shared, nonuniform access, and distributed memories is discussed. In addition the popular programming styles of OpenMP, MPI and mixed programming are highlighted.

We live in an exciting time in the use of high performance computing and a period that promises unmatched performance for those who can effectively utilize the systems for high performance computing. This book presents a balanced treatment of the theory, technology, architecture, and software for modern high performance computers and the use of high performance computing systems. The focus on scientific and engineering problems makes it both educational and unique. I highly recommend this timely book for scientists and engineers, and I believe it will benefit many readers and provide a fine reference.

*Jack Dongarra*

University of Tennessee
Knoxville, Tennessee
USA

# *Preface*

When Konrad Zuse constructed the world's first fully automated, freely programmable computer with binary floating-point arithmetic in 1941 [H129], he had great visions regarding the possible use of his revolutionary device, not only in science and engineering but in all sectors of life [H130]. Today, his dream is reality: Computing in all its facets has radically changed the way we live and perform research since Zuse's days. Computers have become essential due to their ability to perform calculations, visualizations, and general data processing at an incredible, ever-increasing speed. They allow us to offload daunting routine tasks and communicate without delay.

Science and engineering have profited in a special way from this development. It was recognized very early that computers can help tackle problems that were formerly too computationally challenging, or perform *virtual* experiments that would be too complex, expensive, or outright dangerous to carry out in reality. *Computational fluid dynamics*, or CFD, is a typical example: The simulation of fluid flow in arbitrary geometries is a standard task. No airplane, no car, no high-speed train, no turbine bucket enters manufacturing without prior CFD analysis. This does not mean that the days of wind tunnels and wooden mock-ups are numbered, but that computer simulation supports research and engineering as a third pillar beside theory and experiment, not only on fluid dynamics but nearly all other fields of science. In recent years, *pharmaceutical drug design* has emerged as a thrilling new application area for fast computers. Software enables chemists to discover reaction mechanisms literally at the click of their mouse, simulating the complex dynamics of the large molecules that govern the inner mechanics of life. On even smaller scales, *theoretical solid state physics* explores the structure of solids by modeling the interactions of their constituents, nuclei and electrons, on the quantum level [A79], where the sheer number of degrees of freedom rules out any analytical treatment in certain limits and requires vast computational resources. The list goes on and on: Quantum chromodynamics, materials science, structural mechanics, and medical image processing are just a few further application areas.

Computer-based simulations have become ubiquitous standard tools, and are indispensable for most research areas both in academia and industry. Although the power of the PC has brought many of those computational chores to the researcher's desktop, there was, still is and probably will ever be this special group of people whose requirements on storage, main memory, or raw computational speed cannot be met by a single desktop machine. High performance parallel computers come to their rescue.

Employing high performance computing (HPC) as a research tool demands at least a basic understanding of the hardware concepts and software issues involved. This is already true when only using turnkey application software, but it becomes essential if code development is required. However, in all our years of teaching and working with scientists and engineers we have learned that such knowledge is *volatile* — in the sense that it is hard to establish and maintain an adequate competence level within the different research groups. The new PhD student is all too often left alone with the steep learning curve of HPC, but who is to blame? After all, the goal of research and development is to make *scientific progress*, for which HPC is just a tool. It is essential, sometimes unwieldy, and always expensive, but it is still a tool. Nevertheless, writing efficient and parallel code is the admission ticket to high performance computing, which was for a long time an exquisite and small world. Technological changes have brought parallel computing first to the departmental level and recently even to the desktop. In times of stagnating single processor capabilities and increasing parallelism, a growing audience of scientists and engineers must be concerned with performance and scalability. These are the topics we are aiming at with this book, and the reason we wrote it was to make the knowledge about them less volatile.

Actually, a lot of good literature exists on all aspects of computer architecture, optimization, and HPC [S1, R34, S2, S3, S4]. Although the basic principles haven't changed much, a lot of it is outdated at the time of writing: We have seen the decline of vector computers (and also of one or the other highly promising microprocessor design), ubiquitous SIMD capabilities, the advent of multicore processors, the growing presence of ccNUMA, and the introduction of cost-effective high-performance interconnects. Perhaps the most striking development is the absolute dominance of x86-based commodity clusters running the Linux OS on Intel or AMD processors. Recent publications are often focused on very specific aspects, and are unsuitable for the student or the scientist who wants to get a fast overview and maybe later dive into the details. Our goal is to provide a solid introduction to the architecture and programming of high performance computers, with an emphasis on performance issues. In our experience, users all too often have no idea what factors limit time to solution, and whether it makes sense to think about optimization at all. Readers of this book will get an intuitive understanding of performance limitations without much computer science ballast, to a level of knowledge that enables them to understand more specialized sources. To this end we have compiled an extensive bibliography, which is also available online in a hyperlinked and commented version at the book's Web site: http://www.hpc.rrze.uni-erlangen.de/HPC4SE/.

