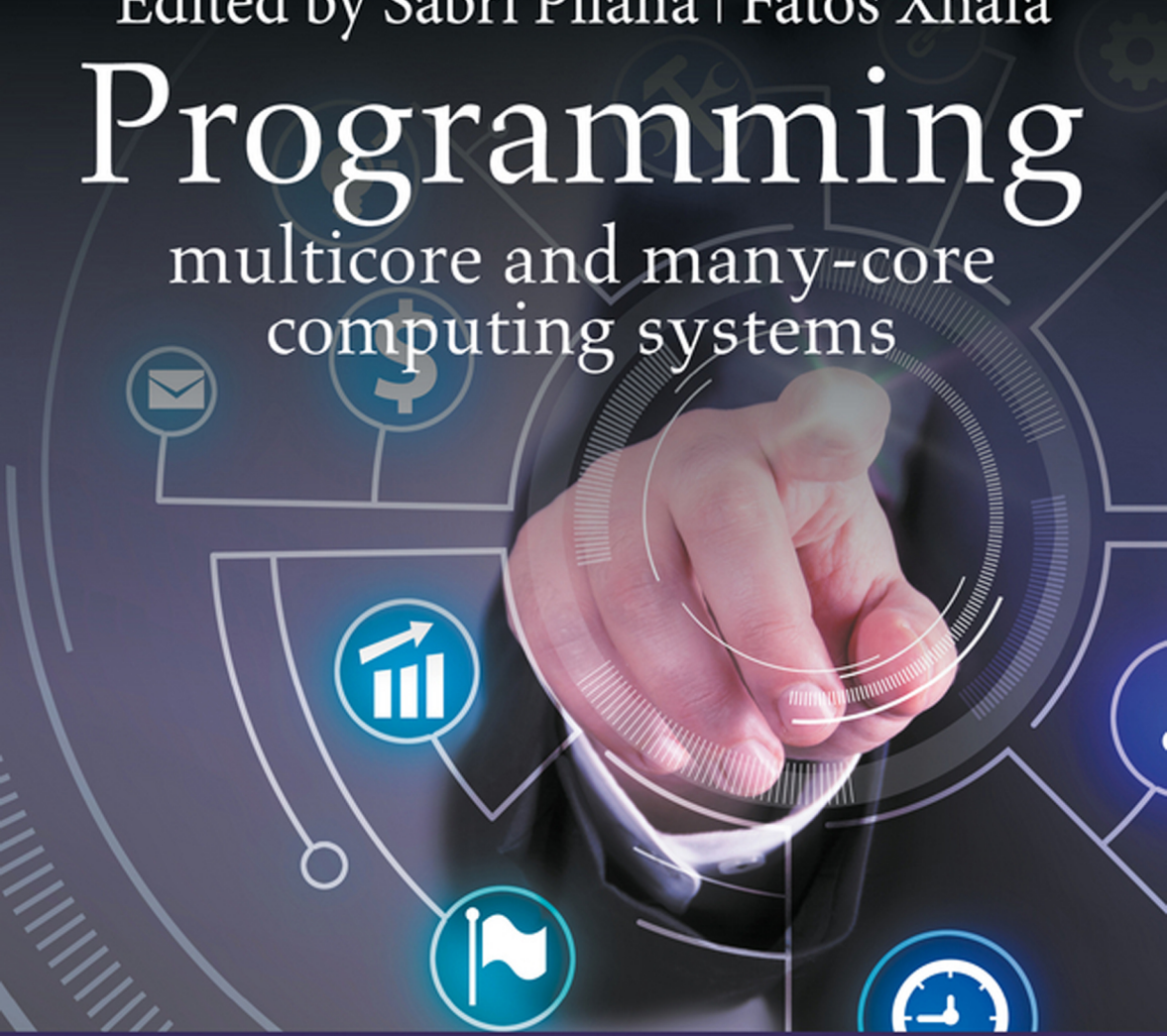
Wiley Series on Parallel and Distributed Computing

Albert Y. Zomaya, Series Editor

Edited by Sabri Pllana | Fatos Xhafa

Programming

multicore and many-core computing systems



WILEY

PROGRAMMING MULTICORE AND MANY-CORE COMPUTING SYSTEMS

Wiley Series on Parallel and Distributed Computing

Series Editor: Albert Y. Zomaya

A complete list of the titles in this series appears at the end of this volume.

PROGRAMMING MULTICORE AND MANY-CORE COMPUTING SYSTEMS

Edited by

Sabri Pllana

Fatos Xhafa

WILEY

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada

Library of Congress Cataloguing-in-Publication Data

Names: Pllana, Sabri, editor. | Xhafa, Fatos, editor.
Title: Programming multicore and many-core computing systems / edited by
Sabri Pllana, Fatos Xhafa.
Description: First edition. | Hoboken, New Jersey : John Wiley & Sons, [2017]
| Includes bibliographical references and index.
Identifiers: LCCN 2016038244| ISBN 9780470936900 (cloth) | ISBN 9781119332008
(epub) | ISBN 9781119331995 (Adobe PDF)
Subjects: LCSH: Parallel programming (Computer science) |
Coprocessors–Programming.
Classification: LCC QA76.642 .P767 2017 | DDC 005.2/75–dc23 LC record available at
https://lccn.loc.gov/2016038244

Cover Image: Creative-idea/Gettyimages
Cover design by Wiley

Set in 10/12pt, TimesLTStd by SPi Global, Chennai, India.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

CONTENTS

LIST OF CONTRIBUTORS

Marco Aldinucci, Computer Science Department, University of Torino, Corso Svizzera 185, 10149 Torino, Italy. [email: aldinuc@di.unito.it]

Jørn Amundsen, Norwegian University of Science and Technology, SemSælandsvei 7-9, NO-7491 Trondheim, Norway. [email: jorn.amundsen@ntnu.no]

Eduard Ayguade, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: eduard.ayguade@bsc.es]

Rosa M. Badia, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: rosa.m.badia@bsc.es]

Henri E. Bal, Department of Computer Science, Vrije Universiteit, De Boelelaan 1081A,1081 HV, Amsterdam, The Netherlands. [email: bal@cs.vu.nl]

Denis Barthou, INRIA Bordeaux Sud-Ouest, 200 avenue de la Vieille Tour 33405 Talence Cedex, France. [email: denis.barthou@labri.fr]

Pieter Bellens, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: pieter.bellens@bsc.es]

Siegfried Benkner, Faculty of Computer Science, University of Vienna, Wahringerstrasse29, A-1090 Vienna, Austria. [email: sigi@par.univie.ac.at]

Daniel Rubio Bonilla, Department for Intelligent Service Infrastructures, HLRS, University of Stuttgart, Nobelstr. 19, 70569 Stuttgart, Germany. [email: rubio@hlrs.de]

Javier Bueno, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: javier.bueno@bsc.es]

Ali R. Butt, Virginia Tech, 2202 Kraft Drive (0106), Blacksburg, VA 24061, USA. [email: butta@cs.vt.edu]

Daniel Cederman, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden. [email: cederman @chalmers.se]

Pierre Collet, Strasbourg University, Pole API BdSébastien Brant BP 10413 67412 Illkirch CEDEX France. [email: pierre.collet@unistra.fr]

Herbert Cornelius, Intel Gmbh, DornacherStrasse 1, D-85622 Feldkirchen, Germany. [email: herbert.cornelius@intel.com]

Tommaso Cucinotta, Retis Lab, ScuolaSuperioreSant'Anna CEIICP, Att.NeFrancesca Gattai, Via Moruzzi 1, 56124 Pisa, Italy. [email: tommaso .cucinotta@sssup.it]

Marco Danelutto, Computer Science Department, University of Pisa, Largo Pontecorvo3, 56127 Pisa, Italy. [email: marcod@di.unipi.it]

Usman Dastgeer, IDA, Linköping University, S-58183 Linköping, Sweden. [email: usman.dastgeer@liu.se]

Pablo De Oliveira Castro, CEA, LIST, Université De Versailles, 45, Avenue DesÉtats Unis, Versailles, 78035 France. [email: pablo.oliveira@exascale-computing.eu]

Alejandro Duran, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: alex.duran@bsc.es]

Mujahed Eleyat, University of Science and Technology (NTU), SemSælandsvei 7-9, NO-7491 Trondheim, Norway. [email: mujahed.eleyat@miriam.as]

Johan Enmyren, IDA, Linköping University, S-58183 Linköping, Sweden. [email: x10johen@ida.liu.se]

Yoav Etsion, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: yoav.etsion@bsc.es]

Eitan Farchi, IBM, 49 Hashmonaim Street, Pardes Hanna 37052, Israel. [email: farchi@il.ibm.com]

Montse Farreras, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: mfarrera@ac.upc.edu]

Min Feng, Department of Computer Science, The University of California At Riverside, Engineering Bldg. Unit 2, Rm. 463, Riverside, CA 92521, USA. [email: mfeng@cs.ucr.edu]

Roger Ferrer, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: roger.ferrer@bsc.es]

Anders Gidenstam, School of Business and Informatics, Högskolan I Borås, S-50190 Borås, Sweden. [email: anders.gidenstam@hb.se]

Sergei Gorlatch, FB10, Universität Münster, D-48149 Münster, Germany. [email: gorlatch@uni-muenster.de]

Vincent Gramoli, LPD, EPFL IC, Station 14, CH-1015 Lausanne, Switzerland. [email: vincent.gramoli@epfl.ch]

Rachid Guerraoui, LPD, EPFL IC, Station 14, CH-1015 Lausanne, Switzerland. [email: rachid.guerraoui@epfl.ch]

Neil Gunther, Performance Dynamics Company, 4061 East Castro Valley Blvd. Suite 110, Castro Valley, CA 95954, USA. [email: njgunther@perfdynamics.com]

Rajiv Gupta, Department of Computer Science, The University of California at Riverside, Engineering Bldg. Unit 2, Rm. 408, Riverside, CA 92521, USA. [email: gupta@cs.ucr.edu]

Phuong Ha, Department of Computer Science, Faculty of Science, University of Tromsø, NO-9037 Tromsø, Norway. [email: phuong@cs.uit.no]

Pieter Hijma, Department of Computer Science, Vrije Universiteit, De Boelelaan1081A, 1081 HV, Amsterdam, The Netherlands. [email: pieter@cs.vu.nl]

Alexandru Iordan, University of Science and Technology (NTU), SemSælandsvei 7-9, NO-7491 Trondheim, Norway. [email: iordan@idi.ntnu.no]

Magnus Jahre, University of Science and Technology (NTU), SemSælandsvei 7-9, NO-7491 Trondheim, Norway. [email: jahre@idi.ntnu.no]

