PARALLEL COMPUTING ON HETEROGENEOUS NETWORKS

Alexey Lastovetsky
University College, Dublin, Ireland

WILEY-INTERSCIENCE

A JOHN WILEY & SONS, INC., PUBLICATION

PARALLEL COMPUTING ON HETEROGENEOUS NETWORKS

PARALLEL COMPUTING ON HETEROGENEOUS NETWORKS

Alexey Lastovetsky
University College, Dublin, Ireland

WILEY-INTERSCIENCE

A JOHN WILEY & SONS, INC., PUBLICATION

Library of Congress Cataloging-in-Publication Data:

ISBN: 0-471-22982-2

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To my wife Gulnara, my daughters Olga and Oksana, and my parents Leonid and Lyudmila.

CONTENTS

Acknowledgments **xiii**
Introduction **1**

PART I EVOLUTION OF PARALLEL COMPUTING **9**

1. Serial Scalar Processor **11**

1.1. Serial Scalar Processor and Programming Model 11
1.2. Basic Program Properties 11

2. Vector and Superscalar Processors **15**

2.1. Vector Processor 15
2.2. Superscalar Processor 18
2.3. Programming Model 21
2.4. Optimizing Compilers 22
2.5. Array Libraries 33
2.5.1. Level 1 BLAS 34
2.5.2. Level 2 BLAS 35
2.5.3. Level 3 BLAS 40
2.5.4. Sparse BLAS 43
2.6. Parallel Languages 44
2.6.1. Fortran 90 45
2.6.2. The C[ ] Language 50
2.7. Memory Hierarchy and Parallel Programming Tools 59
2.8. Summary 63

3. Shared Memory Multiprocessors **65**

3.1. Shared Memory Multiprocessor Architecture and Programming Models 65
3.2. Optimizing Compilers 67
3.3. Thread Libraries 68
3.3.1. Operations on Threads 69
3.3.2. Operations on Mutexes 71

3.3.3. Operations on Condition Variables 73
3.3.4. Example of MT Application: Multithreaded Dot Product 75
3.4. Parallel Languages 78
3.4.1. Fortran 95 78
3.4.2. OpenMP 80
3.5. Summary 94

4. Distributed Memory Multiprocessors 95

4.1. Distributed Memory Multiprocessor Architecture: Programming Model and Performance Models 95
4.2. Message-Passing Libraries 103
4.2.1 Basic MPI Programming Model 104
4.2.2. Groups and Communicators 106
4.2.3. Point-to-Point Communication 111
4.2.4. Collective Communication 120
4.2.5. Environmental Management 127
4.2.6. Example of an MPI Application: Parallel Matrix-Matrix Multiplication 127
4.3. Parallel Languages 130
4.4. Summary 138

5. Networks of Computers: Architecture and Programming Challenges 141

5.1. Processors Heterogeneity 142
5.1.1. Different Processor Speeds 142
5.1.2. Heterogeneity of Machine Arithmetic 146
5.2. Ad Hoc Communication Network 147
5.3. Multiple-User Decentralized Computer System 150
5.3.1. Unstable Performance Characteristics 150
5.3.2. High Probability of Resource Failures 150
5.4. Summary 154

PART II PARALLEL PROGRAMMING FOR NETWORKS OF COMPUTERS WITH MPC AND HMPI 157

6. Introduction to mpC 159

6.1. First mpC Programs 159
6.2. Networks 164
6.3. Network Type 170
6.4. Network Parent 173

6.5. Synchronization of Processes 178
6.6. Network Functions 182
6.7. Subnetworks 186
6.8. A Simple Heterogeneous Algorithm Solving an Irregular Problem 190
6.9. The `RECON` Statement: A Language Construct to Control the Accuracy of the Underlying Model of Computer Network 196
6.10. A Simple Heterogeneous Algorithm Solving a Regular Problem 199
6.11. Principles of Implementation 206
6.11.1. Model of a Target Message-Passing Program 207
6.11.2. Mapping of the Parallel Algorithm to the Processors of a Heterogeneous Network 209
6.12. Summary 211