## Who this book is for

We believe that working in a scientific computing center gave us a unique view of the requirements and attitudes of users as well as manufacturers of parallel computers. Therefore, everybody who has to deal with high performance computing may profit from this book: Students and teachers of computer science, computational engineering, or any field even marginally concerned with simulation may use it as an accompanying textbook. For scientists and engineers who must get a quick grasp of HPC basics it can be a starting point to prepare for more advanced literature. And finally, professional cluster builders can definitely use the knowledge we convey to provide a better service to their customers. The reader should have some familiarity with programming and high-level computer architecture. Even so, we must emphasize that it is an introduction rather than an exhaustive reference; the *Encyclopedia of High Performance Computing* has yet to be written.

## What's in this book, and what's not

High performance computing as we understand it deals with the *implementations* of given algorithms (also commonly referred to as "code"), and the *hardware* they run on. We assume that someone who wants to use HPC resources is already aware of the different algorithms that can be used to tackle their problem, and we make no attempt to provide alternatives. Of course we have to pick certain examples in order to get the point across, but it is always understood that there may be other, and probably more adequate algorithms. The reader is then expected to use the strategies learned from our examples.

Although we tried to keep the book concise, the temptation to cover everything is overwhelming. However, we deliberately (almost) ignore very recent developments like modern accelerator technologies (GPGPU, FPGA, Cell processor), mostly because they are so much in a state of flux that coverage with any claim of depth would be almost instantly outdated. One may also argue that high performance input/output should belong in an HPC book, but we think that efficient parallel I/O is an advanced and highly system-dependent topic, which is best treated elsewhere. On the software side we concentrate on basic sequential optimization strategies and the dominating parallelization paradigms: shared-memory parallelization with OpenMP and distributed-memory parallel programming with MPI. Alternatives like Unified Parallel C (UPC), Co-Array Fortran (CAF), or other, more modern approaches still have to prove their potential for getting at least as efficient, and thus accepted, as MPI and OpenMP.

Most concepts are presented on a level independent of specific architectures, although we cannot ignore the dominating presence of commodity systems. Thus, when we show case studies and actual performance numbers, those have usually been obtained on x86-based clusters with standard interconnects. Almost all code examples are in Fortran; we switch to C or C++ only if the peculiarities of those languages are relevant in a certain setting. Some of the codes used for producing benchmark results are available for download at the book's Web site: http://www.hpc.rrze.uni-erlangen.de/HPC4SE/.

This book is organized as follows: In Chapter 1 we introduce the architecture of modern cache-based microprocessors and discuss their inherent performance limitations. Recent developments like multicore chips and simultaneous multithreading (SMT) receive due attention. Vector processors are briefly touched, although they have all but vanished from the HPC market. Chapters 2 and 3 describe general optimization strategies for serial code on cache-based architectures. Simple models are used to convey the concept of "best possible" performance of loop kernels, and we show how to raise those limits by code transformations. Actually, we believe that performance modeling of applications on all levels of a system's architecture is of utmost importance, and we regard it as an indispensable guiding principle in HPC.

In Chapter 4 we turn to parallel computer architectures of the shared-memory and the distributed-memory type, and also cover the most relevant network topologies. Chapter 5 then covers parallel computing on a theoretical level: Starting with some important parallel programming patterns, we turn to performance models that explain the limitations on parallel scalability. The questions why and when it can make sense to build massively parallel systems with "slow" processors are answered along the way. Chapter 6 gives a brief introduction to OpenMP, which is still the dominating parallelization paradigm on shared-memory systems for scientific applications. Chapter 7 deals with some typical performance problems connected with OpenMP and shows how to avoid or ameliorate them. Since cache-coherent nonuniform memory access (ccNUMA) systems have proliferated the commodity HPC market (a fact that is still widely ignored even by some HPC "professionals"), we dedicate Chapter 8 to ccNUMA-specific optimization techniques. Chapters 9 and 10 are concerned with distributed-memory parallel programming with the Message Passing Interface (MPI), and writing efficient MPI code. Finally, Chapter 11 gives an introduction to hybrid programming with MPI and OpenMP combined. Every chapter closes with a set of problems, which we highly recommend to all readers. The problems frequently cover "odds and ends" that somehow did not fit somewhere else, or elaborate on special topics. Solutions are provided in Appendix B.