Mahmut Kandemir, The Pennsylvania State University, 111 IST Building, University Park, PA 16802, USA. [email: kandemir@cse.psu.edu]

Philipp Kegel, FB10, Universität Münster, D-48149 Münster, Germany. [email:philipp.kegel@uni-muenster.de]

Christoph Kessler, IDA, Linköping University, S-58183 Linköping, Sweden. [email: christoph.kessler@liu.se]

Peter Kilpatrick, School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast BT7 1NN, UK. [email: p.kilpatrick@qub.ac.uk]

Jesus Labarta, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: jesus.labarta@bsc.es]

Nicolas Lachiche, Strasbourg University, Pole API BdSébastien Brant BP 10413 67412 Illkirch CEDEX France. [email: nicolas.lachiche@unistra.fr]

Giuseppe Lipari, Real-Time Systems Laboratory, ScuolaSuperioreSant'Anna, Pisa, Italy. [email: g.lipari@sssup.it]

Stéphane Louise, CEA, LIST, Gif-Sur-Yvette, 91191 France. [email: stephane.louise@cea.fr]

Emilio Luque, Computer Architecture and Operating System Department, Universitat Autonoma De Barcelona, 08193, Barcelona, Spain. [email: emilio.luque@uab.es]

Ogier Maitre, Strasbourg University, Pole API BdSébastien Brant BP 10413 67412 Illkirch CEDEX France. [email: ogier.maitre@unistra.fr]

Sandya Mannarswamy, Xerox Research Centre India, 225 1st C Cross, 2nd Main, Kasturi Nagar, Bangalore, India. 560043. [email: sandya@hp.com]

Vladimir Marjanovic, Barcelona Supercomputing Center, Nexus-2 Building, Jordi Girona 29, 08034 Barcelona, Spain. [email: vladimir.marjanovic@bsc.es]

Lluis Martinell, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: luis.martinell@bsc.es]

Xavier Martorell, Barcelona Supercomputing Center, Nexus-2 Building, Jordi Girona 29, 08034 Barcelona, Spain. [email: xavim@ac.upc.edu]

David Moloney, Movidius Ltd., Mountjoy Square East 19, D1 Dublin, Ireland. [email: david.moloney@movidius.com]

Ronal Muresano, Computer Architecture and Operating System Department, Universitat Autonoma De Barcelona, 08193, Barcelona, Spain. [email: rmuresano@caos.uab.es]

M. Mustafa Rafique, Virginia Tech, 2202 Kraft Drive (0106), Blacksburg, Virginia 24061, USA. [email: mustafa@cs.vt.edu]

Raymond Namyst, Institut National De Recherche En Informatique Et En Automatique (INRIA), Bordeaux Sud-Ouest, Cours De La Liberation 351, F-33405 Talence Cedex, France. [email: raymond.namyst@labri.fr]

Lasse Natvig, University of Science and Technology (NTU), Semsælandsvei 7-9, NO-7491 Trondheim, Norway. [email: lasse@idi.ntnu.no]

Dimitrios S. Nikolopoulos, FORTH-ICS, N. Plastira 100, VassilikaVouton, Heraklion, Crete, Greece. [email: dsn@ics.forth.gr]

Frank Otto, Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany. [email: frank.otto@kit.edu]

Ozcan Ozturk, Department of Computer Engineering, Engineering Building, EA 407B, Bilkent University, Bilkent, 06800, Ankara, Turkey. [email: ozturk@cs.bilkent.edu.tr]

Marina Papatriantafilou, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden. [email: ptrianta@chalmers.se]

Stefan Parvu, Nokia Group, Espoo, Karakaari 15, Finland. [email: stefan.parvu@nokia.com]

Josep M. Perez, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: josep.m.perez@bsc.es]

Judit Planas, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: judit.planas@bsc.es]

Sabri Pllana, Department of Computer Science, Linnaeus University, SE-351 95 Vaxjo, Sweden. [email: sabri.pllana@lnu.se]

Stephane Querry, Pole API BdSébastien Brant BP 10413 67412 Illkirch CEDEX France. [email: stephane.querry@unistra.fr]

Alex Ramirez, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: alex.ramirez@bsc.es]

Dolores Rexachs, Computer Architecture and Operating System Department, Universitat Autonoma De Barcelona, 08193, Barcelona, Spain. [email: dolores.rexachs@uab.es]

Andrew Richards, Codeplay Software Limited, York Place 45, EH1 3HP Edinburgh, United Kingdom. [email: andrew@codeplay.com]

António Rodrigues, TU Lisbon/IST/INESC-ID, Rua Alves Redol 9, 1000-029Lisboa, Portugal. [email: antonio.c.rodrigues@ist.utl.pt]

Nuno Roma, TU Lisbon/IST/INESC-ID, Rua Alves Redol 9, 1000-029 Lisboa, Portugal. [email: nuno.roma@inesc-id.pt]

Peter Sanders, KarlsruherInstitut Für Technologie, Amfasanengarten 5, D-76128Karlsruhe, Germany. [email: sanders@kit.edu]

Lutz Schubert, Department for Intelligent Service Infrastructures, HLRS, University of Stuttgart, Nobelstr. 19, 70569 Stuttgart, Germany. [email: schubert@hlrs.de]

Hazim Shafi, One Microsoft Way, Redmond, WA 98052, USA. [email: hshafi@microsoft.com]

Deepak Sharma, Pole API BdSébastien Brant BP 10413 67412 Illkirch CEDEX, France. [email: deepak.sharma@unistra.fr]

Leonel Sousa, TU Lisbon/IST/INESC-ID, Rua Alves Redol 9, 1000-029 Lisboa, Portugal. [email: leonel.sousa@inesc-id.pt]

Michel Steuwer, FB10, Universität Münster, D-48149Münster, Germany. [email:michel.steuwer@uni-muenster.de]

Shanti Subramanyam, Yahoo! Inc., 701 First Ave, Sunnyvale, CA 94089. [email: shantis@yahoo-inc.com]

Håkan Sundell, School of Business and Infomatics, Högskolan I Borås, S-501 90BorâĂăs, Sweden. [email: hakan.sundell@hb.se]

Xavier Teruel, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: xavier.teruel@bsc.es]

Chen Tian, Department of Computer Science, The University of California at Riverside, Engineering Bldg. Unit 2, Rm. 463, Riverside, CA 92521, USA. [email: tianc@cs.ucr.edu]

Walter F. Tichy, Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany. [email: tichy@kit.edu]

Massimo Torquati, Computer Science Department, University of Pisa, Largo Pontecorvo3, 56127 Pisa, Italy. [email: torquati@di.unipi.it]

Jesper Larsson Träff, Research Group Parallel Computing, Vienna University of Technology, Favoritenstrasse 16/184-5, A-1040 Vienna, Austria. [email: traff@par.tuwien.ac.at]

Ioanna Tsalouchidou, Barcelona Supercomputing Center, Nexus-2 Building, 3rdFloor, Jordi Girona 29, 08034 Barcelona, Spain. [email: ioanna.tsalouchidou @bsc.es]

Philippas Tsigas, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden. [email: philippas.tsigas@chalmers.se]

Philippas Tsigas, Department of Computer Science and Engineering, Chalmers Tekniska Höogskola, SE-41296 Göteborg, Sweden. [email: tsigas@chalmers.se]

Mateo Valero, Barcelona Supercomputing Center, Nexus-2 Building, 3rd Floor, Jordi Girona 29, 08034 Barcelona, Spain. [email: mateo.valero@bsc.es]

Rob V. Van Nieuwpoort, Department of Computer Science, Vrije Universiteit, De Boelelaan 1081A, 1081 HV, Amsterdam, The Netherlands. [email: rob@cs.vu.nl]

Ana Lucia Varbanescu, Department of Software Technologies, Delft University of Technology, Delft, The Netherlands Mekelweg 4, 2628 CD, Delft, The Netherlands. [email: a.l.varbanescu@tudelft.nl]

Stefan Wesner, HLRS, Department of Applications and Visualization, University of Stuttgart, Nobelstr. 19, 70569 Stuttgart, Germany. [email: wesner@hlrs.de]

PREFACE

Multicore and many-core computing systems have emerged as an important paradigm in high-performance computing (HPC) and have significantly propelled development of advanced parallel and distributed applications as well as of embedded systems. Multicore processors are now ubiquitous, indeed, from processors with 2 or 4 cores in the 2000s, the trend of increasing the number of cores keeps the pace, and processors with hundreds or even thousands of (lightweight) cores are becoming commonplace to optimize not only performance but also energy. However, this disruptive technology (also referred to as 'continuum computing paradigm') presents several major challenges such as increased effort and system-specific skills for porting and optimizing application codes, managing and exploiting massive parallelism and system heterogeneity to achieve increased performance, innovative modeling strategies for low-power simulation, etc. Among these, we would distinguish the challenge of mastering the multicore and many-core and heterogeneous systems – this is precisely the focus of this book!

The emergence of multicore processors has helped in addressing several problems that are related to single-core processors – known as memory wall, power wall and instruction-level parallelism wall – but they pose several other 'walls' such as the programmability wall or the coherency wall. Among these, programmability wall is a long-standing challenge. Indeed, on the one hand, program development for multicore processors, especially for heterogeneous multicore processors, is significantly more complex than for single-core processors. On the other hand, programmers have been

traditionally trained for the development of sequential programs, and only a small percentage of them have experience with parallel programming.

In fact, in the past only a relatively small group of programmers interested in HPC was concerned with the parallel programming issues; the situation has changed dramatically with the appearance of multicore processors in commonly used computing systems. Traditionally parallel programs in HPC community have been developed by heroic programmers using a simple text editor as programming environment, programming at a low level of abstraction and doing manual performance optimization. It is expected that with the pervasiveness of multicore processors, parallel programming will become mainstream, but it cannot be expected that a mainstream programmer will prefer to use the traditional HPC methods and tools.

The main objective of this book is to present a comprehensive view of the state-of-the-art parallel programming methods, techniques and tools to aid the programmers in mastering the efficient programming of multicore and many-core systems. The book comprises a selection of twenty-two chapter contributions by experts in the field of multicore and many-core systems that cover fundamental techniques and algorithms, programming approaches, methodologies and frameworks, task/application scheduling and management, testing and evaluation methodologies and case studies for programming multicore and many-core systems. Lessons learned, challenges and road map ahead are also given and discussed along the chapters.

The content of the book is arranged into five parts:

Part I: Foundations

The first part of the book covers fundamental issues in programming of multicore and many-core computing systems. Along four chapters the authors discuss the state of the art on multi- and many-core architectures, programming models, concurrent data structures and memory allocation, scheduling and management.