7. Advanced Heterogeneous Parallel Programming in mpC 215

7.1. Interprocess Communication 215
7.2. Communication Patterns 233
7.3. Algorithmic Patterns 241
7.4. Underlying Models and the Mapping Algorithm 244
7.4.1. Model of a Heterogeneous Network of Computers 244
7.4.2. The Mapping Algorithm 247
7.5. Summary 253

8. Toward a Message-Passing Library for Heterogeneous Networks of Computers 255

8.1. MPI and Heterogeneous Networks of Computers 255
8.2. HMPI: Heterogeneous MPI 257
8.3. Summary 261

PART III APPLICATIONS OF HETEROGENEOUS PARALLEL COMPUTING 263

9. Scientific Applications 265

9.1. Linear Algebra 265
9.1.1. Matrix Multiplication 265
9.1.2. Matrix Factorization 288
9.1.3. Heterogeneous Distribution of Data and Heterogeneous Distribution of Processes Compared 295
9.2. *N*-Body Problem 298

9.3. Numerical Integration 300
9.3.1. Basic Quadrature Rules 301
9.3.2. Adaptive Quadrature Routines 304
9.3.3. The `quanc8` Adaptive Quadrature Routine 307
9.3.4. Parallel Adaptive Quadrature Routine for Heterogeneous Clusters 313
9.4. Simulation of Oil Extraction 323
9.5. Summary 328

10. Business and Software Engineering Applications 331

10.1. Acceleration of Distributed Applications 331
10.1.1. Introduction 331
10.1.2. Distributed Application of a "Supermarket Chain" 332
10.1.3. Parallel Implementation of the Remote Operation `getDistribution` 334
10.1.4. Experimental Results 337
10.2. Parallel Testing of Distributed Software 338
10.2.1. Motivation 338
10.2.2. Parallel Execution of the Orbix Test Suite on a Cluster of Multiprocessor Workstations 339
10.2.3. Experimental Results 350
10.3. Summary 350

APPENDIXES 353

Appendix A. The mpC *N*-Body Application 353

A.1. Source Code 353
A.2. User's Guide 368

Appendix B. The Block Cyclic Matrix Multiplication Routine for Heterogeneous Platforms 371

B.1. Source Code 371
B.2. User's Guide 383

Appendix C. The Parallel Adaptive Quadrature Routine 385

C.1. Source Code 385
C.2. User's Guide 395

Appendix D. The mpC User's Guide 397

D.1. Definition of Terms 397
D.2. Outline of the mpC Programming Environment 397
D.3. Supported Systems 398

D.4. The mpC Compiler 399
D.4.1. Options 399
D.4.2. Pragmas 401
D.5. How to Start Up 401
D.6. Virtual Parallel Machine 402
D.6.1. VPM Description File 403
D.7. Environmental Variables 404
D.7.1. WHICHMPI 404
D.7.2. MPIDIR 404
D.7.3. MPCHOME 404
D.7.4. MPCLOAD 405
D.7.5. MPCTOPO 405
D.8. How to Run mpC Applications 405
D.8.1. mpccreate 406
D.8.2. mpcopen 406
D.8.3. mpcbcast 407
D.8.4. mpcload 407
D.8.5. mpcrun 408
D.8.6. mpctouch 408
D.8.7. mpcclose 408
D.8.8. mpcclean 408
D.8.9. mpcmach 408
D.8.10. mpcdel 409
D.9. How to Debug mpC Applications 409
D.10. Sample mpC Sessions 409
D.10.1. A Simple Session 409
D.10.2. More Complicated Session 412

Bibliography **415**

Index **417**

ACKNOWLEDGMENTS

I would like to express my deep appreciation to my colleagues and friends Alexey Kalinov, Ilya Ledovskih, Dmitry Arapov, and Mikhail Posypkin for the happy days when we were working together on the mpC programming language. Their skills, devotion, and creativity created the basis for its successful implementation. I am very grateful to Victor Ivannikov for his persistent support of both the mpC project and myself in good and bad times. I am also very grateful to Ted Lewis without whose support and contribution the mpC project would not be possible at all. My special thanks are to Hesham El-Rewini and Albert Zomaya for their positive and encouraging attitude to the idea of this book. I also wish to thank Ravi Reddi for his comments and valuable contribution in the material presented in Sections 8.2, 9.1.1.3, and 9.1.1.4.