We certainly recommend reading the book cover to cover, because there is not a single topic that we consider "less important." However, readers who are interested in OpenMP and MPI alone can easily start off with Chapters 6 and 9 for the basic information, and then dive into the corresponding optimization chapters (7, 8, and 10). The text is heavily cross-referenced, so it should be easy to collect the missing bits and pieces from other parts of the book.

## Acknowledgments

This book originated from a two-chapter contribution to a Springer "Lecture Notes in Physics" volume, which comprised the proceedings of a 2006 summer school on computational many-particle physics [A79]. We thank the organizers of this workshop, notably Holger Fehske, Ralf Schneider, and Alexander Weisse, for making us put down our HPC experience for the first time in coherent form. Although we extended the material considerably, we would most probably never have written a book without this initial seed.

Over a decade of working with users, students, algorithms, codes, and tools went into these pages. Many people have thus contributed, directly or indirectly, and sometimes unknowingly. In particular we have to thank the staff of HPC Services at Erlangen Regional Computing Center (RRZE), especially Thomas Zeiser, Jan Treibig, Michael Meier, Markus Wittmann, Johannes Habich, Gerald Schubert, and Holger Stengel, for countless lively discussions leading to invaluable insights. Over the last decade the group has continuously received financial support by the "Competence Network for Scientific High Performance Computing in Bavaria" (KONWIHR) and the Friedrich-Alexander University of Erlangen-Nuremberg. Both bodies shared our vision of HPC as an indispensable tool for many scientists and engineers.

We are also indebted to Uwe Küster (HLRS Stuttgart), Matthias Müller (ZIH Dresden), Reinhold Bader, and Matthias Brehm (both LRZ München), for a highly efficient cooperation between our centers, which enabled many activities and collaborations. Special thanks goes to Darren Kerbyson (PNNL) for his encouragement and many astute comments on our work. Last, but not least, we want to thank Rolf Rabenseifner (HLRS) and Gabriele Jost (TACC) for their collaboration on the topic of hybrid programming. Our Chapter 11 was inspired by this work.

Several companies, through their first-class technical support and willingness to cooperate even on a nonprofit basis, deserve our gratitude: Intel (represented by Andrey Semin and Herbert Cornelius), SGI (Reiner Vogelsang and Rüdiger Wolff), NEC (Thomas Schönemeyer), Sun Microsystems (Rick Hetherington, Ram Kunda, and Constantin Gonzalez), IBM (Klaus Gottschalk), and Cray (Wilfried Oed).

We would furthermore like to acknowledge the competent support of the CRC staff in the production of the book and the promotional material, notably by Ari Silver, Karen Simon, Katy Smith, and Kevin Craig. Finally, this book would not have been possible without the encouragement we received from Horst Simon (LBNL/NERSC) and Randi Cohen (Taylor & Francis), who convinced us to embark on the project in the first place.

*Georg Hager & Gerhard Wellein*

Erlangen Regional Computing Center
University of Erlangen-Nuremberg
Germany

# *About the authors*

*Georg Hager* is a theoretical physicist and holds a PhD in computational physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. His daily work encompasses all aspects of user support in high performance computing such as lectures, tutorials, training, code parallelization, profiling and optimization, and the assessment of novel computer architectures and tools.

*Gerhard Wellein* holds a PhD in solid state physics from the University of Bayreuth and is a professor at the Department for Computer Science at the University of Erlangen. He leads the HPC group at Erlangen Regional Computing Center (RRZE) and has more than ten years of experience in teaching HPC techniques to students and scientists from computational science and engineering programs. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.

# *List of acronyms and abbreviations*

| | |
|---|---|
| ASCII | American standard code for information interchange |
| ASIC | Application-specific integrated circuit |
| BIOS | Basic input/output system |
| BLAS | Basic linear algebra subroutines |
| CAF | Co-array Fortran |
| ccNUMA | Cache-coherent nonuniform memory access |
| CFD | Computational fluid dynamics |
| CISC | Complex instruction set computer |
| CL | Cache line |
| CPI | Cycles per instruction |
| CPU | Central processing unit |
| CRS | Compressed row storage |
| DDR | Double data rate |
| DMA | Direct memory access |
| DP | Double precision |
| DRAM | Dynamic random access memory |
| ED | Exact diagonalization |
| EPIC | Explicitly parallel instruction computing |
| Flop | Floating-point operation |
| FMA | Fused multiply-add |
| FP | Floating point |
| FPGA | Field-programmable gate array |
| FS | File system |
| FSB | Frontside bus |
| GCC | GNU compiler collection |
| GE | Gigabit Ethernet |
| GigE | Gigabit Ethernet |
| GNU | GNU is not UNIX |
| GPU | Graphics processing unit |
| GUI | Graphical user interface |