Natvig et al. in the first chapter, '*Multi- and many-cores, architectural overview for programmers*', provide a broad overview of the fundamental parallel techniques, parallel taxonomies and various 'walls' in programming multicore/many-core computing systems such as 'power wall', 'memory wall' and 'ILP (instruction level parallelism) wall'. The authors also discuss the challenges of the heterogeneity in multicore/many-core computing systems. They conclude by stressing the need for more research in parallel programming models to meet the five P's of parallel processing – performance, predictability, power efficiency, programmability and portability – when building and programming multicore and many-core computing systems.

Varbanescu et al. in the second chapter, '*Programming models for multicore and many-core computing systems*', survey a comprehensive set of programming models for most popular families of many-core systems, including both specific and classical parallel models for multicore and many-core platforms. The authors have introduced four classes of reference features for model evaluation: usability, design support, implementation support and programmability. Based on these features, a

multidimensional comparison of the surveyed models is provided aiming to identify the essential characteristics that separate or cluster these models. The authors conclude by emphasizing the influence that the choice of a programming model can have on the application design and implementation and give a few guidelines for finding a programming model that matches the application characteristics.

The third chapter by **Cederman et al**., '*Lock-free concurrent data structures*', deals with the use of concurrent data structures in parallel programming of multicore and many-core systems. Several issues such as maintaining consistency in the presence of many simultaneous updates are discussed and lock-free implementations of data structures that support concurrent access are given. Lock-free concurrent data structures are shown to support the design of lock-free algorithms that scale much better when the number of processes increases. A set of fundamental synchronization primitives is also described together with challenges in managing dynamically allocated memory in a concurrent environment.

Mannarswamy in the fourth chapter, '*Software transactional memory*', addresses the main challenges in writing concurrent code in multicore and many-core computing systems. In particular, the author focuses on the coordinating access to shared data, accessed by multiple threads concurrently. Then, the software transactional memory (STM) programming paradigm for shared memory multithreaded programs is introduced. STM is intended to facilitate the development of complex concurrent software as an alternative to conventional lock-based synchronization primitives by reducing the burden of programming complexity involved in writing concurrent code. The need for addressing performance bottlenecks and improving the application performance on STM is also discussed as a major research issue in the field.

Part II: Programming Approaches

The second part of the book is devoted to programming approaches for multicore and many-core computing systems. This part comprises seven chapters that cover a variety of programming approaches including heterogeneous programming, skeleton programming, DSL and object-oriented stream programming and programming with transactional memory.

The fifth chapter, '*Heterogeneous programming with OMPSs and its implications*', by **Ayguadé et al**. discusses on programming models for heterogeneous architectures aiming to ease the asynchrony and to increment parallelization, modularity and portability of applications. The authors present the OmpSs model, which extends the OpenMP 3.0 programming model, and show how it leverages MPI and OpenCL/CUDA, mastering the efficient programming of the clustered heterogeneous multi-/many-core systems. The implementation of OmpSs as well as a discussion on the intelligence needed to be embedded in the runtime system to effectively lower the programmability wall and the opportunities to implement new mechanisms and policies is also discussed and some overheads related with task management in OmpSs are pointed out for further investigation.

Kessler et al. in the sixth chapter, '*Skeleton programming for portable many-core computing*', consider skeleton programming ('data parallel skeletons') as a model to

solve the portability problem that arises in multi-and many-core programming and to increase the level of abstraction in such programming environment. After overviewing the concept of algorithmic skeletons, the authors give a detailed description of two recent approaches for programming emerging heterogeneous many-core systems, namely, SkePU and SkelCL. Some other skeleton programming frameworks, which share ideas with SkePU and SkelCL but address a more narrow range of architectures or are used in industrial application development, are also discussed. Adding support for portable task parallelism, such as farm skeletons, is pointed out as an important research issue for future research.

In the seventh chapter, '*DSL stream programming on multicore architectures*', by **de Oliveira Castro et al**., the authors present a novel approach for stream programming, considered a powerful alternative to program multi-core processors by offering a deterministic execution based on a sound mathematical formalism and the ability to implicitly express the parallelism by the stream structure, which leverages compiler optimizations that can harness the multicore performance without having to tune the application by hand. Two families of stream programming languages are analyzed, namely, languages in which the data access patterns are explicitly described by the programmer through a set of reorganization primitives and those in which the data access patterns are implicitly declared through a set of dependencies between tasks. Then, the authors expose the principle of a two-level approach combining the advantages and expressivity of both types of languages aiming to achieve both the expressivity of high-level languages such as Array-OL and Block Parallel and the rich optimization framework, similar to StreamIT and Brook.

The eighth chapter, '*Programming with transactional memory*', by **Gramoli and Guerraoui** addresses similar issues as in Chapter 4, namely, the use of transactional memory to remedy numerous concurrency problems arising in multicore and many-core programming. The chapter analyzes the state-of-the-art concurrent programming advances based on transactional memory. Several programming languages that support TM are considered along with some TM implementations and a running example for software support. The causes for performance limitations that TMs may suffer from and some recent solutions to cope with such limitations are also discussed.

Otto and Tichy in the ninth chapter, '*Object-oriented stream programming*', present an approach unifying the concepts of object orientation (OO) and stream programming aiming to take advantage of features of both paradigms. Aiming for better programmability and performance gains, the object-oriented stream programming (OOSP) is introduced as a solution. The benefits of OO and stream programming are exemplified with XJava, a prototype OOSP language extending Java. Other issues such as potential conflicts between tasks, run-time performance tuning and correctness, allowing for interprocess application optimization and faster parameter adjustments are also discussed.

The tenth chapter, '*Software-based speculative parallelization*', by **Tian et al**. studies the thread level speculative parallelization (SP) approach for parallelizing sequential programs by exploiting dynamic parallelism that may be present in a sequential program. As SP is usually applied to loops and performed at compile time, it requires minimal help from the programmer who may be required to identify loops

to which speculative parallelization is to be applied. The authors have discussed several issues in SP, such as handling misspeculations, recovery capabilities and techniques for identifying parallelizable regions. Some ongoing projects that focus on SP techniques are also briefly discussed along with direction on future research issues comprising energy efficiency in SP, using SP for heterogeneous processors and 3D multicore processors, etc.

Schubert et al. in the eleventh chapter, '*Autonomic distribution and adaptation*', describe an approach for increasing the scalability of applications by exploiting inherent concurrency in order to parallelize and distribute the code. The authors focus more specifically on concurrency, which is a crucial part in any parallelization approach, in the sense of reducing dependencies between logical parts of an application. To that end, the authors have employed graph analysis methods to assess the dependencies on code level, so as to identify concurrent segments and relating them to the specific characteristics of the (heterogeneous, large-scale) environment. Issues posed to programming multicore and many-core computers by the high degree of scalability and especially the large variance of processor architectures are also discussed.

Part III: Programming Frameworks

The third part of the book deals with methodologies, frameworks and high programming tools for constructing and testing software that can be ported between different, possibly in themselves heterogeneous many-core systems under preservation of specific quantitative and qualitative performance aspects.

The twelfth chapter, '*PEPPHER: Performance portability and programmability for heterogeneous many-core architectures*', by **Benkner et al**. presents PEPPHER framework, which introduces a flexible and extensible compositional metalanguage for expressing functional and nonfunctional properties of software components, their resource requirements and possible compilation targets, as well as providing abstract specifications of properties of the underlying hardware. Also, handles for the run-time system to schedule the components on the available hardware resources are provided. Performance predictions can be (automatically) derived by combining the supplied performance models. Performance portability is aided by guidelines and requirements to ensure that the PEPPHER framework at all levels chooses the best implementation of a given component or library routine among the available variants, including settings for tunable parameters, prescheduling decisions and data movement operations.

Aldinucci et al. in the thirteenth chapter, '*Fastflow: high level and efficient streaming on multicore*', consider, as in other chapters, the difficulties of programmability of multicore and many-core systems, but from the perspective of two interrelated needs, namely, that of efficient mechanisms supporting correct concurrent access to shared-memory data structures and of higher-level programming environments capable of hiding the difficulties related to the correct and efficient use of shared-memory objects by raising the level of abstraction provided to application programmers. To address these needs the authors introduce and discuss FastFlow, a programming framework specifically targeting cache-coherent shared-memory multicores. The authors show the suitability of the programming

abstractions provided by the top layer of FastFlow programming model for application programmers. Performance and efficiency considerations are also given along with some real-world applications.

In the fourteenth chapter, Roma et al., '*Programming framework for H.264/AVC video encoding in multicore systems*', the authors bring the example of usefulness of multicore and many-core computing for the video encoding as part of many multimedia applications. As video encoding distinguishes for being highly computationally demanding, to cope with the real-time encoding performance concerns, parallel approaches are envisaged as solutions to accelerate the encoding. The authors have presented a new parallel programming framework, which allows to easily and efficiently implementing high-performance H.264/AVC video encoders. The modularity and flexibility make this framework particularly suited for efficient implementations in either homogeneous or heterogeneous parallel platforms, providing a suitable set of fine-tuning configurations and parameterizations that allow a fast prototyping and implementation, thus significantly reducing the developing time of the whole video encoding system.

The fifteenth chapter, '*Parallelizing evolutionary algorithms on GPGPU cards with the EASEA platform*', by **Maitre et al**. presents the EASEA (EAsy Specification of Evolutionary Algorithm) software platform dedicated to evolutionary algorithms that allows to exploit parallel architectures, that range from a single GPGPU equipped machine to multi-GPGPU machines, to a cluster or even several clusters of GPGPU machines. Parallel algorithms implemented by the EASEA platform are proposed for evolutionary algorithms and evolution strategies, genetic programming and multiobjective optimization. Finally, a set of problems is presented that contains artificial and real-world problems, for which performance evaluation results are given. EASEA is shown suitable to efficiently parallelize generic evolutionary optimization problems to run on current petaflop machines and future exaflop ones.

Part IV: Testing, Evaluation and Optimization

The forth part of the book covers testing, evaluation and optimization of parallel programs, with special emphasis for multicore and many-core systems. Techniques, methodologies and approaches are presented along four chapters.

Farchi in the sixteenth chapter, '*Smart interleavings for testing parallel programs*', discusses the challenges of testing parallel programs that execute several parallel tasks, might be distributed on different machines, under possible node or network failures and might use different synchronization primitives. Therefore, the main challenge is of parallel program testing resides in the definition and coverage of the rather huge space of possible orders of tasks and environment events. The author has presented state-of-the-art testing techniques including parallel bug pattern-based reviews and distributed reviews. The later techniques enable the design of a test plan for the parallel program that is then implemented in unit testing. Coping with the scaling is envisaged as a main challenge for future research.