Introduction

The current situation with parallel programming resembles computer programming before the appearance of personal computers. Computing was concentrated in special computer centers, and computer programs were written by nerds. Soon after PCs appeared, computer programming became available to millions of ordinary people. The result of the change can be clearly seen now.

Similarly parallel computing is now concentrating mainly in supercomputer centers established around specialized high-performance parallel computers or clusters of workstations, and only highly trained people write parallel programs for the computer systems. At the same time, local networks of computers have become personal supercomputers available to millions of ordinary people. They only need appropriate programming languages and tools to write fast and portable parallel applications for the networks. The release of the huge performance potential currently hidden in networks of computers might have even a more significant impact on science and technology than the invention of more powerful processors and supercomputers.

The intent of this book is to introduce into the area of parallel computing on common local networks of computers NoCs.

Nowadays NoCs are a common and widespread parallel architecture. In general, a NoC comprises PCs, workstations, servers, and sometimes supercomputers interconnected via mixed communication equipment. Traditional parallel software was developed for homogeneous multiprocessors. It tries to distribute computations evenly over available processors and therefore cannot utilize the performance potential of this heterogeneous architecture. A good parallel program for a NoC should distribute computations and communications over the NoC unevenly, taking into account actual performances of both processors and communication links. This book mainly introduces in parallel programming for heterogeneous, in other words, NoCs in heterogeneous parallel programming.

To present both basic and advanced concepts of heterogeneous parallel programming, the book extensively uses the mpC language. This is a high-level

Parallel Computing on Heterogeneous Networks, by Alexey Lastovetsky
ISBN 0-471-22982-2 Copyright © 2003 by John Wiley & Sons, Inc.

language aimed at programming portable parallel computations on NoCs. The design of this language allows easy expression in portable form of a wide range of heterogeneous parallel algorithms. The introduction to the mpC language and the accompanying programming model and language constructs serves also to introduce the area of heterogeneous parallel programming. A representative series of mpC programs was carefully selected to illustrate all of the presented concepts. All of the programs can be compiled and executed on a local network of workstations or even on a single workstation with the freely available mpC programming system installed. While the basic concepts are illustrated by very simple programs, a representative set of real-life problems and their portable parallel solutions on NoCs are also included. The problems studied here involve linear algebra, modeling of oil extraction, integration, *N*-body applications, data mining, business applications, and distributed software testing.

It is important for any book to clearly define from the beginning basic terms, especially if the terms are in common use but are understood differently. This is particularly true in parallel computing on distributed memory architectures, where a specific case is parallel computing on heterogeneous networks. Indeed, it is easy to confuse parallel computing on distributed memory architectures with distributed computing, especially high-performance distributed computing. In a distributed memory computer system both parallel and distributed applications are nothing more than a number of processes running in parallel on different computing nodes and interacting via message passing. They both can use the same communication protocols (e.g., TCP/IP) and the same basic software (e.g., sockets).

The key difference between parallel computing technologies and distributed computing technologies lies in the main goal of each of the technologies. The main goal of distributed computing technologies is to make software components, inherently located on different computers, work together. The main goal of parallel computing technologies is to speed up the solution of a single problem on the available computer hardware. Correspondingly, in the case of parallel computing, the partition of an application into a number of distributed components located on different computers is just a way to speed up its execution on the distributed memory computer system; it is not an intrinsic feature of the application nor of the problem that the application solves. The book presents the technology of heterogeneous parallel computing proceeding from this basic understanding of the main goal of parallel computing technologies.

Thus the target computer hardware is the material basis of any parallel computing technology. Correspondingly the evolution of computer hardware is followed by the evolution of parallel computing technologies. In general, the resulting trajectory of computer hardware is aimed at higher performance, that is, at the ability to compute faster and store more data. There exist two ways to make computer systems execute the same volume of computations faster: reduce the time of execution of a single instruction (i.e., to increase the processor clock rate), and increase the number of instructions executed in parallel. The first way is determined by the level of microelectronic technology, and it has some natural limits conditioned by universal physical constants such as the velocity of light. Nowadays the clock rates have reached the magnitude of gigacycles per second. But it is not the higher clock rate that distinguishes high-performance computer systems. Rather, at any stage of the development of microprocessor technology, this index is approximately the same for most manufactured microprocessors. The high-performance computer systems can handle a higher parallelism of computations. Therefore high-performance computer systems are always of parallel architecture.