| | |
|---|---|
| HPC | High performance computing |
| HPF | High performance Fortran |
| HT | HyperTransport |
| IB | InfiniBand |
| ILP | Instruction-level parallelism |
| IMB | Intel MPI benchmarks |
| I/O | Input/output |
| IP | Internet protocol |
| JDS | Jagged diagonals storage |
| L1D | Level 1 data cache |
| L1I | Level 1 instruction cache |
| L2 | Level 2 cache |
| L3 | Level 3 cache |
| LD | Locality domain |
| LD | Load |
| LIKWID | Like I knew what I’m doing |
| LRU | Least recently used |
| LUP | Lattice site update |
| MC | Monte Carlo |
| MESI | Modified/Exclusive/Shared/Invalid |
| MI | Memory interface |
| MIMD | Multiple instruction multiple data |
| MIPS | Million instructions per second |
| MMM | Matrix–matrix multiplication |
| MPI | Message passing interface |
| MPMD | Multiple program multiple data |
| MPP | Massively parallel processing |
| MVM | Matrix–vector multiplication |
| NORMA | No remote memory access |
| NRU | Not recently used |
| NUMA | Nonuniform memory access |
| OLC | Outer-level cache |
| OS | Operating system |
| PAPI | Performance application programming interface |
| PC | Personal computer |
| PCI | Peripheral component interconnect |
| PDE | Partial differential equation |
| PGAS | Partitioned global address space |

| | |
|---|---|
| PLPA | Portable Linux processor affinity |
| POSIX | Portable operating system interface for Unix |
| PPP | Pipeline parallel processing |
| PVM | Parallel virtual machine |
| QDR | Quad data rate |
| QPI | QuickPath interconnect |
| RAM | Random access memory |
| RISC | Reduced instruction set computer |
| RHS | Right hand side |
| RFO | Read for ownership |
| SDR | Single data rate |
| SIMD | Single instruction multiple data |
| SISD | Single instruction single data |
| SMP | Symmetric multiprocessing |
| SMT | Simultaneous multithreading |
| SP | Single precision |
| SPMD | Single program multiple data |
| SSE | Streaming SIMD extensions |
| ST | Store |
| STL | Standard template library |
| SYSV | Unix System V |
| TBB | Threading building blocks |
| TCP | Transmission control protocol |
| TLB | Translation lookaside buffer |
| UMA | Uniform memory access |
| UPC | Unified parallel C |

# Chapter 1

# Modern processors

In the “old days” of scientific supercomputing roughly between 1975 and 1995, leading-edge high performance systems were specially designed for the HPC market by companies like Cray, CDC, NEC, Fujitsu, or Thinking Machines. Those systems were way ahead of standard “commodity” computers in terms of performance and price. Single-chip general-purpose microprocessors, which had been invented in the early 1970s, were only mature enough to hit the HPC market by the end of the 1980s, and it was not until the end of the 1990s that clusters of standard workstation or even PC-based hardware had become competitive at least in terms of theoretical peak performance. Today the situation has changed considerably. The HPC world is dominated by cost-effective, off-the-shelf systems with processors that were not primarily designed for scientific computing. A few traditional supercomputer vendors act in a niche market. They offer systems that are designed for high application performance on the single CPU level as well as for highly parallel workloads. Consequently, the scientist and engineer is likely to encounter such “commodity clusters” first and only advance to more specialized hardware as requirements grow. For this reason, this chapter will mostly focus on systems based on standard cache-based microprocessors. *Vector computers* support a different programming paradigm that is in many respects closer to the requirements of scientific computation, but they have become rare. However, since a discussion of supercomputer architecture would not be complete without them, a general overview will be provided in Section 1.6.