In the seventeenth chapter by **Shafi**, '*Parallel performance evaluation and optimization*', are covered important aspects of shared-memory parallel programming

that impact performance. Guidance and mitigation techniques for diagnosing performance issues applicable to a large spectrum of shared-memory multicore programs in order to assist in performance tuning are also given. Various overheads in parallel programs including thread overheads, cache overheads and synchronization overheads are discussed and mitigation techniques analyzed. Also, optimization-related issues such as nonuniform access memory and latency are described. The chapter overviews diagnostic tools as critical means to achieving good performance in parallel applications.

The eighteenth chapter, '*A methodology for optimizing multithreaded system scalability on multicores*', by **Gunther et al**. presents a methodology which combines controlled measurements of the multithreaded platform together with a scalability modeling framework within which to evaluate performance measurements for multithreaded programs. The authors show how to quantify the scalability using the Universal Scalability Law (USL) by applying it to controlled performance measurements of memcached, J2EE and WebLogic. The authors advocate that system performance analysis should be incorporated into a comprehensive methodology rather than being done as an afterthought. Their methodology, based on the USL, emphasizes the importance of validating scalability data through controlled measurements that use appropriately designed test workloads. Some results from quantifying GPU and many-core scalability using the USL methodology are also reported.

Ozturk and Kandemir in the nineteenth chapter, '*Improving multicore system performance through data compression*', consider some important issues related to accessing off-chip memory in a multicore architecture. Such issues include off-chip memory latencies, large performance penalties, bandwidth limitations between the multicore processor and of the off-chip memory, which may not be sufficient to handle simultaneous off-chip access requests coming from multiple processors. To tackle these issues the authors propose an on-chip memory management scheme based on data compression, aiming to reduce access latencies, reduce off-chip bandwidth requirements and increase the effective on-chip storage capacity. Results are exemplified with empirical data from an experimental study. Building an optimization framework to find the most suitable parameters in the most effective way is planned for future research direction.

Part V: Scheduling and Management

The last part of the book deals with scheduling and resource management in multicore and many-core computing systems. The chapters discuss many-core accelerators as catalysts for HPC systems, nodes management, configuration, efficient allocation and scheduling in multicore clusters as well as operating systems and scheduling support for multicore systems and accelerator-based clusters.

In the twentieth chapter, '*Programming and managing resources on accelerator enabled clusters*', **Rafique et al**. study the use of computational accelerators as catalysts for HPC systems and discuss the challenges that arise in accelerator-based systems (specifically the case of accelerators on clusters), large-scale parallel systems with heterogeneous components for provisioning general-purpose resources

and custom accelerators to achieve a balanced system. The study is exemplified with a study on the implementation of MapReduce, a high-level parallel programming model for large-scale data processing, on asymmetric accelerator-based clusters. Empirical results are presented from an experimental test-bed using three representative MapReduce benchmarks, which shed light on overall system performance.

Muresano et al. in the twenty-first chapter, '*An approach for efficient execution of SPMD applications on multicore clusters*', describe an efficient execution methodology for multicore clusters, which is based on achieving a suitable application execution with a maximum speedup achievable while the efficiency is maintained over a defined threshold. The proposed methodology enables calculating the maximum number of cores that maintain strong application scalability while sustaining a desired efficiency for SPMD applications. The ideal number of tiles that have to be assigned to each core with the objective of maintaining a relationship between speedup and efficiency can also be calculated. It was shown, by experimental evaluation tests using various scientific applications, that the execution methodology can reach an improvement of around 40% in efficiency.

The last chapter, '*Operating system and scheduling for future multicore and many-core platforms*', by **Cucinotta et al**. analyzes the limitations of the nowadays operating system support for multicore systems, when looking at future and emerging many-core, massively parallel and distributed platforms. Therefore, most promising approaches in the literature dealing with such platforms are discussed. The discussion is mainly focused on the kernel architecture models and kernel-level mechanisms, and the needed interface(s) toward user-level code and more specifically on the problem of scheduling in multiprocessor and distributed systems, comprising scheduling of applications with precise timing requirements.

ACKNOWLEDGEMENTS

The editors of the book would like to sincerely thank the authors for their contributions and their patience during the preparation and publication of the book. We would like to appreciate the reviewers' constructive feedback that helped improve the content of the chapters. We would like to express our gratitude to Prof. Albert Y. Zomaya, Founding Editor-in-Chief of the Wiley Book Series on Parallel and Distributed Computing, for his encouragement and the opportunity to edit this book. The help and support from Wiley editorial and publishing team are highly appreciated!

Fatos Xhafa's work is partially supported by research projects from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R).

ACRONYMS

ACML	AMD core math library
ALU	arithmetic logic unit
AMP	asymmetric multicore processor
API	application programming interface
ASF	advanced synchronization facility
ASIC	application-specific integrated circuit
ATLAS	automatically tuned linear algebra software
BE	best effort
BLAS	basic linear algebra subprograms
BLI	bilinear interpolation
BSP	bulk synchronous parallel
CABAC	context-adaptive binary arithmetic coding
CAS	compare-and-swap
CAVLC	context-adaptive variable-length coding
CBS	constant bandwidth server

ccNUMA	cache-coherent nonuniform memory access architecture
Cell/B.E.	cell broadband engine
CellSs	cell superscalar
CG	conjugate gradient
CMP	chip multiprocessor
CorD	copy or discard
COTS	commercial off-the-shelf
CPU	central processing unit
Ct	intel C for throughput
cuBLAS	CUDA BLAS
CUDA	compute unified device architecture
D & C	divide and conquer
DCT	discrete cosine transform
DES	data encryption standard
DMA	direct memory access
DPB	decoded picture buffer
DRAM	dynamic random-access memory
DSL	domain-specific language
DSP	digital signal processor
DTMC	dresden transactional memory compiler
EA	evolutionary algorithm
EASEA	EAsy Specification of Evolutionary Algorithms
EC2	elastic compute cloud
EIB	element interconnect bus
ES	evolution strategy
FFT	fast fourier transform
FIFO	first in, first out
FIR	finite impulse response
FOSS	free open-source software
FPGA	field-programmable gate array
FS	file system
GA	genetic algorithm
GCC	GNU compiler collection

GOP	group of pictures
GP	genetic programming
GP	general-purpose
GPGPU	general-purpose computing on graphics processing units
GPMC	general-purpose multicore
GPOS	general-purpose operating system
GPP	general-purpose processor
GPU	graphics processing units
GPUSs	GPU superscalar
HPC	high-performance computing
HPL	high-performance linpack
HT	hyper-threading
HW	hardware
ILP	instruction-level parallelism
IOIF	input/output interface
IPC	interprocess communications
ISA	instruction set architecture
J2EE	Java 2 platform, enterprise edition
JVM	Java virtual machine
KOPS	kilo operations per second
KPN	Kahn process networks
LAN	local area network
LL/SC	Load-linked/store-conditional
LOC	lines of code
LS	local storage
MB	macroblock
MCD	memcached
MCSTL	multi-core standard template library
ME	motion estimation
MFC	memory flow controller
MG	multigrid
MIC	memory interface controller
MIMD	multiple instruction, multiple data

MKL	math kernel library
MOEA	multiobjective evolutionary algorithm
MP	multiprocessor
MPI	message passing interface
MPMC	multiproducer/multiconsumer
MPMD	multiple process, multiple data
MPSC	multiproducer/single-consumer
MPSoC	multiprocessor system-on-chip
NAS	NASA Advanced Supercomputing
NB-FEB	nonblocking full/empty bit
NFS	network file system
NIC	network Interconnect
NUMA	nonuniform memory access
ODE	ordinary differential equation
OmpSs	openMP superscalar
OO	object-orientation
OOSP	object-oriented stream programming
OpenCL	open computing language
OpenMP	open multiprocessing
OS	operating system
PCIe	PCI express (peripheral component interconnect express)
PEPPHER	performance portability and programmability for heterogeneous many-core architectures
PGAS	partitioned global address space
POSIX	portable operating system interface
PPE	power processing element
PPU	power processing unit
PS3	PlayStation 3
PSNR	peak signal-to-noise ratio
PU	processing unit
QoS	quality of service
RAM	random-access memory
RISC	reduced instruction set computing

RT	real time
RTOS	real-time operating system
SAD	sum of absolute differences
SDF	synchronous data flow
SDK	software development kit
SGL	single global lock
SIMD	single instruction, multiple data
SIMT	single instruction, multiple threads
SISD	single instruction, single data
SIU	system interface unit
SM	streaming multiprocessor
SMP	symmetric multiprocessors
SMPSs	SMP superscalar
SMT	simultaneous multithreading
SoC	system on chip
SP	streaming processor
SP	speculative parallelization
SPE	synergistic processing element
SPM	Scratchpad memory
SPMD	single program, multiple data
SPSC	single producer, single consumer
SPU	synergistic processing unit
SSI	single system image
ST	supertile
STAPL	Standard Template Adaptive Parallel Library
STL	Standard Template Library
STM	software transactional memory
SW	Software
TBB	Threading Building Blocks
TBD	total bandwidth server
TLP	thread-level parallelism
TLS	thread-level speculation
TM	transactional memory
TPA	thread processor array

TPC	thread processing cluster
UMA	uniform memory access
USL	universal scalability law
VLIW	very long instruction word
VLSI	very-large-scale integration

PART I

FOUNDATIONS

CHAPTER 1

MULTI- AND MANY-CORES, ARCHITECTURAL OVERVIEW FOR PROGRAMMERS

Lasse Natvig, Alexandru Iordan, Mujahed Eleyat, Magnus Jahre and Jorn Amundsen

1.1 INTRODUCTION

1.1.1 Fundamental Techniques

Parallelism has been used since the early days of computing to enhance performance. From the first computers to the most modern sequential processors (also called *uniprocessors*), the main concepts introduced by von Neumann [20] are still in use. However, the ever-increasing demand for computing performance has pushed computer architects toward implementing different techniques of parallelism. The von Neumann architecture was initially a sequential machine operating on scalar data with bit-serial operations [20]. *Word-parallel* operations were made possible by using more complex logic that could perform binary operations in parallel on all the bits in a computer word, and it was just the start of an adventure of innovations in parallel computer architectures.

Programming Multicore and Many-core Computing Systems,
First Edition. Edited by Sabri Pllana and Fatos Xhafa.

Prefetching is a 'look-ahead technique' that was introduced quite early and is a way of parallelism that is used at several levels and in different components of a computer today. Both data and instructions are very often accessed sequentially. Therefore, when accessing an element (instruction or data) at address *k*, an automatic access to address *k+1* will bring the element to where it is needed *before* it is accessed and thus eliminates or reduces waiting time. Many clever techniques for *hardware prefetching* have been researched [5, 17] and can be exploited in the context of the new multicore processors. However, the opportunities and challenges given by the new technology in multicores require both a review of old techniques and a development of new ones [9, 21]. *Software prefetching* exploits sequential access patterns in a similar way but either it is controlled by the compiler inserting prefetch operations or it can be explicitly controlled by the programmer [10].