Part I provides an overview of the evolution of parallel computer architectures in relation to the evolution of parallel programming models and tools. The starting-point of all parallel architectures is the serial scalar processor. Main architectural milestones include vector and superscalar processors, a shared memory multiprocessor, a distributed memory multiprocessor, and a common heterogeneous network of computers. The parallel architectures represent the logic of a parallel architecture development rather than its chronology. They represent the main stream of architectural ideas that have proved their viability and effectiveness and made a major impact on the real-life hardware. Compared to a historical approach, the logical approach gives a concise and conceptually clear picture of the evolution of parallel architectures, throwing off a good deal of secondary, nonviable, or simply erroneous architectural decisions, and separating more strictly different architectural concepts often mixed in real hardware. In the series of parallel architectures, each next architecture contains the preceding one as a particular case, and provides more parallelism and, hence, more performance potential.

For each of the listed parallel architectures, its intrinsic model of parallel program is presented and followed by outline of programming tools implementing the model. The models represent all main paradigms of parallel programming. Apart from optimizing compilers for traditional serial programming languages, the programming tools outlined include parallel libraries and parallel programming languages. The book does not pretend to cover all aspects of parallel programming. Many important topics such as debugging of parallel applications and maintenance of fault tolerance of parallel computations are beyond the scope of this book. The book focuses on basic parallel programming models and their implementation by the most popular parallel programming tools.

In Chapter 2 vector and superscalar processors are presented. The architectures provide instruction-level parallelism, which is best exploited by applications with intensive operations on arrays. Such applications can be written in a serial programming language, such as C or Fortran 77, and complied by dedicated optimizing compilers performing some specific loop optimizations. Array libraries allow the programmers to avoid the use of dedicated compilers performing sophisticated optimizations. Instead, the programmers express operations on arrays directly, using calls to carefully implemented subroutines implementing the array operations. Parallel languages, such as Fortran 90 or C[ ], combine advantages of the first and second approaches. They allow the programmer explicitly express operations on arrays, and they therefore do not need to use sophisticated algorithms to recognize parallelized loops. They are able to perform global optimization of combined array operations. Last, unlike existing array libraries, they support general-purpose programming.

In Chapter 3 the shared memory multiprocessor architecture is shown to provide a higher level of parallelism than the vector and superscalar architectures via multiple parallel streams of instructions. Nevertheless, the SMP architecture is not scalable. The speedup provided by this architecture is limited by the bandwidth of the memory bus. Multithreading is the primary programming model for the SMP architecture. Serial languages, such as C and Fortran 77, may be used in concert with optimizing compilers to write efficient programs for SMP computers. Unfortunately, only a limited and simple class of multithreaded algorithms can be implemented in an efficient and portable way by this approach. Thread libraries directly implement the multithreading paradigm and allow the programmers to explicitly write efficient multithreaded programs independent of optimizing compilers. Pthreads are standard for Unix platforms supporting thus efficiently portable parallel programming Unix SMP computers. Thread libraries are powerful tools supporting both parallel and distributed computing. The general programming model underlying the thread libraries is universal and seen too powerful, complicated, and error-prone for parallel programming. OpenMP is a high-level parallel extension of Fortran, C, and C++, providing a simplified multithreaded programming model based on the master/slave design strategy, and aimed specifically at parallel computing on SMP architectures. OpenMP significantly facilitates writing parallel mutlithreaded applications.