## 1.1 Stored-program computer architecture

When we talk about computer systems at large, we always have a certain architectural concept in mind. This concept was conceived by Turing in 1936, and first implemented in a real machine (EDVAC) in 1949 by Eckert and Mauchly [H129, H131]. Figure 1.1 shows a block diagram for the *stored-program digital computer*. Its defining property, which set it apart from earlier designs, is that its instructions are numbers that are stored as data in memory. Instructions are read and executed by a control unit; a separate arithmetic/logic unit is responsible for the actual computations and manipulates data stored in memory along with the instructions. I/O facilities enable communication with users. Control and arithmetic units together with the appropriate interfaces to memory and I/O are called the *Central Processing Unit* (CPU). Programming a stored-program computer amounts to modifying instructions in memory,

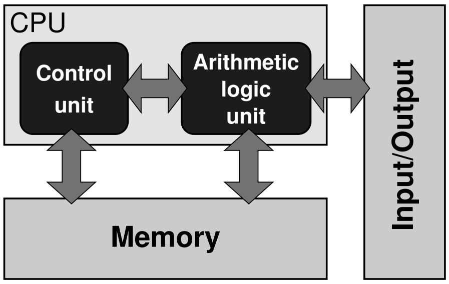


**Figure 1.1:** Stored-program computer architectural concept. The "program," which feeds the control unit, is stored in memory together with any data the arithmetic unit requires.

which can in principle be done by another program; a *compiler* is a typical example, because it translates the constructs of a high-level language like C or Fortran into instructions that can be stored in memory and then executed by a computer.

This blueprint is the basis for all mainstream computer systems today, and its inherent problems still prevail:

- Instructions and data must be continuously fed to the control and arithmetic units, so that the speed of the memory interface poses a limitation on compute performance. This is often called the *von Neumann bottleneck*. In the following sections and chapters we will show how architectural optimizations and programming techniques may mitigate the adverse effects of this constriction, but it should be clear that it remains a most severe limiting factor.

- The architecture is inherently sequential, processing a single instruction with (possibly) a single operand or a group of operands from memory. The term *SISD* (Single Instruction Single Data) has been coined for this concept. How it can be modified and extended to support parallelism in many different flavors and how such a parallel machine can be efficiently used is also one of the main topics of this book.

Despite these drawbacks, no other architectural concept has found similarly widespread use in nearly 70 years of electronic digital computing.

## 1.2 General-purpose cache-based microprocessor architecture

Microprocessors are probably the most complicated machinery that man has ever created; however, they all implement the stored-program digital computer concept as described in the previous section. Understanding all inner workings of a CPU is out of the question for the scientist, and also not required. It is helpful, though, to get a grasp of the high-level features in order to understand potential bottlenecks. Figure 1.2 shows a very simplified block diagram of a modern cache-based general-purpose microprocessor. The components that actually do "work" for a running application are the arithmetic units for floating-point (FP) and integer (INT) operations

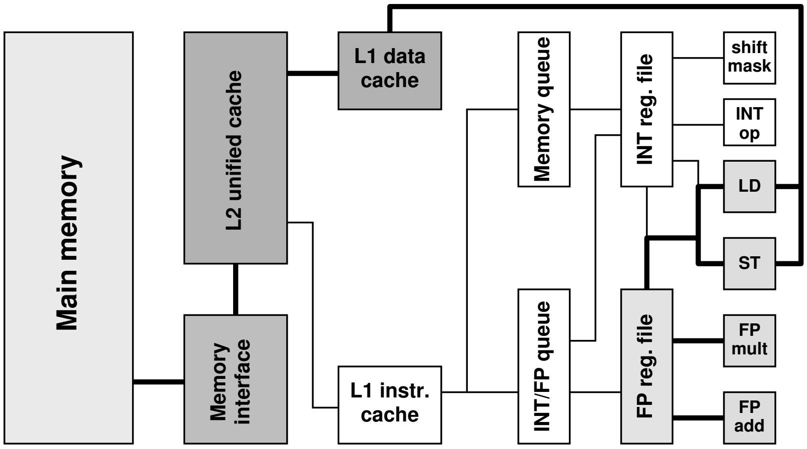


**Figure 1.2:** Simplified block diagram of a typical cache-based microprocessor (one core). Other cores on the same chip or package (socket) can share resources like caches or the memory interface. The functional blocks and data paths most relevant to performance issues in scientific computing are highlighted.

and make up for only a very small fraction of the chip area. The rest consists of administrative logic that helps to feed those units with operands. CPU *registers*, which are generally divided into floating-point and integer (or "general purpose") varieties, can hold operands to be accessed by instructions with no significant delay; in some architectures, *all* operands for arithmetic operations must reside in registers. Typical CPUs nowadays have between 16 and 128 user-visible registers of both kinds. Load (LD) and store (ST) units handle instructions that transfer data to and from registers. Instructions are sorted into several *queues*, waiting to be executed, probably not in the order they were issued (see below). Finally, *caches* hold data and instructions to be (re-)used soon. The major part of the chip area is usually occupied by caches.