Block access is also a fundamental technique that in some sense is a parallel operation. Instead of bringing one word closer to the processor, for example, from memory or cache, a *cache line* (block of words) is transferred. Block access also gives a prefetching effect since the access to the first element in the block will bring in the succeeding elements. The evolution of processor and memory technology during the last 20 years has caused a large and still increasing gap between processor and memory speed-making techniques such as prefetching and block access even more important than before. This *processor–memory gap*, also called the *memory wall*, is further discussed in Section 1.2.

Functional parallelism is a very general technique that has been used for a long time and is exploited at different levels and in different components of almost all computers today. The principle is to have different functional units in the processor that can operate concurrently. Consequently, more than one instruction can be executed at the same time, for example, one unit can execute an arithmetic integer operation while another unit executes a floating-point operation. This is to exploit what has later been called *instruction level parallelism* (ILP).

Pipelining is one main variant of functional parallelism and has been used extensively at different levels and in different components of computers to improve performance. It is perhaps most widely known from the *instruction pipeline* used in almost all contemporary processors. Instructions are processed as a sequence of steps or stages, such as instruction fetch, instruction decoding, execution and write back of results. Modern microprocessors can use more than 20 pipeline stages so that more than 20 instructions are being processed concurrently. Pipelining gives potentially a large performance gain but also added complexity since interdependencies between instructions must be handled to ensure correct execution of the program.

The term *scalar processor* denotes computers that operate on one computer word at a time. When functional parallelism is used as described in the preceding text to exploit ILP, we have a *superscalar processor*. A *k-way superscalar* processor can issue up to *k* instructions at the same time (during one clock cycle). Also instruction fetching, decoding and other nonarithmetic operations are parallelized by adding more functional units.

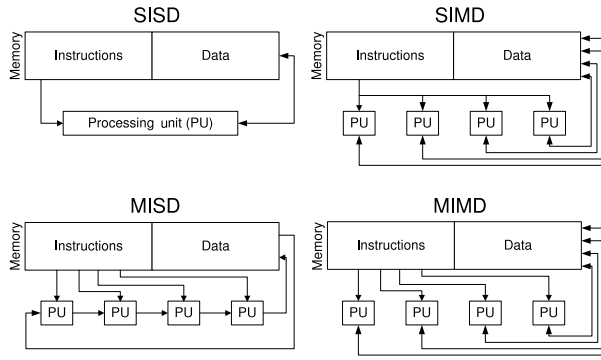


Figure 1.1 Flynn's taxonomy.

1.1.2 Multiprogramming, Multiprocessors and Clusters

Multiprogramming is a technique invented in the 1960s to interleave the execution of the programs and I/O operations among different users by time multiplexing. In this way many users can share a single computer and get acceptable response time, and the concept of a *time-sharing operating system* controlling such a computer was a milestone in the history of computers.

Multiprocessors are computers with two or more distinct physical processors, and they are capable of executing real parallel programs. Here, at the cost of additional hardware, a performance gain can be achieved by executing the parallel processes in different processors.

Many multiprocessors were developed during the 1960s and early 1970s, and in the start most of the commercial multiprocessors had only two processors. Different research prototypes were also developed, and the first computer with a large number of processors was the Illiac IV developed at the University of Illinois [6]. The project development stretched roughly 10 years, and the computer was designed to have 256 processors but was never built with more than 64 processors.

1.1.2.1 Flynn's Taxonomy

Flynn divided multiprocessors into four categories based on the multiplicity of instruction streams and data streams – and this has become known as the famous *Flynn's taxonomy* [14, 15] illustrated in Figure 1.1.

A conventional computer (uniprocessor or von Neumann machine) is termed a *Single Instruction Single Data (SISD)* machine. It has one execution or processing unit (PU) that is controlled by a single sequence of instructions, and it operates on a single sequence of data in memory. In the early days of computing, the control logic needed to decode the instructions into control signals that manage the execution and data traffic in a processor was a costly component. When introducing parallel processing, it was therefore natural to let multiple execution units operate on different data (multiple data streams) while they were controlled by the same single *control unit*, that is, a single instruction stream. A fundamental limitation of these *SIMD archi-*

tectures is that different PUs cannot execute different instructions and, at the same time, they are all bound to one single instruction stream.

SIMD machines evolved in many variants. A main distinction is between SIMD with shared memory as shown in Figure 1.1 and SIMD computers with distributed memory. In the latter variant, the main memory is distributed to the different PUs. The advantage of this architecture is that it is much easier to implement compared to multiple data streams to one shared memory. A disadvantage is that it gives the need for some mechanism such as special instructions for communicating between the different PUs.

The *Multiple Instruction Single Data (MISD)* category of machines has been given a mixed treatment in the literature. Some textbooks simply say that no machines of this category have been built, while others present examples. In our view MISD is an important category representing different parallel architectures. One of the example architectures presented in the classical paper by Flynn [14] is very similar to the variant shown in Figure 1.1. Here a source data stream is sent from the memory to the first PU, then a *derived* data stream is sent to the next PU, where it is processed by another program (instruction stream) and so on until it is streamed back to memory. This kind of computation has by some authors been called a *software pipeline* [26]. It can be efficient for applications such as real-time processing of a stream of images (video) data, where data is streamed through different PUs executing different image processing functions (e.g. filtering or feature extraction).

Another type of parallel architectures that can be classified as MISD is *systolic arrays*. These are specialized hardware structures, often implemented as an application specific integrated circuit (ASIC), and use highly pipelined and parallel execution of specific algorithms such as pattern matching or sorting [36, 22].

The *Multiple Instruction Multiple Data (MIMD)* category comprises most contemporary parallel computer architectures, and its inability to categorize these has been a source for the proposal of different alternative taxonomies [43]. In a MIMD computer, every PU has its own control unit that reads a separate stream of instructions dictating the execution in its PU. Just as for SIMD machines, a main subdivision of MIMD machines is into those having shared memory or distributed memory. In the latter variant each PU can have a local memory storing both instructions and data. This leads us to another main categorization of multiprocessors, –shared memory multiprocessors and message passing multiprocessors.

1.1.2.2 Shared Memory versus Message Passing

When discussing communication and memory in multiprocessors, it is important to distinguish the *programmers view* (logical view or *programming model*) from the actual implementation (physical view or *architecture*). We will use Figure 1.2 as a base for our discussion.

The programmers, view of a *shared memory multiprocessor* is that all processes or threads share the same single main memory. The simplest and cheapest way of building such a machine is to attach a set of processors to one single memory through a bus. A fundamental limitation of a bus is that it allows only one transaction (communication operation or memory access) to be handled at a time. Consequently, its performance does not scale with the number of processors. When multiproces-

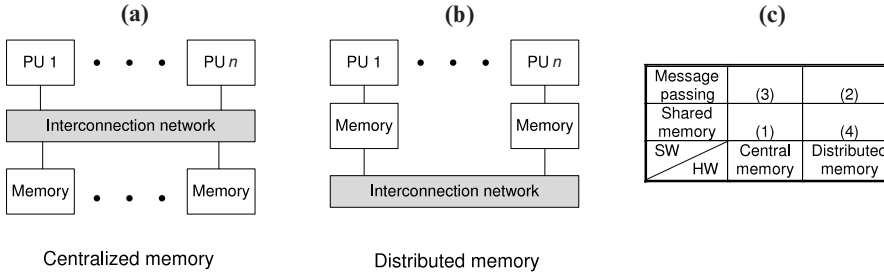


Figure 1.2 Multiprocessor memory architectures and programming models.

sors with higher number of processors were built – the bus was often replaced by an interconnection network that could handle several transactions simultaneously. Examples are a crossbar switch (all-to-all communication), multistage networks, hypercubes and meshes (see [23] Appendix E for more details). The development of these parallel interconnection networks is another example of increased use of parallelism in computers, and they are highly relevant also in multi- and many-core architectures.

When attaching many processors to a single memory module through a parallel interconnection network, the memory could easily become a bottleneck. Consequently, it is common to use several physical memory modules as shown in Figure 1.2(a). Although it has multiple memory modules, this architecture can be called a *centralized memory* system since the modules (memory banks) are assembled as one subsystem that is equally accessible from all the processors. Due to this uniformity of access, these systems are often called *symmetric multiprocessors (SMP)* or *uniform memory access (UMA)* architectures. This programming model (SW) using shared memory implemented on top of centralized memory (HW) is marked as alternative (1) in Figure 1.2(c).

The parallel interconnection network and the multiplicity of memory modules can be used to let the processors work independently and in parallel with different parts of the memory, or a single processor can distribute its memory accesses across the memory banks. This latter technique was one of the early methods to exploit parallelism in memory systems and is called *memory interleaving*. It was motivated by memory modules being much slower than the processors and was together with *memory pipelining* used to speed up memory access in early multiprocessors [26]. As seen in the next section, such techniques are even more important today.

The main alternative to centralized memory is called *distributed memory* and is shown in Figure 1.2(b). Here, the memory modules are located together with the processors. This architecture became popular during the late 1980s and 1990s, when the combination of the RISC processor and VLSI technology made it possible to implement a complete processor with local memory and network interconnect (NIC) on a single board. The machines typically ran multiprocessor variants of the UNIX operating system, and parallel programming was facilitated by *message passing libraries*, standardized with the message passing interface (MPI) [47]. Typical for these machines is that access to a processors local memory module is much

faster than access to the memory module of another processor, thus giving the name *NonUniform Memory Access (NUMA)* machines. This multiprocessor variant with message passing SW and a physically distributed memory is marked as (2) in the right part of Figure 1.2(c).

The distributed architectures are generally easier to build, especially for computers designed to be scalable to a large number of processors. When the number of processors grows in these machines either the cost of the interconnection network will increase rapidly (as with crossbar) or it will become both more costly and slower (as with multistage network). A slower network will make every memory access slower if we use centralized memory.

However, with distributed memory, a slower network can to some extent be hidden if a large fraction of the accesses can be directed to the local memory module. When this design choice is made, we can use cheaper networks and even a hierarchy of interconnection networks, and the programmer is likely to develop software that exploits the given NUMA architecture. A disadvantage is that the distribution and use of data might become a crucial factor to achieve good performance – and in that way making programming more difficult. Also, the ability of porting code to other architectures without loosing performance is reduced.

Shared memory is generally considered to make parallel programming easier compared to message passing, since cooperation and synchronization between the processors can be done through shared data structures, explicit message passing code can be avoided, and memory access latency is relatively uniform. In such *distributed shared memory (DSM)* machines, the programmers view is one single address space, and the machine implements this using specialized hardware and/or system software such as message passing. The last alternative (3) – to offer message passing on top of centralized memory-is much less common but can have the advantage of offering increased portability of message passing code. As an example, MPI has been implemented on multicores with shared memory [42].