In Chapter 4 the distributed memory mutltiprocessor architecture, also known as the MPP architecture, is introduced. It provides much more parallelism than the SMP architecture. Moreover, unlike all other parallel architectures, the MPP architecture is scalable. It means that the speed increase provided by this architecture is potentially infinite. This is due to the absence of principal bottlenecks, such as might limit the number of efficiently interacting processors. Message passing is the dominant programming model for the MPP architecture. As the MPP architecture is farther away from the serial scalar architecture than the vector, superscalar, and even SMP architectures, it is very difficult to automatically generate an efficient message-passing code for the serial source code written in C or Fortran 77. In fact, optimizing C or Fortran 77 compilers for MPPs would involve solving the problem of automatic synthesis of an efficient message-passing program using the source serial code as a specification of its functional semantics. This problem is still a challenge for researchers. Therefore no industrial optimizing C or Fortran 77 compiler for the MPP architecture is now available. Basic programming tools for MPPs are message-passing libraries and high-level parallel languages. Message-passing libraries directly implement the message-passing paradigm and allow the programmers to explicitly write efficient parallel programs for MPPs. MPI is a standard message-passing interface supporting efficiently portable parallel programming MPPs. Unlike the other popular message-passing library, PVM, MPI supports modular parallel programming and hence can be used for development of parallel libraries. MPI is a powerful programming tool for implementing a wide range of parallel algorithms on MPPs in highly efficient and portable message-passing applications. Scientific programmers, who find the explicit message passing provided by MPI tedious and error-prone, can use data parallel programming languages, mainly HPF, to write programs for MPPs. When programming in HPF, the programmer specifies the strategy for parallelization and data partitioning at a higher level of abstraction, based on the single-threaded data parallel model with a global name space. The tedious low-level details of translating from an abstract global name space to the local memories of individual processors and the management of explicit interprocessor communication are left to the compiler. Data parallel programs are easy to write and debug. However, the data parallel programming model allows the programmer to express only a limited class of parallel algorithms. HPF 2.0 addresses the problem by extending purely data-parallel HPF 1.1 with some task parallel features. The resulting multi-paradigm language is more complicated and not as easy to use as pure data parallel languages. Data parallel languages (i.e., HPF) are difficult to compile. Therefore it is hard to get top performance via data parallel programming. The efficiency of data parallel programs strongly depends on the quality of the compiler.

In Chapter 5 we analyze challenges associated with parallel programming for common networks of computers (NoCs) that are, unlike dedicated parallel computer systems, inherently heterogeneous and unreliable. This analysis results in description of main features of an ideal parallel program running on a NoC. Such a program distributes computations and communications unevenly across processors and communications links during the execution of the code of the program. The distribution may be different for different NoCs and for different executions of the program on the same NoC, depending on the work load of its elements. The program keeps running even if some resources in the executing network fail. In the case of resource failure, it is able to reconfigure itself and resume computations from some point in the past. The program takes into account differences in machine arithmetic on different computers and avoids erroneous behaviour of the program that might be caused by the differences.

Part II is the core of the book and presents the mpC parallel programming language.

In Chapter 6 a basic subset of the mpC language is described. It addresses some primary challenges of heterogeneous parallel computing, focusing on uneven distribution of computations in heterogeneous parallel algorithms, and the heterogeneity of physical processors of NoCs. The programmers can explicitly specify the uneven distribution of computations across parallel processes dictated by the implemented heterogeneous parallel algorithm. The mpC compiler will use the provided information to map the parallel processes to the executing network of computers. A simple model of the executing network is used when parallel processes of the mpC program are mapped to physical processors of the network. The relative speed of the physical processors is a key parameter in the model. In heterogeneous environments the speed parameter is sensitive to both the code executed by the processors and the current work load due to external computations. The programmers can control the accuracy of this model at runtime and adjust its parameters to their particular applications. The implementation of the basic version of the mpC language is freely available at *http://www.ispras.ru/~mpc*.

In Chapter 7 some advanced features of the mpC language are presented. In general, the mC language allows the user not only to program computations and communications of the heterogeneous parallel algorithm but also to specify the performance model of the algorithm. This performance model takes into account all the main features of the algotihm that affect its execution time, including the number of parallel processes executing the algorithm, the absolute volume of computations performed by each of the processes, the absolute volume of data transferred between each pair of processes, and the interactions between the parallel processes during the execution of the algorithm. This information is used at runtime to map the algorithm to physical processors of the network of computers. The mpC programming system performs the mapping to minimize the execution time of the algorithm. The implementation of the full mpC language is freely available for research and educational purposes from the author of this book.

In Chapter 8 we very briefly consider how the mpC language approach to optimal heterogeneous distribution of computations and communications can be implemented in the form of a message-passing library. In so doing, we provide a small extension of the standard MPI for heterogeneous NoCs.

Part III presents a number of advanced mpC applications solving different problems on heterogeneous clusters.