A lot of additional logic, i.e., branch prediction, reorder buffers, data shortcuts, transaction queues, etc., that we cannot touch upon here is built into modern processors. Vendors provide extensive documentation about those details [V104, V105, V106]. During the last decade, *multicore* processors have superseded the traditional single-core designs. In a multicore chip, several processors ("cores") execute code concurrently. They can share resources like memory interfaces or caches to varying degrees; see Section 1.4 for details.

### 1.2.1 Performance metrics and benchmarks

All the components of a CPU core can operate at some maximum speed called *peak performance*. Whether this limit can be reached with a specific application code depends on many factors and is one of the key topics of Chapter 3. Here we introduce some basic performance metrics that can quantify the "speed" of a CPU. Scientific computing tends to be quite centric to floating-point data, usually with "double preci-

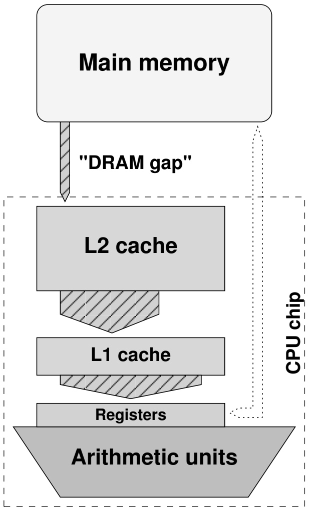
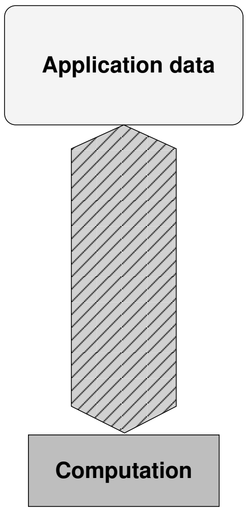


**Figure 1.3:** (Left) Simplified data-centric memory hierarchy in a cache-based microprocessor (direct access paths from registers to memory are not available on all architectures). There is usually a separate L1 cache for instructions. (Right) The "DRAM gap" denotes the large discrepancy between main memory and cache bandwidths. This model must be mapped to the data access requirements of an application.



sion" (DP). The performance at which the FP units generate results for multiply and add operations is measured in *floating-point operations per second* (Flops/sec). The reason why more complicated arithmetic (divide, square root, trigonometric functions) is not counted here is that those operations often share execution resources with multiply and add units, and are executed so slowly as to not contribute significantly to overall performance in practice (see also Chapter 2). High performance software should thus try to avoid such operations as far as possible. At the time of writing, standard commodity microprocessors are designed to deliver at most two or four double-precision floating-point results per clock cycle. With typical clock frequencies between 2 and 3 GHz, this leads to a peak arithmetic performance between 4 and 12 GFlops/sec per core.

As mentioned above, feeding arithmetic units with operands is a complicated task. The most important data paths from the programmer's point of view are those to and from the caches and main memory. The performance, or *bandwidth* of those paths is quantified in GBytes/sec. The GFlops/sec and GBytes/sec metrics usually suffice for explaining most relevant performance features of microprocessors.[1] Hence, as shown in Figure 1.3, the performance-aware programmer's view of a cache-based microprocessor is very data-centric. A "computation" or algorithm of some kind is usually defined by manipulation of data items; a concrete implementation of the algorithm must, however, run on real hardware, with limited performance on all data paths, especially those to main memory.

Fathoming the chief performance characteristics of a processor or system is one of the purposes of *low-level benchmarking*. A low-level benchmark is a program that tries to test some specific feature of the architecture like, e.g., peak performance or

[1]Please note that the "giga-" and "mega-" prefixes refer to a factor of $10^9$ and $10^6$, respectively, when used in conjunction with ratios like bandwidth or arithmetic performance. Since recently, the prefixes "mebi-," "gibi-," etc., are frequently used to express quantities in powers of two, i.e., 1 MiB=$2^{20}$ bytes.

**Listing 1.1:** Basic code fragment for the vector triad benchmark, including performance measurement.

```
 1 double precision, dimension(N) :: A,B,C,D
 2 double precision :: S,E,MFLOPS
 3
 4 do i=1,N                          !initialize arrays
 5   A(i) = 0.d0; B(i) = 1.d0
 6   C(i) = 2.d0; D(i) = 3.d0
 7 enddo
 8
 9 call get_walltime(S)              ! get time stamp
10 do j=1,R
11   do i=1,N
12     A(i) = B(i) + C(i) * D(i)     ! 3 loads, 1 store
13   enddo
14   if(A(2).lt.0) call dummy(A,B,C,D) ! prevent loop interchange
15 enddo
16 call get_walltime(E)              ! get time stamp
17 MFLOPS = R*N*2.d0/((E-S)*1.d6)    ! compute MFlop/sec rate
```