The term *multicomputer* has been used to denote parallel computers built of autonomous processors, often called nodes [26]. Here, each node is an independent computer with its own processor and address space, but message passing can be used to provide the view of *one* distributed memory to the multicomputer programmer. The nodes normally also have I/O units, and today the mostly used term for these parallel machines is *cluster*. Many clusters are built of commercial-off-the -shelf (COTS) components, such as standard PCs or workstations and a fast local area network or switch. This is probably the most cost-efficient way of building a large supercomputer if the goal is maximum compute power on applications that are easy to parallelize. However, although the network technology has improved steadily, these machines have in general a much lower internode communication speed and capacity compared to the computational capacity (processor speed) of the nodes. As a consequence, more tightly coupled multiprocessors have often been chosen for the most communication intensive applications.

1.1.2.3 Multithreading

Multithreading is quite similar to multiprogramming, that is, multiple processes or threads share the functional units of one processor by using

overlapped execution. The purpose can be to execute several programs on one processor as in multiprogramming or can be to execute a single application organized as a multithreaded program (real parallel program). The threads in multithreading are sometimes called HW threads, while the threads of an application can be called SW threads or processes. The HW threads are under execution in the processor, while SW threads can be waiting in a queue outside the processor or even swapped to disk.

When implementing multithreading in a processor, it is common to add internal storage making it possible to save the current architectural state of a thread in a very fast way, making rapid switches between threads possible.

A switch between processes, normally denoted *context switch* in operating systems terminology, can typically use hundreds or even thousands of clock cycles, while there is multithreaded processors that can switch to another thread within one clock cycle. Processes can belong to different users (applications) while threads belong to the same user (application). The use of multithreading is now commonly called *thread-level parallelism (TLP),* and it can be said to be a higher level of parallelism than ILP since the execution of each single thread can exploit ILP.

Fine-grained multithreading denotes cases where the processor switches between threads at every instruction, while in *coarse grained multithreading* the processor executes several instructions from the same thread between switches, normally when the thread has to wait for a lengthy memory access. Both ILP and TLP can be combined as in *simultaneous multithreading (SMT)* processors where the k issue slots of a k-way superscalar processor can be filled with instructions from different threads. In this way, it offers 'real parallelism' in the same way as a multiprocessor. In a SMT processor, the threads will compete for the different subcomponents of the processor, and this might at first sight seem to be a poor solution compared to a multiprocessor where a process or thread can run at top speed without competition from other threads. The advantage of SMT is the good resource utilization of such architectures – very often the processor will stall on lengthy memory operations, and more than one thread is needed to fill in the execution gap. *Hyper-threading* is Intel's terminology (officially called hyper-threading technology) and corresponds to SMT [48].

1.2 WHY MULTICORES?

In recent years, general-purpose processor manufacturers have started to provide chips with multiple processor cores. This type of processor is commonly referred to as a *multicore architecture* or a *chip multiprocessor (CMP)* [38]. Multicores have become a necessity due to four technological and economical constraints, and the purpose of this section is to give a high-level introduction to these.

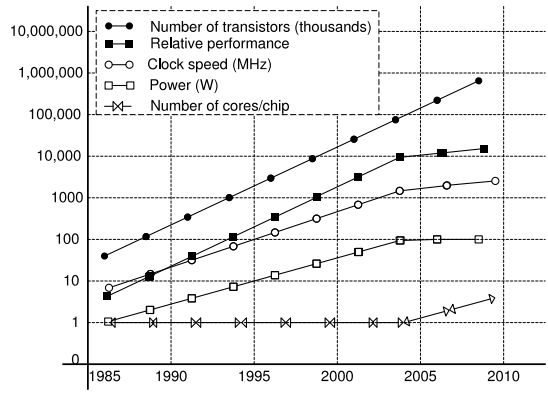


Figure 1.3 Technological trends for microprocessors. Simplified version of Figure 1 in [18].

1.2.1 The Power Wall

High-performance single-core processors consume a great deal of power, and high power consumption necessitates expensive packaging and powerful cooling solutions. During the 1990s and into the 21st century, the strategy of scaling down the gate size of integrated circuits, reducing the supply voltage and increasing the clock rate, was successful and resulted in faster single-core processors. However, around year 2004, it became infeasible to continue reducing the supply voltage, and this made it difficult to continue increasing the clock speed without increasing power dissipation. As a result, the power dissipation started to grow beyond practical limits [18], and the single-core processors were said to hit the *power wall*. In a CMP, multiple cores can cooperate to achieve high performance at a lower clock frequency.

Figure 1.3 illustrates the evolution of processors and the recent shift toward multicores. First, the figure illustrates that Moore's law still holds since the number of transistors is increasing exponentially. However, the relative performance, clock speed and power curves have a distinct knee in 2004 and has been flat or slowly increasing since then. As these curves flatten, the number of cores per chip curve has started to rise. The *aggregate* chip performance is the product of the relative performance per core and the number of cores on a chip, and this scales roughly with Moore's law. Consequently, Figure 1.3 illustrates that multicores are able to increase aggregate performance without increasing power consumption. This exponential performance potential can only be realized for a single application through scalable parallel programming.

1.2.2 The Memory Wall

Processor performance has been improving at a faster rate than the main memory access time for more than 20 years [23]. Consequently, the gap between processor performance and main memory latency is large and growing. This trend is referred to as the *processor–memory gap* or *memory wall*. Figure 1.4 contains the classical plot by Hennessy and Patterson that illustrates the memory wall. The effects of the

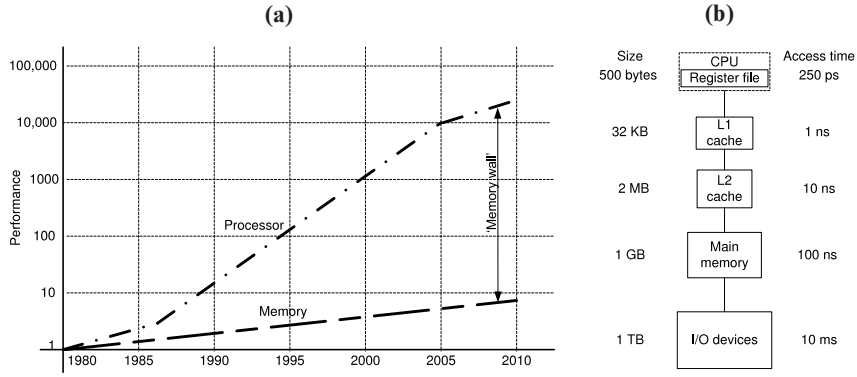


Figure 1.4 The processor–memory gap (a) and a typical memory hierarchy (b).

memory wall have traditionally been handled with latency hiding techniques such as pipelining, out-of-order execution and multilevel caches. The most evident effect of the processor–memory gap is the increasing complexity of the memory hierarchy, shown in Figure 1.4(b). As the gap increased, more levels of cache were added. In recent years, it has been common with a third level of cache, L3 cache. The figure gives some typical numbers for storage capacity and access latency at the different levels [23].

The memory wall also affects multicores, and they invest a significant amount of resources to hide memory latencies. Fortunately, since multicores use lower clock frequencies, the processor–memory gap is growing at a slower rate for multicores than for traditional single cores. However, *aggregate* processor performance is growing at roughly the same rate as Moore's Law. Therefore, multicores to some extent transform a latency hiding problem into an increased bandwidth demand. This is helpful because off-chip bandwidth is expected to scale significantly better than memory latencies [29, 40]. The multicore memory system must provide enough bandwidth to support the needs of an increasing number of concurrent threads. Therefore, there is a need to use the available bandwidth in an efficient manner [30].

1.2.3 The ILP Wall and the Complexity Wall

It has become increasingly difficult to improve performance with techniques that exploit ILP beyond what is common today. Although there is a considerable ILP available in the instruction stream [55], extracting it has proven difficult with current process technologies [2]. This trend has been referred to as the *ILP wall*. Multicores alleviate this problem by shifting the focus from transparently extracting ILP from a serial instruction stream to letting the programmer provide the parallelism through TLP.

Designing and verifying a complex out-of-order processor is a significant task. This challenge has been referred to as the *complexity wall*. In a multicore, a processor core is designed once and reused as many times as there are cores on the chip.

These cores can also be simpler than their single-core counterparts. Consequently, multicores facilitate design reuse and reduce processor core complexity.

1.3 HOMOGENEOUS MULTICORES

Contemporary multicores can be divided into two main classes. This section introduces *homogeneous multicores* that are processors where all the cores are similar, that is, they execute the same instruction set, they run on the same clock frequency and they have the same amount of cache resources. Conceptually, these multicores are quite similar to SMPs. The section starts by introducing a possible categorization of such multicores, before we describe a selected set of modern multicores at a high level. All of these are rather complex products, and both *the scope of this chapter and the space available make it impossible to give a complete and thorough description. Our goal is to introduce the reader to the richness and variety of the market – motivating for further studies.* The other mainclass, heterogeneous multicores, is discussed in the next section. A tabular summary of a larger number of commercial multicores can be found in a recent paper by Sodan et-al. [48].

1.3.1 Early Generations

In the paper *Chip Multithreading: Opportunities and Challenges* by Spracklen and Abraham [50], the authors introduced a categorization of what they called chip multi threaded processors (CMT processors) that also can be used to categorize multicore architectures. As shown in Figure 1.5, the first generation multicores typically had processor cores that did not share any on-chip resources except the off-chip datapaths. It was normally two cores per chip and they were derived from earlier uniprocessor designs. Also the PUs used in the second generation multicores were from earlier uniprocessor designs, but they were more tightly integrated through use of a *shared L2 cache*. It could be more than two processors, and the shared L2 made intracore communication very fast. The cores sometimes run the same program (SPMD), so the demand for cache capacity for storing instructions can be reduced. Both these advantages of the shared L2 cache can reduce the demand of off-chip bandwidth. However, more than one core using the L2 cache introduce new challenges such as cache partitioning, fairness and quality of service (Qos) [12, 11, 30].

The third generation multicores can be said to be those using cores that are designed from the ground up and optimized to sit in a multicore processor. These may typically be simpler cores running at a lower frequency and hence with a much lower power consumption. Further, they are typically using SMT. Olukotun and Hammond [37] call these three generations for simple CMP, shared-cache CMP and multithreaded shared-cache CMP, respectively.