In Chapter 9 we demonstrate that a wide range of scientific problems can be efficiently solved on heterogeneous networks of computers. We consider in details the design of the parallel block cyclic algorithm of matrix multiplication on heterogeneous NoCs and its portable implementation in the mpC language. We also consider parallel algorithms solving on heterogeneous NoCs a more demanding linear algebra problem: Cholesky factorization of a symmetric, positive-definite matrix. We present a relatively simple approach to assessment of a heterogeneous parallel algorithm via comparing its efficiency with the efficiency of its homogeneous prototype. We present two approaches to design of parallel algorithms solving regular problems on heterogeneous NoCs. The first approach supposes a one process per processor configuration of the parallel program with the work load unevenly distributed over the processes. The second approach assumes a mutliple processes per processor configuration of the parallel program, when the work load is evenly distributed over the processes while the number of processes on each processor is proportional to its speed. We experimentally compare the approaches and describe their portable mpC implementation. We present an *N*-body mpC application, which is an example of inherently irregular problem. A heterogeneous parallel algorithm solving such a problem is naturally deduced from the problem itself rather than from the parallel environment executing the algorithm. We also consider in detail the design of the parallel adaptive quadrature routine for numerical approximation to definite integrals on heterogeneous NoCs and its portable mpC implementation. We end with an experience of solving a real-life regular problem—simulation of oil extraction—in a heterogeneous parallel environment.

In Chapter 10 we demonstrate that heterogeneous parallel computing can be used not only to solve scientific problems but also to improve the performance of business distributed applications. Heterogeneous parallel computing also has application in software engineering practice to optimize the maintenance process. An experience of integration of the mpC-based technology of heterogeneous parallel computing and the CORBA-based technology of distributed computing is demonstrated.

The book does not pretend to be an encyclopedia of parallel computing. It presents a subjective view on parallel programming technologies, but Part I of the book can nevertheless be a good basis for an introductory university course on parallel programming systems. All the source code given in this book was carefully tested.

PART I

EVOLUTION OF PARALLEL COMPUTING

CHAPTER 1

Serial Scalar Processor

1.1. SERIAL SCALAR PROCESSOR AND PROGRAMMING MODEL

The starting-point of evolution of parallel architectures is the traditional serial scalar von Neumann architecture. This traditional architecture provides single control flow with serially executed instructions operating on scalar operands. Figure 1.1 depicts schematically the architecture. The processor has one instruction execution unit (IEU). Execution of an instruction can be only started after execution of the previous instruction in the flow is terminated. Except for a relatively small number of special instructions for data transfer between main memory and registers, the instructions take operands from and put results to scalar registers (a scalar register is a register that holds a single integer or float number). The total time of program execution is equal to the sum of execution times of the instructions. Performance of that architecture is determined by the clock rate.

1.2. BASIC PROGRAM PROPERTIES

Many languages and tools have been designed for programming traditional serial scalar processors, but C and Fortran have undoubtedly proved to be the most popular among professionals. What is so special in these two languages that makes them so successful and generally recognized? The answer is that both C and Fortran support and facilitate development of software whose properties are considered basic and necessary by most professionals.

While Fortran is mostly used for scientific programming, the C language is more general purpose widely used for system programming. The C language can be adapted for programming in the Fortran-like style. Moreover any Fortran 77 program can be easily converted into an equivalent C program (in particular, the GNU Fortran 77 compiler is implemented as such a convertor). So it is reasonable to assume that apart from the traditional affection of scientific programmers for Fortran, the same properties make Fortran attractive

Parallel Computing on Heterogeneous Networks, by Alexey Lastovetsky
ISBN 0-471-22982-2 Copyright © 2003 by John Wiley & Sons, Inc.