memory bandwidth. One of the prominent examples is the *vector triad*, introduced by Schönauer [S5]. It comprises a nested loop, the inner level executing a multiply-add operation on the elements of three vectors and storing the result in a fourth (see lines 10–15 in Listing 1.1). The purpose of this benchmark is to measure the performance of data transfers between memory and arithmetic units of a processor. On the inner level, three *load streams* for arrays `B`, `C` and `D` and one *store stream* for `A` are active. Depending on `N`, this loop might execute in a very small time, which would be hard to measure. The outer loop thus repeats the triad `R` times so that execution time becomes large enough to be accurately measurable. In practice one would choose `R` according to `N` so that the overall execution time stays roughly constant for different `N`.

The aim of the masked-out call to the `dummy()` subroutine is to prevent the compiler from doing an obvious optimization: Without the call, the compiler might discover that the inner loop does not depend at all on the outer loop index `j` and drop the outer loop right away. The possible call to `dummy()` fools the compiler into believing that the arrays may change between outer loop iterations. This effectively prevents the optimization described, and the additional cost is negligible because the condition is always false (which the compiler does not know).

The `MFLOPS` variable is computed to be the MFlops/sec rate for the whole loop nest. Please note that the most sensible time measure in benchmarking is *wallclock time*, also called *elapsed time*. Any other "time" that the system may provide, first and foremost the much stressed CPU time, is prone to misinterpretation because there might be contributions from I/O, context switches, other processes, etc., which CPU time cannot encompass. This is even more true for parallel programs (see Chapter 5). A useful C routine to get a wallclock time stamp like the one used in the triad bench-

**Listing 1.2:** A C routine for measuring wallclock time, based on the `gettimeofday()` POSIX function. Under the Windows OS, the `GetSystemTimeAsFileTime()` routine can be used in a similar way.

```
#include <sys/time.h>

void get_walltime_(double* wcTime) {
  struct timeval tp;
  gettimeofday(&tp, NULL);
  *wcTime = (double)(tp.tv_sec + tp.tv_usec/1000000.0);
}

void get_walltime(double* wcTime) {
  get_walltime_(wcTime);
}
```

mark above could look like in Listing 1.2. The reason for providing the function with and without a trailing underscore is that Fortran compilers usually append an underscore to subroutine names. With both versions available, linking the compiled C code to a main program in Fortran or C will always work.

Figure 1.4 shows performance graphs for the vector triad obtained on different generations of cache-based microprocessors and a vector system. For very small loop lengths we see poor performance no matter which type of CPU or architecture is used. On standard microprocessors, performance grows with `N` until some maximum is reached, followed by several sudden breakdowns. Finally, performance stays constant for very large loops. Those characteristics will be analyzed in detail in Section 1.3.

Vector processors (dotted line in Figure 1.4) show very contrasting features. The low-performance region extends much farther than on cache-based microprocessors,

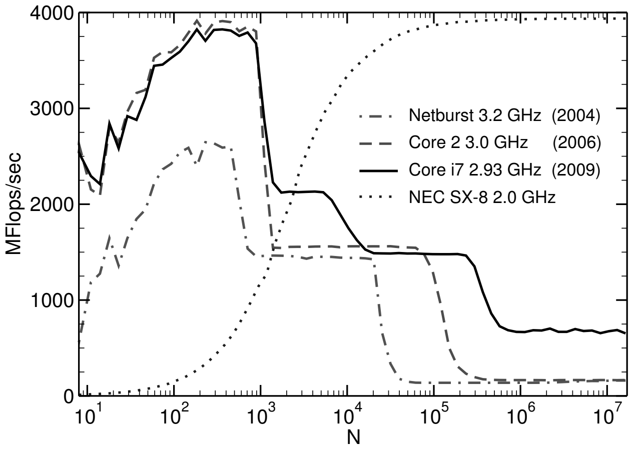


**Figure 1.4:** Serial vector triad performance versus loop length for several generations of Intel processor architectures (clock speed and year of introduction is indicated), and the NEC SX-8 vector processor. Note the entirely different performance characteristics of the latter.

but there are no breakdowns at all. We conclude that vector systems are somewhat complementary to standard CPUs in that they meet different domains of applicability (see Section 1.6 for details on vector architectures). It may, however, be possible to optimize real-world code in a way that circumvents low-performance regions. See Chapters 2 and 3 for details.