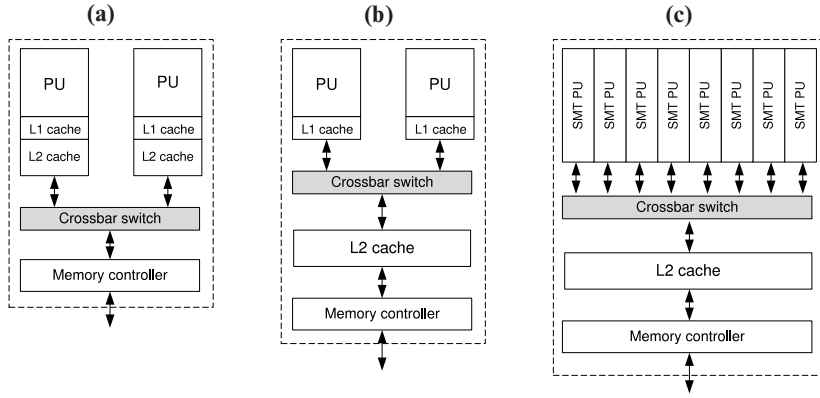


Figure 1.5 Multicore processor generations: first (a), second (b), third (c).

1.3.2 Many Thin Cores or Few Fat Cores?

The choice between a few powerful and many less powerful processors or cores has been discussed widely both during the multiprocessor era and the multicore era. In his classical paper Amdahl [3] gave a simple formula explaining how the serial fraction of an application severely constraints the maximum speedup that can be achieved by a multiprocessor. The serial fraction is a code that cannot be parallelized, and Amdahl's law might motivate for having at least one core that is faster than the others, that is, go for a heterogeneous multicore. For executing the so-called *embarrassingly parallel applications*, that is, applications that are very easy to parallelize since they have no or a very tiny serial part – a multicore with a large number of small cores might be most efficient, especially if power efficiency is in focus. However, if there is significant serial fraction, a smaller number of more powerful cores might be best. A recent paper by Hill and Marty [24] titled *Amdahl's Law in the Multicore Era* demonstrates the influence of Amdahl's law on this trade-off in an elegant way.

1.3.3 Example Multicore Architectures

1.3.3.1 IBM(R) Power(R) Performance Optimization With Enhanced RISC (POWER) is an IBM processor architecture for technical computing workloads implementing superscalar RISC. The POWER architecture was the starting point in 1991 of the Apple®, IBM and Motorola® (now Freescale Semiconductor®) joint effort to develop a new RISC processor architecture, the PowerPC® architecture [49]. The design goals of PowerPC were to create a single chip providing multiprocessing extensions and 64-bit support (addressing and operations). It was later expanded with vector instructions, originally trademarked AltiVec™. In 2006, POWER and PowerPC was unified into a new brand, the *Power Architecture*, owned by Power.org.

The POWER*n* series of processors are IBM's main product line implementing the Power architecture. The first product in this series was the multichip, super-

scalar and out-of-order POWER1 processor, introduced in 1990. The POWER7®, introduced in 2010, is the latest development in this series and is also the processor to power the first DARPA High Productivity Computing System (HPCS) petaflops computer. A stripped-down POWER7-core is expected to be used in the Blue Gene®/Q system, replacing the BlueGene/P massively parallel supercomputer in 2012.

The POWER7 processor provides 4, 6 or 8 cores per chip, each with 4-way hardware multithreading (SMT) [1]. A core might under software control be set to

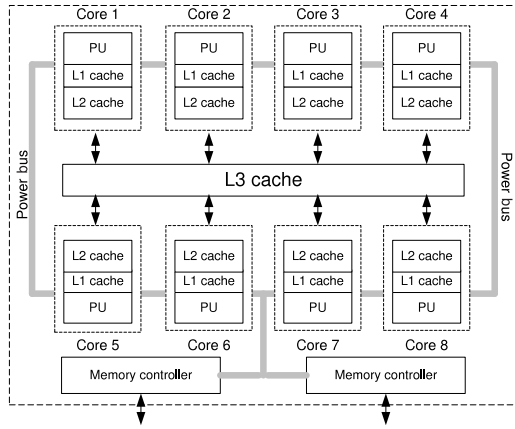


Figure 1.6 Power 7 multicore, simplified block diagram.

operate at different degrees of multithreading from single-threaded mode (ST) to 4-ways SMT.

The chip is implemented in 45 nm technology, with cores running at a nominal frequency of 3.0 – 4.14 GHz, depending on the configuration. The cache hierarchy consists of 32K 4-way L1 data and instruction caches, a 256K 8-way L2 cache and 32 MB shared L3 cache, partitioned into 8×4 MB 8-way partitions. The L3 cache is implemented with embedded DRAM technology (eDRAM). The chip is organized as 8 cores (called *chiplets*), each containing the PU, L1 and L2 caches and one of the 8 L3-cache partitions (Fig. 1.6). A consequence of this design is that the L3 has a nonuniform latency. A pair of DDR3 DRAM controllers, each with four 6.4 GHz channels provides a sustained main memory bandwidth of over 100 GB/s.

In addition to POWER6® VMX (AltiVec) and decimal floating point (DFU), the POWER7 core provides the new VSX vector facility. VSX is mainly an extension for 64-bit vector floating-point arithmetic; it does not provide 64-bit integer arithmetic like Intel® and AMD processors.

Energy efficiency is implemented at the core or chiplet level where each core frequency might be individually changed. The modes *sleep*, *nap* and *turbo* allows dynamic voltage and frequency adjustment, from off, to –50% and +10% for maximum performance.

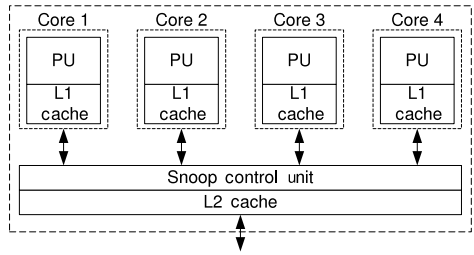


Figure 1.7 ARM Cortex A15, simplified block diagram.

1.3.3.2 ARM(R) Cortex™-A15 MPCoreTM Processor ARM became one of the first companies to implement multicore technology with the launch of the ARM11™ MPCore™ processor in 2004. The latest version of the ARM MPCore technology is the ARM Cortex™-A15 MPCore processor, targeting markets ranging from mobile computing, high-end digital home, servers and wireless infrastructure.

The processor can be implemented to include up to four cores (see Figure 1.7). The multicore architecture enables the processor to exceed the performance of single -core high-performance embedded devices while consuming significantly less power. Every Cortex-A series processor has power management features including dynamic voltage and frequency scaling and the ability for each core to go independently into standby, dormant or power off energy management states. Like its predecessors Cortex-A15 is based on the ARMv7A processor architecture giving full application compatibility with all ARM Cortex-A processors. This compatibility enables access to an established developer and software ecosystem.

Each processor core has an out-of-order superscalar pipeline and low-latency access through a bus to a shared L2 cache that can be up to 4 MB. The cores provide floating-point support and special SIMD instructions for media performance [4].

1.3.3.3 Sun UltraSPARC(R) T2 Sun's UltraSPARC T2 is a homogeneous multithreaded multicore specially designed to exploit the TLP present in almost every server type application. Sun introduced its first multicore, multithreaded microprocessor the UltraSPARC T1 (codenamed Niagara) in November 2005 [33]. The UltraSPARC T1 uses the SPARC V9® instruction set and was available with 4, 6 and 8 processing cores, each able to execute four threads simultaneously [48]. The UltraSPARC T2 includes a network interface unit and a PCI express interface unit, and this is why the T2 is sometimes referred to as a system on chip [45]. It was available in October 2007 and produced in 65 nm technology.

The UltraSPARC T2 is comprised of 8 64-bit cores, and each core can execute 8 independent threads. Thus, T2 is able to execute 64 threads simultaneously. The cores are connected by a crossbar to an 8-banked shared L2 cache, 4 DRAM controllers and 2 interface units (Fig. 1.8).

In order to minimize power requirements and to meet temperature constraints, the UltraSPARC T2 uses a core frequency of only 1.4 GHz. A complete implementation of the UltraSparc T2 processor in Verilog™ (a HW description language) along

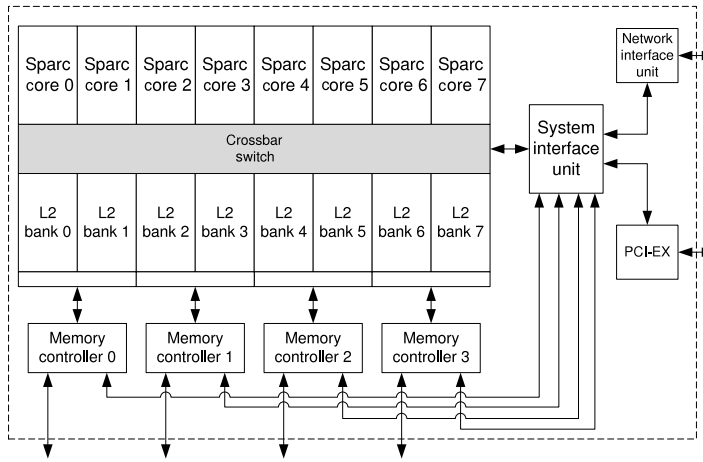


Figure 1.8 Sun UltraSPARC T2 architecture, simplified block diagram.

with tools is freely available from the OpenSPARC® project [54]. This gives the interested researcher a rare opportunity to study the inner details of a modern multicore processor.

In autumn 2010, Oracle launched the SPARC T3, previously known as UltraSPARC T3. It has 16 cores each capable of 8-way SMT giving a total of 128-way multithreading [39].

1.3.3.4 AMD Istanbul The Istanbul processor is the first 6-core AMD Opteron™ processor and is available for 2-, 4- and 8-socket systems, with clock speeds ranging from 2.0 to 2.8 GHz. It was introduced in June 2009 and is manufactured in a 45 nm process and based on the AMD 64-bit K10 architecture. The K10 architecture supports the full AMD64 instruction set and SIMD instructions for both integer and floating-point operations [25].

Figure 1.9 shows a simplified block diagram. The processor has six cores, three levels of cache, a crossbar connecting the cores, the system request interface, the memory controller and the three HyperTransport™ 3.0 links. The memory controller supports DDR2 memory with a bandwidth of up to 12.8 GB/s. In addition, the HyperTransport 3.0 links provide an aggregate bandwidth of 57.6 GB/s and are used to allow communication between different Istanbul processors.

The 6 MB of L3 cache is shared among the 6 cores: there are a 512 KB L2 cache per core and 64 KB L1 data cache and a 64 KB L1 instruction cache for each core.