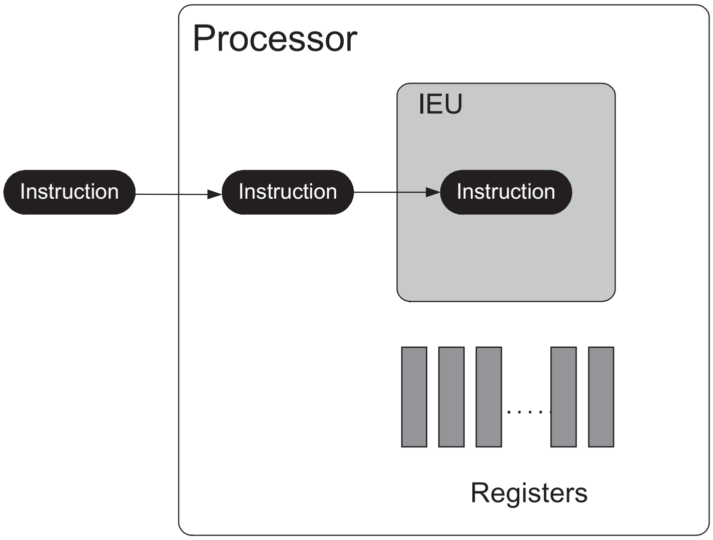

Figure 1.1. Serial scalar processor architecture.

for scientific programming and C for general purpose and especially for system programming. Therefore we will refer mostly to C while analyzing basic software properties that are to be supported by successful programming tools.

First of all, the C language allows one to develop highly efficient software for any particular serial scalar processor. This is because the language reflects all of the main features of the architecture having an effect on the program efficiency such as machine-oriented data types (short, char, unsigned, etc.), indirect addressing and address arithmetics (arrays, pointers and their correlation), and other machine-level notions (increment/decrement operators, the sizeof operator, cast operators, bit-fields, bitwise operators, compound assignments, etc.). The traditional serial scalar architecture is reflected in the C language with a completeness that allows programmers to write, for each serial scalar processor, programs having practically the efficiency of assembly code. In other words, the C language supports efficient programming.

Second, the C language is standardized as ANSI C, and all good C compilers support the standard. This allows programmers to write in C applications that, once developed and tested on one particular platform, will run properly on all platforms. In other words, the C language supports portable programming. Portability of C applications is determined not only by the portability of their source code but by the portability of used libraries as well. The C language provides especially high level of portability for computers running either the same operating system or operating systems of the same family (e.g., different clones of Unix). The point is that in addition to standard ANSI C libraries, a lot of other libraries are de facto standard in the framework of the corresponding family of operating systems. As for Unix systems, the high-quality portable GNU C compiler is often used on many platforms instead of native C compilers, which provides still more portability for source C code.

Third, the C language allows programmer to develop program units that can be separately compiled and correctly used by other programmers, while developing their applications, without knowledge of their source code. In other words, the C language supports modular programming. Obviously packages and libraries can only be developed with tools supporting modular programming.

Fourth, the C language provides a clear and easy-in-use programming model that ensures reliable programming. In addition to modularity it facilitates the development of really complex and useful applications. It is very difficult to find a balance between efficiency and lucidity as well as to combine lucidity and expressiveness. The C language is an exceptionally rare example of such harmony.

Finally, the C language supports not only efficient and portable but efficiently portable programming the serial scalar processors. It has been stressed that the C language reflects all the main features of this architecture that affect program efficiency. On the other hand, the C language hides such peculiarities of each particular processor that have no analogs in other processors of the architecture (the peculiarities of register storage, details of stack implementation, details of instruction sets, etc.). It allows writing portable applications that run efficiently on any particular serial scalar platform having both a high-quality C compiler and efficiently implemented libraries.

Note that any tool supporting efficiently portable programming also supports both efficient and portable programming. However, not every tool enabling both efficient and portable programming of its target architecture, has to enable efficiently portable programming of this architecture as well. Indeed, if some tool allows one to write a portable application as well as manually to optimize the application for every particular representative of the architecture, it does not mean that the application will run efficiently on every system of the architecture without changes in its source code.

Of course, the five properties above do not exhaust all possible properties that can appear important for one or another kind of software (fault tolerance for controlling software, scalability for parallel software, etc.). But these five are primary, and can be summarized in plain language as follows. Not too many programmers will want to use programming tools that subject them to these disadvantages.

- Do not allow them to utilize efficiently the performance potential of their computers.
- Do not allow them to write programs that can run on different computers.

- Do not allow them to write program modules that can be used by other programmers.
- Are based on a sophisticated set of ideas that make applications complex and tedious and also error-prone.
- Do not allow them to write portable applications running efficiently over a target group of computers.

In this book parallel programming tools are mainly assessed from the point of view of how well they support these basic program properties.