Low-level benchmarks are powerful tools to get information about the basic capabilities of a processor. However, they often cannot accurately predict the behavior of "real" application code. In order to decide whether some CPU or architecture is well-suited for some application (e.g., in the run-up to a procurement or before writing a proposal for a computer time grant), the only safe way is to prepare *application benchmarks*. This means that an application code is used with input parameters that reflect as closely as possible the real requirements of production runs. The decision for or against a certain architecture should always be heavily based on application benchmarking. Standard benchmark collections like the SPEC suite [W118] can only be rough guidelines.

### 1.2.2 Transistors galore: Moore's Law

Computer technology had been used for scientific purposes and, more specifically, for numerically demanding calculations long before the dawn of the desktop PC. For more than thirty years scientists could rely on the fact that no matter which technology was implemented to build computer chips, their "complexity" or general "capability" doubled about every 24 months. This trend is commonly ascribed to *Moore's Law*. Gordon Moore, co-founder of Intel Corp., postulated in 1965 that the number of components (transistors) on a chip that are required to hit the "sweet spot" of minimal manufacturing cost per component would continue to increase at the indicated rate [R35]. This has held true since the early 1960s despite substantial changes in manufacturing technologies that have happened over the decades. Amazingly, the growth in complexity has always roughly translated to an equivalent growth in compute performance, although the meaning of "performance" remains debatable as a processor is not the only component in a computer (see below for more discussion regarding this point).

Increasing chip transistor counts and clock speeds have enabled processor designers to implement many advanced techniques that lead to improved application performance. A multitude of concepts have been developed, including the following:

1. *Pipelined functional units*. Of all innovations that have entered computer design, pipelining is perhaps the most important one. By subdividing complex operations (like, e.g., floating point addition and multiplication) into simple components that can be executed using different functional units on the CPU, it is possible to increase instruction throughput, i.e., the number of instructions executed per clock cycle. This is the most elementary example of *instruction-level parallelism* (ILP). Optimally pipelined execution leads to a throughput of one instruction per cycle. At the time of writing, processor designs exist that feature pipelines with more than 30 stages. See the next section on page 9 for details.

2. *Superscalar architecture*. Superscalarity provides "direct" instruction-level parallelism by enabling an instruction throughput of more than one per cycle. This requires multiple, possibly identical functional units, which can operate currently (see Section 1.2.4 for details). Modern microprocessors are up to six-way superscalar.

3. *Data parallelism through SIMD instructions*. SIMD (*Single Instruction Multiple Data*) instructions issue identical operations on a whole array of integer or FP operands, usually in special registers. They improve arithmetic peak performance without the requirement for increased superscalarity. Examples are Intel's "SSE" and its successors, AMD's "3dNow!," the "AltiVec" extensions in Power and PowerPC processors, and the "VIS" instruction set in Sun's UltraSPARC designs. See Section 1.2.5 for details.

4. *Out-of-order execution*. If arguments to instructions are not available in registers "on time," e.g., because the memory subsystem is too slow to keep up with processor speed, out-of-order execution can avoid idle times (also called *stalls*) by executing instructions that appear later in the instruction stream but have their parameters available. This improves instruction throughput and makes it easier for compilers to arrange machine code for optimal performance. Current out-of-order designs can keep hundreds of instructions in flight at any time, using a *reorder buffer* that stores instructions until they become eligible for execution.

5. *Larger caches*. Small, fast, on-chip memories serve as temporary data storage for holding copies of data that is to be used again "soon," or that is close to data that has recently been used. This is essential due to the increasing gap between processor and memory speeds (see Section 1.3). Enlarging the cache size does usually not hurt application performance, but there is some tradeoff because a big cache tends to be slower than a small one.

6. *Simplified instruction set*. In the 1980s, a general move from the *CISC* to the *RISC* paradigm took place. In a CISC (Complex Instruction Set Computer), a processor executes very complex, powerful instructions, requiring a large hardware effort for decoding but keeping programs small and compact. This lightened the burden on programmers, and saved memory, which was a scarce resource for a long time. A RISC (Reduced Instruction Set Computer) features a very simple instruction set that can be executed very rapidly (few clock cycles per instruction; in the extreme case each instruction takes only a single cycle). With RISC, the clock rate of microprocessors could be increased in a way that would never have been possible with CISC. Additionally, it frees up transistors for other uses. Nowadays, most computer architectures significant for scientific computing use RISC at the low level. Although x86-based processors execute CISC machine code, they perform an internal on-the-fly translation into RISC "$\mu$-ops."

In spite of all innovations, processor vendors have recently been facing high obstacles in pushing the performance limits of monolithic, single-core CPUs to new levels.