1.3.3.5 Intel(R) Nehalem In November 2008, with the release of Core™ i7, Intel introduced the new microprocessor architecture Nehalem [28]. The Nehalem architecture (Fig. 1.10) has been used in a large number of processor variants in the mobile, desktop and servers markets and is mainly produced in 45 nm technology. The core count is typically 2 for mobile products, 2 – 4 cores for desktop and 4,

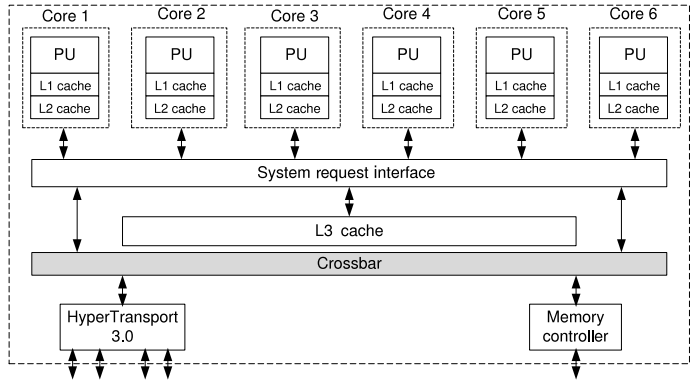


Figure 1.9 AMD Opteron Istanbul processor, simplified block diagram.

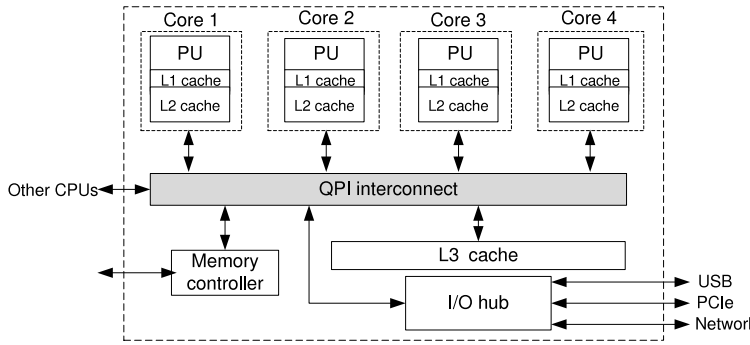


Figure 1.10 Intel Nehalem architecture – 4 cores, simplified block diagram.

6 or 8 for servers. At the high end, the Nehalem architecture shrinked to 32 nm technology (also called Westmere) has been announced to provide a 10-core chip.

Intel introduced with Nehalem the *turbo boost technology* (TBT) to allow adjustments of core frequency at runtime [27]. Considering the number of active cores, estimated current usage, estimated power requirements and CPU temperature, TBT determines the maximum frequency that the processor can run at. Core frequency can be increased in steps of 133 MHz and to a higher level if few cores are active. This allows for a boost in performance while still maintaining the power envelope. To save energy, it is possible to power down cores when they are idle, but when needed again they are turned on, and the frequency of the processor is reduced accordingly [52].

The *QuickPath interconnect* (QPI) was introduced in Nehalem to provide high speed, point-to-point connections between all cores, the I/O hub, the memory controller and the large shared L3 cache (Fig. 1.10). The L3 cache is inclusive. Nehalem-based processors have up to 3.5 times more memory bandwidth than previous generation processors.

The Nehalem architecture reintroduced hyper-threading, a technique that allows each core to run two threads simultaneously, improving on resource utilization and reducing latency. Although it was introduced in Intel processors as early as in 2002, it was not used in the *Intel core* architecture that preceded Nehalem.

For faster computation of media applications, the Nehalem architecture supports the SSE4 instruction set introduced in the previous generation processors. SSE is an abbreviation for *streaming SIMD extensions* and is an SIMD instruction set extension to the ×86 architecture that is used by compilers and assembly coders for vectorization.

1.3.3.6 Tilera(R) TILE64 TM Tilera [53] has developed and is currently shipping the TILEPro36™ and TILEPro64™ series of embedded many-core processors. The Tilera devices may contain up to 64 individual 32-bit processors on a single silicon device and are targeted at embedded markets which require programmability, high performance and demanding power constraints. All Tilera devices contain numerous integrated IO interfaces, allowing system designers to save board real estate and complexity by integrating the IO and processing into a single device. Current target markets for the TILEPro™ family of devices include video and network processing. The TILEPro family of devices is fabricated in TSMC's 90 nm technology and comes in 700 and 866 MHz frequency grades.

Each Tilera device contains multiple individual processor cores. Each core supports the TILE instruction set architecture (ISA), a Tilera proprietary ISA sharing many similarities with modern RISC ISAs. The Tilera ISA is a 3-wide VLIW format, where each 64-bit VLIW instruction encodes three operations. Correspondingly, there are three execution pipelines per processor core, two arithmetic pipelines and one load/store pipeline. When running at 866 MHz, a TILEPro64 is capable of 166 billion 32-bit operations per second. Additionally, the Tilera ISA contains SIMD operations, enabling 32b, 16b and 8b arithmetic. The physical address of the TILEPro devices is 36 bits, giving a TILEPro device access to up to 64 GB of memory. The TILEPro processor is an in-order machine, issuing 64-bit VLIW instructions in program order. However, the TILEPro cache subsystem is out of order, allowing the processor to continue to fetch, issue and execute instructions in the presence of multiple cache misses. The TILE cores do not have HW FPU support.

The TILEPro device is a complete system on a chip, containing multiple integrated IO interfaces. TILEPro64 contains four integrated DDR2 memory controllers, capable of supporting 800 MHz operation. Memory space may be configured to be automatically interleaved across the four controllers or programmatically assigned on a page-by-page mapping from page to controller.

A TILEPro processor core contains a 16 KB L1 instruction cache, an 8 KB L1 data cache and a 64 KB unified L2 cache (used for both instructions and data). All processor cores on a TILEPro device are cache coherent, enabling running of standard, shared-memory programs such as POSIX threads across the entire device. The cores may be configured into multiple coherence domains, allowing a single SMP Linux image to run across all cores within the system, or only a subset. Tilera hypervisor technology enables the ability to run multiple Linux images in parallel. Coherency

is maintained between the processor cores via a unique directory-based coherency protocol, called dynamic distributed cache (DDC). The DDC protocol tracks address sharers within the system via a distributed directory and maintains coherence by properly invalidating/updating shared data upon modification. Additionally, the Tilera cache subsystem provides the ability for one core's L2 cache to serve as a backing L3 cache for another core within the system. In this context, the L2 storage structures may contain both L2 and L3 cache blocks.

The TILEPro processor cores communicate with each other and the IO interfaces via multiple on-chip, packet-switched networks. These networks, called the iMesh™, are proprietary interconnects used to carry communication within the system such as memory read requests, memory read responses, tile-to-tile read responses, etc. The networks are configured in a mesh topology, providing performance scalability as the number of cores is increased. The TILEPro devices contain three separate mesh networks for memory and cache communication, as well as two networks for user-level messaging. These networks are synchronous with the processor cores and run at the same frequency, and the latency for a message through the mesh networks is one processor cycle per node.

1.4 HETEROGENEOUS MULTICORES

This section introduces *heterogeneous multicores* – processors where one or some of the cores are significantly different than the others. The difference can be as fundamental as the instruction set used, or it can be the processor speed or cache/memory capacity of the different cores. We start by introducing some of the main types of heterogeneity, before we present three different contemporary products in this category of processors.

1.4.1 Types of Heterogeneity in Multicores

Single-ISA heterogeneous multicores are processors where all the cores have the same ISA, that is, they can execute the same instructions, but they can have different clock frequencies and/or cache sizes. Also, the cores might have different architectures implementing the same ISA. Typically there is one or a few high-performance cores (fat cores) that are superscalar out-of-order processors and a larger number of smaller and simpler cores that can be in-order processors with a shorter pipeline [34]. As discussed in Section 1.3.2, this can be beneficial for speeding up applications where there is a significant part of the computation that is serial or if some of the threads put more demand on the memory system. This kind of multicores is called by some authors *asymmetric multicore processors (AMP)*. They have gained increased interest lately since they potentially can be more energy efficient than conventional homogeneous multicores [13].

Multiple-ISA multicores such as the Cell/BE™ microprocessor presented in Section 1.4.2.1 have two or more different instruction sets. They require a toolchain for each core type and are in general harder to program. In addition, many of these

processors, including Cell™, have explicitly managed memory hierarchies where the programmer is responsible for placement and transfer of data. This will in general increase programmer effort and code complexity compared to a cache-based system that are automatic and hidden from the programmer. Recent research has shown that comparable performance can be achieved through programming environments where compiler and runtime support implicitly manage locality [44].

In the embedded systems market, there is a long tradition of using highly heterogeneous multicores with different kinds of simple or complex cores and HW units integrated on a single chip. These *multiprocessor system-on-chip* (MPSoC) systems often achieve a very high level of power efficiency through specialization [35], but again the price to pay is often more difficult programming. MPSoC systems have been available as commercial products for longer than multicores, and some few MPSoCs are homogeneous. We refer the reader to a recent survey of MPSoCs by Wolf, Jerraya and Martin for this rich branch of multicore processors [57].

Graphics processing units (GPU) and accelerators are also considered examples of heterogeneous multicores, even though in most cases they in general need a host processor to be able to run a complete application. The principle of *hardware acceleration* – adding a special purpose HW unit to off load the processor or to speed up computation by doing specific functions in HW instead of software - has a long history. About 30 years ago, a common practice for speeding up floating-point operations in a PC was to add a floating-point coprocessor unit (FPU). Today, the inclusion of different accelerator subunits in a CMP is becoming increasingly popular, and IBM has recently announced a processor architecture where processing cores and hardware accelerators are closely coupled [16].

Similarly, the GPU was added to accelerate the processing of graphics. GPUs have during the last two decades been through a substantial development from specialized units for graphics processing only to more programmable units being popular for general-purpose GPU (GPGPU). Their programming has become substantially improved through languages such as CUDA™ and OpenCL™ [32, 8].

1.4.2 Examples of Multicore Architectures

1.4.2.1 The Cell™ Processor Architecture

The Cell Broadband Engine™ (Cell/BE) is a heterogeneous processor that was jointly developed by Sony®, Toshiba® and IBM®. As shown in Figure 1.11, it is mainly composed of one main core (power processing element (PPE)), 8 specialized cores (called synergistic processing elements (SPEs)), an on-chip memory controller and a controller for a configurable I/O interface, all linked together by an element interconnection bus (EIB) [46]. The main core is a 64-bit Power processor with vector processing extensions and two levels of hardware-managed caches, a 32 KB L1 data cache and a 512 KB L2 cache. In addition, it is a dual-issue, dual-threaded processor that has a single-precision peak of 25.6 Gflops/s and a double-precision peak of 6.4 Gflops/s.

The 8 SPEs are SIMD cores (SPU) which each possess a 256 KB local store (LS) for storing both data and instructions, a 128 × 128-bit register file and a memory flow controller (MFC). MFC has the capability to move code and data between main